EFFECTIVE AND EFFICIENT COMPUTATION SYSTEM PROVENANCE TRACKING

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Shiqing Ma

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

August 2019

Purdue University

West Lafayette, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL

STATEMENT OF DISSERTATION APPROVAL

Dr. Xiangyu Zhang, Chair

Department of Computer Science

Dr. Dongyan Xu

Department of Computer Science

Dr. Byoungyoung Lee

Department of Computer Science

Dr. Chunyi Peng

Department of Computer Science

Approved by:

Dr. Voicu S. Popescu

Head of the Department Graduate Program

To my beloved family members.

ACKNOWLEDGMENTS

Firstly, I would like to thank my advisor Professor Xiangyu Zhang for his support of my Ph.D. study and research, for his patience, guidance, motivation, understanding and immense knowledge. In the past six years, I have learned so much from him about being a researcher in computer science, a professor in university, and an advisor for students. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my Ph.D study.

I am deeply grateful to my co-advisor Professor Dongyan Xu, who has given generous help and guidance for me. He has taught me so many skills and he has always been a role model to me. I cannot express how fortunate I am to have him as my co-advisor.

Besides my advisors, I would like to thank the rest of my thesis committee: Prof. Byoungyoung Lee and Prof. Chunyi Peng, for their insightful comments and encouragement, for the suggestions to improve my research and dissertation. My sincere thanks also goes to Professor Somesh Jha from University of Wisconsin, Madison, for serving my preliminary committee, for providing help in my Ph.D. research.

I dedicate this dissertation to my beloved family members. Their unconditional support and love helped me get through hard times in my life. I am always lost to words when it comes to my parents, Shuying Cui and Jianchang Ma, my brother and sister-in-law, Buqing Ma and Hui Kang, my sister and brother-in-law, Linqing Ma and Zheng Wang, and especially, my wife and the love of my life, Juan Zhai who I shared laughs and tears with and together we got through our Ph.D.s. I love them from the deepest part of my heart.

Lastly, I count myself lucky to work with great labmates such as Kyu Hyung Lee, Yingqi Liu, Guanhong Tao, Jianjun Huang, Wei You, Wen-Chuan Lee, Le Yu, Brendan Saltaformmagio, Rohit Bhatia, Hui Lu, Yousra Aafer, Fei Wang, Yonghwi Kwon, Chung Hwan Kim and Kexin Pei and my roommates such as Longyun Guo, Zhang Li and Yongyang Yu. They all helped me a lot in the past six wonderful years in West Lafayette.

TABLE OF CONTENTS

Page

LIST OF TABLES viii

LIST OF FIGURES ix

ABSTRACT xii

1 INTRODUCTION 1
- 1.1 Dissertation Statement 2
- 1.2 Contributions 2
- 1.3 Dissertation Organization 4
- 1.4 Dissertation Overview 4
  - 1.4.1 Provenance System in Enterprise Environment for Attack Forensics 4
  - 1.4.2 Provenance in Deep Learning Based Artificial Intelligence Systems for Adversarial Sample Detection 6

2 PROTRACER: TOWARDS PRACTICAL PROVENANCE TRACING BY ALTERNATING BETWEEN LOGGING AND TAINTING 8
- 2.1 Introduction 8
- 2.2 Motivation 13
- 2.3 System Architecture 19
- 2.4 Tainting and Logging in the User Space Daemon 21
  - 2.4.1 Definitions 23
  - 2.4.2 Run-time Operation Rules 26
  - 2.4.3 Handling Global File Accesses 31
- 2.5 Evaluation 33
  - 2.5.1 Effectiveness 33
  - 2.5.2 Logging Overhead and Scalability 36
  - 2.5.3 Attack Investigation Cases 38
- 2.6 Discussion 42
- 2.7 Related Work 43

3 MPI: MULTIPLE PERSPECTIVE ATTACK INVESTIGATION WITH SEMANTIC AWARE EXECUTION PARTITIONING 45
- 3.1 Introduction 45
- 3.2 Motivation 48
  - 3.2.1 Motivating Example 49
  - 3.2.2 Traditional Solutions 50
  - 3.2.3 Loop Based Partitioning Solutions 51

Page

3.2.4 Our Approach . . . 54
3.3 Design . . . 57
3.3.1 Overview . . . 57
3.3.2 Annotations . . . 57
3.3.3 Runtime . . . 65
3.3.4 Analysis . . . 67
3.3.5 Run Time Optimization . . . 68
3.3.6 Causal Graph Construction . . . 69
3.4 Evaluation . . . 72
3.4.1 Overhead . . . 72
3.4.2 More Overhead Experimental Results . . . 76
3.4.3 Annotation Efforts . . . 78
3.4.4 Annotation Miner . . . 79
3.4.5 Attack Investigation . . . 79
3.4.6 Case Studies . . . 82
3.5 Discussion . . . 85
3.6 Related work . . . 87

4 NIC: DETECTING ADVERSARIAL SAMPLES WITH NEURAL NETWORK INVARIANT CHECKING . . . 89
4.1 Introduction . . . 89
4.2 Background and Related Work . . . 92
4.2.1 Neural Networks . . . 92
4.2.2 Adversarial Samples . . . 93
4.2.3 Existing Attacks . . . 96
4.2.4 Existing Defense and Detection . . . 99
4.3 Observations . . . 103
4.3.1 Observations about DNN Attack Channels . . . 103
4.3.2 Detection As Invariant Checking . . . 105
4.4 System Design . . . 107
4.4.1 Overview . . . 107
4.4.2 Extracting DNN Invariants . . . 110
4.4.3 Training Invariant Models . . . 111
4.4.4 Randomization . . . 112
4.4.5 Threat Model . . . 114
4.5 Evaluation . . . 115
4.5.1 Setup . . . 115
4.5.2 Detecting Gradient Based Attacks . . . 117
4.5.3 Detecting Content Based Attacks . . . 119
4.5.4 NIC Invariants and Comparison . . . 119
4.5.5 Adaptive Adversaries . . . 124
4.5.6 Choice of One Class Classifiers . . . 126
4.5.7 Case Study . . . 126

Page
4.6 Related Work 128
5 CONCLUSION 130
REFERENCES 132
VITA 147

LIST OF TABLES

Table Page

2.1 ProTracer: Logging and Tainting Rules. 25

2.2 Comparison of Effectiveness of Various Provenance Systems 35

2.3 Effectiveness of ProTracer on Real-world Attack Scenarios 40

2.4 Causal Graph Comparison (BEEP/ProTracer). 41

3.1 MPI Space Overhead 73

3.2 MPI Annotation Efforts 78

4.1 Performance of NIC on Gradient Based Attacks 117

4.2 Performance of NIC on Content Based Attacks 118

4.3 NIC: Comparison with State-of-the-art Detectors 122

4.4 NIC: Effects of Different OCC Algorithms 125

4.5 NIC: PI and OPs 127

LIST OF FIGURES

Figure Page

2.1 Basic approaches to provenance tracing. (a) Actual executions in a top-down order; (b) Approach I: audit logging; (c) Approach II: provenance propagation. . 9

2.2 The simplified causal graph of a phishing attack generated by BEEP. (Ovals: execution units; Diamonds: network sessions; Rectangles: files. The nodes inside the dashed areas are those pruned away by ProTracer. 14

2.3 Linux Audit system architecture 16

2.4 ProTracer architecture overview (Dashed lines: control flow; Solid lines: data flow; Numbers: the order of the events. 19

2.5 An abstract diagram to illustrate the logging and tainting run-time. The numbers represent the order of the events. The logging/tainting behavior is highlighted in red on edges.$UT(u)$ and $MT(a)$ are simplified representations of the taint set of a unit u and address a, respectively. P denotes the current process and id_0 an ID denoting a network session. 21

2.6 Definitions for Logging and Tainting. 22

2.7 Example for the logging and tainting run-time. The shaded statements correspond to syscalls. The statements in red are those instrumented by ProTracer to generate special events. 29

2.8 Accumulated log size from the one-day execution of BEEP and ProTracer. Note the size of BEEP log is measured by *megabytes*, whereas the ProTracer log is measured by *kilobytes*. 37

2.9 Run-time overhead with different concurrent thread(s) for server programs. . . . 38

2.10 Run-time overhead for client programs. 38

2.11 Part of the graphs generated by BEEP and ProTracer for the backdoor case. . . 42

3.1 Simplified causal graph for case in §3.2.1 generated by traditional solutions . . 50

3.2 Simplified backtracking causal graph for the case in §3.2.1 with event loop based partitioning technique. It only shows the causal relationship within the Firefox process (runs for 5 minutes with 11 tabs and 7 websites) 52

3.3 Event handling loop of *Transmission* (version 2.6) 53

Figure Page

3.4 Different perspectives of Firefox partitioning 55

3.5 Simplified MPI causal graph for the case in §3.2.1 (Firefox in tabs) 55

3.6 Simplified MPI causal graph for the case in §3.2.1 (Firefox in websites) 56

3.7 MPI workflow 58

3.8 Vim data structure and our annotations 60

3.9 Simplified Firefox execution model 61

3.10 Tab and window annotations in Firefox 63

3.11 Firefox main thread and socket thread communication example 64

3.12 Example of annotation miner in MPI 65

3.13 Instrumentation example (Vim, *op_yank* function) 68

3.14 An example of constructing backward causal graph. (*UCX* is short for *Unit Context Switch*, *FDR* is short for *File Descriptor Read*, and *SKW* is short for *Socket Write*.) 72

3.15 MPI: Run-time overhead for each application 76

3.16 MPI: Run-time overhead for different partitioning perspectives 76

3.17 MPI: Space overhead for a whole day 77

3.18 Annotation miner results 79

3.19 FTP server partitioning perspectives 80

3.20 FTP server partitioned by BEEP 81

3.21 FTP server partitioned by each connection 81

3.22 FTP server partitioned by users 82

3.23 Causal graph for insider attack using BEEP 83

3.24 Causal graph for insider attack using MPI 84

3.25 Firefox in page perspective 85

4.1 **Adversarial images of different attacks on MNIST.** T: targeted. UT: untargeted. L_0, L_2 and L_∞ stand for different distance metrics (gradient based attacks) and CB stands for content based attacks. Captions are the attack names. For gradient based attacks, Figure 4.1(a) is the original image. For each attack, the first image is adversarial and the second image highlights the perturbations. For content based attacks, red boxes highlight the added contents. 95

Figure Page

4.2 Input A 102

4.3 Input B 102

4.4 PI and VI 102

4.5 Input that violates PI 102

4.6 Input that violates VI 102

4.7 Program invariant and DNN invariant 105

4.8 Overview of NIC 108

4.9 Effects of using different SVM kernels 112

4.10 **Adversarial Samples.** 1st line shows seed inputs, 2nd line shows without detector, 3rd line shows with 1 detector. Gray boxes mean no adversarial samples found in 1000 iterations. 124

4.11 Adversarial samples for the Trojan attack 127

4.12 NIC: VI and OVs 127

ABSTRACT

Ma, Shiqing Ph.D., Purdue University, August 2019. Effective and Efficient Computation System Provenance Tracking. Major Professor: Xiangyu Zhang.

Provenance collection and analysis is one of the most important techniques used in analyzing computation system behaviors. For forensic analysis in enterprise environment, existing provenance systems are limited. On one hand, they tend to log many redundant and irrelevant events causing high runtime and space overhead as well as long investigation time. On the other hand, they lack the application specific provenance data, leading to ineffective investigation process. Moreover, emerging machine learning especially deep learning based artificial intelligence systems are hard to interpret and vulnerable to adversarial attacks. Using provenance information to analyze such systems and defend adversarial attacks is potentially very promising but not well-studied yet.

In this dissertation, I try to address the aforementioned challenges. I present an effective and efficient operating system level provenance data collector, ProTracer. It features the idea of alternating between logging and tainting to perform on-the-fly log filtering and reduction to achieve low runtime and storage overhead. Tainting is used to track the dependence relationships between system call events, and logging is performed only when useful dependencies are detected. I also develop MPI, an LLVM based analysis and instrumentation framework which automatically transfers existing applications to be provenance-aware. It requires the programmers to annotate the desired data structures used for partitioning, and then instruments the program to actively emit application specific semantics to provenance collectors which can be used for multiple perspective attack investigation. In the end, I propose a new technique named NIC, a provenance collection and analysis technique for deep learning systems. It analyzes deep learning system internal variables to generate system invariants as provenance for such systems, which can be then used to as a general way to detect adversarial attacks.

1 INTRODUCTION

Computation system provenance collection and analysis is one of the most important techniques in computer security. In enterprise environment, there is an increasing need of more advanced forensics analysis techniques because of the rising of recent complex cyber attacks, such as advanced persistent threats (APTs). The attackers usually targets giant companies or governments. They intrude target networks and spread the malware by leveraging advanced social engineering techniques, zero-day vulnerabilities or highly customized malware, and the malware has very few activities for months or even years to avoid being detected. Due to its long attack cycle, the damages it makes is usually huge. Once detected, it is also very challenging to understand the root cause or the ramifications. Provenance systems play an important role in such scenarios. These systems collect runtime information of system activities and log them into log files. Once the attack is detected, investigators perform *backward analysis*, which traces the dependencies between the logged system events to find how the attack happens to understand the root cause of the attack; and *forward analysis*, which locates the affected system objects (e.g. files, sockets) and system subjects (e.g., processes) to understand the ramifications of the attack. Existing provenance systems are not capable of handling these cases. The challenges mainly lie in two aspects: the challenge of building an efficient and effective provenance system architecture to support collecting, transferring and storing provenance data, and the challenge of transforming existing applications to make them provenance-aware to enable application semantic rich investigation.

Artificial intelligence (AI) systems, including deep learning systems, are vulnerable to adversarial samples which are inputs carefully crafted by the adversary to fool AI systems. With the success of AI systems in many areas such as image classification and natural language processing (NLP), the security of AI system is becoming more and more important. On one hand, such deep learning AI systems usually have a large number of internal neurons, which represent behaviors of the AI system. Potentially, there is a great opportunity to leverage such information to analyze AI systems. On the other hand, there is no existing method to understand or interpret the internal activation values. This calls for new provenance collection and analysis techniques to help researchers and users understand or interpret the internal neurons and thus defend the adversarial samples.

In my research, I worked on solving security and software engineering problems [1, 2, 3, 4, 5, 6, 7, 8, 9]. In this dissertation, I will present my work on effective and efficient computation system provenance tracking techniques to help computer forensics tasks [10, 11, 12, 13, 14, 15, 16, 17] and AI system security [18, 19, 20, 21].

1.1 Dissertation Statement

This dissertation addresses important issues in system provenance research for traditional enterprise system and AI system. For traditional enterprise system, it first proposes ProTracer, a newly designed operating system provenance collector that features alternating between logging and tainting to enable on-the-fly event reduction to achieve low run time and storage overheads. Second, it proposes an automatic approach to transfer existing applications to be provenance-aware so that they can actively expose application specific semantics to the low level provenance systems to help better attack investigations. For AI system, it proposes a neuron activation distribution analysis based method to detect adversarial samples.

The thesis of this dissertation is as follows: *effective and efficient provenance analysis can be achieved by fine-grained system internal analysis, i.e., semantics aware execution partitioning and on-the-fly log reduction for traditional software and layer-wise neuron activation analysis for deep learning system.*

1.2 Contributions

The contributions of this dissertation are as follows:

- We propose the novel idea of combining both logging and unit level tainting for cost-effective provenance tracing. It collects system call events, performs on-the-fly filtering to remove redundant and irrelevant events by leveraging a small size logging buffer, and post-processes out-of-order events through a thread pool to achieve low runtime and storage overheads. We re-designed the kernel tracing and transmission facilities to help reduce the runtime overhead, and propose the idea of alternating between logging and tainting to locate and filter out redundant and irrelevant events. We built a prototype, ProTracer, based on the proposed idea, and the experimental results show a great improvement over existing provenance systems, such as the Linux Audit system.

- We propose and implement MPI based on the novel idea of partitioning execution according to annotated internal program data structures to enable multi-perspective, application semantics aware attack investigations. Given a small number of annotations on program data structures, MPI automatically analyzes and instruments the program to emit run time information to lower level operating system provenance systems. Such information can be used for multi-perspective investigations. We also develop an annotation miner that can be used to help programs identify internal data structures to annotate.

- We analyze the attack channels used by adversarial samples in deep learning systems, and observe two popular attack channels: the internal neuron activation value distribution channel and the provenance channel (i.e., the value distribution change channel). Based on this analysis, we propose to analyze deep learning model internal neuron behaviors of individual layers and their transitions to abstract the allowed and benign model behaviors, and use such abstractions to detect adversarial samples. Our approach has shown to be very effective in terms of detecting existing attack methods and it also increases the attack bar for adaptive attacks.

1.3 Dissertation Organization

This dissertation is organized as follows: Chapter 2 discusses the design, implementation and evaluation of ProTracer, which is a new efficient and effective operating system level provenance tracking system. Chapter 3 presents MPI, which is a LLVM based analysis and instrumentation framework to automatically transfer normal programs into provenance-aware with multiple possible perspectives to enable more powerful attack investigation. In Chapter 4, we will discuss the adversarial sample problem in existing AI systems and our proposed adversarial sample detection method.

1.4 Dissertation Overview

1.4.1 Provenance System in Enterprise Environment for Attack Forensics

Recent cyber attacks, especially APT attacks are becoming increasingly sophisticated and targeted, leading to significant damages such as data leak (e.g., huge swaths of confidential documents were stolen in the Sony Picture attack), financial loss (e.g., the Carbanak attack causes over $1 billion loss) and even threatening human life (e.g., the Stuxnet attack targets nuclear power equipment). OS level provenance tracking is a very important approach for APT attack investigation. Starting from the symptom events (e.g., the creation of a malicious process), investigators perform backward or forward tracking on the provenance data based on dependency relations (e.g., a process reads/writes a file). Traditional techniques suffer from two problems: low analysis accuracy, and significant run time and space overhead. My techniques try to solve these two problems.

Effective and Efficient Provenance Collection System Design

Prior to my work, widely used provenance collection systems like the Linux Audit framework have substantial run time and storage overhead. According to our profiling results, the root cause is the large amount of redundant events which require excessive computing resources to transfer, log and store. To address these problems, I proposed

ProTracer, a lightweight provenance system that features online redundancy reduction. It uses memory ring buffers as the data channel to speedup the data transfer process, and a novel online log redundancy detection and reduction algorithm to reduce both the run time overhead and space overhead via alternating between tainting system objects/subjects and logging system events. It treats all incoming inputs to the system as taints, and propagates them through dependency relations. Following the principle that persistent system events (e.g., process creation) should be logged and temporary system events (e.g., duplicated file reads) are used to propagate taints, we carefully designed a set of logging and tainting rules to detect and reduce redundant events without affecting the completeness and correctness of dependency analysis. The evaluation with different realistic system workloads and a number of attack cases show that ProTracer only consumes 1.28% of the space used by state-of-the-art provenance systems, 7 times smaller than a recent offline garbage collection technique with less than 7% run time overhead for servers and less than 5% run time overhead for regular applications. Moreover, the investigation process is also 7 times faster using reduced log files.

Accurate Dependency Analysis

The low analysis accuracy problem in traditional techniques is mainly caused by the dependence explosion problem: for long running processes, a subject (i.e., process) is causally related to all the objects it has accessed so far, which introduces many false dependencies. Prior to my work, the state-of-the-art work BEEP partitions these long running executions to units based on individual iterations of event handling loops which represent a common design pattern in these programs. An event handling loop accepts external requests as inputs and then dispatches them to different functions for processing in individual iteration. Each iteration constitutes a unit. With the partitioning, events in different execution units are considered independent, which helps remove many false dependencies. However, BEEP has many limitations. Firstly, execution units in BEEP expose low level semantics, which are hard for investigators to understand. Secondly, BEEP generates excessive units (e.g., units for GUI events like key strokes or mouse movements). Thirdly, to find dependency relations across units, it requires a heavyweight training process. Moreover, it performs binary rewriting to instrument the partitioning logic, which is not practical for complex programs like Firefox. To solve these problems, I proposed MPI, a multiple perspective attack investigation technique with semantics aware execution partitioning. Users first annotate the target program to define desired investigation perspectives with the assistance of our annotation helper tool. These perspectives reflect high level semantic partitioning of the target program, such as individual tabs or individual web pages for Firefox. MPI then automatically analyzes and instruments the target program to produce provenance aware applications that can proactively report its current context (e.g., current tab ID in Firefox) to the OS provenance collection system, such that events in different contexts are independent. Doing so, MPI provides accurate, multiple perspective and semantics rich attack investigations with less instrumentation and run time overhead.

1.4.2 Provenance in Deep Learning Based Artificial Intelligence Systems for Adversarial Sample Detection

DNNs are vulnerable to adversarial samples that are generated by perturbing correctly classified inputs to cause DNN models to misbehave (e.g., misclassification). This can lead to disastrous consequences in security sensitive applications. Existing defense and detection techniques work well for specific attacks but do not generalize well on different types of attacks. The execution of a DNN model is just like a program execution consisting of a sequence of matrix computations and activation functions. In the traditional software security domain, researchers proposed program invariants checking to limit the program behaviors to benign ones. For example, control flow invariants are used to check control flow integrity, and a violation indicates an attack. Inspired by this, I proposed DNN invariants to detect adversarial samples. Similar to program invariants, DNN invariants represent benign behaviors of a given model. Unlike traditional program invariants, which are logical constraints, DNN invariants are probabilistic models. There are two types of

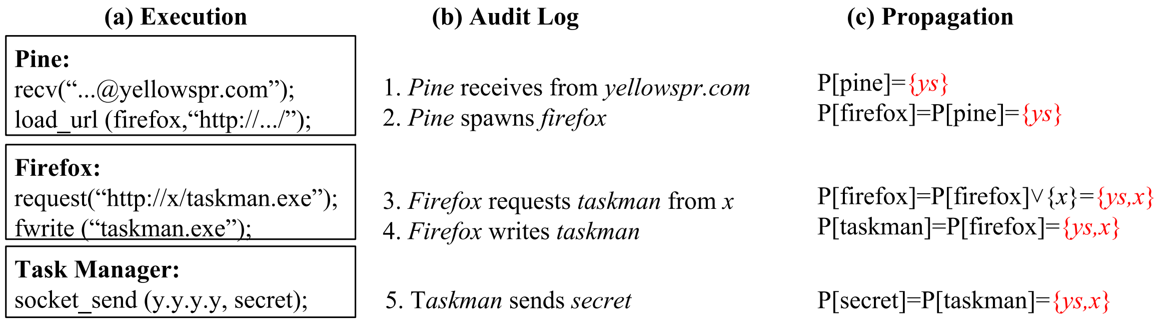
DNN invariants, the value invariant and the provenance invariant. A value invariant is activation value distributions of a specific layer for benign inputs, and a provenance invariant is activation transition patterns of two consecutive layers for benign inputs. Together, value and provenance invariants model the benign behaviors of a DNN model. A violation of any invariant indicates an adversarial sample input. We empirically compared our technique, NIC, with four other state-of-the-art detection methods (i.e., LID, MagNet, HGD, and Feature Squeezing), and found that NIC is more general to different types of attacks with over 96% detection accuracy on 11 different attacks and 13 different models.

2 PROTRACER: TOWARDS PRACTICAL PROVENANCE TRACING BY ALTERNATING BETWEEN LOGGING AND TAINTING

Provenance tracing is a very important approach to *Advanced Persistent Threat* (APT) attack detection and investigation. Existing techniques either suffer from the dependence explosion problem or have non-trivial space and run-time overhead, which hinder their application in practice. We propose ProTracer, a lightweight provenance tracing system that alternates between system event logging and unit level taint propagation. The technique is built on an *on-the-fly* system event processing infrastructure that features a very lightweight kernel module and a sophisticated user space daemon that performs concurrent and out-of-order event processing. The evaluation with different realistic system workloads and a number of attack cases show that ProTracer only produces 13MB log data per day, and 0.84GB(Server)/2.32GB(Client) in 3 months without losing any important information. The space consumption is only $< 1.28\%$ of the state-of-the-art, 7 times smaller than an off-line garbage collection technique. The run-time overhead averages $<7\%$ for servers and $<5\%$ for regular applications. The generated attack causal graphs are a few times smaller than those by existing techniques while they are equally informative.

2.1 Introduction

There is an increasing need of detecting and investigating APT attacks in an enterprise environment. A very important approach to addressing this problem is provenance tracking. According to previous works [22, 23], provenance captures multiple aspects of information about an entity in a system: *what* the entity's origin is; *how* the entity is derived; and *when* it originated. In the context of APT defense, entities with trackable provenance information are of various granularity, such as processes, network connections, files, and data items within files. Correspondingly, the *what*-provenance of such an entity e is the set of external



(a) Execution

Pine:
recv("...@yellowspr.com");
load_url (firefox,"http://.../");

Firefox:
request("http://x/taskman.exe");
fwrite ("taskman.exe");

Task Manager:
socket_send (y.y.y.y, secret);

(b) Audit Log

1. *Pine* receives from *yellowspr.com*
2. *Pine* spawns *firefox*
3. *Firefox* requests *taskman* from *x*
4. *Firefox* writes *taskman*
5. *Taskman* sends *secret*

(c) Propagation

P[pine]=$\{ys\}$
P[firefox]=P[pine]=$\{ys\}$
P[firefox]=P[firefox]$\vee\{x\}=\{ys,x\}$
P[taskman]=P[firefox]=$\{ys,x\}$
P[secret]=P[taskman]=$\{ys,x\}$

Figure 2.1.: Basic approaches to provenance tracing. (a) Actual executions in a top-down order; (b) Approach I: audit logging; (c) Approach II: provenance propagation.

entities that have causally influenced e's value or state (e.g., if one file's content comes from a number of network connections, then its *what*-provenance contains the IDs of the corresponding sessions); whereas, the *how*-provenance of entity e consists of events and their causal ordering – which can be organized as a causal graph – that demonstrates how (and when) other entities influence e's value or state.

Existing Approaches. Existing techniques fall into two categories: *audit logging* and *provenance propagation (or tainting)*. Audit logging [24, 10, 25, 26, 27, 28, 29, 30, 31] records events during system execution and then causally connects events during attack investigation. They treat processes as subjects; files, sockets, and other passive entities as objects; and assume causality between subjects and objects involved in the same syscall event (e.g., a process reading a file). In general, audit logging incurs much lower overhead than per-instruction provenance propagation. Causal graphs can be constructed to denote both *what*- and *how*-provenance. Provenance propagation, or tainting [32, 33, 34, 35, 36, 37, 38, 39] works by first assigning IDs/tags to *provenance sources* (e.g., network sessions), and then propagating the IDs through program dependencies captured during execution. Provenance propagation usually entails set operations at the instruction level. Eventually, the set of provenance IDs that reaches a *sink* (e.g., a socket for send) denotes the sink's provenance. Provenance propagation usually only captures the *what*-provenance.

Consider the example in Figure 2.1. Figure 2.1 (a) denotes a simple attack. The user received a phishing email from attacker "Yellow Spring" and opened the URL in the email through `Firefox`. Upon visiting the website, a Trojan executable for task management was saved on the local disk. Later, the malware is executed and sends some secret to a remote host. Fig. 2.1 (b) shows the events captured by audit logging. Causality can be derived from events. Depending on the precision demanded and the scope of the analysis, events can be captured at different granularity (e.g., syscalls or memory accesses) and different scopes (e.g., host or whole enterprise).

Fig. 2.1 (c) shows the provenance propagation approach. IDs `ys` and `x` denote the different provenance sources. Observe that when `pine` spawns `Firefox`, the latter inherits the provenance of the former. The malware `taskman`'s provenance is the union of the provenance set of `Firefox` and the download URL `x`. At the end, we know the origins of the stolen secret, but we do not know its history. Such propagation can be exhibited within an application, across applications, and across hosts.

Both approaches have pros and cons, and neither meets the requirements for enterprise-wide APT detection/forensics. Logging has the following limitations:

(1) Dependence explosion is a major limitation of most audit logging. For a long-running process, an output event is assumed to be causally dependent on all preceding input events, and an input event is assumed to have causal influence on all subsequent output events. Such conservative assumptions create excessive false positive causal relations, making it difficult to reveal the true causality. In our previous work, we proposed to divide an execution to autonomous units [26] such that an output is only dependent on the preceding inputs *within the same unit*.

(2) High storage overhead. According to [40], audit logging easily generates gigabytes of log data per host every day. This is particularly problematic for APT defense, as APT malware tends to lurk in the victim host for a long time.

(3) Non-trivial run-time overhead. Although logging has relatively lower run-time overhead compared to provenance propagation because it does not require expensive per-instruction set operations, many existing logging systems [26, 40] are built on the default Linux audit logging infrastructure that can cause up to 40% slow-down to the whole system due to its poor design (Section 2.5). This makes it undesirable in a production environment.

Researchers have proposed advanced infrastructures [29, 30, 31] that can achieve much lower overhead. However, to achieve the low overhead, these systems usually do not perform any online event processing, but rather just record the events, leading to substantial space consumption and dependence explosion.

The propagation-based approach features much lower space overhead compared to logging as it does not generate log. It also has higher precision due to its fine-grained instrumentation. However it has many limitations that hinder its application in the real world:

(1) Substantial run-time overhead. Because propagation based techniques track individual instructions' execution and propagate (potentially) large provenance sets (Fig. 2.1 (c)), they usually incur substantial run-time overhead. State-of-the-art implementations without hardware support incur multiple factor of slow-down [33].

(2) Lack of implicit flow handling. Many propagation based techniques have difficulty handling *implicit flow*, which is information flow through control dependencies [41] (usually induced by program predicates).

(3) Complexity in implementation. Developers have to define provenance propagation logic for each instruction, a task which is tedious and error-prone. This problem is exacerbated when programs rely on third-party libraries; internal run-time engines (e.g., VMs); and various languages and their run-times, which all require specific instrumentation/tracking mechanisms.

In this paper, we develop ProTracer that leverages the advantages of both approaches and overcomes their respective limitations. It collects system events and processes them on the fly. The cost-effective online processing filters out events that are redundant or irrelevant for provenance analysis, substantially reducing the space consumption and the size of the generated causal graphs without affecting effectiveness.

System Goals. The goal of ProTracer is to provide efficient support for both the *what*-provenance and the *how*-provenance queries on any system objects such as processes and files. For example, given a corrupted file x, two *what*-provenance queries are: (1) *"What is the source/entry point of x?"* and (2) *"which other files in the enterprise were derived* *from (and corrupted by) x?"* A sample *how*-provenance query is: *"Construct a causal graph showing the events/entities that led to the corruption of x and those that have been further corrupted by x."* We aim to achieve *completeness*. In particular, the result of a what-provenance query on x must include all the external entities that directly/transitively affected x; the result of a how-provenance query must capture the set of internal and external entities that affected x and their causal relations with x.

The technique works as follows. It first leverages a selective instrumentation technique similar to BEEP [26] to partition an execution to units, by emitting special syscalls denoting the unit boundaries. Intuitively, an unit is an iteration of the event handling loop that processes an external request or a UI event. Different from [26], ProTracer does not simply log all the syscalls and the unit related events. Instead, it alternates between logging and provenance propagation. Logging is conducted when changes are made to the permanent storage or the external environment such as writing a file and sending a packet. For other events such as file reads and network receives, ProTracer performs *coarse-grained provenance propagation (tainting)*, which taints at the level of a unit and an system object (e.g. file) instead of an instruction and a memory byte. For example, if a unit receives packets from two network sessions x_1 and x_2, the unit is tainted with both sources. If later the same unit writes to a file on disk, a log entry is emitted containing the two sources. Then if the file is read by another unit, the unit is tainted with the two sources too. Note that avoiding logging as much as possible reduces the space overhead, and performing unit level and system object level taint propagation substantially reduces the run-time overhead compared to instruction level tainting. Unit level tainting does not lose any precision compared to a log-all-events strategy. Furthermore, ProTracer decouples its implementation from the expensive Linux audit logging system. It builds from scratch a highly optimized system. It has a lightweight kernel module that simply saves events to a ring buffer. The buffer is shared with a user space daemon that retrieves these events and processes them using a thread pool. ProTracer features out-of-order event processing, meaning that the event processing order does not need to be identical to the event order, maximizing concurrency.

Our contributions are summarized as follows.

- We propose the novel idea of combining both logging and unit level tainting to achieve cost-effective provenance tracing.

- We develop an efficient run-time that features on-the-fly event processing. It not only collects system events, but also filters out the redundant and irrelevant events on the fly. It achieves low run-time overhead by out-of-order event processing through a thread pool.

- We build a prototype and evaluate it on different systems with various users and workloads for over 3 months, and on a number of real-world attacks that we reproduce. Our results show that the space consumption of ProTracer is $<1.28\%$ of BEEP's on average, and about 7 times smaller than our previous offline log garbage collection technique LogGC [40]. The log generated per day is roughly 13MB without losing precision compared to BEEP. The run-time overhead averages $<7\%$ for servers and $<5\%$ for user systems, which is 4-10 times lower than the default Linux Audit Logging system, on which many techniques including BEEP were built, and comparable to light-weight logging systems such as Hi-Fi [29, 30, 31] that simply record events without processing them.

Like most existing audit logging systems [42, 26, 40], ProTracer trusts the kernel and any user space daemon associated with the provenance tracing system. More discussion about the assumptions, limitations and security analysis of ProTracer can be found in Section 2.6.

2.2 Motivation

Scenario: We will use a cyber attack scenario to motivate our technique. It is a *phishing attack*, in which an employee received an phishing email with a malicious link via *pine*, an email client. The email mentions that a free beta version of a costly program that the employee has been hoping to own is released on the Internet. The employee was excited and decided to try it out. He clicked the link; a new tab in `Firefox` was opened; he then downloaded the file to the local machine. However, the file is actually a back-door malware.

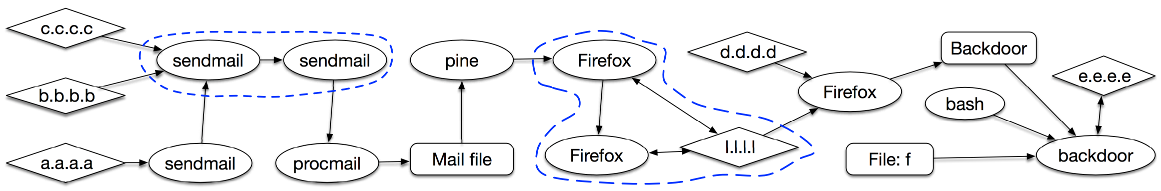


Figure 2.2.: The simplified causal graph of a phishing attack generated by BEEP. (Ovals: execution units; Diamonds: network sessions; Rectangles: files. The nodes inside the dashed areas are those pruned away by ProTracer.

Later when it is executed, a back-door process is started and sends some local file to a remote IP address.

State-of-The-Art: In BEEP [26], we observed that many programs share a common property: their execution is dominated by event handling loops. More importantly, individual iterations of these loops tend to handle relatively independent tasks such as serving a client request or handling a UI event. These observations were made by a study of more than 100 widely used open-source applications such as servers, browsers, and social networking applications. It was then proposed to partition an execution to autonomous units, each corresponding to an iteration of some event handling loop. In particular, program analysis was developed in [26] to recognize the unit-inducing loops, leveraging the following three observations: (1) such loops tend to be at the top level; (2) their loop bodies must make some I/O syscalls; and (3) their loop bodies dominate the execution time. Binary instrumentation is hence used to instrument the loop entry and exit points such that special syscalls are generated to indicate unit boundaries. *An output syscall is considered only dependent on the preceding input syscalls in the same unit,* whereas in other logging techniques [42, 25, 27], it is dependent on all the preceding input syscalls in the whole execution, leading to dependence explosion.

In some cases, a unit by itself may not fully cover the sub-execution that handles an independent input. Instead, a few inter-dependent units together constitute a semantically independent sub-execution. In practice, there are *memory dependencies* across unit boundaries. However, only some of them – called *workflow dependencies* – are helpful in connecting units that belong to the same sub-execution. Examples include the dependencies caused by the `enqueue` and `dequeue` operations of a task queue. In [26], inter-unit dependencies are identified via program analysis. A small number of memory operations that induce inter-unit dependencies are instrumented to emit special syscalls that help constructing the dependencies during off-line processing.

Fig. 2.2 shows the causal graph constructed by BEEP. The ovals on the left represent the units of `sendmail`, which checks the IP address through the Domain Name System (DNS) (a.a.a.a), and then interacts with the authentication server (b.b.b.b) and mail server (c.c.c.c) to fetch all emails. An email is further processed by a separate thread, whose unit is the one on the right of the dashed circle. The email is further filtered by `procmail` before it is opened by `pine`. Inside `pine`, the user clicks the phishing link, which triggers `Firefox`. `Firefox` uses multiple threads to process a request. The units in the dashed area correspond to units of the main thread and the tab thread, which uses an IPC channel `i.i.i.i` to communicate with a worker thread that downloads the backdoor file from `d.d.d.d`. The malware is later executed through `bash` and sends a file `f` to `e.e.e.e`.

Limitations of the State-of-the-Art. Although the causal graphs generated by BEEP (e.g. Fig. 2.2) are usually precise and concise, there are a few critical limitations that hinder the application of BEEP in practice.

(1) Substantial space overhead. BEEP generates a few GB log per-day for a system with a normal workload. This is because it logs all the provenance related syscalls including those generated by instrumentation. In [40], an offline garbage collection (GC) technique LogGC was proposed to prune redundant events from BEEP logs. However, it still requires storing all the events before pruning them. During pruning, it traverses the large log file back and forth in order to identify the redundant events. Due to the high cost of processing large files, one cannot afford running the GC technique frequently.

(2) Non-trivial run-time overhead. Although BEEP's instrumentation is lightweight, like many other audit logging systems (e.g. [42]) it is unfortunately built on the Linux Audit system that has non-trivial run-time overhead by itself. According to our experiment (Section 2.5), the overhead can be as high as 43%.

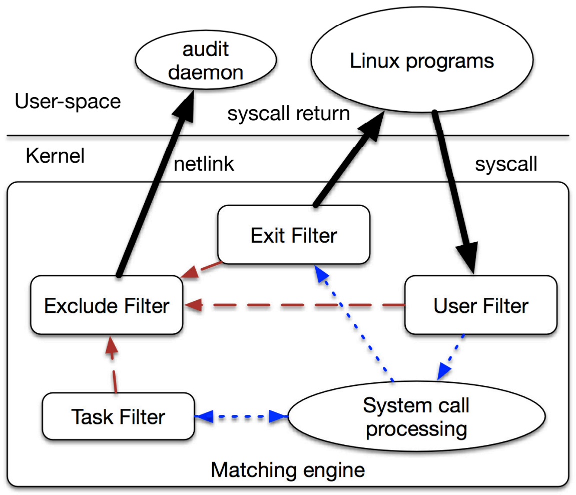


Figure 2.3.: Linux Audit system architecture

A further inspection reveals that the Linux Audit logging system is unnecessarily heavy-weight. Fig. 2.3 illustrates the architecture of the Linux Audit logging system. It consists of two main parts: a kernel module for system call processing and a few audit applications that process/store auditing events, managed by an dispatcher daemon `audisp`. The kernel module receives syscalls from Linux programs. A syscall first goes through the *user* filter that decides if the syscall will be further sent to the other kernel modules for further processing. The *user* filter also forwards the syscall to the *exclude* filter to determine if the syscall should be prevented from being sent to the audit apps. After the syscall is processed (by other modules), the return state needs to go through the *exit* filter and then the *exclude* filter to determine if the state is interesting for auditing. The control is only given back to the Linux program after all these activities.

Note that most of the filtering work is done on the kernel side, which blocks the application execution for a long time. Second, all types of syscalls have to go through filters even if they are not interesting. It uses *netlink* to send data from the kernel to the user-space daemon, which is slow. Finally, the audit applications write to the log file, which also generates a lot of events that need to go through the costly procedure.

In Hi-Fi [29], researchers have developed a more advanced logging infrastructure with a substantially lower run-time overhead (3% in a representative workload). They leverage the *Linux Security Modules* (LSM) that allow adding light-weight hooks before accesses to kernel objects such as `inodes`, and use a high performance buffer to deliver kernel object access events to a user space logging application. Despite its low overhead, Hi-Fi does not perform event processing on the fly hence it records all events. Furthermore, kernel object access events are at a level lower than syscalls. As a result, some commonly used syscalls such as file read may lead to many kernel object accesses, which induce additional overhead. Finally, LSM hooks may have difficulty handling customized syscalls introduced by BEEP as those syscalls do not lead to any kernel object accesses. As such, the capability of solving dependence explosion cannot be easily ported to Hi-Fi.

The Basic Idea of ProTracer. We improve the practicality of provenance tracing by the following two aspects.

In the first aspect, We develop a lightweight kernel module. We will leverage a kernel facility called Tracepoints [43]. A tracepoint can be placed in both user and kernel code to provide a hook to call a kernel function (*probe*). In ProTracer, we will insert tracepoints to kernel functions that process provenance related syscalls (e.g., `sys_clone`). The tracepoint driver is extremely lightweight and simply stores the events to a ring buffer, which will be processed by the user space daemon through a pool of threads. More details can be found in Section 2.3.

In the second aspect, we avoid logging as much as possible by alternating between tainting and logging. We only log when files are written to the disk or packets are sent through sockets for either IPC or real network communication. For other syscalls that only lead to intra-process information flow such as file reads and network receives, we perform unit level taint propagation. Tainting has the following benefits:

(1) Avoid logging redundant events. Consider the dashed area for `Firefox` in the middle of Fig. 2.2. At the entry point to the area on the left, ProTracer will introduce a new taint to represent the provenance of the hyper link, which is essentially the sub-graph to the left of the area. The taint is further propagated through the nodes inside the dashed area. Note that since no external accesses are performed inside the area, the taint remains the same until it gets out the area. As such, we avoids logging events in that duration without

losing any provenance information. The same applies to the dashed box for `sendmail`. Similarly, consider an FTP server. Each unit of the server corresponds to processing a client request. Assume the client request is to upload a large file, which entails many network receive syscalls. In a pure logging system such as [26], all the syscalls need to be logged. In ProTracer, logging these events is avoided by taint propagation. In fact, all these syscalls have the same taint and do not add to the taint set.

(2) Avoid logging dead events. Tainting also allows ProTracer to handle the large number of syscalls that do not have any permanent effects on the system. We call them the *dead events*. For example, syscalls related to temporary files represent a large portion of a raw audit log [40]. However, since these files are just used internally and never accessed by others, their taint propagation usually does not reach any other file writes or network sends and hence does not trigger any logging. Lets consider the FTP server example again. Assume during the processing of the file upload request, the connection is lost. The FTP server will eventually timeout and exit the execution unit without writing any data. In this case, the taint source representing the data session with the client IP is not propagated to any updates on the storage. Thus nothing needs to be logged.

Note that the aforementioned two kinds of reductions are different from the reduction in LogGC [40], which is an *offline* log garbage collection technique. LogGC is based on reachability so that all the events in the dashed area of Fig. 2.2 cannot be pruned as they are reachable from the backdoor process. Furthermore, it requires first acquiring the entire log file and then traversing the large file back and forth to identify unreachable items, incurring high cost.

(3) Allow concurrent event processing. Introducing a new taint to represent a provenance set allows out-of-order event processing. For example, by introducing a new taint when the dashed region of `Firefox` in Fig. 2.2 is entered, the processing of the `Firefox` events does not have to wait for the processing of the events in the sub-graph on the left of the shaded area. This maximizes the utilization of the thread pool. More details can be found in Section 2.4.

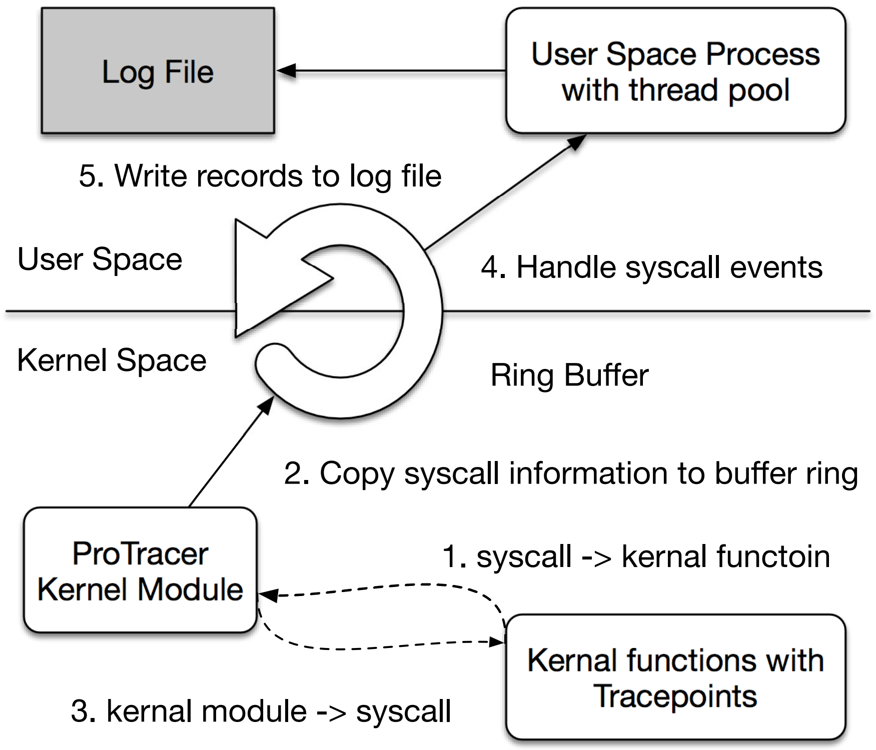

Log File
User Space Process with thread pool
5. Write records to log file
User Space
4. Handle syscall events
Kernel Space
Ring Buffer
2. Copy syscall information to buffer ring
ProTracer Kernel Module
1. syscall -> kernal functoin
Kernal functions with Tracepoints
3. kernal module -> syscall

Figure 2.4.: ProTracer architecture overview (Dashed lines: control flow; Solid lines: data flow; Numbers: the order of the events.

2.3 System Architecture

The architecture of ProTracer is shown in Fig. 2.4. The system consists of two main parts: a kernel module and a user space daemon process. The kernel module is responsible for collecting syscall events and writing them to the ring buffer. The user space process fetches and handles these events, including deciding to log the events or perform taint propagation.

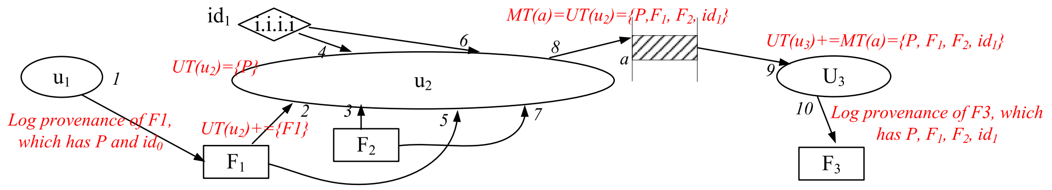
When a Linux application makes a syscall, the execution is trapped to the kernel space and the application is blocked until the kernel finishes processing the syscall. It is hence critical to ensure the kernel module is lightweight. ProTracer makes use of an existing lightweight Linux kernel trace facility, Tracepoints [43]. In particular, we identify the set of kernel functions that handle syscalls that can induce causality with system objects or other processes. They mainly fall into the following categories.

- All syscalls that operate on file descriptors (representing regular files, network sockets, device files, pipes and so on), including creation, read, write, and close.
- Special syscalls that help trace taints on certain types of objects. For example, *sys_bind* for sockets.
- IPC syscalls operating on pipes, semaphores, message queues, shared memory, and UNIX domain sockets.
- Process manipulation syscalls including process creation, termination, and privilege changes (escalation or degradation).
- Syscalls generated by program instrumentation to denote unit boundaries and inter-unit workflow.

The syscalls that are not monitored are mainly for time management (e.g. timer_create), fetching information from file system or kernel (e.g. getpid), and those not implemented (e.g. getpmsg). To our knowledge, the set is complete for provenance tracing with certain assumptions. Detailed discussion can be found in Section 2.6. Tracepoints are inserted at the entry and exit points of the kernel functions. They are lightweight hooks that can hand over the execution to our kernel module so that the syscall and its context can be copied to the ring buffer. The trace points at the entries are to collect the parameters while those at the exits are to collect the syscall results. We separate the two to allow better concurrency in event processing. Our kernel module is also responsible for managing the ring buffer to avoid any event loss.

ProTracer uses a user space daemon process to process the syscall events. To increase throughput, the daemon process uses a pool of worker threads, which is different from most existing works. All events are time-stamped so that we do not need to worry about the event order in the buffer and in the log file. A general worker thread assignment policy is that *syscalls from the same application cannot be processed by more than one worker thread.* In other words, event processing is in order for the same application. But it may be out-of-order for events from different applications. For each event, the daemon process needs to decide to log it or to perform taint propagation. More details are discussed in Section 2.4. All threads share a log buffer that stores the log records. The log records are written to disk only when the buffer is nearly full or the system is in a relatively idle state so that we can reduce the number of disk I/O operations.

To achieve good performance, ProTracer uses a ring buffer to share data between the kernel module and the user space daemon. The ring buffer is similar to the high performance buffer in Hi-Fi [29], which is also memory-mapped to the user space so that it can be accessed without any copy operations. However, we choose to use tracepoints for syscall interception instead of LSM hooks, to support customized syscalls and to trace at the syscall level instead of the lower kernel object access level.


id1
i.i.i.i
MT(a)=UT(u2)={P,F1, F2, id1}
UT(u3)+=MT(a)={P, F1, F2, id1}
u1
1
UT(u2)={P}
u2
4
6
8
a
9
U3
Log provenance of F1, which has P and id0
UT(u2)+={F1}
2
3
5
7
F1
F2
10
Log provenance of F3, which has P, F1, F2, id1
F3

Syscalls: ***1.*** *write to F1;* ***2.*** *read F1;* ***3.*** *read F2;* ***4.*** *recv from i.i.i.i;* ***5.*** *read F1;* ***6.*** *recv from i.i.i.i;* ***7.*** *read F2;* ***8.*** *write to mem addr. a;* ***9.*** *read from a;* ***10.*** *write to F3*

Figure 2.5.: An abstract diagram to illustrate the logging and tainting run-time. The numbers represent the order of the events. The logging/tainting behavior is highlighted in red on edges.$UT(u)$ and $MT(a)$ are simplified representations of the taint set of a unit u and address a, respectively. P denotes the current process and id_0 an ID denoting a network session.

2.4 Tainting and Logging in the User Space Daemon

In this section, we explain the user space daemon that alternates between tainting and logging. The basic scheme is intuitively illustrated by Fig. 2.5. When receiving a syscall event, the daemon checks if it is a syscall that makes permanent changes to the external state (e.g., a file write or a socket send). If so, it logs the current taint set of the event to disk, which denotes the provenance of the associated object (e.g., the logging action in red on

$$
\begin{array}{lll}
OT \in ObjectTaintStore & ::= & (IPC \mid File) \rightarrow Taint \\
UT \in UnitTaintStore & ::= & Process \rightarrow \mathcal{P}(Taint) \\
MT \in MemTaintStore & ::= & Address \rightarrow \mathcal{P}(Taint) \\
t \in Taint & ::= & (IPC \mid File \mid ID) \times TimeStamp \\
TSrc \in TaintSource & ::= & ID \rightarrow (Session \mid Email) \\
LB \in LogBuffer & ::= & \overline{(\text{WRITE} \times (File \mid Session \mid IPC) \times \mathcal{P}(Taint) \times TimeStamp \mid} \\
 & & \overline{\text{DEL} \times File \times \mathcal{P}(Taint) \times TimeStamp \mid \text{FORK} \times Process \times \mathcal{P}(Taint) \times TimeStamp)} \\
e \in Event & ::= & FileOpen(Process, File) \mid FileRead(Process, File) \mid FileWrite(Process, File) \mid \\
 & & FileDel(Process, File) \mid \\
 & & IPCRead(Process, IPC) \mid IPCWrite(Process, IPC) \mid \\
 & & SessionCreate(Process, Session) \mid SessionRead(Process, Session) \mid \\
 & & SessionWrite(Process, Session) \mid \\
 & & Fork(Process, Process) \mid \\
 & & MemWrite(Process, Address) \mid MemRead(Process, Address) \mid \\
 & & UnitEnter(Process) \mid UnitExit(Process) \mid \\
 & & EmailRecv(Process, Email)
\end{array}
$$

$$f \in File \quad c \in IPC \quad a \in Address \quad p \in Process \quad x \in Session \quad m \in Email \quad ts \in TimeStamp$$

Figure 2.6.: Definitions for Logging and Tainting.

edge 1). When a new unit starts (i.e., the event handling loop starts to process a new and independent request), the taint set associated with the unit is reset to only containing the process itself (e.g., $UT[u_2] = \{P\}$ to the left of u_2), meaning the provenance of the unit only contains the current process.

Upon an input event, a new taint is created to denote the current provenance set of the input object (e.g., the new taint F_1 on edge 2 denoting the current provenance set of F_1 and ID id_1 on edge 3 denoting the network session). The taint is then added to the taint set of the unit, denoting that now the unit is causally related to the corresponding input source. Input syscalls only trigger taint propagation instead of logging. Upon a memory write representing workflow, the current unit taint set is propagated to the memory (e.g., the highlighted behavior on edge 8). Later, when another unit loads from the same memory location, the memory taint set is propagated to the unit (e.g., the behavior on edge 9). Eventually, when unit u_3 writes to F_3, the provenance of F_3 is the current unit taint set. Note that F_3's provenance set contains F_1, implying a causal edge between this event and the previously logged event about F_1. In our implementation, we associate timestamps with taints and events to facilitate recovery of such causality. It is worth noting that although there are 10 syscall events, only two entries are logged, which are sufficient to disclose both the what- and how-provenance of F_1 and F_3. In particular, the how-provenance is represented by the causal graph.

Next, we discuss the details of our design using an abstraction of the system.

2.4.1 Definitions

The definitions related to our discussion are presented in Fig. 2.6. To support tainting, three data structures are introduced to store taints for objects, units, and memory, respectively. In particular, we use an *ObjectTaintStore* structure to associate a singleton taint to an object of two possible kinds: *Inter-Process Communication objects* (IPCs) that are essentially a special kind of sockets, and memory-mapped files. We use a *UnitTaintStore* structure to associate a process with a set of taints, denoting the taints of the current execution unit, which is usually an iteration of the event handling loop. *MemTaintStore* associates a set of taints with a memory address, which is to support intra-process taint propagation. ProTracer selectively instruments a very small number of critical memory reads and writes that denote the inter-unit workflow (i.e., high level data flow [26]) of the application, e.g. the reads and writes of a task queue that is used to pass user requests across execution units. A taint can be a time-stamped IPC, file, or an ID that represents a taint source, which can be either a network session or an email received. In other words, we use IDs to denote external sources. The mapping is maintained by a *TaintSource* structure. In contrast, for objects internal to the system we may use a taint consisting of the object and a timestamp ts to denote the provenance of that object at ts, which may represent a set of IDs (e.g., in event 2 in Fig. 2.5 F_1 denotes the current provenance of file F_1, which contains P and id_0).

As mentioned in Section 2.1, we cannot capture all important provenance by taint propagation alone, which does not record the history of an object or a process. As such, in addition to taint propagation, we also log important events. More specifically, we log all the permanent changes to the system, such as file writes, file deletes, outgoing network traffic, and process creation, together with their taints. *LogBuffer* is a memory buffer to store these changes. We use a memory buffer to avoid frequent disk accesses. More importantly, the memory buffer allows us to easily avoid logging events related to temporary files, which are often in a large number. More discussion can be found later.

As mentioned in Section 2.3, ProTracer intercepts all syscalls related to provenance, including those related to units. We abstract these syscalls to a few representatives as shown in Fig. 2.6. In particular, they denote file, IPC, network session, process spawn, memory reads/writes denoting inter-unit workflow, unit enter/exit, and taint source related operations. The run-time behavior corresponding to these events will be discussed next. Note that although our implementation intercepts both the entry and the exit of a kernel function that handles a syscall, our abstraction combines the two events into one abstract event for discussion simplicity.

Table 2.1.: ProTracer: Logging and Tainting Rules.

Rule #	Event	Action
1	$FileOpen(p, f)$	$OT[f] = \langle f, \text{getTime}() \rangle;$
2	$FileRead(p, f)$	$UT[p] \cup = \{OT[f]\}$
3	$FileWrite(p, f)$	$LB = LB + \langle \text{WRITE}, f, UT[p] \cup \{OT[f]\}, \text{getTime}() \rangle$
4	$FileDel(p, f)$	if (f is owned by p) $LB = LB - \langle *, f, *, * \rangle;\ LB = LB + \langle \text{DEL}, f, UT[p], \text{getTime}() \rangle;$ $OT[f] = nil;$
5	$IPCRead(p, c)$	$OT[c] = \langle c, \text{getTime}() \rangle;\ UT[p] = UT[p] \cup \{OT[c]\}$
6	$IPCWrite(p, c)$	$LB = LB + \langle \text{WRITE}, c, UT[p], \text{getTime}() \rangle$
7	$SessionCreate(p, x)$	$t = \text{newSource}();\ \ OT[x] = \langle t, \text{gettime}() \rangle$
8	$SessionRead(p, x)$	$UT[p] = UT[p] \cup \{OT[x]\}$
9	$SessionWrite(p, x)$	$LB = LB + \langle \text{WRITE}, x, UT[p], \text{getTime}() \rangle$
10	$Fork(p_1, p_2)$	$LB = LB + \langle \text{FORK}, p_2, UT[p_1], \text{getTime}() \rangle$
11	$MemWrite(p, a)$	$MT[a] = UT[p];$
12	$MemRead(p, a)$	$UT[p] \cup = MT[a]$
13	$UnitEnter(p)$	$UT[p] = \{\langle p, - \rangle\}$
14	$EmailRecv(p, m)$	$t = \text{newSource}();\ \ UT[p] = UT[p] + \{\langle t, \text{getTime}() \rangle\}$

2.4.2 Run-time Operation Rules

Table 2.1 describes the taint propagation and logging operations conducted by the threads in the user space daemon. A worker thread (in the daemon) receives an event from the ring buffer and processes it based on the rules in the table.

File Operations. Rules 1-4 are for file related event processing. For a file open event with process p opening a file f, ProTracer creates a new taint that consists of the file object and the current timestamp. The taint denotes the provenance set of the file at this moment, which may include multiple external sources. The principle is that ProTracer uses a singleton taint to represent a provenance set for a system object that can propagate information across processes, including file and IPC. This is a critical design decision which will be further explained. The taint is then associated with f through the *ObjectTaintStore* OT. Upon a file read, the taint set of the current execution unit of p is enhanced with the taint of f, meaning the current execution of p is also affected by the provenance of f. Upon a file write, a log entry is inserted to the log buffer denoting the write operation and the associated taint set, which is the union of the current unit taint set and the file taint (Rule 3). Intuitively, after the write, the file inherits the taints of all the preceding input syscalls in the same unit. The design choice of using a singleton taint to denote the provenance (taint) set of an object on external storage has a few critical advantages over the design of directly propagating provenance sets.

- An object may be transitively dependent on a large set of taint sources. It is expensive to propagate taint sets, which entails allocating space and performing set unions. Hence, ProTracer uses a singleton taint consisting of the object and a timestamp to denote the current taint set of the object and propagates the taint.
- The design allows out-of-order processing of events in the ring buffer. As mentioned earlier, the kernel inserts events with timestamps to the ring buffer and the user space daemon retrieves and handles these events from the same buffer. Events from different processes may be dispatched to different threads that execute concurrently. As such, event processing across processes may be out-of-order. For example, assume two

applications A and B. A writes to a file f and closes it before B reads it. The file read event (in B) may be processed before the file write (in A) is processed. If we directly propagate the taint set of f from A to B, we have to wait for the file write to be processed before processing the file read, substantially limiting concurrency. With the current design, the file read will use a fresh taint, without waiting for the computation of the set. The timestamps of the taint set (recorded to the log buffer at the write event) and the fresh taint (introduced at the file read) would allow ProTracer to infer the proper mapping between the set and the new taint during the offline causal analysis.

- The design allows us to record not only the *what* provenance, but also the *how* provenance. Traditional techniques based on standard tainting [24] can only record the set of taint sources associated with an object, missing the history about how the object was created and updated. With the current design, each time an object is updated (i.e. written to the permanent storage), a log entry representing the set of taints of the object is recorded.

Upon deleting a file (Rule 4), ProTracer not only resets the taint of f, but also removes all the log entries in the buffer related to f if the process p is the exclusive owner of f, meaning f is a temporary file that does not escape the lifespan of its owner. We say the p is the owner of f if p creates f and f is never read by another process. If p is not the owner, the log entries related to f cannot be removed as the history of f may still be of interest. For example, an APT attack may remove a malicious library generated in an earlier phase of the attack (by another process) to cover its trail. The history of the malicious library is still valuable although it is deleted. In addition, the deletion event itself needs to be logged as it is part of the malicious behavior. The log buffer is flushed to the disk when it is close to full. It often takes a long time for the log buffer to reach its capacity so that most temporary file deletes happen before the buffer is flushed, allowing the pruning of dead log events (Section 2.2) such as temporary file reads and writes.

IPC Operations. Processes may use IPC (e.g. pipes) to communicate with each other. Upon an IPC write (Rule 6), a log entry is added to denote the write and the provenance of the write, which is essentially the current unit taint set. Following the design policy of using singleton taints to allow out-of-order processing, upon an IPC read a new taint consisting of the IPC object and the current timestamp is created and added to the unit taint set of the receiver process (Rule 5).

Network Operations and Process Spawn. Network operations are handled similar to file operations. We consider a network session as a unique taint source. As such, each time a session is created, a new taint ID is created and associated with the session. When a process p receives packets from a session, the taint of the session is added to the unit taint set of p (Rule 8). When p sends a packet through a session, the provenance of the network send is denoted by the unit taint set of p. A log entry containing the taint set is recorded. Such entries allow ProTracer to construct causality across hosts. When a process p_1 spawns another process p_2 (Rule 10), the provenance of the child is the unit taint set of its parent. A log entry is added to record the fork and the corresponding taint set.

Execution Unit Related Operations. These events are generated by selective program instrumentation [26]. Application executables are instrumented in a very small number of places to emit special syscalls to indicate the beginning and the end of an execution unit, and memory operations that denote the high level workflow between units. ProTracer needs to propagate taints through the memory object involved. Upon a write to a memory object, the unit taint set is propagated to the object (Rule 11). Later, when the same memory object is read in another unit, its taint set is inserted to the taint set of the new unit (Rule 12). As mentioned in Section 2.2, execution units are considered autonomous and their correlations are only through the workflow related memory objects explicitly monitored by ProTracer. Therefore, when the execution leaves a unit and enters a new unit, the unit taint set is reset to only containing the process itself (Rule 13).

Taint Source Operations. Upon events such as receiving an email, a new ID representing the source is created and inserted to the unit taint set (Rule 14). Note that these events may be at a higher level than syscalls. In our implementation, the corresponding protocol libraries are instrumented to generate these high level events.

(a) Browser code

```
   UI Thread
1  …
2  /*UI event dispatching loop*/
3  while (true) {
4     UnitEnter ();
5     ui_event = poll (…)
6     if (ui_event.type==…)
7        …
8     if (ui_event.type==HYPER_LNK) {
9         MemWrite(q.tail( ));
10        q.enqueue(ui_event.url);
11    }
12    …
13    UnitExit ();
14 }
```

```
   Worker Thread
20 …
21 /*Event processing loop*/
22 while (!q.empty( )) {
23   UnitEnter( );
24   MemRead(q.head( ));
25    u = q.dequeue( );
26    if (u==URL_REQ) {
27       f= FileCreate("tmp");
28       x=SessionCreate (u, …);
29       buf=SessionRead(x);
30       FileWrite (f, buf…);
31
32       c=IPCCreate(reader, ...);
33       IPCWrite(c, f);
34       FileDel(f);
35       ...
36 }
```

(b) PDF reader code

```
50 …
51 /*Event processing loop*/
52 while (true) {
53   UnitEnter( );
54   e = poll(…);
55   if (e==OPEN_BY_IPC) {
56      c=IPCCreate(browser,…);
57      buf=IPCRead(c);
58      MemWrite(buf);
59      pdf_render(buf);
60   }
61   if (e==SAVE_AS) {
62       o= FileCreate("a.pdf");
63       MemRead(buf);
64       FileWrite (o, buf…);
65       ...
66 }
67
```

(c) A sample system execution

Program	TimeStamp	Event	Rule	OT[]/MT[]	UT[]	LB
Browser	1	UnitEnter(b)	13		UT[b]={OT[b]}={<b,->}	
	2	MemWrite(b,q[0])	11	MT[q[0]]=UT[b]={<b,->}		
	3	UnitEnter(b)	13		UT[b]={OT[b]}={<b,->}	
	4	MemRead(b,q[0])	12		UT[b]=UT[b] U MT[q[0]]={<b,->}	
	5	SessionCreate(b,x)	7	OT[x]=<t_1,5>		
	6	SessionRead(b,x)	8		UT[b]=UT[b] U {OT[x]}={<b,->, <t_1,5>}	
	7	FileWrite(b,f)	3			LB=<W,f,{<b,->,<t_1,5>},7>
	8	IPCWrite(b,c)	6			LB=<W,f,{<b,->,<t_1,5>},7>; <W,c,{<b,->,<t_1,5>},8>
	9	FileDel(b,f)	4	OT[f]=nil		LB= <W,c,{<b,->,<t1,5>},8>
Reader	10	UnitEnter(r)	13		UT[r]={OT[r]}={<r,->}	
	11	IPCRead(r,c)	7	OT[c]=<c,11>	UT[r]=UT[r] U OT[c] ={<c,11>,<r,->}	
	12	MemWrite(r,buf)	11	MT[buf]=UT[r]={<c,11>,<r,->}		
	13	UnitEnter(r)	13		UT[r]={<r,->}	
	14	MemRead(r,buf)	12		UT[r]=UT[r] U MT[buf]={<c,11>,<r,->}	
	15	FileWrite(r,o)	3			LB=<W,c,{<b,->,<t_2,5>},8>; <W,o,{<c,11>,<r,->},15>

Figure 2.7.: Example for the logging and tainting run-time. The shaded statements correspond to syscalls. The statements in red are those instrumented by ProTracer to generate special events.

Example. Consider the example in Fig. 2.7. We have two programs running in the system: a browser and a PDF reader. Parts of the code snippets of the two applications are shown. Although the code snippets simulate the workflow in a real-world browser and a real-world PDF reader, they are substantially simplified and abstracted to be consistent with our definitions in Fig. 2.6. Specifically, the browser has two threads: the UI thread that handles UI events and the worker thread that performs background operations such as downloading a file. The event handling loop dominates the execution of the UI thread. The beginning of the loop is instrumented by a function `UnitEnter()` that will produce an event denoting the start of a unit. In lines 8-11, if the UI event is the click of a hyper link, the URL is added to the work queue. Since the queue operations denote the workflow across units, the enqueue operation is instrumented to generate a memory write event (line 9). The worker thread execution is dominated by the loop in lines 22-36, which acquires a request from the work queue and processes it. Lines 23-24 denote the unit instrumentation and the memory read instrumentation. If the request is to access a URL, a temporary file "`tmp`" is created to store the downloaded content. A session is created and used to download the resource (lines 28-29). The downloaded content is written to the file (line 30). An IPC object is created to communicate with the PDF reader to display the PDF file (lines 32-33). The temporary file is deleted at the end (line 34).

The PDF reader is also event driven. If it receives an IPC request to render a PDF file, it acquires the file through IPC and saves it to `buf` (lines 56-57) before rendering it. If it receives a UI request to save the PDF file, it creates a file and writes `buf` to the file (lines 62-64). ProTracer detects that `buf` carries workflow across units (i.e. the loop iterations corresponding to the IPC and the save-as-a-file operations), the read and write of `buf` are instrumented (lines 58 and 63).

Fig. 2.7 (c) shows a sample execution of the system, in which the user clicks a hyper link denoting a remote PDF file, the file is then downloaded and rendered by the reader, and finally the user further saves the file. The table shows the events generated by ProTracer and how the run-time processes these events. The second column shows the timestamps; the third column shows the events with process b and r denoting the browser and the reader, respectively. The fourth column shows the rules applied and the last three columns show the state of the various data structures.

Observe that in the first unit corresponding to the click of the hyper link, the `UnitEnter` event causes the unit taint of b to be reset to $\{\langle b, -\rangle\}$. Upon the `MemWrite` at 2, the taint of the queue is updated to contain the taint of the current unit. The execution then proceeds to the unit from the worker thread that downloads the file. At 3, the unit taint set is reset. At 4, the taint set of the queue is unioned with the unit taint set. At 5, since a network session is considered an external source, an ID t_1 is generated to denote the source. The taint of the session is inserted to the unit taint set at 6 due to the `SessionRead` event. At 7, the downloaded content is saved to the temporary file f, and thus a log entry is inserted to the log buffer LB to denote the write and the provenance. The file is further passed to the reader through an IPC c. The `IPCWrite` event leads to another log entry at 8. At 9, the deletion of f leads to the removal of the first log entry as f is a temporary file.

Timestamps 10-15 correspond to the execution of the reader, which consists of two units. The first one renders the file and the second one saves the file. At 11, a new taint is created to denote the provenance set of the IPC object c at timestamp 11, which essentially denotes the set $\{\langle b, -\rangle,\ \langle t_1, 5\rangle\}$. The taint is inserted to the unit taint set of r. The unit taint set is propagated to `buf` at 13. In the second unit (timestamps 13-15), the taint set of `buf` is retrieved and inserted to the unit taint set. When the file is written, another log entry is added to denote the write.

There are three important things that we need to point out. (1) Although there are 15 events, ProTracer only needs to log two of them, which are the two in LB at the end. In other words, on-the-fly taint propagation avoids storing a lot of events. (2) ProTracer introduced a new taint $\langle c, 11\rangle$ to denote the provenance set of c at 11 such that the processing of the reader events and the processing of the browser events can be performed concurrently by different threads. And from the timestamp 11 and the log, we know that $\langle c, 11\rangle$ must represent the taint set in the first log entry. (3) The log entries reflect the history of the file, whereas existing techniques only track the sources of the file.

2.4.3 Handling Global File Accesses

A long running execution can often be divided to three parts. The first one is the start phase, which is responsible for loading configurations, allocating resources like file descriptors for application log files. The second one is the event handling loop, which handles a large number of external requests. The third one is the closing phase, where all resources are deallocated before the process terminates. In the previous sections, we mainly focus on the event handling loops, which dominate and generate units. However, handling the other two phases, especially file operations in those phases, is equally important. Files opened in the start phase are often used throughout the whole execution. For example, the `Apache httpd` server opens its application log files (e.g. access log and error log) in the start phase. The access log will be written within any unit that handles an external request. This log file is a shared object across most of the units, which would cause unnecessary dependence between units. To address this problem, we apply a special policy to objects opened in the start phase. In particular, these objects are stated as global in the log. During execution, they are not considered as shared objects and operations on these files are not logged.

Unlike log files, which are opened in the start phase and not closed until the end phase, some objects used in the start phase have a very short life time. They are usually opened, read and then closed. Typical examples include configuration files and libraries used by an application. Our policy is to log these events, because the data read from these files can be possibly utilized for a malicious purpose. An example is that a malicious library downloaded from a remote site is loaded by a normal application in the start phase.

Discussion: Completeness of ProTracer. As introduced in Section 2.1, we aim to capture all the external and internal entities that affect a system object and their casual relations. With the assumption that all the provenance related syscalls are intercepted by ProTracer, we want to show that the alternation between tainting and logging and the pruning of events (e.g., through file deletions) do not affect completeness. According to the rules in Table 2.1, within a unit, all input events are captured and propagated to the taint set of the unit. With the assumption that all the inter-unit workflows through memory accesses are captured[1], the taint set is properly propagated across units. Upon an outgoing syscall, the set is logged. During offline analysis, causal edges are introduced by connecting a log entry containing a taint (of an input file), such as taint F_1 on edge 2 in Fig. 2.5, and a preceding log entry containing the provenance set corresponding to the taint (e.g., the log entry generated by the action on edge 1 in Fig. 2.5). This ensures not only that the set of external sources is complete, but also the history of the object is captured (by the causal graph). The way that ProTracer prunes log entries related to temporary files is safe because only the log entries related to a file owned by the process are removed. When the file is owned, its information cannot reach other processes. In the case that a temporary file is copied to another permanent

[1]We will discuss situations where our assumptions may not hold in Section 2.6.

file, its provenance is completely inherited by the permanent file so that the temporary file log entries are no longer useful. ProTracer also precludes syscalls caused by application logging as they introduce bogus causality across units. Note that application log usually records a subset of what ProTracer is already recording and is hence redundant.

2.5 Evaluation

In this section, we will show the evaluation results of ProTracer. The experiments were conducted on five identical machines with four cores and 4GB RAM. Most of the binaries in our experiment machines have been instrumented by BEEP [26].

2.5.1 Effectiveness

Daily Usage: In the first experiment, we emulated the daily (24 hours including break time) usage of five computer users, and collected logs. To compare the space consumption, we run both ProTracer and BEEP [26] at the same time during the experiment. After the logs were generated, we also applied LogGC [40] to garbage collect the BEEP logs to acquire the reduced logs. To create workload diversity, we emulated users exhibiting different usage patterns: User 1 uses the system to run a web server for a group project. An FTP server is also running in the same system; User 2 is new to Linux. He just tries out various applications in the system and accesses his personal emails and the Internet; User 3 is preparing for an exam. She is mainly reading documents and watching video lectures; User 4 uses *vim* a lot to finish his course project report besides accessing the Internet; User 5 mainly uses the system for watching movies and communicating with friends. In addition, we performed an extended 3-month emulation of user 1, whose machine is used as a server; and user 4, who actively uses client programs.

We compare the number of log entries and the log file sizes for BEEP, LogGC, and ProTracer. The results are presented in Table. 2.2. Columns 8 and 9 show the ratios between the ProTracer logs and the BEEP logs, whereas the last two columns show the same ratios between the LogGC logs and the BEEP logs. Observe that ProTracer can significantly reduce the number of events that need to be logged. On average, ProTracer only needs to log 1.85%(Daily)/1.45%(3 months) of the entries in BEEP. Since ProTracer records taint IDs, our log file format is slightly more efficient than BEEP and LogGC. On average, ProTracer's disk space consumption is only 1.28%(Daily)/1.02%(3 months) of BEEP's. The results vary for different users because of the different workloads and use patterns. Even in the worst case (user 1 that hosted servers), ProTracer generated less than 4% of the log entries, and consumed less than 2% of the disk space.

Compared to LogGC, ProTracer has better space efficiency in most cases. This is reasonable because LogGC garbage-collects events based on their reachability from live system objects. In other words, events that do not contribute to any live system objects are removed. ProTracer not only avoids logging such dead events (Section 2.4), but also precludes redundant events that affect live system objects (e.g. repetitive socket reads from the same session and execution units that do not access any taint sources but rather serve as part of the information flow path). In some cases, LogGC is more space efficient (e.g. user 3). This is mainly due to temporary files. LogGC is an offline log reduction method, which has sufficient information to precisely decide if a file is temporary. ProTracer uses the log buffer to delay writing log entries to the disk, hoping that the temporary file related entries will be removed by the time the buffer is flushed. However, some temporary files related log entries will be flushed to disk if the files are not deleted by the time the buffer is flushed. Besides, ProTracer also logs events that belong to the start and end phases of an execution, whereas BEEP/LogGC ignores those events. For user 3, `MPlayer` and `Xpdf` were frequently used and they produced a large number of temporary files. LogGC was able to remove all the records that belong to these two programs, whereas ProTracer keeps some of them. Also observe that on average, ProTracer only generates 13MB log per day, which is very affordable.

Application Perspective: We also compare the space consumption of the different systems on various applications. The logs specific to applications are extracted from the whole system logs. The results are presented in the lower half of Table 2.2. Observe that the ProTracer logs are significantly smaller compared to BEEP Logs. The number of records

Table 2.2.: Comparison of Effectiveness of Various Provenance Systems

	Logs	BEEP		ProTracer		LogGC		ProTracer/BEEP		LogGC/BEEP	
		# Items	Size(KB)	# Items	Size(KB)	# Items	Size(KB)	# Item	Size	# Item	Size
3 Months	Server	274,242,874	157,839,158.66	3,371,453	880,584.40	45,429,888	21,311,715.08	1.23%	0.56%	16.57%	13.50%
	Client	500,928,294	168,269,687.89	7,844,783	2,437,009.51	30,449,339	10,037,472.39	1.57%	1.45%	6.08%	5.97%
	Average	387,585,584	163,054,423.27	5,608,118	1,658,796.95	37,939,614	15,674,593.74	1.45%	1.02%	9.79%	9.61%
Daily	1	3,610,501	1,673,830.40	133,159	30,830.20	637,596	280,576.00	3.69%	1.84%	17.66%	16.76%
	2	1,730,339	581,389.08	37,205	8,873.10	293,957	98,768.73	2.15%	1.53%	16.99%	16.99%
	3	2,221,662	743,986.67	9,558	2,245.02	3,382	1,351.68	0.43%	0.33%	0.15%	0.18%
	4	3,768,783	1,265,993.45	52,009	16,078.02	183,947	60,139.52	1.38%	1.27%	4.88%	4.75%
	5	2,471,859	829,968.11	23,998	7,226.04	22,248	8,970.24	0.97%	0.87%	0.90%	1.08%
	Average	2,760,629	1019033.54	51,186	13,086	228226	89961.23	1.85%	1.28%	8.27%	8.83%
Applications	Apache	3,262,452	4,570,766.23	47,305	31,380.62	288,482	404,797.44	1.45%	0.69%	8.84%	8.86%
	Vim	1,089,732	370,508.82	11,215	3,053.91	99,452	34,017.28	1.03%	0.82%	9.13%	9.18%
	Firefox	3,672,740	4,655,331.54	302,873	288,344.52	352,587	447,406.08	7.70%	6.62%	9.60%	9.61%
	W3M	368,752	259,001.72	25,793	18,068.22	82,959	57,671.68	6.99%	6.98%	22.50%	22.27%
	ProFTPD	48,374	12,183.53	689	124.80	4,539	1,228.80	1.42%	1.02%	9.38%	10.09%
	Wget	873,205	189,782.34	7,938	897.81	4212,847	88,811.52	0.91%	0.47%	47.28%	46.80%
	Mplayer	858,236	188,811.93	240	16.00	0	0.00	0.03%	0.01%	0.00%	0.00%
	Pine	59,125	21,285.72	648	286.72	642	327.68	1.10%	1.35%	1.09%	1.54%
	Xpdf	56,083	9,534.20	64	16.00	0	0.00	0.11%	0.17%	0.00%	0.00%
	MC	7,823	9,026.81	26	8.00	0	0.00	0.33%	0.09%	0.15%	0.00%

is reduced to less than 8%, and the log size shrinks to less than 7% for all programs. The results vary across different applications due to the different behavioral patterns of the applications. For browsers like `Firefox`, accessing a single web page can introduce many taints as it may access the web server, advertisement server, image storage server and so on. Since each resource request will cause a log entry, the log buffer is filled up much faster and more frequently, compared to other applications. As a result, ProTracer records more temporary files related events. Programs like `Xpdf` interact with files and the screen. There is no outgoing information via sockets or other files. ProTracer only needs to record a small number of events in the start and the end phases. For most programs, ProTracer occupies less space than LogGC. But for some of them (e.g. `Xpdf`), LogGC performs better. This is because LogGC ignores the events in the start and the end phases of a process. However, the results also show that the overhead is minor for these applications.

2.5.2 Logging Overhead and Scalability

We also perform experiments to study the run-time overhead and scalability of ProTracer. Fig. 2.8 shows the accumulated log size over time for user 1. The solid line shows the growth of the BEEP log size over time, and the dashed line shows ProTracer's. In general, the growth is similar although the scales of the sizes are different. The sharpest growth occurs in the 15th-20th hour, indicating the user was intensively using the system. Even in this period, the growth of the ProTracer log is about 13MB, suggesting very good scalability. There are some shape differences between the two lines near the 20th hour. This is mainly because ProTracer has better log reduction for the applications used during that time period, compared to other applications.

Fig. 2.9 and Fig. 2.10 show the run-time overhead comparison between ProTracer and the default Linux Audit system, with the same set of syscalls monitored. Note BEEP is built on the Linux Audit system and hence more expensive. We perform two sets of experiments. The first one is for server programs. We use the `Apache Benchmark` [44] to test two web servers `Apache` and `MiniHttp`, and `ftpbench` to test `ProFTPD`. We also test different

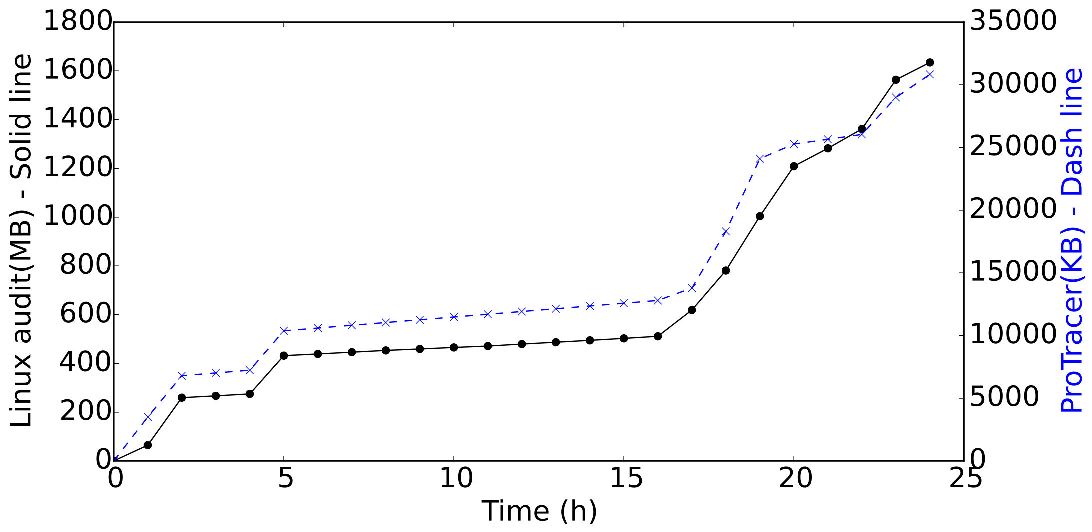


Figure 2.8.: Accumulated log size from the one-day execution of BEEP and ProTracer. Note the size of BEEP log is measured by *megabytes*, whereas the ProTracer log is measured by *kilobytes*.

concurrency configurations, with the number of requests sent at the same time being 1, 2, 4, and 8. The results are shown in Fig. 2.9. The benchmarks tend to give the system a lot of pressure, which would cause higher overhead than regular usage. The baseline we use is the native Linux system without running the Linux Audit system. Observe that the overhead of ProTracer is less than 7%, whereas the Linux Audit system has a much more significant overhead (more than 5 times larger).

We also perform experiments for client programs. We use standard benchmarks if they are available such as `SunSpider` for `Firefox`. Otherwise, we use the batch mode for programs like `vim` or `W3M`. We perform the experiments with ProTracer and with the Linux Audit system. The baseline we use here is the native Linux system without any logging system running. The results are shown in Fig. 2.10. Observe that ProTracer has less than 3.5% run-time overhead for all these programs, whereas the overhead of the Linux Audit system is 7-8 times larger.

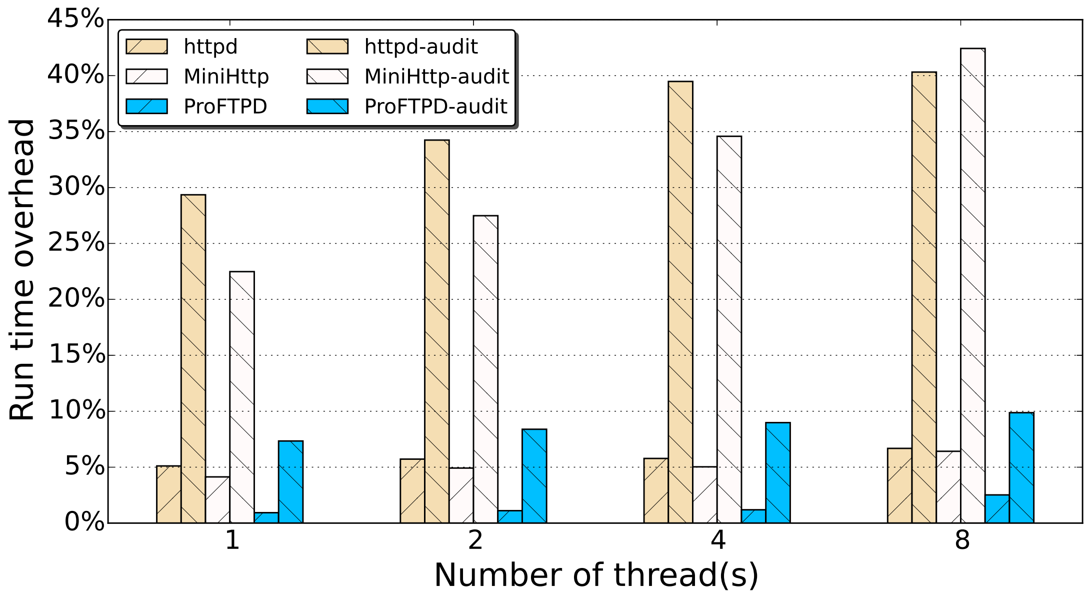


Figure 2.9.: Run-time overhead with different concurrent thread(s) for server programs.

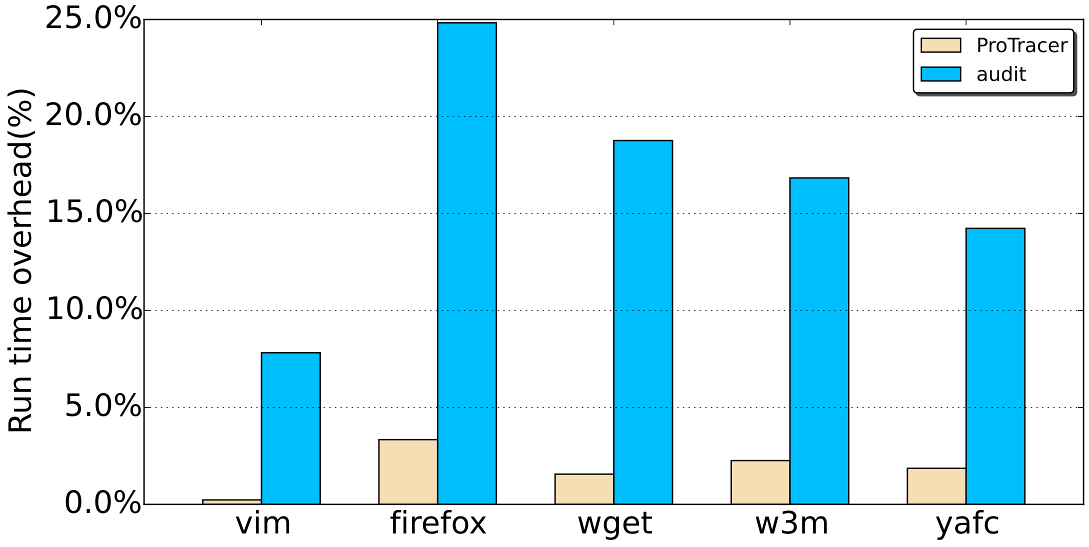


Figure 2.10.: Run-time overhead for client programs.

2.5.3 Attack Investigation Cases

In this section, we use a number of attack cases to show that the causal graphs generated by ProTracer during attack analysis are smaller than those by BEEP, but equally informative, and the time taken to generate the graphs is much less. We reproduce a few realistic attack scenarios for our experiment. With each scenario, we perform two *what*-provenance queries to understand the sources and the ramifications of the attacks, and also the *how*-provenance query to understand the attack path (Section 2.1). We compare the query results by BEEP and ProTracer, and also cross-check with our prior knowledge. To emulate real-world attack scenarios, each experiment lasted for a few hours with the attack performed in the middle.

The first case is a *backdoor attack*. The attacker detected that the running FTP server was ProFTPD-1.3.3c, which had a backdoor command [45]. He compromised the server, and was able to get a *bash* shell. He then downloaded a backdoor program using *wget*, and started this backdoor to get permanent access. A few days later, the administrator got a warning that the FTP server had a backdoor, and decided to check if the backdoor had been exploited. If so, what damages have been inflicted.

The second scenario is *information theft* [26]. An employee had a under-the-table deal with one competitor of his own company: he copied some information from the company, and leaked it to the competitor by pasting it to a public page via vim. When the company found that the information was leaked, they should be able to pair the file that contained the information with the web page that leaked the information, among thousands of files. ProTracer shall also allow them to prove that the attacker leaked it among all the other employees that have the access to the file.

The third scenario is *illegal storage* [27]. One of the server administrators wanted to store some illegal files on a server. However, he did not want the files to be in his own directory. Instead he created a directory under another user's home directory, and downloaded the illegal files to the directory. He replaced the `ls` program to hide the existence of this directory. When the files were eventually found, the victim user was considered a suspect because of the presence of those files in his/her directory. The investigator should be able to identify that the administrator was the one that committed the crime. Note here we assume that the administrator cannot tamper with the log file generated by ProTracer.

The forth scenario is *cheating student* [46]. An instructor's password was stolen by a student. The student downloaded a file containing midterm scores from `Apache`, and uploaded a modified version. The instructor noticed that the average score became higher, and started to suspect someone had modified the file. Luckily, students used static IP addresses on campus and off-campus IPs were forbidden to connect to the server. So finding the IP from which the current file was uploaded would help identify the student. Moreover, the administrator should be able to find other suspicious activities of the student. In our case, the student also downloaded a few files containing answers to future quizzes.

The fifth is *phishing email* [47] that was discussed in Section 2.2.

Parts of the results are shown in Table 2.3. The second column shows the experiment duration. The next three columns show the size of the logs by different systems. Then we show the time it takes to perform the queries. The last two columns show if BEEP and ProTracer produce matched results for the two *what*-provenance queries (i.e., the *backward* query for attack sources and the *forward* query for attack ramifications). For the first scenario, the backward query is not applicable. Observe that ProTracer produces much smaller logs and the query processing time is much shorter. The query results show no qualitative differences and precisely disclose the provenance.

Table 2.3.: Effectiveness of ProTracer on Real-world Attack Scenarios

Scenario	Duration	Log file size(KB)			Run time(s)		Investigation	
		BEEP	LogGC	ProTracer	BEEP	ProTracer	Backward	Forward
Backdoor attack	3h54min	832,753	174,693	79,834	74	11	-	Match
Information theft	4h22min	587,494	94,759	13,938	39	5	Match	Match
Illegal storage	2h58min	369,585	63,375	10,864	32	5	Match	Match
Cheating student	1h17min	179,748	29,485	9,385	17	3	Match	Match
Phishing email	4h36min	975,753	183,795	82,343	64	8	Match	Match

Backward means backward *what*-provenance query; and *forward* means forward query; *match* means ProTracer is able to precisely and concisely uncover the attack path.

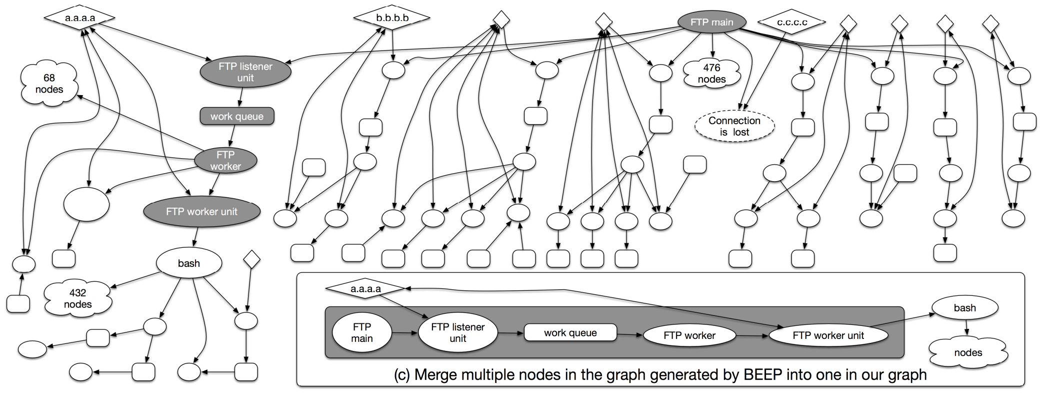
To further compare the quality of the query results. We compare the causal graphs generated by BEEP and ProTracer in answering the forward queries. The results are shown in Table 2.4, which shows the number of taint sources, the number of processes (i.e. the internal nodes along the attack path), the number of files affected by the attacks, and the number of nodes in the graphs. Note that LogGC would produce the same graphs as BEEP.

Table 2.4.: Causal Graph Comparison (BEEP/ProTracer).

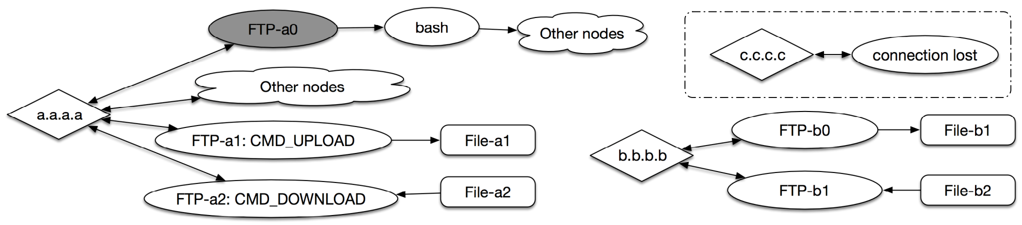
Scenario	#source	#process	#file	#nodes
Backdoor	33/33	23/23	37/66	580/128
Infor theft	1/1	4/4	21/36	148/82
Illegal storage	24/24	6/6	56/72	388/208
Student hacker	2/2	2/2	67/85	432/226
Phishing email	5/5	8/8	12/12	864/305

Observe that the two systems produce the same set of taint sources and processes. The differences in the files are due to the fact that BEEP does not log file accesses in the start and the end phases (e.g. loaded libraries). In this sense, we argue that the ProTracer-induced graphs are more complete. Finally, the ProTracer-induced graphs are much smaller than graphs generated by BEEP, reducing the human inspection efforts.

To acquire an intuitive understanding of the differences between ProTracer graphs and BEEP graphs, we present parts of the graphs by BEEP and ProTracer for the forward query in the backdoor attack case. The query aims to find all the reachable items from the external IPs connected to the FTP server. There are three connections. The one from `a.a.a.a` downloaded and uploaded a file, and exploited the backdoor. The one from `b.b.b.b` simply downloaded and uploaded a file. The one from `c.c.c.c` lost its connection. Observe that the ProTracer graph is a lot more concise and clear. This is because using tainting, ProTracer avoids logging many events and generating nodes for those events. For example, the box on the bottom of the BEEP graph shows a zoom-in view of part of the graph. It is reduced to a single node (`FTP-a0`) in our graph. And our graph is still equally informative. Moreover, the events related to `c.c.c.c` are precluded from our log in the first place as the taints did not propagate to any permanent changes.



(a) BEEP



(b) ProTracer

Figure 2.11.: Part of the graphs generated by BEEP and ProTracer for the backdoor case.

2.6 Discussion

In this section, we will discuss the limitations of ProTracer.

(1) While the execution of many programs can be divided autonomous units which are only connected through workflow memory dependencies, this may not hold for all programs. For programs that do not have unit structure (i.e., does not have event handling loops), ProTracer treats the entire execution as a unit, which may cause dependence explosion.

(2) Similar to BEEP [26], ProTracer relies on training runs to identify unit loops and workflow dependencies. However, the training may not be complete. If unit loops cannot be properly identified, ProTracer treats the entire execution as a unit. Once a unit loop is identified, the corresponding workflow dependencies can be identified as they rarely change [26]. In theory, however, ProTracer may miss such dependencies hence the corresponding memory accesses are not instrumented, leading to broken causal paths.

(3) Just like most audit logging systems, ProTracer requires that the kernel and the user space daemon are not compromised. This limitation can be mitigated by porting ProTracer to a hypervisor. Furthermore, if a system is clean to begin with and an attacker successfully subverts the system at a later time, the initial subversion will be accurately captured by ProTracer. But the log entries after the system's subversion cannot be trusted.

(4) Similar to most logging systems, ProTracer excels at capturing provenance through benign and commonly used applications, such as browsers and editors, as many attacks leverage these applications. In contrast, malware usually makes use of various methods to protect themselves such as obfuscation and self-modification, which may create trouble for ProTracer's analysis. As a result, ProTracer usually treats malware execution as a single unit. We argue that this is reasonable because all malware actions are – by definition – of interest (instead of noise) to attack investigation.

2.7 Related Work

System logging: Lots of works [24, 27, 48, 49, 50, 31, 51, 29, 30, 31, 52, 42, 10] have been done in tracking provenance using system-level audit logs. However, many of them suffer from *dependence explosion* and have high overhead. BEEP [26] and LogGC [40] are the most closely related work. ProTracer uses similar unit partitioning to avoid dependence explosion. Compared to BEEP and LogGC, ProTracer has much lower space and run-time overhead, due to the new infrastructure and the integration of tainting and logging. The graphs by ProTracer are also much more concise. Some of these techniques [29, 30, 31] provide high performance. However, they do not perform any on-the-fly reduction but rather simply store the whole traces. They may also be susceptible to dependence explosion.

Dynamic information flow tracking and tainting: Tainting and dynamic information flow tracking [53, 54, 33, 55, 56, 57, 58, 59, 60, 61, 62] have been studied from different aspects (e.g., file system, kernel object level, network flow) on different platforms (e.g., Linux, Android) including some new operating system prototypes like Asbestos [53] or HiStar [54] to precisely trace provenance. They can trace provenance with high precision. But their run-time overhead tends to be on the high end, due to the heavy-weight instrumentation. ProTracer borrows the basic concept of tainting, optimizes it at the unit level to avoid the heavy-weight instrumentation used in existing approaches. Moreover, tainting alone cannot answer how-queries.

Log storage and presentation: Provenance data can be represented as graphs, researchers have done a lot of work [63, 64, 65] on reducing the size of these graphs by borrowing ideas from Web graph compression and dictionary based encoding. In [66], researchers leverage Mandatory Access Control (MAC) policies to reduce the storage cost of provenance based on the Hi-Fi [29] system. We envision such policies can also be adopted in ProTracer to achieve more sophisticated reduction. G^2 [67] stores logs in databases, and provides execution graphs that can be analyzed using LINQ queries or user-defined programs. However, it depends on printed messages by the applications. Some techniques [40, 52] try to reduce events offline. Since ProTracer performs online reduction, the whole trace is not visible to ProTracer. Some reduction that is easy for offline analysis cannot be applied online. In other words, these offline reduction techniques are complementary to ProTracer.

Log integrity: In [68], researchers proposed a real-time server/client audit model. The client sends integrity-assured log to the server side for post-mortem detection of infections. ProTracer can provide pre-analyzed logs with small size, which help [68] gain more accurate results and better performance with lower network traffic. In [69], researchers proposed a primitive that provides the integrity of execution trace. It works on instruction-level execution traces. The same idea can be applied in ProTracer to guarantee the integrity of the log and provide better attack resilience. Recently in [70], researchers propose a novel generic framework for the development of provenance-aware systems based on LSM to secure such systems. Other researchers also try to enhance the storage system to provide the integrity of the provenance data. For example, [71] suggests using an isolated versioning system – working at the disk level – to store provenance; [72] develops a storage system that allows repetitive reads but only a single write to guarantee data integrity.

3 MPI: MULTIPLE PERSPECTIVE ATTACK INVESTIGATION WITH SEMANTIC AWARE EXECUTION PARTITIONING

Traditional auditing techniques generate large and inaccurate causal graphs. To overcome such limitations, researchers proposed to leverage execution partitioning to improve analysis granularity and hence precision. However, these techniques rely on a low level programming paradigm (i.e., event handling loops) to partition execution, which often results in low level graphs with a lot of redundancy. This not only leads to space inefficiency and noises in causal graphs, but also makes it difficult to understand attack provenance. Moreover, these techniques require training to detect low level memory dependencies across partitions. Achieving correctness and completeness in the training is highly challenging. In this paper, we propose a semantics aware program annotation and instrumentation technique to partition execution based on the application specific high level task structures. It avoids training, generates execution partitions with rich semantic information and provides multiple perspectives of an attack. We develop a prototype and integrate it with three different provenance systems: the Linux Audit system, ProTracer and the LPM-HiFi system. The evaluation results show that our technique generates cleaner attack graphs with rich high-level semantics and has much lower space and time overheads, when compared with the event loop based partitioning techniques BEEP and ProTracer.

3.1 Introduction

Provenance tracking is critical for attack investigation, especially for *Advanced Persistent Threats* (APTs) that are backed by organizations such as alien governments and terrorists. APT attacks often span a long duration of time with a low profile, and hence are difficult to detect and investigate. A provenance tracking system records the causality of system objects (e.g. files) and subjects (e.g. processes). Once an attack symptom is detected, the analyst can utilize the provenance data to understand the attack including its root cause and ramifications. Such inspection is critical for timely response to attacks and the protection of target systems. Most existing techniques [25, 11, 30, 58, 28] entail hooking and recording important system level events (e.g. file operations), and then correlating these events during an offline investigation process. The correlations have multiple types: between two processes such as a process creating a child process through *sys_clone()*; between a process and a system object, e.g., a process reads a file through *sys_read()*. However, these techniques suffer from the *dependence explosion* problem, especially for long running processes. The reason is that a long running process may have dependencies with many objects and other processes during its lifetime although only a small subset is attack related. For instance, a Firefox process may visit numerous pages over its lifetime while only one page is related to a drive-by-download attack.

Researchers proposed to partition execution to units so that only the events within a unit are considered causally related [26, 11]. For instance, the execution of a long running server is partitioned to individual units, each handling a request. Although existing execution partitioning based systems such as BEEP [26] and ProTracer [11] have demonstrated great potential, they partitioned execution based on event handling loops. That is, each iteration of an event handling loop is considered a unit. Despite its generality, such a partitioning scheme has inherent limitations. (1) Event loop iterations are too low level and cannot denote high level task structure. For instance, in UI programs, an event loop iteration may be to handle some user interaction. (2) There are often inter-dependencies across units. Therefore, BEEP and ProTracer rely on a training phase to detect such dependencies in the form of low level memory reads and writes. Achieving completeness in training is highly challenging. Note that the problem could not be addressed even when source code is provided because there are typically a lot of program dependencies across event loop iterations and only a subset of them are important. (3) A high level task is often composed of many units (e.g., those denoting event loop iterations in multiple worker threads that serve the same high level task). Ideally, we would like to partition execution based on the high level task structure.

Note that high level task structure is application specific. Therefore, developers' input on what denotes a task/unit is necessary. We observe that a high level task/unit has its corresponding data structure in the software. Our proposal is hence to allow the developer/user to inform our system what task/unit structure they desire by annotating a small number of data structures (e.g., the tab data structure in Firefox). Our system MPI[1] takes the annotations and automatically instruments (a large number of) program locations that denote unit boundaries through static program analysis. The analysis handles complex threading models in which the executions of multiple tasks/units interleave. The instrumentation emits special syscalls upon unit context switches so that the application specific task/unit semantics is exposed to the underlying provenance tracking systems. MPI allows annotating multiple task/unit structures simultaneously so that the forensic analyst can inspect an execution from multiple perspectives (e.g., tab and domain perspectives for Firefox). This is highly desirable for attack investigation as we will show later in the paper. Asking for developers/users input in audit logging is a strategy adopted in practice. For example, the audit system on Windows, *Event Tracing for Windows* (ETW) requires the developers to explicitly plant auditing API calls in their source code if they would like to perform any customized logging. Nonetheless, reducing manual efforts is critical to the real world deployment of the technique. MPI is highly automated as the user only needs to annotate a few data structures and then the invocations to logging commands are automatically inserted through program analysis. Most of the programs we use in our experiment require only 2-3 annotations for each perspective. In addition, MPI provides a data structure profiler, called the *annotation miner*, to recommend the potential data structures to annotate. As shown in §3.4.3, it makes the correct recommendations in most cases.

MPI is a general execution partitioning scheme orthogonal to the underlying OS-level provenance collection system. We integrate it with three different provenance collection systems: the widely adopted Linux audit framework, and two state-of-the-art research

[1]MPI is short for "Multiple Perspective attack Investigation"

projects, ProTracer [11][2] and the LPM [70] enabled HiFi [29] system (LPM-HiFi) which features secure audit logging.

In summary, we make the following contributions:

- We propose the novel idea of partitioning execution based on data structures to support different granularities and facilitate multi-perspective, application-semantics-aware attack investigation.
- We develop program analysis and runtime techniques to enable such partitioning. Given a small number of annotations on data structure definitions, program analysis is conducted to identify places that need to be instrumented to emit events at runtime that denote unit boundaries and unit inheritance. The number of such places may be very large, rendering manual instrumentation infeasible.
- We develop an annotation miner that can recommend the data structures to annotate with high accuracy, substantially alleviating the manual efforts.
- We develop a prototype based on LLVM. The evaluation on a set of commonly used Linux applications and three different provenance systems shows that our approach can effectively partition program execution in different granularities. We also use a number of case studies that simulate real-world attacks to demonstrate the strength of the proposed technique, in comparison with BEEP [26] and ProTracer [11].

3.2 Motivation

In this section, we use an example to illustrate the differences between the classic provenance tracking systems [30, 58, 29, 70], the existing event loop based execution partitioning approaches [26, 11], and the proposed approach. This example simulates an important kind of real-world attacks, *watering hole attack* [73],

[2]ProTracer is based on BEEP, we replace the BEEP with MPI.

3.2.1 Motivating Example

Watering hole is a popular attack strategy targeting large enterprises such as Apple [74] and Google [75]. The adversaries do not directly attack the enterprise networks or websites, which are well protected. Instead, they aim to compromise the websites that are frequently visited by the employees of the target enterprise, which are usually much less protected. Recently, there have been a number of real incidents of watering hole attacks, e.g., by compromising Github [76] and CSDN [77]. There are exploit kits (e.g., BeEF [78]) to make it easy to conduct such attacks.

In our example case, a developer in an enterprise opens *Firefox*, and then uses Bing to look for a utility program for file copying. The search engine returns a number of relevant links to technical forums, blogs, wikis and online articles. Some of these links further lead to other relevant resources such as pages comparing similar programs. Some pages host software for download. In many cases, the software was uploaded by other developers. After intensive browsing and researching, the developer settles down on a forum that hosts not only the wanted software, but also many other interesting resources, including torrents for a few tutorial videos. The developer downloads the program and also a few torrents from the forum. After the download, he starts to use the program. He also uses a p2p software *Transmission* to download the videos described by the torrents.

Unfortunately, the forum website was compromised, targeting enterprises whose developers tend to use the forum for technical discussion and information sharing. The program downloaded, *fcopy*, is malicious. In addition to the expected functionality, the malware creates a reverse TCP connection and provides a shell to the remote attacker. The malware causes unusual network bandwidth consumption and is eventually noticed by the administrator of the enterprise. To understand the attack and prepare for response, the administrator performs forensic analysis, trying to identify the root cause and assess the potential damage to the system. At the very beginning, the binary file *fcopy* is the only evidence. Hence, the creation of the file is used as the symptom event.

3.2.2 Traditional Solutions

Traditional techniques such as backtrackers [79, 25], audit systems [80] and provenance-aware file systems [29, 30] track the lineage of system objects or subjects without being aware by the applications. These techniques collect system subjects (e.g. processes and threads) and objects (e.g. files, network sockets and pipes) information at run time with system call hooking or Linux Security Modules (LSM) [81], and construct dependency graph or causal graph for inspection. Note that these two terms are interchangeable in this paper. While they use different approaches to trace system information, the graphs generated by these systems are similar.

A general workflow for these techniques is as follows. Starting from the given *symptom* subject or object, they identify all the subjects and objects that the symptom directly and indirectly depends on using backtracking. They also allow identifying all the effects induced by the root cause using forward tracking. For the case mentioned in §3.2.1, the administrator identifies the Firefox process and all its data sources by backtracking, and then discloses the downloaded files and the operations on these files with forward tracking. Figure 3.1 shows the simplified graph generated. In this graph and also *the rest of the paper*, we use *diamonds* to represent *sockets*, *oval nodes* to represent *files*, and *boxes* to represent *processes* or *execution units*. In Figure 3.1, many network sockets point to the Firefox process, and the process points to a large number of files including the torrent files and others like *fcopy*, which reflect the browsing and downloading behaviors of Firefox.

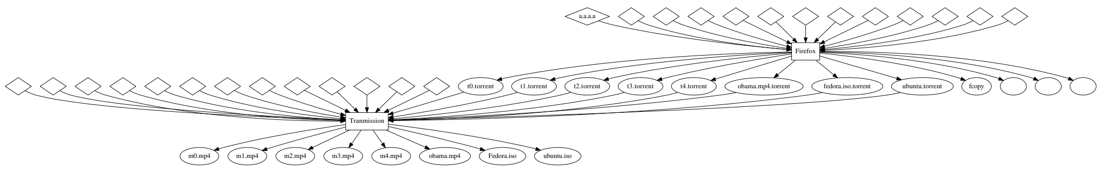


Figure 3.1.: Simplified causal graph for case in §3.2.1 generated by traditional solutions

While we only show part of the original graph in Figure 3.1 for readability, the original graph contains more than 500 nodes in total, with most files and network socket accesses being (undesirably) associated with the Firefox and Transmission nodes. These bogus dependencies make manual inspection extremely difficult.

3.2.3 Loop Based Partitioning Solutions

It was observed in [26] that the inaccuracy of traditional approaches is mainly caused by long running processes, which interact with many other subjects and objects during their lifetime. Traditional approaches consider the entire process execution as a node so that all the input/output interactions become edges to/from the process node, resulting in considerably large and inaccurate graphs. Take the Transmission process as an example. It has dependencies with many torrent files and network sockets, obfuscating the true causalities (e.g., a torrent file and the corresponding downloaded file).

Event loop based partitioning techniques [26, 10, 11] leverage the observation that long running processes are usually event driven and the whole process execution can be partitioned by the event handling loops (through binary instrumentation). They proposed the concept of *execution unit*, which denotes one iteration of an event handling loop. This fine-grained execution abstraction enables accurate tracing of dependency relationship. It was shown that these techniques can generate much smaller and more accurate dependency graphs. However, these techniques still have the following limitations that hinder their application in the real-world.

Units Are Too Low Level. Assume the administrator applies BEEP/ProTracer to the motivation case in §3.2.1. He constructs the causal graph starting from the file *fcopy*. He acquires the download event in Firefox, which is associated with the web socket *a.a.a.a*. Then, he traces back to the forum website, and eventually the search engine. As part of the investigation, the administrator applies forward tracking from the search engine page to understand if other (potentially malicious) pages were accessed and if other (potentially malicious) programs were downloaded and used. Since the developer visited many links returned by the search engine, the forward tracking includes many web pages and their follow-ups in the resulting graph. The simplified graph is shown in Figure 3.2.

In this case, Firefox is used for 5 minutes with 11 tabs containing 7 websites. There are thousands of nodes in the graph. This is because all user interactions like scrolling the web pages, moving mouse pointer over a link and clicking links are processed by unique event loop iterations, each leading to a unit/node. Moreover, Firefox has internal events including timer events to refresh pages. As these events operate on DOM elements, they are connected in the dependency graph due to memory dependencies, making the graph excessive.

The root cause of the limitation lies in that BEEP exposes very low level semantics (i.e., event loop iterations) in partitioning. The onus is on the user to chain low level units to form high level tasks. Unfortunately, BEEP graphs have little information to facilitate this process as they lack high level semantic information such as which high level task (e.g., tab) a low level unit belongs to.

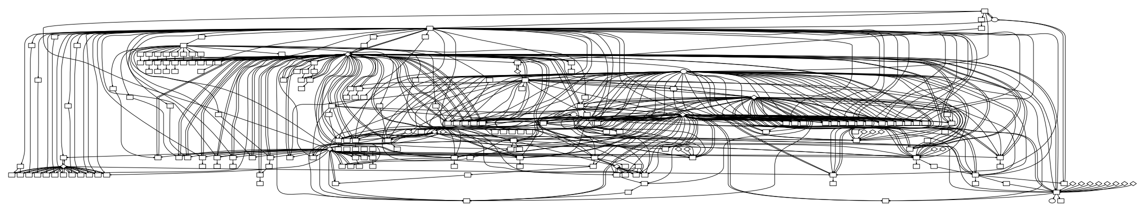


Figure 3.2.: Simplified backtracking causal graph for the case in §3.2.1 with event loop based partitioning technique. It only shows the causal relationship within the Firefox process (runs for 5 minutes with 11 tabs and 7 websites)

Depending on Training. BEEP and ProTracer are training based due to the difficulty of binary analysis. It requires intensive training to identify the event handling loops and memory accesses that disclose dependencies across units (e.g., one event loop inserts a task to the queue which is later loaded and processed by another event loop). The completeness of the training inputs is hence critical. Otherwise, there may be missing or even wrong causal relations. Note that providing source code does not address this problem as identifying event handling loops and cross-unit dependencies requires in-depth understanding of low level program semantics, which is much easier through dynamic analysis by observing concrete states than static analysis, in which everything is abstract. Specifically, there are a large number of loops in a program. Statically determining which ones are event handling loops

```
1  int main( int argc, char ** argv ) {
2      // parse options and session, load torrents
3      torrents = tr_sessionLoadTorrents(mySession, ctor, NULL);
4      // event loop
5      while( !closing ) {
6          tr_wait_msec( 1000 ); /* sleep one second */
7          // update and log and so on
8      }
9      // close program and sessions
10     return 0;
11 }
```

Figure 3.3.: Event handling loop of *Transmission* (version 2.6)

is difficult. Furthermore, while static analysis can identify memory dependencies, a lot of cross-unit dependencies should be ignored as they have nothing to do with the high level work flow (e.g., those caused by memory management or statistics collection).

In our motivating example, we did not use the “Go back" button in the initial training of Firefox. As a result, we were not able to get the full causal chain in Figure 3.2, which was broken at one web page that contains a lot of clicking-link and going-back actions. We had to enhance our training set by providing a going-back case.

Excessive Units. Partitioning based on event handling loops works nicely for server programs, in which one event loop iteration handles an external request and hence corresponds to a high level task. However, in many complex programs, especially those that heavily use threads to distribute workloads or involve intensive UI operations, event loop iterations do not align well with the high level tasks. As a result, it generates excessive small units that do not have much meaning. For example, in GUI programs, units are generated to denote the large number of GUI events (e.g., key strokes), even though all these events may serve the same high level task.

Consider the p2p program Transmission. Figure 3.3 shows its event handling loop in the main function of the daemon process. After parsing options, loading settings and torrent files (line 2-3), the daemon goes to a loop which exits only when the user closes the program (i.e., set *closing* to *TRUE*). In each iteration of the loop, it waits for 1 second (line 6), updates the torrent status and logs some information (line 7). Due to the nature of p2p protocol, downloading a single file requires thousands of loop iterations, leading to thousands of units in BEEP.

In many situations, there may not be any system events within these small units. For example, GUI programs monitor and handle frequent events such as page scroll. However, not all of them lead to system calls. Thus BEEP ends up with many “UNIT_ENTER” and “UNIT_EXIT” events without any system calls in between. These useless units waste a lot of space and CPU cycles. While existing techniques [40, 11, 66, 82] can remove redundant events, they cannot prevent these events from being generated in the first place.

These limitations are rooted at the misalignment between the rigid and low level execution partitioning scheme based on event loops. Ideally, the units generated by a partitioning scheme would precisely match with the high level logic tasks. MPI *aims to achieve this goal.*

3.2.4 Our Approach

The overarching idea of this paper is that high level tasks are reflected as data structures. MPI *allows the user to annotate the data structures that correspond to such tasks. It then leverages program analysis to instrument a set of places that indicate switches and inheritances of tasks to achieve execution partitioning.* Note that there may be multiple perspectives of the high level tasks involved in an execution, denoted by different data structures. Hence, MPI allows annotating multiple data structures, each denoting an independent perspective. To reduce the annotation efforts, MPI provides a profiler that can automatically identify the critical data structures (Figure 3.3.2). Note that allowing developers/users to insert logging related annotations/commands to software source code is a practical approach for system auditing. The Windows auditing system, *Event Tracing for Windows* (ETW), requires the developers to explicitly plan customized events to their software before deployment [83, 84]. These commands generate system events at runtime. In our design, we only require the developer to annotate (a few) task oriented data structures, MPI automatically instruments a much larger number of code places based on the annotations.

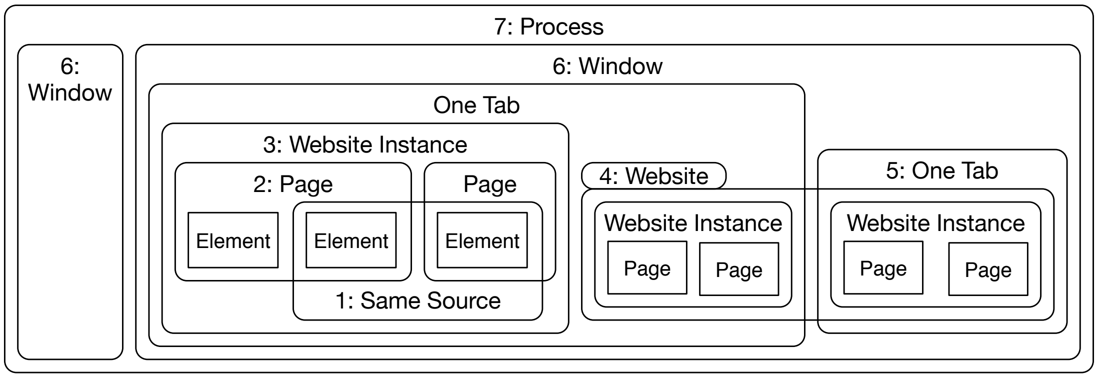


Figure 3.4.: Different perspectives of Firefox partitioning

Figure 3.4 presents a few possible perspectives of Firefox execution. By annotating the appropriate data structures, we can partition a Firefox execution into sub-executions of various windows (perspective 6), tabs (perspective 5), websites/domains (perspective 4), website instances (perspective 3), individual pages (perspective 2), and even the sources of individual DOM elements (perspective 1). Observe that some of the perspectives are cross-cutting. For instance, a tab may show pages from multiple domains whereas pages from the same domain may appear in multiple tabs. *A prominent benefit of such partitioning is to expose the high level semantics of the application to the underlying provenance tracking system.*

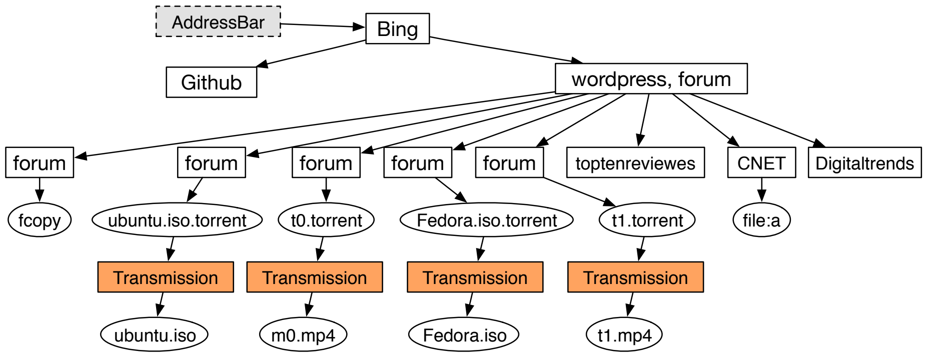


Figure 3.5.: Simplified MPI causal graph for the case in §3.2.1 (Firefox in tabs)

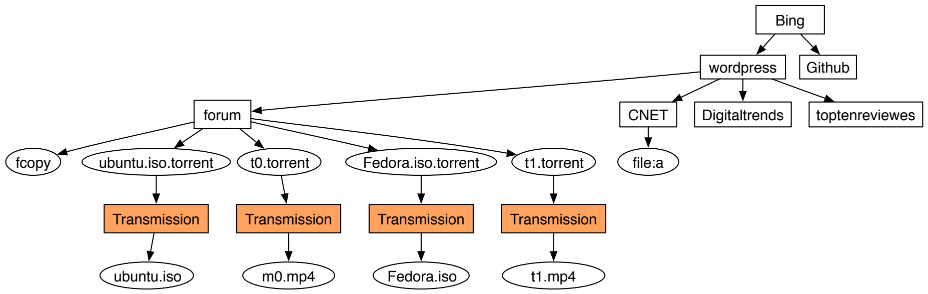


Figure 3.6.: Simplified MPI causal graph for the case in §3.2.1 (Firefox in websites)

Figure 3.5 shows the causal graph for the attack example when we partition the execution of Firefox by its tabs and Transmission by the files being downloaded. Each rectangle represents the life time of a tab. Observe that the Bing tab leads to the wordpress tab, which also shows the forum main page. A number of forum pages are displayed on separate tabs, each of which leads to the download of a torrent file through a Transmission unit. In contrast Figure 3.6 shows the causal graph when we partition the execution of Firefox by the websites/domains it visits. Observe that all the forum tabs are now collapsed to a single forum node. It clearly indicates that fcopy and the torrent files are downloaded from the same domain. Compared to the BEEP graph in Figure 3.1, these graphs are much smaller and cleaner, precisely capturing the high level workflow of the execution. *Note that these graphs cannot be generated by directly querying/operating-on the BEEP log, which has only very low level semantic information (i.e., event loop iterations).*

Advantages Over Event Loop Based Partitioning. We can clearly see data structure based partitioning system MPI addresses the limitations of event loop based partitioning. ① Units are no longer based on low level loop iterations. The inspector does not need to manually chain many such low level units to form a high level view of the execution. ② Dependency identification is made easy. Training is no longer needed. The memory dependencies that are needed to chain the low level event loop units are no longer necessary because these low level units are automatically classified to a high level unit in MPI. The incidents of missing causality due to incomplete training can be avoided. For instance, Firefox uses multiple threads to load and render the many elements on a page, which induces lots of memory dependencies across event loop units. But if we look at the execution from the tab perspective, these memory dependencies are no longer inter-unit dependencies that need to be explicitly captured. ③ Excessive (small and non-informative) units are prevented from being generated. All nodes representing timer event for Transmission will be merged into one node. Moreover, MPI provides great flexibility for attack investigation by supporting multiple perspectives. Enabling these perspectives is impossible if the appropriate semantic information is not exposed through MPI.

One may argue that event loop based partitioning can be enhanced by annotating event loops and cross-unit memory dependencies. However, such annotations are so low-level that (1) they require a lot of human efforts due to the large number of places that need to be annotated (e.g., the memory dependencies), and (2) they expose low-level and sometimes non-informative semantics such as mouse moves and timer events. In addition, the partitioning is solely based on event handling and hence cannot provide multiple perspectives.

3.3 Design

3.3.1 Overview

The overall process of analysis and instrumentation is shown in Figure 3.7. The user first annotates the program source code to indicate unit related data structures under the help of the annotation miner, which is essentially a data structure profiler. The analysis component, implemented as a LLVM pass, takes the annotations and analyzes the program to determine the places to instrument (e.g., data structure accesses denoting unit boundaries). The graph construction is using a standard algorithm, and details can be found in §3.3.6.

3.3.2 Annotations

Basic Annotations. Let us review how the Linux kernel conducts context switching internally, which inspires our approach to unit switching. Specifically, ① a *task_struct* with a unique *pid* identifies an individual process; ② a variable *current* is used to indicate the

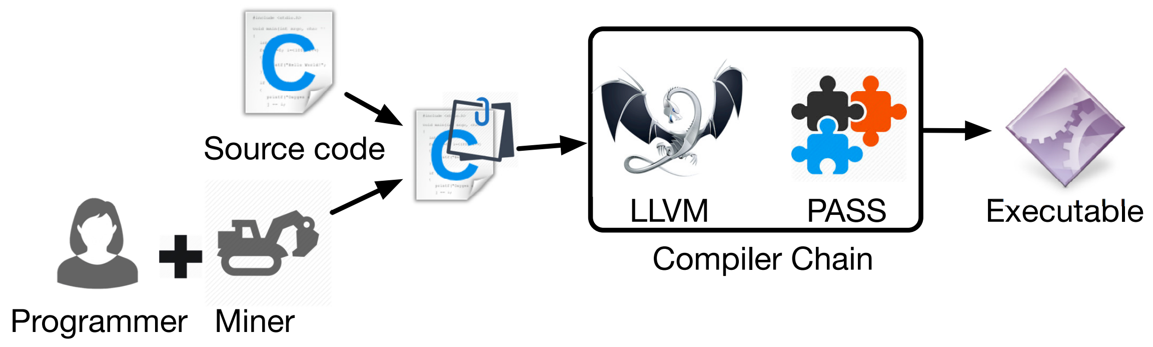


Figure 3.7.: MPI workflow

current active process. Processes can communicate through inter-process communication (IPC) channels like *pipes*. In order to perform unit switching, we need to identify the *unit data structure* that is analogous to *task_struct* and used to store per-unit information, a field/expression that can be used to differentiate unit instances as the identifier, and a variable that stores the current active unit. Note that there may not be an explicit task data structure in a program. Any data structure that allows us to partition an execution to disjoint autonomous units can serve as a unit data structure. Also, we need to know the variables that serve as communication channels between different unit instances. Thus we need the following types of annotations.

① @*indicator* annotates the variable/field that is used to indicate the possible switches between different unit data structure instances (similar to the variable *current* in Linux kernel). The user can choose to annotate multiple indicator variables/fields, one for each perspective. A unique id is assigned to each type of indicator.

② @*identifier* is an expression used to differentiate the instances of a unit data structure (similar to the data field *pid*). This expression can be a field in the data structure or a compound operation over multiple fields. Since an identifier must be paired up with the corresponding indicator, we allow providing an indicator id as part of the identifier annotation.

③ @*channel* annotates the variables/fields that serve as "IPC channels" between two different *unit data structure* instances (similar to *pipes*). It contains a unique id number, and a parameter indicating which field stores the data that induces inter-unit dependencies.

□ *Example.* Vim is a *tabbed* editor with each *tab* containing one or multiple *windows*. Each *window* is a viewpoint of a *buffer*, with each *buffer* containing the in-memory text of a file [85]. A file buffer can be shared by multiple *windows* in the backend, and buffers are organized as a linked list. A natural way to partition its execution is to partition according to the file it is working on, each represented by a *file_buffer* data structure. Figure 3.8 shows a piece of code which demonstrates our annotations. Vim uses the variable *curbuf* to represent the current active buffer. Consequently, we use *curbuf* as our *indicator* variable. Line 2 shows the indicator annotation. The annotation has an id to distinguish different indicators for various granularities/perspectives. The id is used to match with the corresponding @*identifier* annotation. Vim creates a buffer for each file. We can hence use the absolute file path in the OS to identify each file buffer instance. Line 8 shows the @*identifier* annotation. It has two parts: ① an expression used to differentiate instances; and ② an indicator id used to match with the corresponding @*indicator* annotation. In this case, field *b_ffname* is the identifier with id 1. Vim maintains its own clip board to support internal copy(cut)-and-paste operations. When the user cuts or copies data from a *file_buffer*, it sets the field *y_current→array*. When the user performs a paste operation, it reads data from the variable and puts the data to the expected position. In this case, *y_current→array* can be considered as the IPC channel between the two different *file_buffer* instances. Line 25 shows the channel annotation. It contains a unique id for the channel (analogous to a file descriptor), and the reference path to the field. Note that this is to support communication using the Vim clip board. Our system also supports inter- or intra-process operations through the *system* clip board by tracking system level events.

Threading Support. In order to improve responsiveness, modern complex applications heavily rely on threads to perform asynchronous sub-tasks. More specifically, the main thread divides a task into multiple subtasks that can proceed asynchronously and dispatches them to various (background) worker threads. A worker thread receives sub-tasks from

```
// in file src/globals.h
@indicator=1
EXTERN buf_T*curbuf INIT(= NULL);

// in file src/structs.h
typedef struct file_buffer buf_T;
// buffer: structure that holds information about one file
@identifier=b_ffname, indicator=1
struct file_buffer{
  // associated memline
  memline_T b_ml;
  // buffers are orgnized as a linked list
  buf_T *b_next;
  buf_T *b_prev;
  char_u *b_ffname;      // full path file name
  // TRUE if the file has been changed and not written out
  int b_changed;
  // variables for specific commands or local options
  char_u *b_u_line_ptr;  // for 'U' command
  int b_p_ai;            // 'autoindent', local opts
  // other data field like change time or so
}; /* file_buffer */

// in file src/ops.c
@channel=channelID, data=(y_current->array)
static struct yankreg *y_current;

while(True) {
  e = fetch_pending_event();
  switch(e) {
    case MouseMove:
      ...
      break;
    case MouseClick:
      ...
      break;
      ...
  }
}
```

Figure 3.8.: Vim data structure and our annotations

the main thread and also other threads and processes them in the order of reception. It can also further break a sub-task to many smaller sub-tasks and dispatch them to other threads, including itself. This advanced execution model makes partitioning challenging because *we need to attribute the interleaved sub-tasks to the appropriate top level units*. In event loop based partitioning techniques [26, 11], all the event handling loops from various threads need to be recognized during training. More importantly, multiple event loop iterations (across multiple threads but within an application) may be causally related as they belong to the same task. The correlations are reflected by memory dependencies. As such, the training process needs to discover all such dependencies. Otherwise, the provenance may be broken. Unfortunately, memory dependencies are often path-sensitive and it is very difficult to achieve good path coverage. It is hence highly desirable to directly recognize the logic tasks, which are disclosed by corresponding data structures, instead of chaining low level event loop based units belonging to a logic task through memory dependencies.

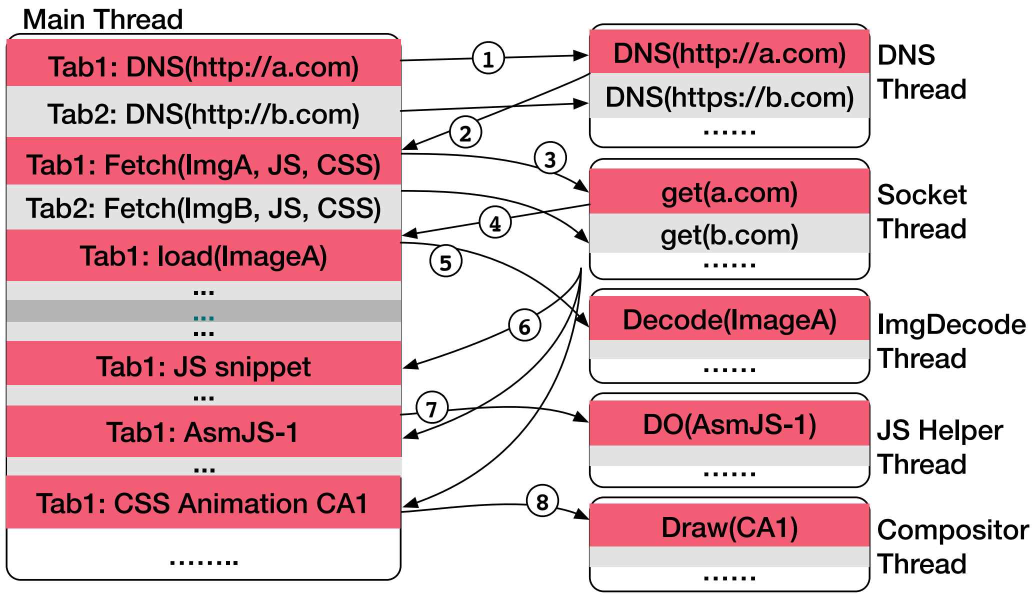


Figure 3.9.: Simplified Firefox execution model

☐ *Example.* Figure 3.9 illustrates a substantially simplified example of the Firefox execution model. It corresponds to an execution that loads two pages (in two respective tabs). Specifically, each box represents a thread and each colored bar (inside a box) denotes an iteration of the event handling loop (and hence a unit in BEEP/ProTracer). Observe that at step ①, the loading of tab1 first dispatches a Domain Name Server (DNS) query to a DNS thread, and then (step ③) posts a connection request to the socket thread to download the page. At step ④, the socket thread informs the main thread that the data is ready. The main thread leverages other threads such as the image decode thread, JS helper thread, and compositor thread to decode/execute/render the individual page elements. Note that every thread has interleaved sub-tasks belonging to various tabs. Edges denote memory dependencies across sub-tasks that need to be disclosed during training and instrumented at runtime in BEEP/ProTracer. □

Different from BEEP/ProTracer, our solution is to leverage annotations and static analysis to partition directly according to the logic tasks (e.g. tabs). In order to precisely determine the membership of a sub-task. We introduce the *@delegator* annotation. This annotation is associated with a data structure to denote a sub-task (e.g., the HTTP connection request posted to the socket thread). Intuitively, it is a *delegator* of a top level task (e.g., the HTTP connection request delegates the unit of its owner tab). At runtime, upon the dispatching of a delegator data structure instance (e.g., adding a sub-task to a worker thread event queue), it inherits the current (top level) unit identification. Later when the delegator is used (in a worker thread), the system knows which top level unit the current execution belongs to. There could be multiple layers of delegation. Similar to a unit, a delegator data structure also has an indicator, which is a variable like *current* whose updates may indicate delegation switches. More details can be found in Section 3.3.3.

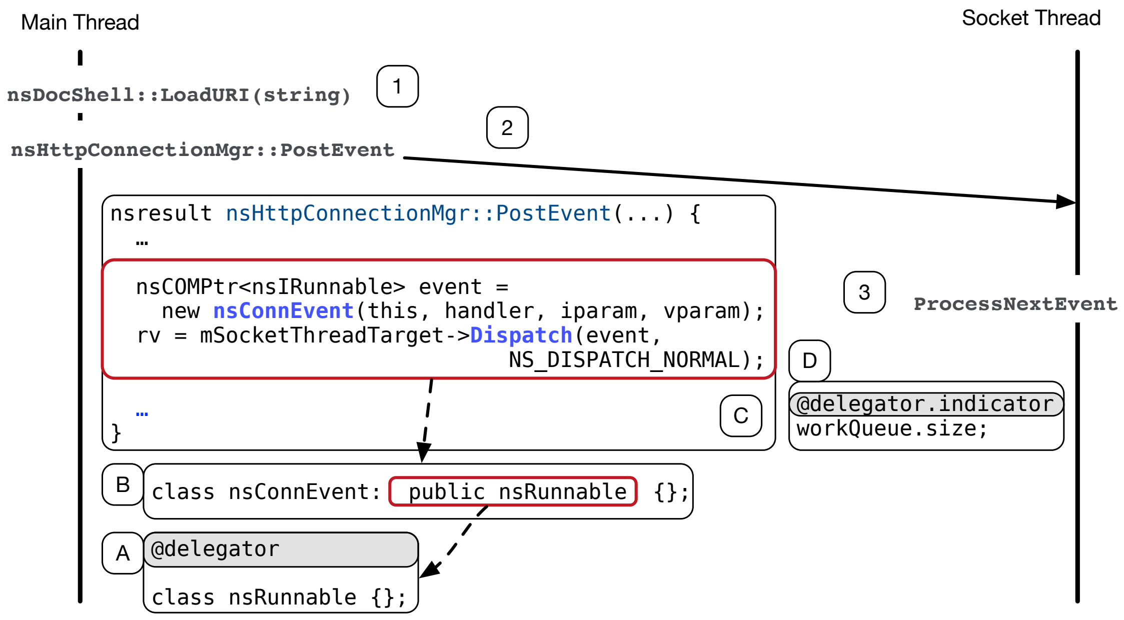
□ *Example.* Consider the Firefox execution model. The user can annotate a tab, a window, and/or an iframe as a top level unit. Internally, these are all represented by the same *nsPIDOMWindow* class. They are differentiated by the internal field values. Hence, we provide multiple perspectives by annotating the *nsPIDOMWindow* data structure and using different expressions in the identifier annotations to distinguish the perspectives. Figure 3.10 shows the annotations for tabs and windows. The indicator id 1 is for tabs and 2 for windows. Any tab or window changes must entail the change of the *mDoc* field, which is used as the indicator. The expressions in the corresponding identifier annotations mean that we can

```
@identifier=this->GetOuterWindow(2)->mWindowID, indicator=1
@identifier=this->GetTop()->mWindowID, indicator=2
class nsPIDOMWindow {
  @indicator=1
  @indicator=2
  nsCOMPtr<nsIDocument> mDoc;
  // Tracks activation state
  bool mIsActive;
  virtual already_AddRefed<nsPIDOMWindow> GetTop() = 0;
  nsPIDOMWindow *GetOuterWindow()
  { return mIsInnerWindow ? mOuterWindow.get() ?  this; }
  // The references between inner and outer windows
  nsPIDOMWindow         *mInnerWindow;
  nsPIDOMWindow         *mOuterWindow;
  // A unique  (64-bit counter)
  // id for this window.
  uint64_t mWindowID;
  /* other methods and data fields */
};
```

Figure 3.10.: Tab and window annotations in Firefox

acquire the tab of any given window by getting the second layer outer window, and the top level window by calling *GetTop()*.

The connection request data structure (in the SocketThread), the image data structure (in the image decoder thread), etc. are annotated as delegators. As such, when a connection request is created in the main thread, the request inherits the current tab/window id. When the request is used/handled in a SocketThread, the execution duration corresponding to the request belongs to the owner tab/window of the request. An example is shown in Figure 3.11. In Firefox, all delegator data structure classes have the same base class *nsRunnable*. As such, we only need to annotate *nsRunnable* as the delegator class (box A). When the main thread tries to load a new URI (step 1), it posts an *nsConnEvent* to the SocketThread (step 2) by calling the *PostEvent* method (box C). Since *nsConnEvent* is a sub-class of *nsRunnable* (box B), the delegator class, the newly created *nsConnEvent* inherits the tab/window id. The *nsRunnable* class provides a function *Run()*, which is implemented by its child classes to perform specific tasks. And each thread maintains its own work queue containing all such class instances. Thus the size of the worker queue is annotated as the indicator of the delegator. Whenever it changes, there may be a unit context switch. □



```
nsresult nsHttpConnectionMgr::PostEvent(...) {
  …
  nsCOMPtr<nsIRunnable> event =
    new nsConnEvent(this, handler, iparam, vparam);
  rv = mSocketThreadTarget->Dispatch(event,
                                     NS_DISPATCH_NORMAL);
  …
}
```

```
@delegator.indicator
workQueue.size;
```

```
class nsConnEvent: public nsRunnable {};
```

```
@delegator
class nsRunnable {};
```

Figure 3.11.: Firefox main thread and socket thread communication example

Annotation Miner. We develop an annotation miner to recommend unit and delegator data structures to annotate. The miner works as follows. The user provides a pair of executions to denote an intended unit task, one execution containing one unit and the other containing two units. Then, differential trace analysis is performed to prune data structures that are common in both traces and hence irrelevant to the unit (e.g., global data structures). The miner leverages the points-to relations between data structures to narrow down to the top data structures (i.e., those that are not pointed-to by other data structures). PageRank is further used to determine the significance of individual top data structures. A ranked list of data structures is returned to the user. Note that this mining stage is much less demanding than the training process in BEEP/ProTracer, which requires extracting code locations that induce low level memory dependencies. Since we focus on identifying high level data structures, which are covered by the provided inputs, completeness is not an issue for us in practice.

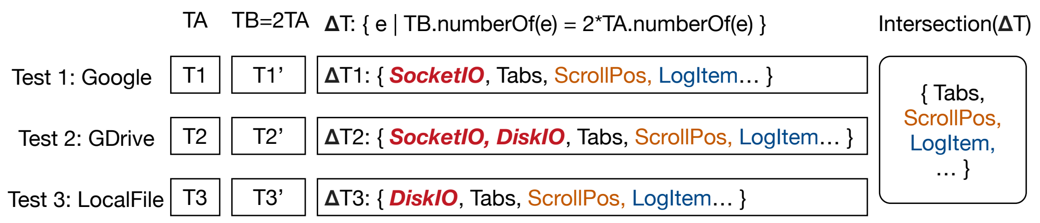

TA
TB=2TA
ΔT: { e | TB.numberOf(e) = 2*TA.numberOf(e) }
Intersection(ΔT)
Test 1: Google
T1
T1'
ΔT1: { SocketIO, Tabs, ScrollPos, LogItem… }
Test 2: GDrive
T2
T2'
ΔT2: { SocketIO, DiskIO, Tabs, ScrollPos, LogItem… }
Test 3: LocalFile
T3
T3'
ΔT3: { DiskIO, Tabs, ScrollPos, LogItem… }
{ Tabs, ScrollPos, LogItem, … }

Figure 3.12.: Example of annotation miner in MPI

Next, we show how to mine the tab data structure in Firefox (Figure 3.12). We first use a pair of runs to visit the Google main page. `T1` has one tab and `T1'` has two tabs. Δ`T` shows the data structures in the trace differences. Note that there are data structures specific to the page content but irrelevant to the intended unit, such as `SocketIO`. To further prune those, we use another two pairs of executions that visit Google Drive and a local file, respectively. The miner then takes the intersection of the trace differences to prune out `SocketIO` and `DiskIO`. The resulting set contains the top level data structures and their supporting meta data structures (e.g., the `ScrollPos` data structure to support scrolling in a tab). The trace-based points-to analysis then filters out the low level supporting data structures. There may be multiple top level data structures remained, many not related to units (e.g., for logging). Hence in the last step, PageRank is used to rank the several top data structures. In our case, the tab data structure is correctly ranked the top.

3.3.3 Runtime

Unit Context. At runtime, each thread maintains a vector called the *unit context*. Each element of the vector denotes the current unit instance for each unit type (or each perspective). Note that MPI allows partitioning an execution in different ways by annotating multiple unit data structures. If the user has annotated n unit data structures (with n indicators and n identifiers), there are n elements in the vector. Each time the indicator of a unit data structure is updated, the identifier of the data structure is copied to the corresponding vector element.

Delegation. MPI runtime provides a global hash map that is shared across all threads, called the *delegation table*. The delegation table projects a delegator data structure instance to a unit context vector value, denoting the membership of the delegator. Upon the creation/initialization of a delegator data structure instance, MPI inserts a key-value pair into the delegation table associating the delegator to the current unit context. Upon an update of the indicator of a delegator data structure (in a worker thread that handles the subtask represented by the delegator), the unit context of the current thread is set to the unit context of the delegator, which is looked up from the delegation table. Intuitively, it means the following execution belongs to the unit of the delegator until a different delegator is loaded to the indicator variable. The optimization of this process can be found in §3.3.5.

☐ *Example.* Let us revisit the Firefox example in Figure 3.9. We want to attribute all subtasks to their corresponding tabs (shown in different colors). In Figure 3.11, we show a detailed workflow of the main thread posting the connection event to the socket thread. The main thread first calls the *LoadURI* method (step 1), which invokes the *PostEvent* method. Within *PostEvent* (box C), it creates an *nsConnEvnet* and posts it to the socket thread. Since data structure *nsRunnable* (box A) is annotated as a delegator and the HTTP connection request *nsConnEvent* (box B) is a subclass of *nsRunnable*, MPI propagates the current unit id in the main thread to the worker thread, namely, the socket thread. Specifically, the request is associated with the current unit context of the main thread in the delegation table. Inside the socket thread that receives and processes the request (i.e., step 3), loading the request from the task queue causes the change of the queue size indicating a possible unit context switch. As a result, the current unit context of the socket thread is set to that of the request, namely, tab1. With a chain of delegations, MPI is able to recognize all the tab1 subtasks performed by different threads, namely, all the red bars in Figure 3.9 belong to the same tab1 unit. ☐

3.3.4 Analysis

The analysis component of MPI is a pass in LLVM responsible for adding instrumentation to realize the runtime semantics mentioned earlier. It takes a program with the four kinds of annotations mentioned in §3.3.2, and produces an instrumented version of the program that emits additional syscall events denoting unit context switches and channel operations.

MPI needs to identify the following a few kinds of code locations: (1) all the updates (i.e., definitions) to indicator variables, including unit indicators and delegator indicators, to add instrumentation for unit context updates; (2) all the creation/initialization locations of delegator data structures to add instrumentation for the inheritance of unit context; (3) reads/writes of channel variables/fields to add instrumentation for channel event emission and redundancy detection; (4) all the system/library calls that may lead to system calls to add instrumentation for unit event emission and redundancy detection. We use a type based analysis to identify (2) and (3). For (4), we pre-define a list of library functions (e.g., libc functions) that may lead to system calls of interest and then scan the LLVM bitcode to identify all the system calls and the library calls on the list. Details are elided. A naive solution to (1) is to perform a walk-through of the LLVM bitcode to identify all definitions to indicator variables or to their aliases (using the default alias analysis in LLVM). However, this may lead to redundant instrumentation. Specifically, an indicator may be defined multiple times and there may not be any system calls (or library calls that can lead to system calls) in between. As such, the unit context switch instrumentation for those definitions are redundant.
□ *Example.* The function *im_regexec_multi()* in Figure 3.13 searches for a regular expression in Vim. The *indicator* variable is updated at line 6, and then again at line 8. The operations inside function *vim_regexec_both()* are all on memory. In other words, it does not make any system calls directly or indirectly. As such, the instrumentation for line 6 is redundant. □

The problem is formulated as a reaching-definition problem, which determines the set of definitions (of a variable) that can reach a program point. We say a definition of variable x

```
1  /* Match a regexp against multiple lines. */
2  long im_regexec_multi(...) {
3    buf_T        *save_curbuf = curbuf;
4    // initilize local variables
5    // switch to buffer "buf" to make vim_iswordc() work
6    curbuf = buf;
7    r = vim_regexec_both(NULL, col, tm);
8    curbuf = save_curbuf;
9    return r;
10 }
```

Figure 3.13.: Instrumentation example (Vim, *op_yank* function)

can reach a program point ℓ, if x is not redefined along any paths from the definition to ℓ. In our context, we only instrument the definitions that can reach a system call or a library call that can lead to a system call. In Figure 3.13, the definition at line 6 cannot reach any point beyond line 8. Since line 7 does not denote any system call, line 6 is not instrumented. §3.3.6 discusses how to construct attack graphs from MPI logs. The algorithm is elided.

3.3.5 Run Time Optimization

MPI emits special syscall events to denote unit context switches, and channel reads/writes. During causal graph construction §3.3.6, the unit context switch events are used to derive unit boundaries and the channel events are used to derive inter-unit dependencies. Note that channel operations are essentially memory reads and writes that need to be exposed as system events. Otherwise, they are invisible to MPI. Inter-unit communication through system resources such as files, sockets, and the system clipboard can be captured by the default underlying system event tracking module without the intervention of MPI.

A naive solution is to emit a unit context switch event upon any indicator update and a channel event upon any channel read/write. However in practice, we observe that (1) an indicator update may not imply the change of the unit context and (2) even though the unit context changes, there may not be any system events that happen in between the two unit context switches. Both cases lead to redundant unit context switch events. Similarly, there are often multiple accesses to the same channel object within the same unit. These accesses must induce the same causality and hence cause redundancy. Since emitting an event entails a system call and hence a context switch, preventing redundant event emission is critical to the efficiency of MPI. We have two approaches to address this problem. One is through the static analysis (§3.3.4) and the other is runtime optimization. MPI does not emit any event upon an indicator update. Instead, it simply updates the current unit context (in memory), which has much lower overhead compared to a system call. Upon a regular system call (e.g., file read), it checks if the current unit context is the same as the previous context that was emitted. If not, it emits a unit context switch event right before the system call. Otherwise, it does not emit. Similarly, upon a channel operation, MPI checks if a channel operation by the same unit was logged before. If so, it avoids logging the channel operation.

3.3.6 Causal Graph Construction

In this section, we discuss the causal graph construction algorithms for backward tracking starting from a symptom event and forward tracking starting from a root cause event. Algorithm 1 shows how to generate the *backward tracking* causal graph for a specific perspective with a given log file and a symptom event. Generating the graphs for all perspectives only requires an easy extension.

We use an $objs$ set to represent the system objects, subjects, and channels between units that are directly or indirectly related to the symptom event. The overall procedure of the algorithm is to traverse the log in a reverse order to populate the set and identifies events causally related to the symptom by correlating to some entity in $objs$. At line 1, the algorithm initializes the set to contain the system object accessed by the symptom event and the system subject (i.e., the process of the event). It also marks the current unit as correlated to the symptom (line 2). Then it traverses all the events in the log file in a reverse order, starting from the symptom event (lines 3-17). If the current event e is not a unit context switch event, the algorithm saves it in a temporary list of events for the current unit (line

Algorithm 1 Backward Causal Graph Construction

```
Input:    $L$ - the event log
          $l$ - unit type (i.e., perspective) given in the @*indicator* annotation
          $e_s$ - symptom event
Output:   $G_l$ - the generated causal graph for perspective $l$
Variable: $objs$ - system objects/subjects relevant to $e_s$
          $s_e, pid_e$ - the system object/pid of event $e$
          $bUnit$ - if the current unit causally related with $e_s$
          $eventUnit[pid]$ - the events in the current unit of process $pid$
1:  $objs \leftarrow \{\, pid_{e_s}, s_{e_s}\}$
2:  $bUnit \leftarrow true$
3:  for each event $e \in L$ in reverse order, starting from $e_s$ do
4:      if $e$ is not a unit context switch event then
5:          $eventUnit[pid].add(e)$
6:      if $e$ updates any object or subject in $objs$ then
7:          $bUnit \leftarrow true$
8:      if $e$ is a unit context switch event then
9:          if $e$ does not switch to a $l$ unit then
10:             continue
11:         else
12:             if $bUnit$ then
13:                 add events in $eventUnit[pid_e]$ to $G_l$
14:                 add accessed objects/subjects in $eventUnit[pid_e]$ to $objs$
15:         $eventUnit[pid_e] \leftarrow \varnothing$
16:         $bUnit \leftarrow false$
17: return $G_l$
```

4-5). If e updates an object (e.g., file and pipe) or spawns a subject (i.e., process) that was identified as related to the symptom (and hence in the $objs$ set), a flag is set to indicate that the current unit is correlated (lines 6-7). If e is a unit context switch, the algorithm further tests if e switches to a unit in the given perspective. If not, the switch event is irrelevant and simply skipped (lines 9-10). Otherwise, it indicates a unit boundary of our interest. The algorithm checks the flag to see if the current unit is causally related to the symptom (lines

11-12). If so, it adds all the events in the current unit to the result graph. It also updates $objs$ with all the objects read by any event in the current unit and all the subjects spawned in the unit (lines 13-14). The temporary event list and the flag are then reset (lines 15-16). Note that when the events are added to the graph, nodes are created and further connected to existing nodes in the graph by the dependencies implied by the events. For example, a file read event entails connecting to the (previously created) file node. Details are elided for brevity.

☐ *Example.* Figure 3.14 shows an example of constructing the backward causal graph. The simplified log entries are shown on the left while the generated graph is shown on the right. The graph is also annotated with events to explain why nodes/edges are introduced. The algorithm generates the graph starting from the symptom event at line 8, which is a write event to the socket *a.a.a.a*. It traverses back and reaches line 7, which is a unit context switch (UCX) event whose *indicator* is 5 and the *identifier* value is 7. Two nodes are hence created representing that a process (node) wrote to a socket (node) whose value is *a.a.a.a*. Going backward, the algorithm further identifies another unit represented in lines 4-6 with the *indicator* value 5 and the *identifier* value 3. This is a different unit instance of the same type and it has no causal relation with the object set that currently contains the socket object and the process. Therefore, all the events in this unit are dropped. The algorithm continues to traverse backward and encounter another unit in lines 1-3. Line 2 indicates that it reads file *index.html*, so the subgraph for lines 1-3 is file *index.html* being read by the process. Note that the value of *identifier* indicates lines 1-3 and lines 7-8 belong to the same unit (instance), which means that the application is working on the same task. Hence, the global causal graph is updated by joining the two subgraphs. The result graph is shown on the right hand side.

The forward graph construction algorithm is similar and hence omitted.

Essence of MPI and Memory Dependencies. From the graph construction Algorithm 1, one can observe that all the events in a unit are considered correlated. If there is a single event (within a unit) that has any direct/indirect dependency with the symptom, all the events in the unit are added to the graph and all the objects/subjects accessed by the unit

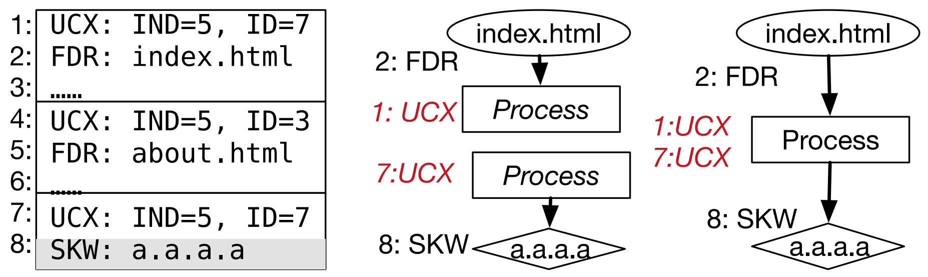


Figure 3.14.: An example of constructing backward causal graph. (*UCX* is short for *Unit Context Switch*, *FDR* is short for *File Descriptor Read*, and *SKW* is short for *Socket Write*.)

are considered correlated. As such, MPI does not need to track any fine-grained (memory) dependencies *within a unit*. Dependencies across units are either captured through system level dependencies (e.g., file/socket reads and writes) or explicitly indicated by the user through the channel annotation.

3.4 Evaluation

In this section, we present the evaluation results including the annotation efforts needed, the runtime and space overheads of the prototype, and a number of attack cases to show the advantages of MPI compared to the event loop based partitioning technique in BEEP [26] and ProTracer [11]. For comprehensive comparison, we integrate both MPI and event loop based partitioning with three underlying provenance tracking systems, the Linux Audit system, ProTracer and LPM-HiFi.

3.4.1 Overhead

Space overhead: We measure the space overhead of MPI and compare it with the overhead of event loop based partitioning, on the aforementioned three provenance tracking systems. We measure the overhead of MPI and BEEP on Linux Audit and LPM-HiFi by comparing the logs generated by the original binaries and the instrumented binaries. ProTracer requires unit information to eliminate redundant system events (e.g., multiple reads

Table 3.1.: MPI Space Overhead

Application	Level	BEEP Space Overhead			MPI Space Overhead			BEEP	MPI
		Linux Audit	LPM-HiFi (Raw - Gzip)		Linux Audit	LPM-HiFi (Raw - Gzip)		ProTracer (MB)	
Apache	HTTP Connection	15.38%	12.87%	0.64%	5.37%	3.75%	0.16%	22.12	20.08
Bash	Command	0.45%	0.34%	0.01%	0.41%	0.34%	0.01%	1.01	0.78
Evince	Document File	3.72%	4.98%	0.25%	0.04%	0.04%	0.00%	0.22	0.21
Firefox	Tab	42.16%	38.23%	1.01%	18.20%	13.24%	0.52%	593.23	228.54
Krusader	Command	26.54%	24.53%	0.09%	5.71%	4.89%	0.24%	2.31	2.31
Wget	Request	0.43%	0.33%	0.01%	0.42%	0.33%	0.01%	4.33	4.33
Most	File	0.05%	0.04%	0.00%	0.05%	0.04%	0.00%	1.78	1.78
MC	Command	0.93%	0.75%	0.01%	0.90%	0.75%	0.01%	3.43	1.89
Mplayer	Video File	0.04%	0.04%	0.00%	0.04%	0.04%	0.00%	0.34	0.34
MPV	Video File	0.09%	0.03%	0.00%	0.09%	0.03%	0.00%	0.58	0.58
Nano	File	0.29%	0.11%	0.01%	0.01%	0.01%	0.00%	8.23	2.46
Pine	Command	8.11%	6.09%	0.27%	7.28%	4.09%	0.13%	34.23	14.32
ProFTPd	FTP Connection	4.61%	3.45%	0.17%	2.11%	1.27%	0.06%	24.98	20.35
SKOD	FTP Connection	5.99%	3.89%	0.17%	2.68%	1.99%	0.10%	25.35	22.73
TinyHTTPd	HTTP Connection	8.94%	5.32%	0.32%	2.72%	1.08%	0.04%	43.24	37.48
Transmission	Torrent File	18.41%	18.33%	1.03%	0.12%	0.12%	0.01%	8.34	8.23
Vim	File	2.23%	2.32%	0.12%	0.13%	0.13%	0.01%	17.23	9.48
W3M	Tab	38.74%	30.45%	1.07%	24.67%	18.23%	0.19%	145.26	73.26
Xpdf	Document File	0.03%	0.07%	0.00%	0.03%	0.07%	0.00%	0.45	0.45
Yafc	FTP Connection	3.44%	1.78%	0.09%	2.60%	0.87%	0.04%	26.34	18.27

of a file within a unit). Therefore, it needs to work with an execution partitioning scheme. We hence compare the ProTracer logs by BEEP and by MPI. Note that BEEP+ProTracer is equivalent to the original ProTracer system [11] and in MPI+ProTracer we retain the efficient runtime of the original ProTracer but replace the partitioning component with MPI. Since BEEP supports only one low-level perspective, we only annotate one perspective in MPI during comparison. The overhead of multiple perspectives is in §3.4.2.

The results are shown in Table 3.1. The table contains the following information (column by column): 1) Application. 2) Perspective for partitioning. 3) Overhead of BEEP on Linux Audit, i.e., comparing the Linux Audit log sizes with and without BEEP. 4) Overhead of BEEP on LPM-HiFi with the raw log format. 5) Overhead of BEEP on LPM-HiFi with its Gzip enabled user space reporter tool. 6-8) Overhead of MPI on BEEP and LPM-HiFi. 9) Log size of BEEP on (original) ProTracer. 10) Log size of MPI on ProTracer. Note that Linux Audit and LPM-HiFi have different provenance collection mechanisms, i.e. system call interception for Linux Audit and LSM for LPM-HiFi. This leads to different space overheads. LPM-HiFi provides different user space reporters, and the Gzip enabled reporter has less space overhead.

Observe that for most programs our approach has less overhead on all the three platforms. For programs like document readers and video players, both approaches show very little overhead. These programs do not need to switch between different tasks frequently, which means they rarely trigger the instrumented code. Our approach shows significant better results for many programs like web browsers, P2P clients, HTTP and FTP programs including servers and clients due to a few reasons. Firstly, in these programs, the events handled by the event handling loop are at a very low level, whereas MPI can partition execution at a much higher level. Thus there are fewer unit context switches in our system, and multiple execution units in BEEP are grouped into one in our system without losing precision. For example in Apache, a remote HTTP request can lead to redirection, and the Apache server needs a few BEEP execution units to handle it. This triggers the instrumented code several times. But in MPI, multiple requests, including their redirections, of a same connection are grouped together. Thus, the instrumentation (for unit context switch) is triggered less frequently. Another reason is that we avoid meaningless execution units. For example in benchmark Transmission, BEEP execution units are based on time events, leading to many redundant units. This is avoided in MPI. Firefox has high overhead in both systems. When multiple tabs are opened, Firefox processes them in the background with threads. Since most of the requests involve network or file I/O, a lot of system/unit context switches are triggered, leading to the overhead. Despite this, the overhead of our system is about one third of that of BEEP. Note that there is another advantage of MPI that cannot be quantified –MPI does not require extensive training to detect low level memory dependencies. During our experiments, we had to add test inputs to the training sets of BEEP to ensure the provenance was not broken for a number of applications (e.g., Firefox).

We want to point out that with MPI, we can even reduce space overhead for the highly efficient ProTracer system and the reduction is substantial for a few cases. This is because MPI produces higher level execution units (compared to BEEP/ProTracer), leading to fewer units, more events in each unit and hence more redundancies eliminated by the ProTracer runtime. Also note that all the advantages of MPI over BEEP (e.g., without requiring extensive training and rich high-level semantics) are also advantages over ProTracer as the original ProTracer system relies on BEEP. We have ran MPI for 24 hours with a regular workload. The generated audit log has 680MB with 80MB by MPI. Details can be found in §3.4.2.

Run time overhead: We measure the run time overhead caused by our instrumentation. For server programs, we use standard benchmarks. For example, for the Apache web server, we use the *ab* [44] benchmark. For programs that do not have standard test benchmarks, but support batch mode (e.g., Vim), we translate a number of typical use cases to test scripts to drive the executions. We preclude highly interactive programs.

For each application, we choose the same perspectives as the previous experiment, and the results are shown in Figure 3.15. For each program, we have eight bars. ①MPI-Native: the overhead of MPI without any provenance system over native run. ②MPI-ProTracer: the overhead of MPI over ProTracer. ③MPI-LPM: the overhead of MPI over LPM-HiFi. ④MPI-Audit: the overhead of MPI over Linux Audit. The other four bars denote the

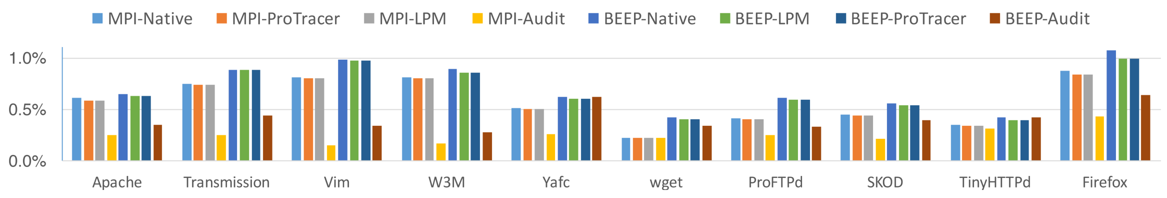


Figure 3.15.: MPI: Run-time overhead for each application

overhead of BEEP. As we can see from the graph, most applications have less than 1% run time overhead for all situations, which is acceptable. Comparing with BEEP, MPI shows less overhead in all cases. The low run time overhead is due to the following factors. Firstly, compared with the original program, the number of instrumented instructions is quite small. Secondly, most of the instructions are rarely triggered. Thirdly, our instrumentation mainly contains memory operations like comparing the newly assigned *identifier* value with the cached value.

3.4.2 More Overhead Experimental Results

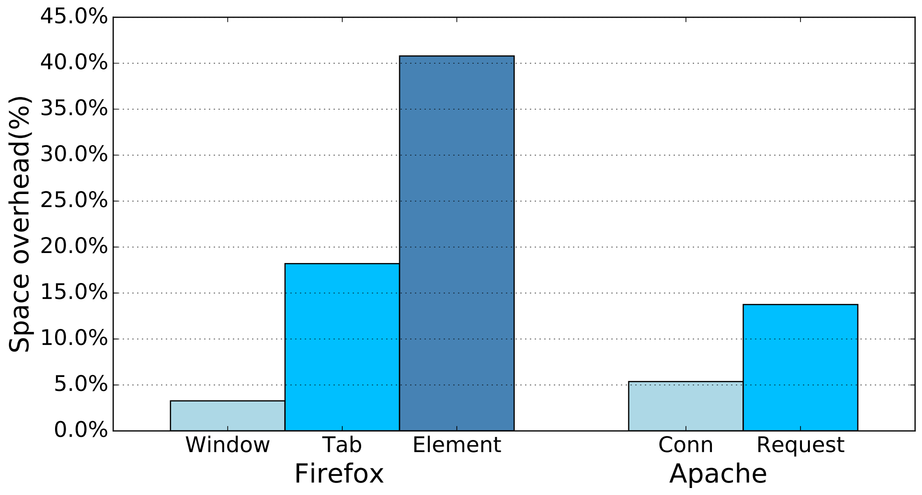


Figure 3.16.: MPI: Run-time overhead for different partitioning perspectives

We also conduct experiments to measure space overhead for the same application with different partitioning choices, and the results are shown in Figure 3.16. We select two programs, Firefox and Apache. For Firefox, we choose three different ways to instrument: windows, i.e., a unit for a top level residence window for tabs (note that multiple windows may be driven by the same Firefox process internally); tabs and elements (inside a page). We do not show the numbers for each web site instance, because the instrumentations are similar to those of tabs, and the only difference lies in the expressions used in the *@identifier* annotation (see §3.3). For Apache, we use two ways to instrument: each connection (each client instance), and each request. The results show that with different levels of instrumentation, the overhead is significantly different. Instrumenting the applications at a higher level causes less overhead. For both cases, a lower level suggests 2-3 times overhead increase.

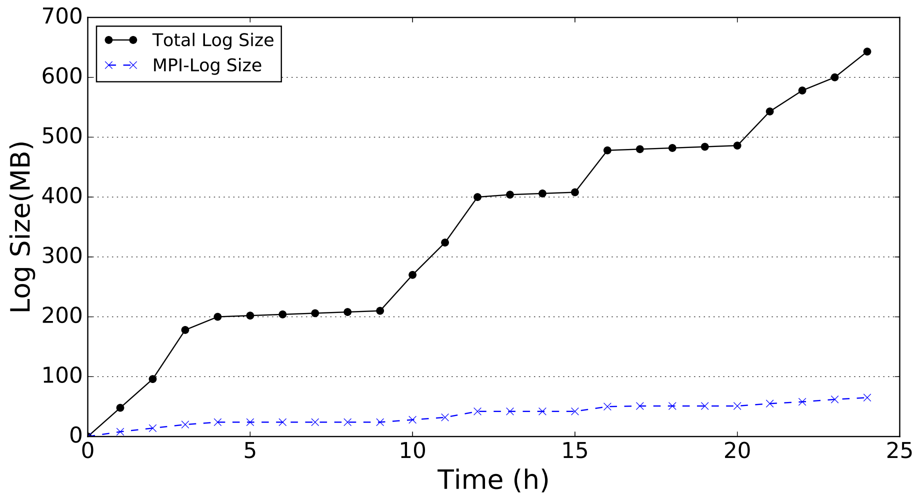


Figure 3.17.: MPI: Space overhead for a whole day

The last space overhead experiment we did is to run the instrumented applications on our machine for a whole day with Linux audit system enabled and measure the events generated by MPI. The workload includes regular uses such as web surfing, checking and responding emails. The result is shown in Figure 3.17. The black solid line shows the log size generated by the Linux audit system, and the dashed blue line shows the log size generated by MPI. From the graph, we can see that the log size generated by the Linux audit is more than 600 MB while our instrumentation issues less than 80 MB.

Table 3.2.: MPI Annotation Efforts

Application	LOC	Annotation				Inst
		ID	IND	Chann	DEL	
Vim	313,283	3	3	2	0	878
Yafc	22,823	2	3	0	1	111
Firefox	8,073,181	3	32	0	1	6,867
TuxPaint	41,682	2	2	0	0	121
Pine	353,665	2	2	2	0	746
Apache	168,801	2	2	0	1	2,437
MC	135,668	2	2	1	0	3,332
ProFTPd	307,050	3	3	0	1	4,905
Transmission	111,903	2	4	0	1	66
W3M	67,291	2	2	0	1	3,718

3.4.3 Annotation Efforts

In this experiment, we measure the effectiveness of the annotation miner and the number of annotations eventually added. The annotation results are shown in Table 3.2. We only show some representative programs as the others have similar results. We present the applications in the first column, and their sizes (measured by SLOCCount [86]) in the second column. In the next four columns, we show the number of annotations needed for @identifier, @indicator, @channel, and @delegator. For each program, we provide two or more perspectives, as denoted by the number of @identifier annotations. In the last column, we show the instrumentation places automatically identified by our compiler pass. Less than 20% of these places were covered by our profiling runs. In other words, a training based method like that in BEEP/ProTracer would not be able to cover all these places.

3.4.4 Annotation Miner

To evaluate the annotation miner, we use the 20 programs in Table 3.1. For each program, we report the ranking of the unit/delegator data structures that we eventually choose to annotate. There are totally 52 of them. All the 6 delegator data structures are correctly ranked the top. That is because they are mainly used in worker threads, which have relatively fewer data structures. For the 46 unit data structures that we eventually annotate, 36 of them are ranked at the first place, 8 at the second place, and the remaining 2 at the third place. Figure 3.18 shows the reported data structures for Vim, Firefox and HTTPd. Each plane denotes the results for a perspective. The highlighted data structures are the ones that we eventually choose to annotate. The reason why we do not always annotate the top data structures is that they are typically the *shadow* data structures of the real unit data structures. They usually store meta-data related to units, causing them to have higher ranks than the real unit data structure. With the help of the miner, we spent minutes to hours to finalize the annotations. We argue that such efforts are manageable. More importantly, they are one-time efforts.

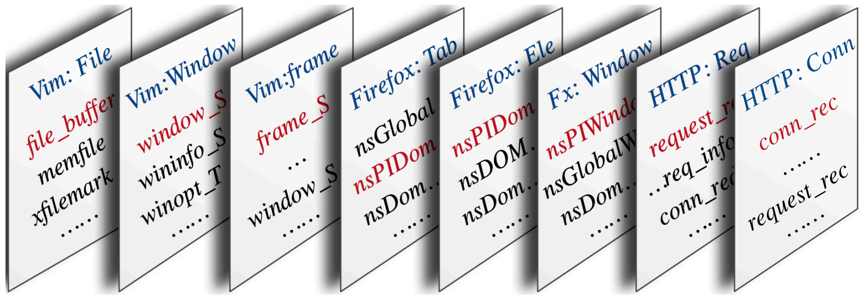


Figure 3.18.: Annotation miner results

3.4.5 Attack Investigation

To evaluate MPI's effectiveness in attack investigation, we apply it on 13 realistic attack cases used in previous works [27, 26, 40, 11]. The results show that MPI is able to correctly identify the root causes with very succinct causal graphs for all cases. Moreover, MPI generates fewer execution units using the perspectives in Table 3.1, when compared to BEEP/ProTracer. On average, the number of units generated by MPI is only 25% of that by BEEP/ProTracer. For attacks involving GUI programs (e.g., Firefox), the number is 8%, and in an extreme attack case involving Transmission, it is less than 1%. In terms of the generated attack graphs, MPI can reduce the number of nodes to 92% and the number of edges to 83% on average. Note that it is because these attacks have simple propagation paths such that the BEEP/ProTracer graphs are quite succinct. For complicated cases, MPI can reduce the graphs to 76%(nodes)/62%(edges). In addition, we evaluate it on a few other realistic attack cases. Next, we show one such case. Two more cases are presented in §3.4.6 to demonstrate the advantages of MPI over BEEP/ProTracer in an insider threat and in tracking complex browsing behaviors in Firefox.

Case: FTP Data Leak. Exploiting system misconfiguration to acquire valuable sensitive information is a common attack vector [87, 88]. It is important to assess and control damages once the problem is noticed. In the following incident, an FTP administrator accidentally configured the root directory of many users to a folder containing classified files, and gave them read accesses. After noticing the problem, he shut down the server and then conducted investigation to figure out the significance of the potential information leak. In the duration of the misconfiguration, there are thousands of connections from a large number of users. The number of classified files is also large.

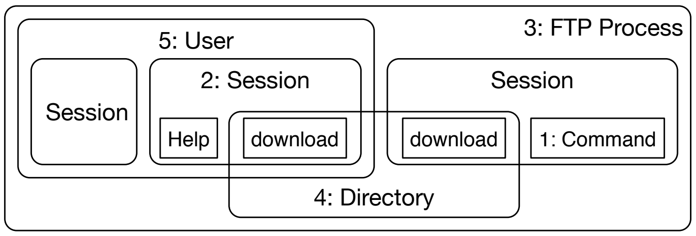


Figure 3.19.: FTP server partitioning perspectives

In Figure 3.19, we show a number of possible investigation perspectives for the FTP server application. Event loop based partitioning techniques are based on each command or user request (box 1), and traditional auditing approaches are based on the whole process (box

3). MPI provides choices that align better with the logical structures of the application, such as the session perspective (box 2), i.e., all the commands/requests from a session belong to a unit, the directory perspective (box 4), i.e., all the commands on a given directory are considered a unit, and the user perspective (box 5), i.e., all commands/requests from a user (not limited to an IP address) belong to a unit. Note that all FTP commands are associated with some file or directory as part of its context, and hence we can partition FTP execution based on this information.

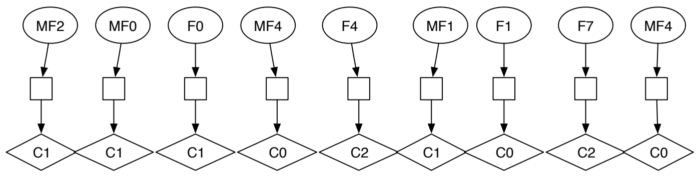


Figure 3.20.: FTP server partitioned by BEEP

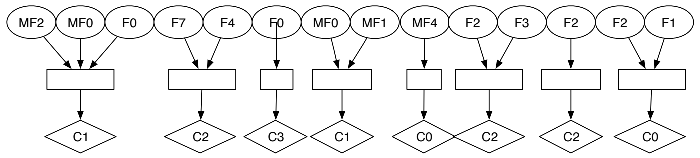


Figure 3.21.: FTP server partitioned by each connection

Part of the BEEP graph is shown in Figure 3.20. Observe that each user command is captured as a unit. The simplified graph by MPI with connection based partitioning is shown in Figure 3.21, and user based partitioning in Figure 3.22. The connection perspective alleviates the inspector from going through the individual commands. The user perspective can aggregate all the behaviors from a specific user over multiple sessions so that the inspector can hold individual users for responsibilities. Note that a user can use various IP addresses to connect to the server. Without MPI, such semantic information cannot be exposed to the provenance tracking system. The number of nodes in the BEEP, connection (MPI), and user (MPI) graphs are 962, 224, and 78, respectively. We want to point out that the MPI graphs cannot be generated from the BEEP graph by post-processing because of the subtask delegation in this program, i.e., it is difficult to attribute a sub-task to the top level unit that it belongs to with only the low level semantic information in the BEEP graph.

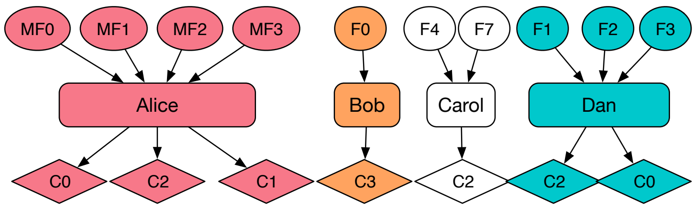


Figure 3.22.: FTP server partitioned by users

3.4.6 Case Studies

We have presented a watering hole attack through Firefox and Transmission in §3.2. Next, we will present more attack cases and one scenario with complex user behaviors to show the advantages of MPI. These are attacks simulating real stories. Note that since APT attacks are staged (and hence last for a long time), involve many (vulnerable) applications with some cannot be found any more, and rely on human mistakes, it is very difficult to apprehend or reproduce real APT attacks. Therefore, we simulate attacks following the same strategies used in real incidents. based on attack stories by repeating the similar human behaviors (e.g., clicking phishing emails) and replacing unavailable apps with apps with similar functionalities. This is consistent with the experiments conducted by existing works [26, 11, 27].

Case: Insider Threat. In attacks such as watering hole and phishing emails, the adversaries apply external influences and wait for the employees to make mistakes. However, it is also very common that attacks are launched from inside the enterprise (e.g., by malicious or former employees). In fact, a large number of such cases had been reported [89, 90, 91, 92]. Next, we simulate such an attack.

A computer game development company noticed that the graphical design of a to-be-announced game was leaked on an online gaming forum. The company started investigation, trying to understand how this design was leaked and who should be held responsible. The investigator first conducted forward tracking from the design file but found that the file was neither sent outside by any email nor copied by any employee to their own devices. She further suspected that some old version of the file was leaked instead of the current version. Even though the old versions of the design file did not explicitly exist any more, the provenance of the file was tracked by the audit system.

She first conducted backward tracking to disclose all the past versions (with the name "p_v" plus the version number) and then forward tracking to see how these versions were propagated/used. Assume that she used BEEP first. She quickly noticed a number of problems in the BEEP graph that makes manual inspection difficult.

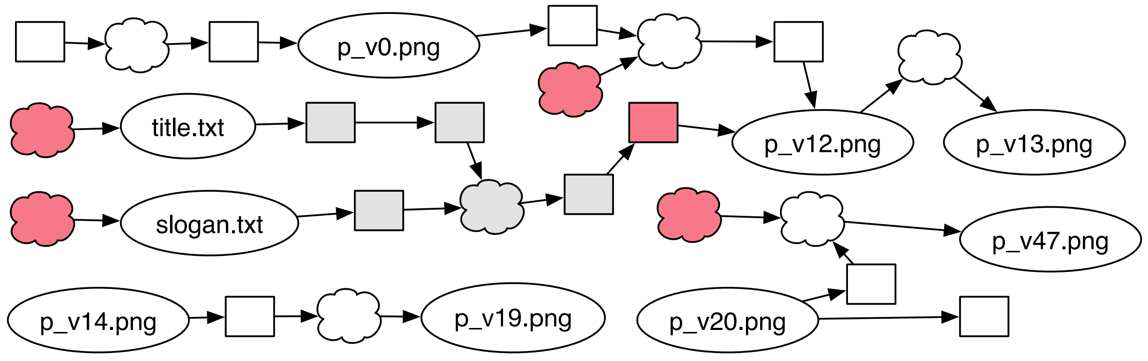


Figure 3.23.: Causal graph for insider attack using BEEP

The resulting graphs by BEEP are shown in Figure 3.23. White boxes represent units for TuxPaint [93], gray boxes are for the editor, Vim, and red boxes are for other apps. *First of all*, the graph is very large (containing 1832 nodes). This is because many people had contributed to the file in the past using TuxPaint, a graph drawing tool. There were a lot of interactions (e.g., copy & paste) among multiple image files, some of which were from Internet. The various historic versions of the design file were propagated to other places. *Second*, there are many "empty" execution units, which are execution units just have boundary events. This is because many operations in UI intensive program TuxPaint have no real effects on the provenance. These operations include, but are not limited to, switching painting tools (frequently), clicking menu bars and so on. *Third*, she found that most execution units for TuxPaint only have memory dependency events. This is because

TuxPaint stores the image buffers in memory, and flushes them to disk only when the user clicks the save button. In the editing units (e.g., choosing tools and drawing figures), TuxPaint only operates on the image buffers. These units are only connected by memory dependency and do not invoke any system calls. However, these units are important as they are responsible for chaining up the important behaviors.

After inspecting such a large graph, the inspector still could not spot any suspicious behavior. The reason is that there are broken links in the graph such that some updates to the design file are missing from the graph. Specifically, some of the editing actions were not in the BEEP training set such that the corresponding memory dependencies are not visible, leading to broken provenance, e.g., “p_v14.png” and “p_v20.png”.

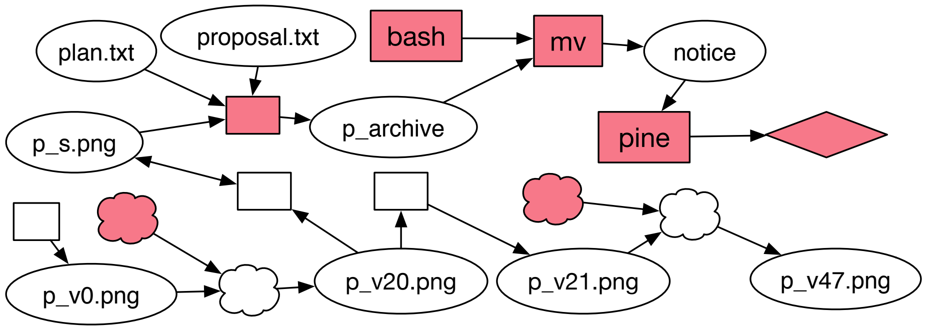


Figure 3.24.: Causal graph for insider attack using MPI

The inspector switched to MPI. She used individual image files as the perspective. The resulting (simplified) graph is in Figure 3.24. Now each white box represents all the editing operations on a single file. It can be clearly seen that a version of the design file, “p_v20.png”, was read by a TuxPaint unit that operated on file “p_s.png”, which was later archived with a number of text files. The archive was renamed and sent through an email. The link from the design file to file “p_s.png” was missed by BEEP because the attacker opened the design file, conducted a few editing actions whose memory dependencies are missed by BEEP such that the later *save-as* unit is disconnected from the file read unit. Note that all these actions are individual units in BEEP that need to be chained up by memory dependencies, whereas they belong to the same unit in MPI. Overall, the MPI graph is precise, much smaller (152 nodes) and cleaner. We also want to point out that a graph similar

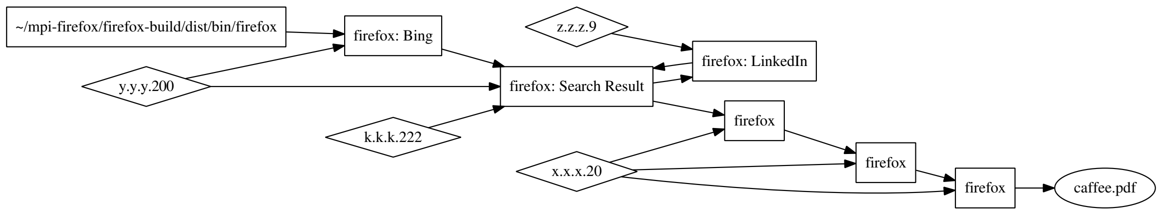


Figure 3.25.: Firefox in page perspective

to the MPI graph cannot be generated by post-processing the BEEP graph as the missing links cannot be inferred and it is difficult to determine which low-level nodes belong to an image file.

Case: Complex Browsing Behavior in Firefox. In this case study, we show how MPI precisely captures the causality of complex browsing behavior of Firefox. During browsing, the user first opened Bing from the bookmark bar, and searched a key word, and then used different ways to open new pages including clicking links, choosing "open page in a new tab/window" in the right-click menu, going back to the previous page, and opening new pages from Javascript code automatically. In the end, the user downloaded a PDF file. We collected the log with the page perspective and generated a causal graph by conducting backward traversal starting from the PDF file. The graph is shown in Figure 3.25. Observe that the entire browsing history is precisely captured by the graph, including visiting the LinkedIn page from the search result page and then going back to the search result page. In contrast, the BEEP's graph only includes the page hosting the PDF file, missing all the other pages along the causal chain, due to missing memory dependencies.

3.5 Discussion

Similar to many existing works [26, 40, 11, 70, 29], MPI trusts the Linux kernel and the components associated with the audit logging system. Attacks that can bypass the security mechanisms of these systems may cause problems for MPI. Moreover, attacks that target the underlying audit system, such as audit log blurring and log filling, may inject noise to logs, making log inspection difficult. As our system is built on top of existing provenance and operating systems, MPI leverages existing features provided by these systems to mitigate some of the problems. For example, operating systems like Ubuntu now leverages Ubuntu Software Center to deliver trustworthy software which can be used to protect the MPI binaries for benign software. Provenance systems like Hi-Fi uses reference monitor guarantees to protect audit logs, and LPM provides a general framework for trustworthy provenance collection. We argue these are orthogonal challenges to all existing provenance tracking techniques and a complete solution to all these challenges is not the focus of our paper. Instead, the emphasis of MPI is to address dependence explosion caused by long running processes with accuracy and flexibility.

MPI is essentially an add-on service to the OS-level provenance collection system (e.g. the Linux Audit system, LPM-HiFi, and ProTracer). System calls can be too coarse-grained. Fine-grained events, such as library calls or even instruction level dependencies, may need to be captured for some sophisticated attacks. We argue that the multiple perspective partitioning enabled by MPI is orthogonal. It is independent of the granularity of the events captured by the underlying provenance system. It can be easily integrated with systems of various granularities.

MPI requires program source code. We believe that the semantic information needed to enable multiple perspective partitioning is difficult to acquire through binary analysis for complex programs such as Firefox. If it is necessary to partition the execution of a binary, training and event loop based approaches such as BEEP could be used together with MPI. In the worst cases, MPI treats the entire process execution as a unit. Note that this approximation is only problematic for long running processes. Many malware executables are likely not long running such that treating a whole process as a unit does not introduce a lot of bogus dependencies. Also note that such approximation does not miss provenance so that the attack path is still captured. It is just that more efforts may be needed to go through the causal graph.

MPI relies on source code annotations, which are widely used in practice. Windows developers explicitly plant logging commands in their software source code to customize ETW auditing. Both GCC and LLVM provide advanced language features [94, 95, 96] that are triggered by annotations. For example, Firefox has 926 different types of annotations. The stack-only class annotation "NS_STACK_CLASS" has 406 uses through out the code base. In contrast, we only introduce 36 annotations (of 4 types) in Firefox. As MPI is based on source code level annotation and compiler instrumentation, it cannot find units within dynamic code. However, in practice, we find that unit boundaries mostly lie in static code. For example, JavaScript code can be grouped into different tabs. Thus, dynamic code can be attributed to tab units.

3.6 Related work

Many approaches have been proposed for system level provenance tracking. Detailed comparison of MPI with existing audit systems [10, 26, 40, 42, 80] can be found in §3.2. Another important approach is to monitor the internal kernel objects (e.g., the file system [30, 52, 58, 28, 50, 71, 97], or LSM objects [27, 29, 70]) to track lineages. The capabilities of these techniques are similar to those of the audit systems. Thus MPI is complementary to such systems. For example in §3.4, we showed the integration of MPI and LPM-HiFi. System wide record-and-replay techniques [79, 25, 49, 98] can also track provenance. These systems record the inputs for all programs, and replay the whole system execution when needed. Such systems require deterministic record-and-replay techniques, which are open research problems, and cause more space overhead. Whole system tainting [31, 51, 54, 24] is another method of tracking provenance. By tainting all inputs to a system and tracking their propagation, such systems can record the needed provenance data. These techniques need to deal with the granularity problem as the taint set may be explosive for a long living system objects/subjects. MPI can be applied to such systems to overcome the dependency explosion problem and enable multiple perspective inspection.

In [99], researchers propose to develop provenance aware applications. Muniswamy-Reddy *et. al.* [58] provide a library with provenance tracking APIs so that programmers can develop provenance aware applications. Such an approach relies on the programmers to intensively modify their code to leverage the APIs. In contrast, MPI aims to address the partitioning problem. Provenance tracking is through the underlying audit system.

Many works [40, 52, 66, 82] are proposed to reduce the space overhead of provenance tracking based on reachability analysis, Mandatory Access Control (MAC) policies and so on. Provenance visualization [100, 101, 102, 103, 104] and graph compression [63, 64, 65, 105, 67, 106] are also proposed to correlate events and reduce graph size to facilitate investigation. These approaches work on generated graphs to compress them for better visualization. As such, they are complementary to MPI, and can be directly applied to MPI, its provenance logs and graphs. Researchers proposed many machine learning methods [107, 108, 109, 110, 111, 112, 113] to investigate provenance data to find abnormal behaviors. We envision that the multiple perspectives provided by MPI may substantially improve their effectiveness.

4 NIC: DETECTING ADVERSARIAL SAMPLES WITH NEURAL NETWORK INVARIANT CHECKING

Deep Neural Networks (DNN) are vulnerable to adversarial samples that are generated by perturbing correctly classified inputs to cause DNN models to misbehave (e.g., misclassification). This can potentially lead to disastrous consequences especially in security-sensitive applications. Existing defense and detection techniques work well for specific attacks under various assumptions (e.g., the set of possible attacks are known beforehand). However, they are not sufficiently general to protect against a broader range of attacks. In this paper, we analyze the internals of DNN models under various attacks and identify two common exploitation channels: the provenance channel and the activation value distribution channel. We then propose a novel technique to extract DNN invariants and use them to perform runtime adversarial sample detection. Our experimental results of 11 different kinds of attacks on popular datasets including ImageNet and 13 models show that our technique can effectively detect all these attacks (over 90% accuracy) with limited false positives. We also compare it with three state-of-the-art techniques including the Local Intrinsic Dimensionality (LID) based method, denoiser based methods (i.e., MagNet and HGD), and the prediction inconsistency based approach (i.e., feature squeezing). Our experiments show promising results.

4.1 Introduction

Deep Neural Networks (DNNs) have achieved very noticeable success in many applications such as face recognition [114], self-driving cars [115], malware classification [116], and private network connection attribution [117]. However, researchers found that DNNs are vulnerable to adversarial samples [118]. Attackers can perturb a benign input (i.e., correctly classified input) so that the DNN would misclassify the perturbed input. Existing attack methods use two types of perturbation strategies: *gradient based approach* and *content based approach*. In the gradient based approaches, attackers view generating an adversarial sample as an optimization problem and conduct gradient guided search to find adversarial samples [119, 120, 121, 122, 123] (§4.2.3). In the content based approaches, attackers craft patches to inputs that are consistent with real world content of the inputs such as watermarks on images and black spots caused by dirt on camera lens to perturb the inputs [5, 124] (§4.2.3).

Real world applications, including many security critical applications, are trending to integrate DNNs as part of their systems. For example, iPhone X uses a face recognition system for authentication (unlocking phone, authenticating purchase, etc.). Many companies, e.g., Google and Uber, are developing self-driving cars that use DNNs to replace human drivers. However, the existence of adversarial examples is a fatal threat to the thriving of these applications because misbehaved DNNs would incur severe consequences such as identity theft, financial losses, and even endangering human lives. Thus detecting or defending against DNN adversarial samples is an important and urgent challenge.

There are existing works aiming to defend against or detect adversarial samples. Defense techniques try to harden DNNs using various methods such as *adversarial training* [119] and *gradient masking* [125]. The former tries to include adversarial samples in the training set so that the hardened models can recognize them. This technique is effective when the possible attacks are known beforehand. The latter aims to mask gradients so that attackers can hardly leverage them to construct adversarial samples. Recently, attackers have developed more advanced attacks against this type of defense. Some other works [126, 127, 128, 129, 130] do not aim to harden or change the models but rather detect adversarial samples during operation. For example, Ma et al. [131] proposed to use the Local Intrinsic Dimensionality (LID), a commonly used anomaly detection metric to detect adversarial samples. Xu et al. [127] proposed to examine the prediction inconsistency on an original image and its transformed version with carefully designed filters. MagNet [126] and HGD [132] propose to train encoders and decoders to remove the added noises of the adversarial samples. More detailed discussion of recent advances in defending and detecting adversarial samples is presented later (§4.2.4).

In this paper, we analyze the internals of individual DNN layers under various attacks and summarize that adversarial samples mainly exploit two attack channels: the *provenance channel* and the *activation value distribution channel*. The former means that the model is not stable so that small changes of the activation values of neurons in a layer may lead to substantial changes of the set of activated neurons in the next layer, which eventually leads to misclassification. The latter means that while the provenance changes slightly, the activation values of a layer may be substantially different from those in the presence of benign inputs. We then propose a method NIC (Neural-network Invariant Checking) that extracts two kinds of invariants, the *value invariants* (VI) to guard the value channel and the *provenance invariants* (PI) to guard the provenance channel. Due to the uncertain nature of DNNs, neural network invariants are essentially probability distributions denoted by models. Constructing DNN value invariants is to train a set of models for individual layers to describe the activation value distributions of the layers from the benign inputs. Constructing provenance invariants is to train a set of models, each describing the distribution of the causality between the activated neurons across layers (i.e., how a set of activated neurons in a layer lead to a set of activated neurons in the next layer). At runtime, given a test input which may be adversarial, we run it through all the invariant models which provide independent predictions about whether the input induces states that violate the invariant distributions. The final result is a joint decision based on all these predictions.

We develop a prototype and evaluate it on 11 different attacks (including both gradient based attacks and content based attacks), 13 models, and various datasets including MNIST [133], CIFAR-10 [134], ImageNet [135] and LFW [136]. The experimental results show that we can effectively detect all the attacks considered on all the datasets and models, with consistently over 90% detection accuracy (many having 100% accuracy) and an acceptable false positive rate (less than 4% for most cases and less than 15% for ImageNet, a very large dataset). We also compare our approach with state-of-the-arts, LID [131], MagNet [126]/HGD [132], and feature squeezing [127]. The results show that our proposed method can achieve consistently high detection accuracy with few false positives, while the other three detectors can achieve very good results on some attacks but cannot consistently detect all different types of attacks. We make the following contributions:

- We analyze the internals of DNNs under various attacks and identify the two main exploit channels: the provenance channel and the value channel.
- We propose a novel invariant based technique to detect adversarial samples, including novel methods of extracting value invariants and provenance invariants.
- To the best of our knowledge, we are the first one to explore detecting content based adversarial samples, which have significant different attack patterns.
- We also evaluate the robustness of our technique by designing a strong white-box adaptive attack, in which the attacker has full knowledge of our technique. Compared to existing techniques, NIC significantly increases the difficulty of constructing adversarial samples, with 1.4x larger mean L_2 distortion and 1200x longer construction time for MNIST.
- We implement a prototype [137]. The evaluation results show that our method can detect the 11 types of attacks considered with over 90% accuracy (many 100%) and limited false positives. Comparing with three state-of-the-art detectors, NIC shows consistently good results for all these attacks whereas other techniques perform well on a subset.

4.2 Background and Related Work

4.2.1 Neural Networks

A DNN can be represented as a function $F : X \to Y$, where X denotes the input space and Y the output space. It consists of several connected layers, and links between layers are weighted by a set of matrices, w_F. The training phase of a DNN is to identify the numerical values in w_F. Data engineers provide a large set of known input-output pairs (x_i, y_i) and define a loss function (or cost function) $J(F(x), y^*)$ representing the differences between a predicted result $F(x)$ and the corresponding true label y^*. The training phase is hence to minimize the loss function by updating the parameters using the *backpropagation* technique [133]. The training phase is governed by *hyper-parameters* such as the learning rate. In our settings (detecting adversarial inputs), models are given and thus the hyper-parameters are fixed. In the testing phase, a trained model is provided with unseen inputs X_t, and for each input $x_t \in X_t$, the classification model assigns its label to be $C(x_t) = \text{argmax}_i F_i(x_t)$, where $F_i(x_t)$ represents the probability of x_t being recognized as class i. The classification result is correct if the predicated result $C(x_t)$ is the same as the correct label $C^*(x_t)$.

In this paper, we focus on m-class DNN classification models. For such models, a model output is a vector with m elements, and each of them represents the probability of the input being in a specific class. We use notations from previous papers [138, 122] to define a neural network:

$$y = F(x) = \text{softmax}(Z(x))$$

, where $x \in X$ and $y \in Y$. In such models, $Z(x)$ is known as the $logits$ and the softmax function normalizes the values such that for an output vector $y \in \mathbb{R}^m$, $y_i \in (0, 1)$ and $\sum y_i = 1$, where y_i represents the probability of the input being class i. The final output will be the class with the highest probability.

4.2.2 Adversarial Samples

DNNs are vulnerable to adversarial samples [118]. Intuitively, an adversarial sample is an input that is very similar to one of the correctly classified inputs (or benign inputs), but the machine learning model produces different prediction outputs for these two inputs. Existing works try to generate adversarial samples in two different kinds of approaches: *gradient based approach* and *content based approach*.

Gradient based approach. Formally, given a correctly classified input $x \in X$ with class $C(x)$ (notice that $C^*(x) = C(x)$), we call x' an adversarial sample if

$$x' \in X \wedge C(x') \neq C(x) \wedge \Delta(x, x') \leq \epsilon$$

, where $\Delta(\cdot)$ denotes a distance function measuring the similarity of the two inputs and ϵ is the adversarial strength threshold which limits the permissible transformations. The equation means that x' is a valid input ($x' \in X$) which is very similar to x ($\Delta(x, x') \leq \epsilon$), but the model gives a different prediction output ($C(x') \neq C(x)$). Such adversarial samples are usually referred to as *untargeted adversarial samples*. Another type of attack is known as the *targeted attack*. In such attacks, the output of the adversarial sample $C(x')$ can be a particularly wanted class. For a given correctly classified input $x \in X$ ($C(x) = C^*(x)$) and a target class $t \neq C(x)$, a targeted adversarial sample x' satisfies

$$x' \in X \wedge C(x) \neq C(x') \wedge C(x') = t \wedge \Delta(x, x') \leq \epsilon.$$

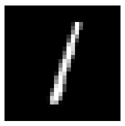
Thus, finding/generating adversarial samples can be viewed as optimization problems:

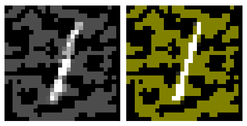
$$\textbf{min} \;\; \Delta(x, x') \;\; \textbf{s.t.} \;\; C(x') = t \wedge x' \in X \;\; \textbf{(Targeted)}$$

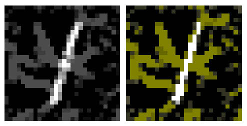
$$\textbf{min} \;\; \Delta(x, x') \;\; \textbf{s.t.} \;\; C(x') \neq C(x) \wedge x' \in X \;\; \textbf{(Untargeted)}.$$

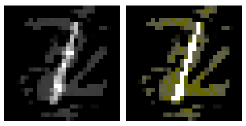
Recent and popular adversarial attacks usually use the L_p-norm distance metric as the $\Delta(\cdot)$ function. Three p values are commonly used in adversarial attacks, leading to L_0, L_2 and L_∞ attacks. Previous works [122, 127, 126] have detailed comparison between them. In summary, (1) L_0 *distance limits the number of dimensions an adversarial sample can alter, but does not limit the degree of change for each dimension*. As a result, such adversarial samples tend to have fewer perturbation regions, but the change of each region is significant. (2) L_∞ *distance limits the maximum change in any dimension, but does not limit the number of altered dimensions*. That is to say, the generated adversarial samples tend to have a large number of altered regions, but the change of each region is not substantial. (3) L_2 *distance (Euclidean distance) bounds the accumulated change to achieve a balance of the number of altered dimensions and the degree of change in each dimension*.

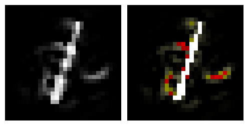
Content based approach. Another way of generating adversarial samples is to leverage the input content and provide semantically consistent/restrained perturbations on an original input to simulate real world scenarios. For example, images may have black spots because of dirt on camera lens and attackers can intentionally add multiple black spots to an image

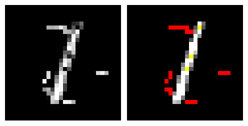

(a) Original

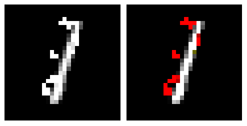

(b) FGSM (UT, L_∞)

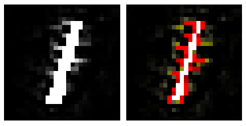

(c) BIM (UT, L_∞)

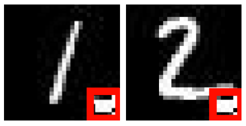

(d) CW$_\infty$ (T, L_∞)

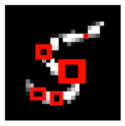

(e) CW$_2$ (T, L_2)

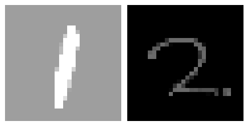

(f) CW$_0$ (T, L_0)


(g) JSMA (T, L_0)


(h) DeepFool (UT, L_2)


(i) Trojan (T, CB)


(j) Dirt(UT, CB)


(k) Lighting (UT, CB)

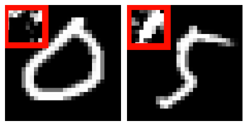

(l) Rectangle Patching (UT, CB)

Figure 4.1.: Adversarial images of different attacks on MNIST. *T: targeted. UT: untargeted. L_0, L_2 and L_∞ stand for different distance metrics (gradient based attacks) and CB stands for content based attacks. Captions are the attack names. For gradient based attacks, Figure 4.1(a) is the original image. For each attack, the first image is adversarial and the second image highlights the perturbations. For content based attacks, red boxes highlight the added contents.*

to simulate this. Under such assumptions, the attackers do not need to calculate the $\Delta(\cdot)$ distance value, and avoid optimizing the distance. As such, many such adversarial samples are human recognizable but have large $\Delta(\cdot)$ distances. There are a number of such recent attacks [124, 5, 139, 140].

4.2.3 Existing Attacks

In this section, we discuss 11 existing representative attacks for DNN models, including both gradient based attacks and content based attacks. Note that while there are adversarial attacks for machine learning models in general [141], we focus on adversarial samples on DNN models in this paper. In Figure 4.1, we show sample images of various attacks using the MNIST dataset [133] to facilitate intuitive understanding. For gradient based attacks, we show the generated adversarial samples as well as the perturbations made (i.e., the colored regions). We use yellow to highlight small changes and red to highlight large changes. The lighter the color, the less the perturbation. For the content based attacks, the boxed regions represent the newly added elements to the original image.

Fast Gradient Sign Method (FGSM): Goodfellow et al. proposed the *fast gradient sign method* to efficiently find adversarial samples [119]. Existing DNNs usually use piece-wise linear activation functions such as ReLU. These functions are easy to optimize, making it feasible to train a DNN. For such models, any change of the input is propagated to later hidden layers till the output layer. As such, errors on inputs can accumulate and lead to misclassification. The FGSM attack is based on this analysis and assumes the same attack strength at all dimensions and hence an L_∞ attack. Also, it is an untargeted attack. Formally, an adversarial sample is generated by using the following equation:

$$x' = x - \epsilon \cdot \text{sign}(\nabla_x J(F(x)))$$

, where ∇ represents the gradient and $J(\cdot)$ is the cost function used to train the model. Essentially, it changes all pixels simultaneously at a fix scale along the gradient direction. It can quickly generate adversarial samples as it only needs the gradient sign which can be calculated by backpropagation. An adversarial sample for the attack is shown in Figure 4.1(b). Observe that it makes small changes to many pixels.

Basic Iterative Method (BIM): The *basic iterative method* [120] is an improved version of the FGSM attack. It generates adversarial samples with many iterations, and in each iteration it updates the generated sample with the following equation:

$$x'_i = x'_{i-1} - \text{clip}_\epsilon(\alpha \cdot \text{sign}(\nabla_x J(F(x'_{i-1}))))$$

, where x'_0 represents the original correctly classified input. In FGSM, a single step is taken along the direction of the gradient sign. BIM takes multiple steps, and in each iteration, it performs clipping to ensure the results are in the L_∞ ϵ-neighborhood of the original input. As such, it is also known as the Iterative FGSM (IFGSM). In practice, it can generate superior results than FGSM (Figure 4.1(c)).

DeepFool: Moosavi et al. designed the DeepFool attack [123] by starting from the assumption that models are fully linear. Under this assumption, there is a polyhedron that can separate individual classes. Composing adversarial samples becomes a simpler problem, because the boundaries of a class are linear planes and the whole region (for this class) is a polyhedron. The DeepFool attack searches for adversarial samples with minimal perturbations within a specific region by using the L_2 distance. The authors also adopt methods from geometry to guide the search. For cases where the model is not fully linear, the attack tries to derive an approximated polyhedron by leveraging an iterative linearization procedure, and it terminates the process when a true adversarial sample is found. In Figure 4.1(h), observe that the changes are in the vicinity of the original object. It is an untargeted attack.

Jacobian-based Saliency Map Attack (JSMA): Papernot et al. proposed the *Jacobian-based Saliency Map Attack* [121] which optimizes L_0 distance using a greedy algorithm. The attack is an iterative process, and in each iteration, it picks the most important pixel to modify, and terminates when an adversarial sample is found. To measure the benefits of changing pixels, the attack introduces the concept of *saliency map*, which calculates how much each output class label will be affected when individual pixels are modified by using the Jacobian matrix. The attack then changes the pixel with the largest benefits (getting the target label). As an L_0 attack, JSMA modifies a very small number of pixels as shown in Figure 4.1(g), but the perturbation is more substantial than L_∞ attacks such as FGSM or BIM. In the mean time, it has very high computation cost (due to the need of computing the Jacobian matrix), making it impractical for high dimension inputs, e.g., images from the ImageNet dataset.

Carlini and Wagner Attack (CW): Carlini and Wagner proposed three different gradient based attacks using different L_p norms, namely, L_2, L_∞ and L_0. We refer to them as CW$_2$, CW$_\infty$ and CW$_0$ attacks, respectively. In the CW$_2$ attack, Carlini and Wagner are also solving the (targeted) optimization problem using the L_2-norm as the distance measurement. However, it features a few new designs to make it very effective. First of all, they use the logits $Z(\cdot)$ instead of the final prediction $F(\cdot)$ in the loss function. This design was shown to be critical for the robustness of the attack against defensive distillation methods [122]. Secondly, they introduced a variable α (also known as the optimal constant) to control the confidence of the adversarial samples so that they can get a better trade-off between the prediction and distance values. Another key design in this attack is that it maps the target variable to the *argtanh* space and then solves the optimization problem. This design enables the attack to use modern optimization solvers such as the Adam solver [142]. These techniques allow the CW$_2$ attack to very efficiently generate superior adversarial samples as shown in Figure 4.1(e). Comparing with the other L_2 attack DeepFool, CW$_2$ perturbs fewer regions with smaller changes on most pixels. Most adaptive attacks are based on similar approaches (see §4.4.5 and §4.5).

DeepXplore Attack: In [124], Pei et al. observed that not all neurons are activated during testing, and these (inactivated) neurons can contain unexpected behaviors. Thus they proposed a systematic way of generating adversarial samples to activate all neurons in the network so that it can uncover unexpected behaviors in DNNs. The proposed attacks are leveraging semantic information (known as domain-specific constraints) to perform the perturbation. Specifically for the image type of inputs, they propose three methods: the *Lighting attack, Rectangle Patching attack* and the *Dirt attack*.

In the *Lighting attack*, the attackers change all the pixels' values by the same amount to make them darker or brighter. This action simulates the images taken under different lighting conditions (Figure 4.1(k)). In the *Rectangle Patching attack*, a single small rectangle patch is placed on the original image to simulate that the camera lens are occluded. Notice that in this attack, the content in the rectangle region are controlled by the adversary, and can be different for different images (Figure 4.1(l)). The *Dirt attack* adds multiple tiny black rectangles to the image for simulating the effects of dirt on camera lens (Figure 4.1(j)).

Trojan Attack and Adversarial Patch Attack: Liu et al. [5] proposed a systematic way to perform Trojan attacks on DNNs. The attacker first reverse-engineers the DNN to generate a well-crafted image patch (known as the *Trojan trigger*), and then retrains a limited number of neurons with benign inputs and trojaned inputs (i.e., benign inputs patched with the Trojan trigger, marked with the target output label) to construct the trojaned DNN. The trojaned DNN will perform the trojaned behavior (i.e., misclassifying images to the target label) for trojaned inputs, while retaining a comparable (sometimes even better) performance on benign inputs. This attack essentially simulates many real world scenarios such as images patched with watermarks or stamps.

Brown et al. [139] presented a method to create *adversarial patches* (conceptually similar to Trojan triggers) which achieves similar effects with the Trojan attack mentioned above. The difference is that it does not require retraining. From the perspective of detecting adversarial samples, there is no difference between this attack and the Trojan attack, as we are not aware if a model is trojaned and what the trigger is. Thus we consider them as the same type of attack.

4.2.4 Existing Defense and Detection

Existing Defense. Defense techniques try to harden the NN models to prevent adversarial sample attacks [143, 144]. Papernot et al. [145] comprehensively studied existing defense mechanisms, and grouped them into two broad categories: *adversarial training* and *gradient masking*. Goodfellow et al. introduced the idea of adversarial training [119]. It extends the training dataset to include adversarial samples with ground truth labels. There are also similar ideas that try to integrate existing adversarial sample generation methods into the training process so that the trained model can easily defend such attacks [146]. However, it requires prior knowledge of the possible attacks and hence may not be able to deal with new attacks.

The basic idea of gradient masking is to enhance the training process by training the model with small (e.g., close to 0) gradients so that the model will not be sensitive to small changes in the input [125]. However, experiments showed that it may cause accuracy degradation on benign inputs. Papernot et al. introduced *defensive distillation* to harden DNN models [138]. It works by smoothing the prediction results from an existing DNN to train the model and replacing the last softmax layer with a revised softmax-like function to hide gradient information from attackers. However, it was reported that such models can be broken by advanced attacks [122, 147, 148]. Athalye et al. [149] also shows that hardening or obfuscating gradients can be circumvented by gradient approximation. Papernot et al. [145] concluded that controlling gradient information in training has limited effects in defending adversarial attacks due to the transferability of adversarial samples, which means that adversarial samples generated from a model can be used to attack a different model.

Existing Detection. Adversarial sample detection is to *determine if a specific input is adversarial*. Many previous researches have studied to build detection systems [150, 151, 152, 129, 130, 153, 154, 155, 156]. We classify existing state-of-the-art works into three major categories.

Metric based approaches. Researchers have proposed to perform statistical measurement of the inputs (and activation values) to detect adversarial samples. Feinman et al. [128] proposed to use the *Kernel Density estimation* (KD) and *Bayesian Uncertainty* (BU) to identify adversarial subspace to separate benign inputs and adversarial samples. Carlini and Wagner [157] showed this method can be bypassed but also commented this method to be one of the promising directions. Safetynet [158] quantizes activations in late-stage ReLU neurons and use SVM-RBF to find the patterns that distinguish adversarial and natural samples. This approach is only tested on FGSM, BIM and DeepFool and requires adversarial samples to find the patterns. Inspired by ideas from the anomaly detection community, Ma et al. [131] recently proposed to use a measurement called *Local Intrinsic Dimensionality* (LID). For a given sample input, this method estimates a LID value which assesses its space-filling capability of the region surrounding the sample by calculating the distance distribution of the sample and a number of neighbors for individual layers. The authors empirically show that adversarial samples tend to have large LID values. Their results demonstrate that LID outperforms BU and KD in adversarial sample detection and currently represents the state-of-the-art for this kind of detectors. A key challenge of these techniques is how to define a high-quality statistical metric which can clearly tell the difference between clean samples and adversarial samples. Lu et al. [159] have shown that LID is sensitive to the confidence parameters deployed by an attack and vulnerable to transferred adversarial samples. Our evaluation in §4.5 also shows that although LID is capable of detecting many attacks, it does not perform well for a number of others.

Denoisers. Another way of detecting adversarial samples is to perform a pre-process (denoiser) step for each input [126, 125, 160]. These approaches train a model or denoiser (encoders and decoders) to filter the images so that it can highlight or emphasize the main component in the image. Doing so, it can remove the *noises* of an image including those added by attackers and thus correct the classification result. For example, MagNet [126] uses detectors and reformers (trained auto-encoders and auto-decoders) to detect adversarial samples. The method has been shown to work well on many attacks. But it is only tested on small datasets like MNIST and CIFAR-10. Liao et al. [132] argued that these pixel guided denoisers (e.g., MagNet) do not scale to large images such as those in the ImageNet dataset. Thus they proposed a *high-level representation guided denoiser* (HGD) for large images and achieved state-of-the-art results on ImageNet. A limitation of this kind of techniques is that denoisers are essentially trained neural networks. Training them remains a highly challenging problem (e.g., very time-consuming). Also, they are end-to-end differentiable, making them potentially vulnerable to white-box attacks [127, 161]. Moreover, the quality of denoisers depends on the training dataset and collecting a high-quality training set is also very demanding.

Prediction inconsistency based approaches. Many other works are based on prediction inconsistency [162, 163, 164]. Tao et al. [164] propose to detect adversarial examples by measuring the inconsistency between original neural network and neural network enhanced with human perceptible attributes. However, this approach requires human defined attributes for detection. The state-of-the-art detection technique *Feature Squeezing* [127] achieves very high detection rates for various attacks. The authors pointed out that DNNs have unnecessarily large input feature space, which allows an adversary to produce adversarial

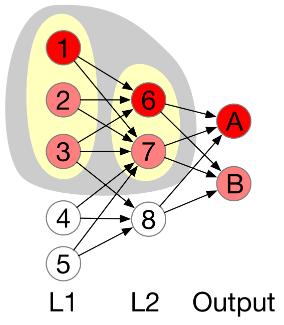


Figure 4.2.: Input A

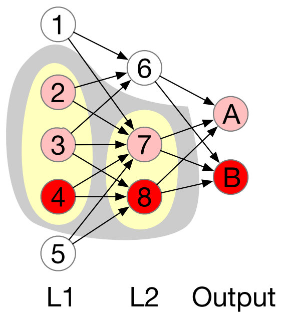


Figure 4.3.: Input B

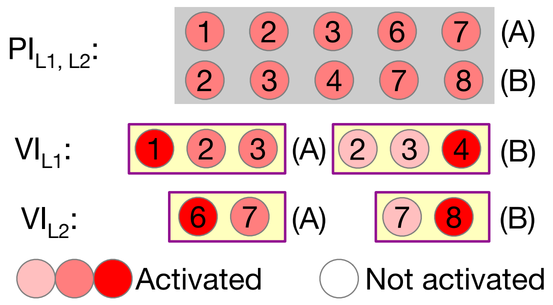


Figure 4.4.: PI and VI

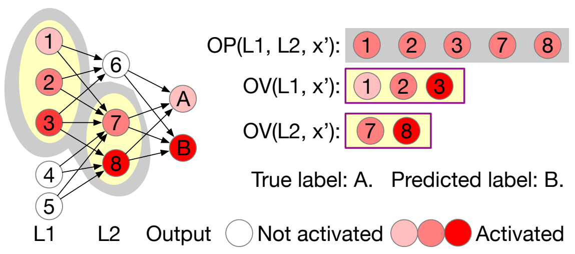


Figure 4.5.: Input that violates PI

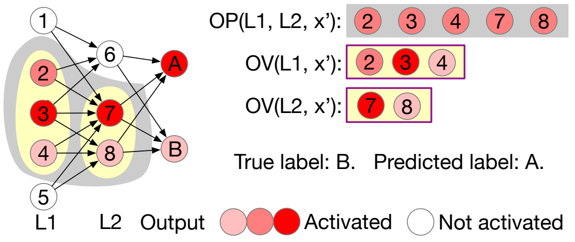


Figure 4.6.: Input that violates VI

samples. They hence proposed to use *squeezing* techniques (i.e., reducing the color depth of images and smoothing the images) to generate a few *squeezed* images based on the seed input. Feature squeezing essentially limits the degree of freedom available to an adversary. Then the DNN model takes all the squeezed images and the original seed image, and makes predictions individually. Adversarial samples are detected by measuring the distance between the prediction vectors of the original seed input and each of the squeezed images. If one of the distances exceeds a threshold, the seed input is considered malicious. However, according to [127] and our experiments in §4.5, the technique does not perform well on FGSM, BIM and some content based attacks on CIFAR and ImageNet. This is because its performance highly depends on the quality of the designed squeezers, which remains an open research challenge.

Furthermore, most existing works focus on detecting gradient based attacks. It is unclear if they can detect content based attacks such as DeepXplore and Trojaning.

4.3 Observations

In this section, we first explain existing attacks are essentially exploiting two channels (§4.3.1). Then we discuss that guarding these channels can be achieved by invariant checking (§4.3.2). Finally, we introduce our design details (§4.4).

4.3.1 Observations about DNN Attack Channels

We study the internals of individual layers of DNNs under the various kinds of attacks discussed in §4.2.3. We identify they mostly exploit two channels in the models: the *provenance channel* and the *activation value distribution channel*.

Exploiting the Provenance Channel. In our discussion, we say a neuron is activated if its activation function (e.g., ReLU) returns a non-zero value. A (hidden) layer of a DNN can be considered as a function that takes the activated neurons from the previous layer, performs matrix multiplication with the weight values in the layer, and then applies an activation function to determine what neurons are activated in the layer. *We consider the relation that the activated neurons in the preceding layer lead to the activated neurons in a given layer the provenance of the layer*.

Many attacks entail changing provenance. Intuitively, given an adversarial sample x' that is similar to x of class A, if the goal is to make the model to misclassify x' to B, the provenance of x' is often different from the typical provenance of A or B. This is usually due to the internal instability of the model such that small changes lead to a different set of activated neurons.

Figure 4.2 and Figure 4.3 present normal operations of a DNN. For simplicity, we only show three layers, two hidden layers $L1$ and $L2$ and the output layer (that classifies A and B). Figure 4.2 shows an input belonging to A and Figure 4.3 to B. The colored nodes represent activated neurons, and white nodes represent the inactivated neurons. Darker colors denote larger values. Observe that some neurons are only activated by inputs belonging to a certain class [118, 165], such as nodes 1 and 6 for class A and nodes 4 and 8 for class B. There are also a lot of neurons that are activated by both kinds of inputs such as neurons 2, 3, and

7. The gray regions across the first two layers denote the provenance. Observe that for A, neurons 1, 2, and 3 in $L1$ lead to 6 and 7 in $L2$; for B, neurons 2, 3, 4 lead to 7 and 8. Note that there are many possible provenance relations even for the same class. Our example is just for illustration.

Figure 4.5 shows the operation of an adversarial sample which is very similar to A although it is misclassified as B. Observe neurons 1, 2, 3 in $L1$ lead to 7 and 8 in $L2$ and eventually the misclassified label B. Namely, the provenance is different from a typical B provenance as in Figure 4.3. The root cause is that the model is very sensitive to small changes of neuron 1.

Trojan attack [5] is an example attack that exploits the provenance channel. The objective of the attack is that a trojaned model must not have performance degradation for benign inputs and the model should misclassify any input with a Trojan trigger to the target class. As such, it changes weight values at specific layers so that the provenance has substantial deviation in the presence of the Trojan trigger. A case study can be found in §4.5.7. L_0 attacks also tend to exploit the provenance channel. These attacks introduce substantial changes to a limited number of elements in an input (see §4.2.3). As a result, different neurons (features) got triggered but they are not typical neurons (features) for the target class.

Exploiting Activation Value Distribution. Some attacks may not exploit the provenance channel. In other words, the provenance of an adversarial sample is no different from that of a benign input. In such cases, to cause misclassification, the activation value distribution of the activated neurons has to be different from that of a benign input. The exploitation can be illustrated by Figure 4.6, in which the adversarial input has the same provenance as a benign B input (Figure 4.3). However, the substantially different activation values in $L2$ lead to the output value of class A larger than B.

L_∞ attacks (e.g., FGSM attack in Figure 4.1(b)) limit the change degree made to individual dimensions but do not limit the number of changed dimensions. For example, they tend to alter a lot of pixels, but the change of each pixel is small as shown in Figure 4.1. Thus they lead to substantially different value distributions in the first few layers. Note that in the first few layers, many neurons tend to be activated for both benign and adversarial inputs so that their provenance relations do not differ much.

Some adversarial samples exploit both channels such as L_2 attacks, which do not limit the number of perturbations or the degree of each perturbation, but rather constrain the total change using the Euclidean distance. Depending on the search procedure of an L_2 attack, it may exploit either value or provenance. Some attacks (e.g., DeepXplore) even exploit neurons that do not represent unique features for any class, leading to unique value distribution.

4.3.2 Detection As Invariant Checking

The aforementioned DNN exploit channels are analogous to the ways that traditional attacks exploit software defects. Exploiting the provenance channel shares similarity to control flow hijacking [166], in which a regular execution path is hijacked and redirected to malicious payload. Exploiting the activation value distribution is analogous to state corruption [167], which represents a prominent kind of defective software behaviors that cause incorrect outputs but not necessarily crashes.

A classic approach to detecting software attacks is *invariant checking*. Specifically, to detect control flow hijacking, *control flow integrity* (CFI) techniques check control flow invariants (i.e., the control flow predecessor of an instruction must be one of a predetermined set of instructions). To detect state corruption, programmers explicitly add assertions to

```
01: def fib(n):
02:   assert(n>=0)
03:   assert(from line 6 or 10)
04:   if n == 0 or n == 1:
05:     return n
06:   return fib(n-1)+fib(n-2)
07:
08: def main():
09:   x = input('Input a number:')
10:   print fib(x)
```

```
01: def dnn(x, M): # input:x, model:M
02:   L = M.layers
03:   for i in range(1, L.size):
04:     L[i] = L.w[i]*L[i-1]+L.b[i]
05:     for j in range(1, L[i].size):
06:       if L[i][j] < 0:
07:         L[i][j] = 0
08:     assert(PI, L[i], L[i-1])
09:     assert(VI, L[i])
10: return M
```

Figure 4.7.: Program invariant and DNN invariant

check pre- and post-conditions of individual program statements. Figure 4.7 shows an example of program invariants. It is a program that computes Fibonacci number. Observe that at line 2, the programmer uses an assert to ensure n cannot be negative. Line 3 ensures that there are only two possible statements that can invoke the `fib()` function, namely, statements 6 and 10. Note that in many cases, such invariants are merely approximation due to various difficulties in program analysis (e.g., dealing with variable aliasing).

DNN computation is essentially a process of taking a model input and producing the corresponding classification output, which has a program structure like the one on the right of Figure 4.7. It iterates through the individual layers (lines 3 to 9). For each layer, it computes the activation values (line 4) from those of the previous layer, the weight of this layer (i.e., `L.w[i]`) and a bias factor `L.b[i]`. After that, an activation function (we use ReLU as an example, line 6-7 in Figure 4.7) is applied and a conditional statement is used to determine what neurons are activated in this layer (lines 5-7). Observe that exploiting the provenance channel is essentially altering the branch outcome of the conditional (line 6) while exploiting activation values is changing the distribution of the computed values at line 4.

Therefore, *our overarching idea is to check invariant violations during DNN computation* (lines 8 and 9 on the right). Our invariants are approximate as they cannot be precisely specified due to the uninterpretability of DNN. They are essentially probabilistic distributions constructed through learning. Different from other learning based techniques that require both benign and adversarial samples, *our invariants are only learned from benign samples and their computation*, and hence can provide general protection against a wide spectrum of attacks.

Intuitively, we define *the distribution of the activation values of two consecutive layers as the provenance invariant (PI) of the layers*. Later in §4.4, we show that learning such a distribution is too expensive so that we learn a reduced model. We further define *the activation value distribution of a given layer as its value invariant (VI)*. Figure 4.4 shows the PIs and VIs for the sample DNN (derived from benign inputs). Observe that the PI of $L1$ and $L2$ is denoted as two vectors, representing neurons 1, 2, 3 leading to neurons

6, 7 (for class A), and neurons 2, 3, 4 leading to 7 and 8 (for class B). In Figure 4.5, the *observed provenance* (OP) for the adversarial input x', which is misclassified as B, violates the provenance invariant (i.e., does not fit the distribution). In Figure 4.6, the *observed values* (OV) for layer $L1$ is substantially different from VI_{L1} (denoted by their different colors), even though the OP is consistent with PI. Note that our VI technique is different from estimating LID [131]. The estimated LID measures the distances between a sample and a number of its neighbors whereas VI trains models to explicitly describe activation value distributions. More importantly, as shown in the evaluation section, our PI and VI methods are complementary, together constituting an effective and robust solution outperforming LID.

4.4 System Design

4.4.1 Overview

Figure 4.8 shows the overview of our approach. It is a training based approach, and we do not make any modification to the original trained model so that it does not affect the performance of the original model. We only use the benign inputs as our training data so that our technique is not specific to a certain attack. In step Ⓐ (Figure 4.8), we collect activation values in each layer for each training input. We then train a distribution for each layer for all benign inputs. These distributions (e.g., VI_{L1} and VI_{L2}) denote the value invariants.

Ideally for PIs, we would train on any two consecutive layers to construct distributions of a concatenation of the activation values of the layers. However, a hidden layer may have many neurons, (e.g., 802,816 in one of our evaluated models). Such a distribution is often of high dimension (e.g., 2*802,816), very sparse and hence has limited predictive precision. Thus we train a reduced model that is of lower dimension and denser. Particularly, in Ⓑ, for each layer l (e.g., $L1$, $L2$), we create a derived model by taking a sub-model from the input layer to l and appending a new softmax layer with the same output labels as the original model. Observe that $L1$'s derived model has the input layer, $L1$ of the original model and a new softmax layer (dashed arrows). Then we freeze the sub-model weights, and re-train the

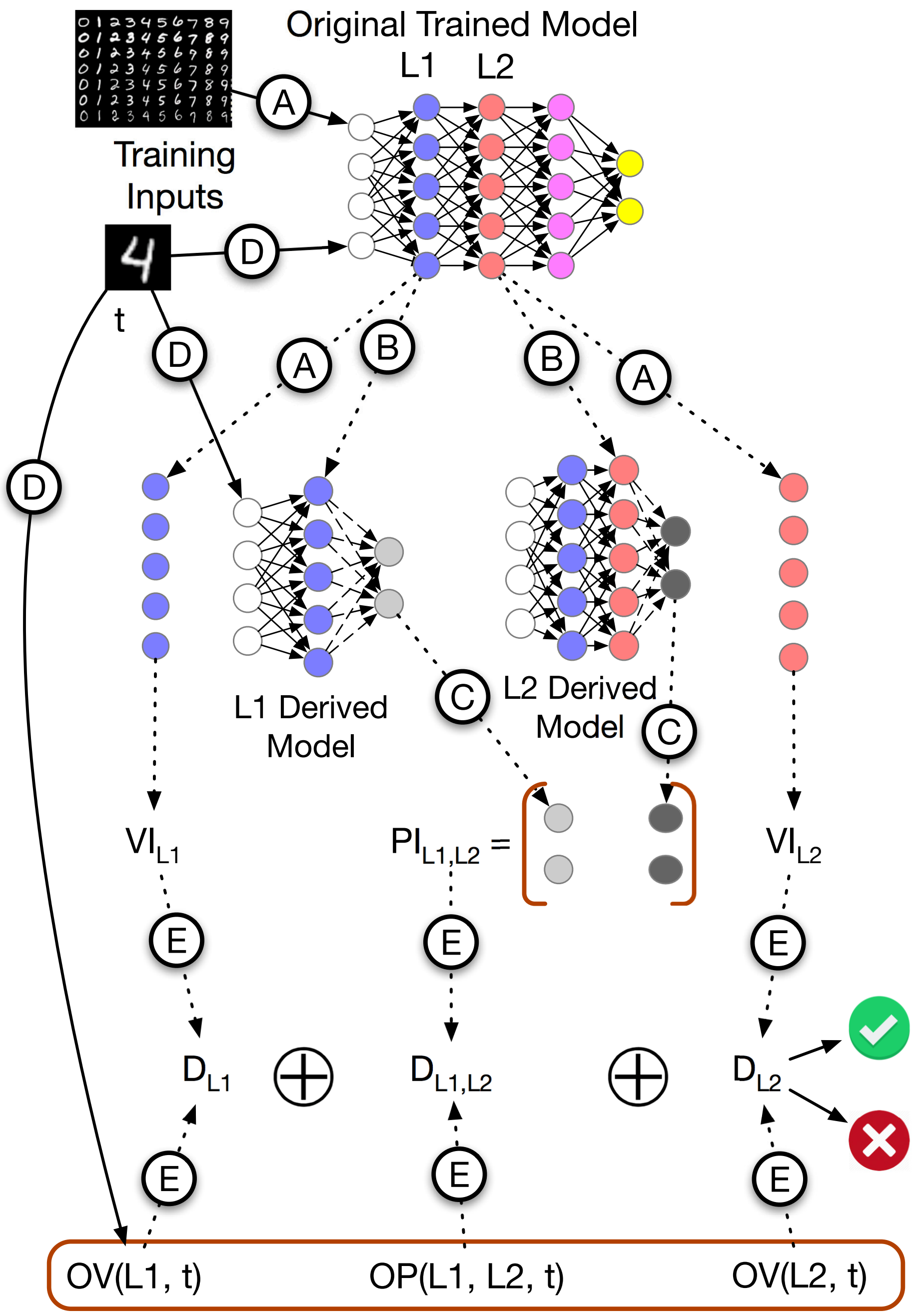


Figure 4.8.: Overview of NIC

softmax layer in the derived model. As it only trains one layer, it usually takes very limited amount of time to train a derived model. Intuitively, a derived model of layer l is to predict the output class label based on the features extracted at l. Hence, a derived model of an earlier layer (e.g., $L1$) uses more primitive features while a derived model of a later layer (e.g., $L2$) uses more abstract features.

In step Ⓒ, we run each benign training input through all derived models, we collect the final outputs of these models (i.e., the output probability values for individual classes). For each pair of consecutive layers, we train a distribution for classification results of their derived models. The trained distribution is the PI for these two layers. Essentially, we leverage the softmax functions in derived models to reduce the dimensions of the PI models. Intuitively, we are computing an approximate notion of the provenance, namely, *how the prediction results may change from using features in a specific layer to using features in the next layer*.

For example, assume in a model that predicts a bird, bat, or a dog, an early layer extracts low level features such as beak, wing, tooth, feather, fur, tail, and claws whereas its next layer extracts more abstract features such as four-leg and two-wing. The derived model of the early layer would have a softmax that introduces strong connections from beak, feather, claws to bird; from wing, tooth, fur to bat; and from tooth, fur, claws, tail to dog. The derived model of the next layer associates four-leg to dog and two-wing to both bird and bat. The PI model of the two layers hence would associate the prediction results of the two respective layers such as $\langle bat, bat \rangle$ (meaning the derived models of both layers predict bat), $\langle bat, bird \rangle$, and $\langle dog, dog \rangle$. Note that while the early layer may classify an input as bat and then the next layer classifies it as bird, the PI model tells us that it is unlikely the early layer says dog whereas the later layer says bird. Essentially, the PI model provides a way of summarizing the correlations between the features across the two layers. Ideally, we would like to train the PI models to associate the primitive features to the corresponding abstract features (e.g., beak, feather to two-wing). However, such models would have input space of very high dimensions, much higher than our current design. Our results in §4.5 indicate our design is very effective.

At runtime (step Ⓓ), for each test input t (e.g., the image of 4 in Figure 4.8), besides running the input through the original model, we also run it through all the derived models. We collect the activation values of each layer in the original model as the *observed values* (OV) (e.g., $OV(L_1, t)$ on the bottom of Figure 4.8) and the classification results of derived models of consecutive layers (in pairs) as the *observed provenance* (OP) (e.g., $OP(L_1, L_2, t)$). Then (step Ⓔ), we compute the probabilities D that OVs and OPs fit the distributions of the corresponding VIs and PIs. All these D values are aggregated to make a joint prediction if t is adversarial.

In the remainder of the section, we discuss more details of individual components.

4.4.2 Extracting DNN Invariants

We extract two kinds of invariants: provenance invariants (PI) and activation value invariants (VI) from the internals of the DNN on the benign training inputs. These invariants are probabilistic, represented as distributions.

Recall that each layer of a DNN model is a function. We use f_k to represent the k-th layer, and $f_k = \sigma(x_k \cdot w_k^T + b_k)$ where σ is the activation function, w_k is the model weight metrics, b_k is the bias and x_k is the input to this layer. Using these notations, a DNN with $n+1$ layers can be written as:

$$F = \text{softmax} \circ f_n \circ \ldots \circ f_1.$$

The output of layer l is denoted as $f_l(x_l)$. The value invariant of layer l is a function that describes the distribution of the activation values at l for benign inputs. Hence, $0 \leq VI_l(x) \leq 1$ predicts if a model input x is benign based on the activation values at layer l . Hence, learning the weight w of the VI_l model is to solve the following optimization problem:

$$\textbf{min} \sum_{x \in X_B} J(f_l \circ f_{l-1} \circ \ldots \circ f_2 \circ f_1(x) \cdot w^T - 1)$$

, where X_B represents the benign inputs and $J(\cdot)$ an error cost function. Intuitively, we aim to maximize the chance that $VI_l(x)$ predicts 1 for a benign input x.

As mentioned in §4.4, to compute PIs, we leverage derived models to reduce learning complexity and improve prediction accuracy. A derived model is constructed by taking the first few layers of the original model and then appending them with a new softmax layer. Thus the derived model D_l for layer l is the softmax layer, and D_l can be defined as follows:

$$D_l = \text{softmax} \circ f_l \circ f_{l-1} \circ \ldots \circ f_2 \circ f_1.$$

$PI_{l,l+1}(x)$ predicts the probability of x being benign based on the classification outputs of the derived models of l and $l+1$ layers. Thus, training $PI_{l,l+1}$ (i.e., deriving its weights w) is an optimization problem as follows.

$$\mathbf{min} \sum_{x \in X_B} J(\mathbf{concat}(D_l(x), D_{l+1}(x)) \cdot w^T - 1)$$

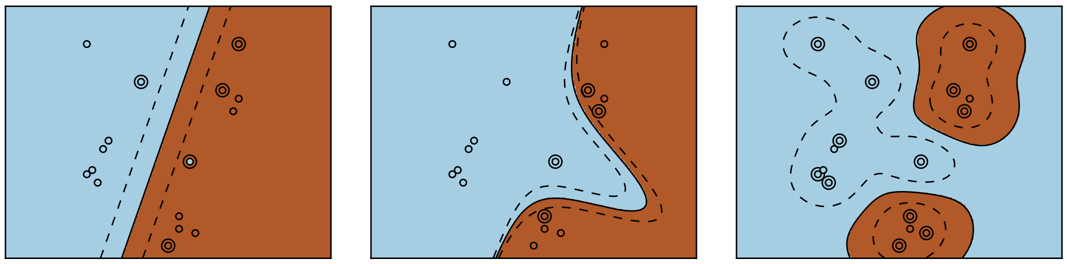
, where **concat()** concatenates two vectors to a larger one.

4.4.3 Training Invariant Models

An important feature of our technique is that the invariant models (i.e., VIs and PIs) are trained from only benign inputs, which makes our technique a general detection approach different from existing training based approaches (see §4.2.4) that require adversarial samples in training and hence prior knowledge about the attack(s).

We model the training problem without adversarial samples as a *One-Class Classification* (OCC) problem. In OCC, most (or even all) training samples are positive (i.e., benign inputs in our context), while there will be all types of inputs (e.g., adversarial samples from various attacks in our context) during testing. OCC is a well studied problem [168, 169, 170]. Most existing classification algorithms can be extended for OCC. While OCC in general is not as accurate as techniques that leverage both positive and negative samples, it is sufficient and particularly suitable in our context as we use multiple invariant models to make joint decisions, which allows effectively mitigating inaccuracy in individual OCC models.

We use the *One-class Support Vector Machine* (OSVM) algorithm [169], which is the most popular classification algorithm for OCC problems and has a lot of applications [171]. The basic idea of OSVM is to assume a shape of the border between different classes



(a) Linear (b) Ploy (c) RBF

Figure 4.9.: Effects of using different SVM kernels

(represented by different kernel functions), and calculate the parameters describing the border shape. Figure 4.9(a) uses a linear kernel function and the border of a class is assumed to be a line. Figure 4.9(b) uses a polynomial kernel and Figure 4.9(c) uses a radial basis function kernel (RBF). For OSVMs, the most widely used kernel is RBF. In our case, since most values in the input space are invalid (e.g., most random images are not realistic) and the valid inputs cluster in small sub-spaces, using the RBF kernel to achieve good accuracy. Moreover, although RBF is expensive, we train models for individual layers and consecutive layer pairs, which essentially breaks the entire invariant space into sub-spaces that have more regularity. While standard OSVM outputs a single number 0 or 1 to indicate if an input belongs to a group, we change the algorithm to output the probability of membership. In other words, we use the trained OSVM models to measure similarity.

During production runs, given an input x, we compute $VI_l(x)$ and $PI_{l,l+1}(x)$ for all possible l. In other words, we collect the probabilities of any invariant violations across all layers. A standard OSVM classifier is further used to determine if t is an adversarial sample based on these probabilities and report the results. Details are elided.

4.4.4 Randomization

Adversarial samples may be *transferable*, meaning that some adversarial samples generated for a model may cause misclassification on another model as well [172, 173]. Such a property makes it possible to attack a black-box system (assuming no knowledge about training data, method, process, hyper-parameters and trained weights). For example, Papernot et al. [173, 148] proposed to use synthesized data that are labeled by querying the target model to train a substitute model, and demonstrated that the generated adversarial samples for the substitute model can be used to attack the target model. Moosavi-Dezfooli et al. [174] found the existence of a universal perturbation across different images and demonstrated that such images can transfer across different models of ImageNet. Liu et al. [175] proposed an ensemble-based approach to generate transferable adversarial examples for large datasets such as ImageNet. Intuitively, a DNN model is an *approximation* (or fitting) of the real distribution. There is always a discrepancy between the approximation (the trained DNN model) and reality, which is the space of adversarial samples.

While our invariant checker essentially prunes the adversarial space, the fact that our checker is training based suggests that it is still an approximation. In other words, it is susceptible to transferable adversarial samples just like other detection techniques. Note that even proof-based techniques [176] suffer the same problem as they can only prove that inputs must satisfy certain conditions. They cannot prove a model is precise as having a precise model itself is intractable.

An effective and practical solution is to introduce randomization in the detectors. For example, MagNet [126] trains multiple encoders beforehand, and at runtime, they randomly select one encoder to use. In feature squeezing, Xu et al. also introduce random clip operations in their squeezer to make the attack harder. Even though theoretically such randomization mechanisms cannot prevent all possible attacks (especially when facing adaptive adversaries), they were shown to be very effective in practice. With randomization, attacks take much longer time and generate human unrecognizable inputs [126, 127].

As an integral part of our system, we also use randomization. Our randomization is based on the observation that DNNs make decisions by looking at all activated neurons rather than a subset of them [118, 165]. Due to the large number of contributing neurons, benign inputs are more robust to small neuron value changes than adversarial samples. Thus when we train VIs and PIs, we first apply a transformation function. It means that we use $g_l \circ f_l$ instead of f_l as the input to train VIs and PIs. We prefer non-differentiable g_l functions. Because if the attacker wants to craft adversarial samples under the assumption that the adversary knows the presence of detector, non-differentiable functions would significantly increase the difficulty [177] due to the lack of gradient information. In contrast, our technique directly uses the output of $g_l \circ f_l$ to train VIs and PIs, thus the introduction of g_l does not affect our training. In this paper, typical g_l functions include random scaling combined with discrete floor and ceiling functions. At runtime, we randomly select a set of g_l functions and the corresponding trained invariant models in our detector. The results of randomization are shown in §4.5.

4.4.5 Threat Model

We assume that the adversary knows everything about the original classifier including the trained weights so that they can construct strong attacks such as the CW attacks, and the detector is not aware of the methods used for generating the adversarial samples. Depending on the information of the detector exposed to the adversary, there are multiple scenarios.

The weakest attack scenario is that the adversary knows nothing about the detector. In this case, the adversary generates the attack purely based on the original classifier. In §4.5, we show that our detector can successfully detect a wide spectrum of such attacks.

The strongest adversary has full knowledge of our detector. Since the detector itself is also a classifier, this makes it vulnerable to adversarial samples too [178]. Note that this limitation is not specific to our technique as other existing detection techniques suffer the same problem too. Under such a strong threat model, our technique has better resilience compared to other techniques. In particular, as discussed in §4.4.4, we introduce randomization in our detector to improve its robustness. During training of the detectors, we first apply different transform functions g_l on activated neurons to generate multiple randomized activation vectors. This allows us to have the flexibility of generating multiple detectors. At runtime, we can use different detectors (or their combinations) to detect adversarial samples. This substantially elevates the difficulty level of generating adversarial samples. In §4.5, we show the effects of using the randomization technique. Note that complete prevention of adversarial samples is intractable for approximate models to which almost all practical DNNs belong. Our goal is to have a general and practical solution to substantially raise the bar for attackers.

4.5 Evaluation

In this section, we discuss the results of a large scale evaluation. Most experiments were conducted on two servers. One is equipped with two Xeon E5-2667 2.3GHz 8-core processors, 128 GB of RAM, 2 Tesla K40c GPU, 2 GeForce GTX TITAN X GPU and 4 TITAN Xp GPU cards. The other one has two Intel Xeon E5602 2.13GHz 4-core processors, 24 GB of RAM, and 2 NVIDIA Tesla C2075 GPU cards.

4.5.1 Setup

Datasets. For gradient based attacks, we performed our experiments on three popular image datasets: MNIST [133], CIFAR-10 [134] and ImageNet [135]. MNIST is a grayscale image dataset used for handwritten digits recognition. CIFAR-10 and ImageNet are colored image datasets used for object recognition. For the ImageNet dataset, we used the ILSVRC2012 samples [179]. We chose these datasets because they were the most widely used datasets for this task and most existing attacks were carried out on them.

For content based attacks, Liu et al. evaluated Trojan attack on different datasets [5] (e.g., face recognition on LFW [136]). Besides using their face recognition data in our case study, we also ported the attack to two MNIST models, one Carlini model [122] and one Cleverhans model [180]. The trojaned Carlini model achieves 93% test accuracy on normal images and 100% attack accuracy, and the trojaned Cleverhans model has around 95% test accuracy on the original dataset with 100% attack accuracy. The evaluation of DeepXplore were conducted on their provided datasets (i.e., MNIST and ImageNet), their pre-trained models (i.e., LeNet-1, LeNet-4 and LeNet-5 for MNIST, and VGG16, VGG19 and ResNet50 for ImageNet), and their pre-generated adversarial samples on Github [124].

Attack. We evaluated our detection method on all the eleven attacks described in §4.2.3. For FGSM, BIM, and JSMA attacks, we used the implementations from the Cleverhans library [180] to generate adversarial samples, and for the other attacks, we used implementations from the authors [122, 123, 124, 5]. For the four targeted gradient based attacks (JSMA and three CW attacks), we evaluated on two different target settings: the next-class setting (denoted as *Next* in result tables) in which the targeted label is the next of the original label (e.g., misclassify an input of digit 2 to digit 3); and the least-likely class setting (denoted as *LL*), in which the target label is the most dissimilar to the original label (e.g., misclassify 1 to 8). Besides the four targeted attacks, the other seven included three untargeted gradient based attacks (FGSM, BIM and DeepFool) and four content based attacks (Trojaning, Dirt, Lighting and Rectangle Patching, with the last three from DeepXplore).

By default, we reused the same attack parameters used in feature squeezing [127] to construct attack images for gradient based attacks. DeepFool attack generates unrecognizable adversarial examples for the MNIST dataset and it is ignored by feature squeezing. As discussed in §4.2.3, the JSMA attack requires heavyweight computation and does not scale to large datasets like ImageNet. Thus we cannot test it on ImageNet. To generate Trojan triggers for the Trojan attack, we used the same configurations (e.g., size, shape and transparency) in their original paper [5]. DeepXplore targets to cover inactivate neurons, and it does not have extra attack parameters. For each dataset, we randomly partitioned the whole dataset to training data (60%, for training the detector), validation data (20%, for evaluating the detector with various g functions mentioned in §4.4.4) and test data (20%, to test performance). For each attack, we generated 100 adversarial samples for each model.

Models. We evaluated our technique on thirteen popular models. These models are representatives of their kinds and used in the attacks under study [122, 124, 5]. For the MNIST dataset, we have collected six models from the work by Carlini et al. [122], the Cleverhans library (2 models) [180], and three different DNNs in the LeNet family [133], i.e., LeNet-1, LeNet-4 and LeNet-5. For the CIFAR-10 dataset, we used two models, one from Carlini and Wagner [122] and the other from DenseNet [181, 182]. For ImageNet, we

Table 4.1.: Performance of NIC on Gradient Based Attacks

Dataset	Model	OH	FP	FGSM	BIM	CW_∞		DF	CW_2		CW_0		JSMA	
						Next	LL		Next	LL	Next	LL	Next	LL
MNIST	Cleverhans	4%	2.5%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
	Cleverhans2	3%	3.8%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
	Carlini	4%	3.7%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
CIFAR	Carlini	9%	4.2%	100%	100%	100%	100%	91%	96%	100%	98%	100%	94%	100%
	DenseNet	8%	3.8%	100%	100%	100%	100%	93%	97%	100%	97%	100%	92%	99%
ImageNet	ResNet50	28%	14.6%	100%	100%	99%	100%	92%	96%	100%	95%	100%	-	-
	VGG19	19%	9.9%	100%	100%	97%	100%	89%	93%	100%	100%	100%	-	-
	Inceptionv3	21%	13.8%	100%	100%	99%	100%	90%	96%	100%	100%	100%	-	-
	MobileNets	18%	7.2%	100%	100%	100%	100%	100%	100%	100%	100%	100%	-	-

used five models, ResNet50 [183], VGG16 [184], VGG19 [184], Inceptionv3 [185] and MobileNets [186, 187].

Comparison. We also compared NIC with other state-of-the-art detectors. For the metric based approach, we compared with LID [131]. Denoisers require a lot of parameter turning. Thus we chose to compare with well-tuned models, MagNet [126] for MNIST and CIFAR-10, and HGD [132] for ImageNet. For prediction inconsistency based techniques, we chose to compare with feature squeezing [127]. This paper only shows part of the comparison results due to the space limit. The remaining results are similar. For gradient based attacks on ImageNet, we used the Inceptionv3 model, and for MNIST and CIFAR-10, we adopted the same model used by Carlini and Wagner [122]. The Trojan attack on MNIST was also based on this model and the one from Cleverhans, and the attack on LFW was evaluated on the VGG19 model provided by the original authors. For DeepXplore attacks, the paper only shows the results for LeNet-4 and LeNet-5 on MNIST and VGG19 on ImageNet.

4.5.2 Detecting Gradient Based Attacks

Table 4.1 shows the detection accuracy of our approach on gradient based attacks. Columns 1 and 2 show datasets and models. Column 3 shows the runtime overhead. It is

Table 4.2.: Performance of NIC on Content Based Attacks

Dataset	FP	Model	Dirt	Lighting	RP
MNIST	3.4%	LeNet-1	100%	100%	100%
	2.5%	LeNet-4	100%	100%	100%
	2.5%	LeNet-5	100%	100%	100%
ImageNet	13.8%	ResNet50	100%	100%	100%
	15.2%	VGG19	100%	100%	100%
	15.9%	VGG16	100%	100%	100%
Dataset	FP	Model	Trojan	-	-
MNIST	3.7%	Carlini	100%	-	-
	2.5%	Cleverhans	100%	-	-
LFW	2.4%	VGG19	100%	-	-

measured by the average time used to run through the whole test dataset and adversarial samples (over 10 runs). Note that since we reuse the inner layer activation values of the original model when computing the results of VIs and PIs, invariant checking can be largely parallel with the original model execution. Thus the overhead is not significant. As shown in the table, for simple models (e.g., MNIST and CIFAR-10), we have less than 10% overhead. For complex models (e.g., ImageNet), it is less than 30%. Considering that the inference time for each image is very short (usually within a second), this is reasonable. Column 4 presents the false positive rates (FP) using default settings on each model, i.e., the number of benign inputs that are incorrectly detected as adversarial samples. The remaining columns show the detection results for each attack.

As shown in Table 4.1, for the three MNIST models, NIC can detect all gradient based attacks with 100% accuracy with less than 3.8% false positive rate on benign inputs. Also the false positive rates of NIC on CIFAR-10 models are 3.8% and 4.2%, and the detection rates are all over 90%. For L_∞ attacks on CIFAR-10, our approach can get 100% detection rates for all attack settings. For L_2 attacks, NIC can achieve nearly 100% detection rate on CW$_2$ attacks, and 91%/93% detection rates on DeepFool. The results of DeepFool are not 100% because it is an untargeted attack, which has much smaller perturbations compared to targeted attacks. For L_0 attacks, we can detect CW$_0$ and JSMA attacks with over 92% success rate.

For the ImageNet dataset, the detection rates show a similar pattern as the CIFAR-10 dataset. The images have a larger size and are more diverse, which makes the detection harder. For different models, the false positive rates range from 7.2% to 14.6%. In terms of detection rate, our detector can still achieve very high accuracy (over 90% for most cases).

4.5.3 Detecting Content Based Attacks

We also evaluated our approach on content based approaches, namely the Trojan and DeepXplore attacks. The results are summarized in Table 4.2. The above part shows the results of DeepXplore attacks, and the bottom part shows the results of the Trojan attack. The table has the same structure with Table 4.1. The runtime overhead is similar and hence elided.

As shown in the table, NIC is able to detect all adversarial samples generated by the Trojan attack and DeepXplore attacks (100% for all cases) with a relatively low false positive rate (less than 4% on MNIST and LFW, less than 16% on ImageNet). More details of how they can be detected are described in §4.5.4. Similar to the gradient based attacks, NIC has relatively high false positive rate on ImageNet than other datasets (e.g., MNIST), because of the large image size and content diversity. The results show that NIC is capable of detecting such content based attacks with high confidence and effectiveness.

4.5.4 NIC Invariants and Comparison

In Table 4.3, we show the comparison between NIC and other detectors (with gray background). Each row represents a detection method and the columns contain the results for different types of attacks (including both gradient based and content based attacks). For NIC, we also studied the effects of using only PIs or VIs on each attack to understand how they work. The rows are marked as PI and VI in Table 4.3. From the table, we make the following observations.

NIC Invariants. For most L_∞ attacks, using VIs alone has better results than using PIs alone, and using both produces the best results. This is because these attacks try to modify a lot of pixels with a small degree of change. As a result, the OVs in the first few layers violate VIs significantly, which makes it easy to detect. On the other hand, as the features extracted by the first few layers are still very primitive, the generated PIs are not of high quality. For most L_0 attacks where the perturbations happen for a very limited number of pixels, but the per-pixel change is more substantial, using PIs alone produces good results. Using VIs are not as good because the number of activated neurons in each layer is large but the ones perturbed are in a small number. As such, the distance between VIs and OVs may not be sufficiently large. On the other hand, these perturbed neurons alter the set of activated neurons in later layers, leading to PI violations. L_2 attacks bound the total Euclidean distance. As such, using PIs or VIs alone has mixed results, depending on which channel is being exploited. Our results indicate that either of them is (or both are) violated. Overall, using one kind of invariants alone is often not enough, and both kinds of invariants are very important to detecting adversarial samples.

For the Trojan attack, the adversarial samples tend to have the same activation pattern before the trojaned layer, and substantially change the prediction results at the trojaned layer, allowing them to be detected by PIs. Thus, Trojan attacks often lead to substantial invariant violations, which will be illustrated by a case study later in §4.5.7. The DeepXplore attacks are designed to generate adversarial samples that try to activate inactivated neurons to uncover hidden behaviors. Thus, it often significantly violates the invariants. In most cases, such changes significantly alter the prediction results and violate the PIs. In the Lighting and Rectangle Patching attacks, the perturbations are substantial, leading to significant changes in the activation value distributions. Thus using VIs can detect most of them. In contrast, the Dirt attack adds a limited number of black dots to the image, and some changes are not significant. Thus using VIs alone cannot achieve very good performance.

Denoisers. Denoisers (i.e., MagNet and HGD) work well for L_∞ attacks on all three datasets. In the mean time, we found that they do not perform well on L_0 attacks. This is because these denoisers are not guaranteed to be able to remove all the noise in an image, and the L_0 constraint limits the number of modified pixels and lowers the chance of being denoised. For most L_2 attacks, its performance is also not very good due to the same reason. As NIC also considers the effects of such noises (e.g., a few noisy pixels will change the activation pattern and violate the PIs), it can detect them as adversarial samples when the effects are amplified in hidden layers.

For content based attacks, this approach can detect all adversarial samples on MNIST and the RP attacks on ImageNet. But it does not perform well on other attacks. For Dirt attack which modifies relatively limited number of pixels, denoisers cannot remove some of the noises. For the Lighting attack, the whole color scheme is shifted with the same degree and denoisers do not perform well. In many cases they actually remove parts of the real objects. We believe it is because these parts have color patterns that are rare in the original training set, leading to some random behaviors of the denoisers. Denoisers show relatively good performance on the Trojan attacks as they transform the trigger to another representation, which lowers the chance to trigger the trojaned behaviors.

Denoisers tend to have relatively high false positive rates in most cases, meaning that the trained encoders and decoders affect the clean images especially for the colored datasets like ImageNet. Recall that denoisers are trained models. Training them is challenging and time-consuming. It may take more time than training the original classifier. One advantage of NIC is that it breaks the problem into small sub-problems. The detector consists of many small models and training them is much easier, making it more practical.

LID. For the gradient based attacks, the results of LID show that it is an effective approach for MNIST and CIFAR-10 with 93% and 90% average detection rates, respectively. However, it does not scale well on ImageNet, with 82% average detection rate. For the content based attacks, LID achieves relatively good results for both the Trojan attack and the DeepXplore attacks on the MNIST dataset, but relatively bad results on ImageNet. This is because images in ImageNet contain more noises and the clean images also have relatively large distances. This makes it more difficult identifying the boundaries between clean images and adversarial images. In the mean time, we observed that its false positive rates are relatively

Table 4.3.: NIC: Comparison with State-of-the-art Detectors

		Gradient-based												Content-based									
Detector	Dataset	FP	FGSM	BIM	CW_∞		DF	CW_2		CW_0		JSMA		Dataset	Model	FP	DeepXplore			Dataset	Model	FP	Trojan
					Next	LL		Next	LL	Next	LL	Next	LL				Dirt	Lighting	RP				
PI	MNIST	1.8%	84%	92%	94%	99%	95%	99%	96%	100%	100%	100%	100%	MNIST	LeNet-4	1.3%	100%	100%	100%	MNIST	Carlini	0.5%	100%
VI		1.9%	100%	100%	100%	100%	93%	100%	95%	90%	88%	85%	83%			1.2%	100%	100%	100%			3.2%	76%
NIC		3.7%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%			2.5%	100%	100%	100%			3.7%	100%
MagNet		5.1%	100%	100%	96%	97%	91%	85%	87%	85%	85%	84%	84%			4.3%	100%	100%	100%			5.6%	100%
LID		4.4%	97%	96%	93%	92%	92%	92%	91%	92%	92%	96%	96%			4.2%	100%	100%	100%			4.5%	100%
FS		4.0%	100%	98%	100%	100%	-	100%	100%	91%	94%	100%	100%			3.9%	97%	39%	72%			4.0%	82%
PI	CIFAR	2.6%	52%	64%	68%	63%	69%	89%	89%	98%	100%	94%	100%		LeNet-5	1.6%	100%	100%	100%		Cleverhans	0.4%	100%
VI		1.2%	100%	100%	100%	100%	78%	82%	88%	63%	68%	58%	62%			0.9%	100%	100%	100%			2.1%	72%
NIC		3.8%	100%	100%	100%	100%	91%	96%	100%	98%	100%	94%	100%			2.5%	100%	100%	100%			2.5%	100%
MagNet		6.4%	100%	100%	85%	87%	87%	89%	91%	76%	74%	72%	74%			4.2%	100%	100%	100%			5.2%	99%
LID		5.6%	94%	96%	91%	91%	84%	86%	88%	90%	91%	92%	92%			4.0%	99%	98%	99%			3.9%	97%
FS		4.9%	21%	55%	98%	100%	77%	100%	100%	98%	100%	84%	89%			3.9%	96%	41%	71%			3.6%	78%
PI	ImageNet	5.8%	30%	25%	53%	49%	81%	84%	86%	100%	100%	-	-	ImageNet	VGG19	1.5%	97%	95%	100%	LFW	VGG19	0.2%	100%
VI		1.4%	99%	100%	99%	100%	53%	90%	92%	29%	12%	-	-			1.0%	43%	100%	93%			2.2%	46%
NIC		7.2%	100%	100%	99%	100%	90%	96%	100%	100%	100%	-	-			2.5%	100%	100%	100%			2.4%	100%
HGD		9.7%	97%	95%	92%	92%	83%	83%	85%	81%	82%	-	-			6.2%	88%	69%	100%			4.7%	80%
LID		14.5%	82%	78%	85%	86%	83%	78%	80%	79%	80%	-	-			9.6%	83%	89%	91%			4.8%	75%
FS		8.3%	43%	64%	98%	100%	79%	92%	100%	98%	100%	-	-			3.9%	89%	40%	82%			3.3%	67%

high for all cases (highest on ImageNet and LFW), meaning that it tends to mis-identify clean images as adversarial samples, which makes the detection results less confident.

Feature Squeezing. For gradient based attacks, feature squeezing achieves very good results for a set of attacks, but does not perform very well for FGSM, BIM and DeepFool attacks on colored datasets (CIFAR-10 and ImageNet). For extreme cases like FGSM attacks on CIFAR-10, the detection rate of feature squeezing is about 20%. According to [127], this is likely because that the designed squeezers are not suitable for such attacks on colored datasets. It illustrates a limitation of feature squeezing: requiring high quality squeezers for various models. To this end, our approach is more general.

Feature squeezing does not perform well on the Trojan attack with the detection rate ranging from 67% to 82%, which is much lower than using our method. Through manual inspection, it appears that after applying the squeezers, the effects of the Trojan triggers (which lead to mis-classification) are degraded but not eliminated. As a result, many adversarial samples and their squeezed versions produce the same (malicious) classification results. For DeepXplore attacks, feature squeezing can detect almost all the adversarial samples generated by the Dirt attack, which adds black dots to the images to simulate the effects of dirt. This attack is very similar to L_0 attacks, on which feature squeezing performs very well. It cannot handle the Lighting attacks and Rectangle Patching (RP) attacks very well. The Rectangle Patching attack is very similar to the Trojan attack except that it generates a unique rectangle for each adversarial sample. Lighting attacks are similar to FGSM attacks but they change pixel values more aggressively. The changes can hardly be squeezed away.

We notice that the false positive rates of feature squeezing is relatively low. This shows that the designed squeezers are carefully chosen to avoid falsely recognize clean images as adversarial samples, which makes its results more trustworthy.

4.5.5 Adaptive Adversaries

The experiments in the previous sections are to detect adversarial samples under the assumption that the attacker is not aware of the existence of the detector (black-box attack). As discussed in §4.4.4, we need to deal with a stronger threat model, in which the attacker knows everything of the dataset, the trained model and the trained detector including the method used to train them as well as the model parameters after training. Our method against this attack model is to train multiple detectors using different g functions (§4.4.4). At runtime, NIC randomly chooses the detector(s) to use for the final decision. During this evaluation, we use three detectors and perform majority voting for the final decision.

We designed a CW_2 based white-box attack. The basic idea is to view the two classifiers (i.e., the original model and the detector) as a whole. In other words, we consider the output of the detector as part of the loss function to generate adversarial samples. The original CW_2 attack tries to minimize the L_2 distance of the generated adversarial sample and the original benign image such that the two images will lead to different prediction results. With the presence of NIC, the new attack modifies the optimization objective function. Now we minimize the L_2 distance of the two images as well as the L_2 distance of the activation neurons of the two images in each hidden layer and the prediction output vectors of the two images from all derived models (intuitively, minimizing VI and PI violations). We choose to make the attack untargeted, which tends to have less perturbation compared with targeted

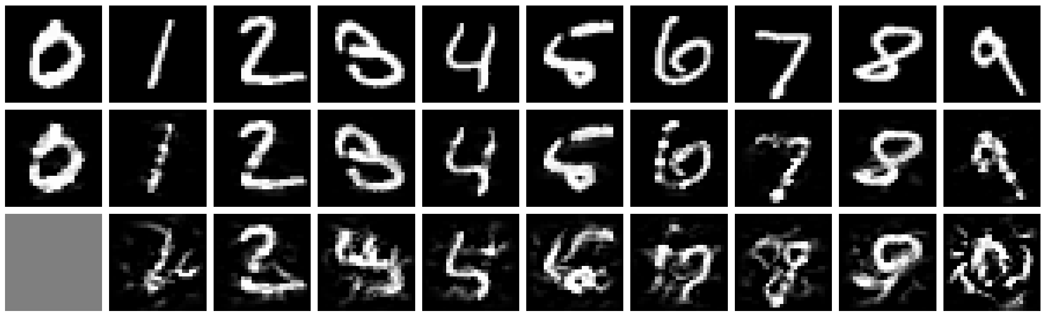


Figure 4.10.: Adversarial Samples. *1st line shows seed inputs, 2nd line shows without detector, 3rd line shows with 1 detector. Gray boxes mean no adversarial samples found in 1000 iterations.*

Table 4.4.: NIC: Effects of Different OCC Algorithms

Detector	FGSM	BIM	\cwi Next	\cwi LL	DF	\cwt Next	\cwt LL	\cwo Next	\cwo LL	JSMA	JSMA	Trojan	Dirt	Lighting	RP
5NN	14%	11%	9%	10%	4%	5%	4%	3%	1%	1%	2%	26%	12%	2%	24%
Linear	22%	20%	17%	19%	8%	12%	15%	8%	12%	22%	24%	32%	34%	93%	45%
Poly	89%	86%	79%	82%	85%	88%	89%	84%	85%	84%	86%	94%	89%	99%	95%
RBF	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	1005	100%	100%

attacks. As the g functions are not differentiable, we leverage a random process to find an adversarial sample that cannot be detected before the start of the optimization procedure.

If we limit the optimal constant (used to balance the amount of perturbation and the adversarial strength) to a rationale range, $[10^{-3}, 10^{6}]$, the same settings used in LID [131], the attack fails to generate adversarial images for all cases. This demonstrates that NIC is robust to such normal confidence attacks even under such strong white-box attack model. Without such limits, the attack can achieve 97% success rate on MNIST and CIFAR-10. The mean L_2 distortion of successful samples is 3.98 for MNIST and 2.76 for CIFAR-10, which is higher compared with other detectors (e.g., the L_2 distortion of feature squeezing is 2.80 on MNIST for adaptive, untargeted attacks [127]). This shows that our technique increases the difficulty of constructing adversarial samples.

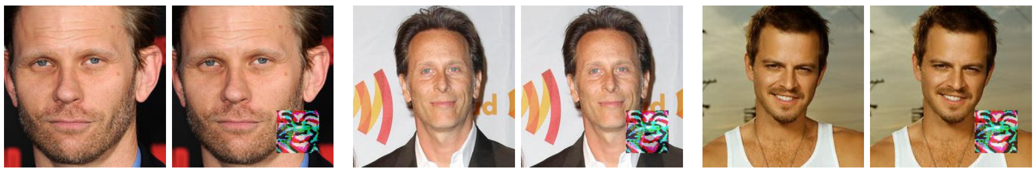
Because of the complexity of the optimization objective function, it takes significantly longer time to generate adversarial samples. We performed an experiment on a desktop equipped with an i7-3770 3.40GHz CPU and 28 GB of RAM. Specifically, generating a normal adversarial sample (without a detector) takes about 1 second, while generating an attack image takes about 420 seconds on average in the presence of one detector and around 1200 seconds on average in the presence of three detectors. This shows that such randomization approaches do increase the difficulty of generating adversarial samples. This result is also consistent with the findings in [177, 127]. Furthermore, the adversarial samples generated in the presence of a randomized detector usually have low confidence and are human unrecognizable. Figure 4.10 shows some generated attack images. The first row is the seed inputs, the second row shows the attack images without any detector and the last row shows the attack images to bypass one NIC detector. We can see that many of the last row images are not human recognizable.

4.5.6 Choice of One Class Classifiers

In this part, we evaluated the effects of using different OCC algorithms. We compared four algorithms using the Carlini model on MNIST dataset: KNN-OCC (k=5), OSVM with linear kernel, OSVM with poly kernel and OSVM with the RBF kernel. The results are shown in Table 4.4. As we can see, KNN-OCC performs badly for this task, and the more complex the kernel is, the better results the OSVM algorithm can achieve. This is because complex kernels are more expressive and can identify the differences between clean images and adversarial samples. Potentially, the model accuracy may be further improved with better one-class classification algorithms.

4.5.7 Case Study

We use the Trojan attack on a face recognition model (with 2622 output labels) to demonstrate invariant violations. The pre-trained VGG-face recognition model can be found at [5]. We use the same datasets (VGG and LFW) to generate adversarial samples to be consistent with the original paper [5]. Figure 4.11 shows the benign images (on the left of each subfigure) and the image patched with the Trojan trigger (on the right of each subfigure). The captions show the ground truth label. The trojaned model misclassifies all the images with the Trojan trigger to the target label `A.J. Buckley`. The attack triggers both value and provenance invariant violations at layer $L18$. The scale of the provenance invariant violation is more substantial. Table 4.5 shows part of the PI for layers $L17$ and $L18$ (rows 3,4), and the three OPs for the three images with the Trojan trigger (i.e., the three rows on the bottom). Each row consists of the classification results of the derived



(a) Mark Pellegrino (b) Steven Weber (c) Carmine Giovinazzo

Figure 4.11.: Adversarial samples for the Trojan attack

	A.J. Buckley	Mark Pellegrino	Steven Weber	Carmine Giovinazzo
		PI		
L17	0.0001	0.8857	0.0001	0.0004
L18	0.0001	0.9023	0.0001	0.0004
		OP		
L17	0.001	0.8734	0.0001	0.0004
L18	0.9273	0.0001	0.0001	0.0001
L17	0.0001	0.0001	0.7234	0.0001
L18	0.9736	0.0001	0.0001	0.0001
L17	0.0001	0.0001	0.0001	0.9243
L18	0.9652	0.0001	0.0001	0.0004

Table 4.5.: NIC: PI and OPs

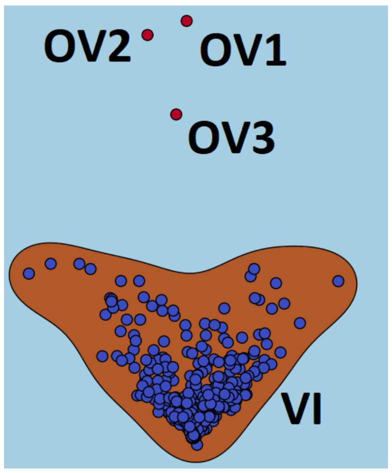


Figure 4.12.: NIC: VI and OVs

model of $L17$ and the results of $L18$. Since it is difficult to visualize PI, we show a typical vector value in the PI instead. Observe that the OPs are substantially different from the PI with the differences highlighted in red. Intuitively, the PI indicates that if $L17$ predicates `Mark` with a high confidence, $L18$ very likely predicts `Mark` with high confidence (slightly higher or lower than the previous layer result, no significant change) too. However, the first OP says that $L17$ predicts `Mark` with a high confidence but $L18$ predicts `A.J. Buckley` with a high confidence. This is a clear violation. The three adversarial samples cause value invariant violations as well (Figure 4.12). The brown region describes the VI distribution for $L18$. The adversarial samples (red dots) are clear outliers. However, NIC is not able to distinguish Trajoned models from those are simply vulnerable.

4.6 Related Work

We have covered many adversarial sample attacks, defense and detection works in §4.2. There are other types of attacks on machine learning models especially DNNs [188, 189, 190, 191, 192, 193]. Liu et al. [194] noticed that existing works study the perturbation problem by assuming an ideal software-level DNN, and argued this is not enough. They investigated adversarial samples by considering both perturbation and DNN model-reshaping (used by many DNN hardware implementations). Xiao et al. [195] utilized generative adversarial networks (GANs) to learn and approximate the distribution of original inputs so as to generate adversarial samples. Some works focus on applying poisoning attacks on machine learning models [196, 197, 198, 199]. Poisoning attacks aim at downgrading or compromising machine learning models by providing malicious training instances. Researchers have also proposed attacks to steal the parameters or hyper-parameters of machine learning models from open online services [200, 201]. Some works focus on studying the potential privacy leakage issues of machine learning [189, 190, 202, 203, 204, 205]. Fredrickson et al. [206, 207] proposed membership inference attacks on machine learning models by reverse engineering models to infer information of training data. Sharif et al. [208] focused on facial biometirc systems. They defined and investigated attacks that are physically realizable and inconspicuous, and allow an attacker to evade recognition or impersonate anther individual.

Recently, researchers proposed a few DNN verification frameworks [176, 209] that can verify DNN properties. Katz et al. [209] proposed an SMT-solver based solution, Reluplex, which can prove a network is δ-local-robust at input x, namely, for every x' such that $||x - x'||_\infty \leq \delta$, the network always assigns the same output label. And Gehr et al. [176] proposed AI^2 based on abstract interpretation and over-approximation, and they also used a similar local robustness property, namely, the network must assign the same label for a region and the region is defined for the Lighting attack in DeepXplore [124]. While these works demonstrate the potential of DNN verification, they aim to defend specific attacks. For example, the δ-local-robustness property proposed by Reluplex may not handle attacks beyond the L_∞ attack category whereas the local robustness property used in AI^2 was mainly for the Lighting attack in DeepXplore. To prove the aforementioned properties, the techniques require the knowledge of attack parameters, such as δ for Reluplex and the degree of change in Lighting attacks in DeepXplore, which may not be feasible. Moreover, these techniques prove that DNN does not misbehave *for a specific given input*. In practice, it is difficult to enumerate all possible benign inputs. Due to the cost of verification, experiments have been conducted on small datasets. It remains unclear if such techniques can scale to larger sets such as the ImageNet. In contrast, our technique is more general and practical.

We use one class classification (OCC) in our technique. Researchers have extensively studied OCC [168, 169, 210, 211, 212, 213]. The most widely used method is OSVM [169, 210]. The basic idea of OSVM is to fit the data into Support Vector Machine classifiers to find the correct boundary between positive and negative samples. There are also many works that try to use other classifiers for OCC, such as decision trees [211], k-nearest-neighbors [212, 213] and neural networks [170].

5 CONCLUSION

Provenance collection and analysis is one of the most important methods for computer forensic tasks and analyzing AI systems. In computer forensics analysis, recent APT attacks are becoming more and more persistent and sophisticated, which calls for the new provenance collection and analysis techniques to improve the effectiveness and efficiency of such systems. In AI systems, deep learning model internal neuron activations potentially can be used for adversarial sample detection. In this dissertation, I propose effective and efficient computation system provenance tracking techniques to reduce the space and runtime overheads and improve the dependency analysis results for enterprise environment cyber attack forensics especially for APTs, and detecting adversarial samples in AI systems by using internal neuron activation patterns as provenance.

In particular, we develop ProTracer, a cost-effective provenance tracing system that features the capabilities of alternating between logging and unit-level taint propagation, and event processing through a lightweight kernel module and a sophisticated concurrent user space daemon. Our evaluation results show that ProTracer substantially improves the state-of-the-art. In our experiments, it only generates 13MB audit log per day, and 0.84GB(Server)/2.32GB(Client) in 3 months with less than 7% overhead, while the generated logs do not lose any attack-related information.

Also, we analyzed execution partitioning which is important for addressing dependency explosion in audit logging. However, existing techniques are event loop based. They generate too many small units, require training to detect dependencies across units, and lack information about high level logic tasks. We propose MPI, a technique that partitions based on high level tasks. It allows the user to annotate the data structures corresponding to these task, and leverages compiler to instrument operations of the data structures in order to capture unit context switches and delegations. We implemented a prototype and evaluated it on three existing systems: Linux Audit, ProTracer and LPM-HiFi. The results show that

MPI generates much smaller graphs with lower overhead comparing to the state-of-the-art, and avoids broken provenance due to incomplete training.

Lastly, we propose a novel invariant based detection technique eagainst DNN adversarial samples, based on the observation that existing attacks are exploiting two common channels: the provenance channel and the activation value distribution channel. We develop innovative methods to extract provenance invariants and value invariants which can be checked at runtime to guard the two channels. Our evaluation on 11 different attacks shows that our technique can accurately detect these attacks with limited false positives, out-perform three state-of-the-art techniques based on different techniques and significantly increases the difficulty of constructing adversarial samples.

REFERENCES

[1] Wei You, Xuwei Liu, Shiqing Ma, David Mitchel Perry, Xiangyu Zhang, and Bin Liang. SLF: fuzzing without valid seed inputs. In Gunter Mussbacher, Joanne M. Atlee, and Tevfik Bultan, editors, *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 712–723. IEEE / ACM, 2019.

[2] Wei You, Xueqiang Wang, Shiqing Ma, Jianjun Huang, Xiangyu Zhang, Xiaofeng Wang, and Bin Liang. Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery. In *IEEE Security and Privacy 2019*, 2019.

[3] Juan Zhai, Jianjun Huang, Shiqing Ma, Xiangyu Zhang, Lin Tan, Jianhua Zhao, and Feng Qin. Automatic model generation from documentation for java API functions. In Laura K. Dillon, Willem Visser, and Laurie Williams, editors, *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 380–391. ACM, 2016.

[4] Zhenhao Tang, Juan Zhai, Minxue Pan, Yousra Aafer, Shiqing Ma, Xiangyu Zhang, and Jianhua Zhao. Dual-force: understanding webview malware via cross-language forced execution. In Marianne Huchard, Christian Kästner, and Gordon Fraser, editors, *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, pages 714–725. ACM, 2018.

[5] Yingqi Liu, Shiqing Ma, Yousra Aafer, Wen-Chuan Lee, Juan Zhai, Weihang Wang, and Xiangyu Zhang. Trojaning attack on neural networks. In *25nd Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-221, 2018*. The Internet Society, 2018.

[6] Wen-Chuan Lee, Yingqi Liu, Peng Liu, Shiqing Ma, Hongjun Choi, Xiangyu Zhang, and Rajiv Gupta. White-box program tuning. In Mahmut Taylan Kandemir, Alexandra Jimborean, and Tipp Moseley, editors, *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2019, Washington, DC, USA, February 16-20, 2019*, pages 122–135. IEEE, 2019.

[7] Wen-Chuan Lee, Peng Liu, Yingqi Liu, Shiqing Ma, and Xiangyu Zhang. Programming support for autonomizing software. In Kathryn S. McKinley and Kathleen Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019.*, pages 702–716. ACM, 2019.

[8] Zhaogui Xu, Shiqing Ma, Xiangyu Zhang, Shuofei Zhu, and Baowen Xu. Debugging with intelligence via probabilistic inference. In Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman, editors, *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 1171–1181. ACM, 2018.

[9] Yaozu Dong, Wei Ye, Yunhong Jiang, Ian Pratt, Shiqing Ma, Jian Li, and Haibing Guan. COLO: coarse-grained lock-stepping virtual machines for non-stop service. In Guy M. Lohman, editor, *ACM Symposium on Cloud Computing, SOCC '13, Santa Clara, CA, USA, October 1-3, 2013*, pages 3:1–3:16. ACM, 2013.

[10] 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. ACSAC '15.

[11] Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. Protracer: towards practical provenance tracing by alternating between logging and tainting. NDSS '16.

[12] 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 *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 1111–1128, 2017.

[13] Kexin Pei, Zhongshu Gu, Brendan Saltaformaggio, Shiqing Ma, Fei Wang, Zhiwei Zhang, Luo Si, Xiangyu Zhang, and Dongyan Xu. Hercule: Attack story reconstruction via community discovery on correlated log graph. In *Proceedings of the 32Nd Annual Conference on Computer Security Applications*, ACSAC '16, pages 583–595, New York, NY, USA, 2016. ACM.

[14] Yonghwi Kwon, Fei Wang, Weihang Wang, Kyu Hyung Lee, Wen-Chuan Lee, Shiqing Ma, Xiangyu Zhang, Dongyan Xu, Somesh Jha, Gabriela F. Ciocarlie, Ashish Gehani, and Vinod Yegneswaran. MCI : Modeling-based causality inference in audit logging for attack investigation. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018.

[15] Shiqing Ma, Juan Zhai, Yonghwi Kwon, Kyu Hyung Lee, Xiangyu Zhang, Gabriela F. Ciocarlie, Ashish Gehani, Vinod Yegneswaran, Dongyan Xu, and Somesh Jha. Kernel-supported cost-effective audit logging for causality tracking. In Haryadi S. Gunawi and Benjamin Reed, editors, *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018.*, pages 241–254. USENIX Association, 2018.

[16] Fei Wang, Yonghwi Kwon, Shiqing Ma, Xiangyu Zhang, and Dongyan Xu. Lprov: Practical library-aware provenance tracing. In *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*, pages 605–617. ACM, 2018.

[17] Chonghua Wang, Shiqing Ma, Xiangyu Zhang, Junghwan Rhee, Xiaochun Yun, and Zhiyu Hao. A hypervisor level provenance system to reconstruct attack story caused by kernel malware. In Xiaodong Lin, Ali Ghorbani, Kui Ren, Sencun Zhu, and Aiqing Zhang, editors, *Security and Privacy in Communication Networks - 13th International Conference, SecureComm 2017, Niagara Falls, ON, Canada, October 22-25, 2017, Proceedings*, volume 238 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 778–792. Springer, 2017.

[18] Shiqing Ma, Yingqi Liu, Guanhong Tao, Wen-Chuan Lee, and Xiangyu Zhang. NIC: detecting adversarial samples with neural network invariant checking. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019.

[19] Guanhong Tao, Shiqing Ma, Yingqi Liu, and Xiangyu Zhang. Attacks meet interpretability: Attribute-steered detection of adversarial samples. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada.*, pages 7728–7739, 2018.

[20] Shiqing Ma, Yingqi Liu, Wen-Chuan Lee, Xiangyu Zhang, and Ananth Grama. MODE: automated neural network model debugging via state differential analysis and input selection. In Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu, editors, *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, pages 175–186. ACM, 2018.

[21] Shiqing Ma, Yousra Aafer, Zhaogui Xu, Wen-Chuan Lee, Juan Zhai, Yingqi Liu, and Xiangyu Zhang. LAMP: data provenance for graph based machine learning algorithms through derivative computation. In Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman, editors, *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, pages 786–797. ACM, 2017.

[22] S. Miles, P. Groth, M. Branco, and L. Moreau. The requirements of recording and using provenance in e-science experiments. *Journal of Grid Computing*.

[23] P. Groth, S. Miles, W. Fang, S. C. Wong, K. P. Zauner, and L. Moreau. HPDC'05.

[24] Xuxian Jiang, AAron Walters, Dongyan Xu, Eugene H. Spafford, Florian Buchholz, and Yi-Min Wang. Provenance-aware tracing of worm break-in and contaminations: A process coloring approach. ICDCS.

[25] Samuel T. King and Peter M. Chen. Backtracking intrusions. SOSP '03.

[26] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. High accuracy attack provenance via binary-based execution partition. NDSS '13.

[27] Ashvin Goel, Kenneth Po, Kamran Farhadi, Zheng Li, and Eyal de Lara. The taser intrusion recovery system. SOSP '05.

[28] Sriranjani Sitaraman and S. Venkatesan. Forensic analysis of file system intrusions using improved backtracking. IWIA '05.

[29] Devin J. Pohly, Stephen McLaughlin, Patrick McDaniel, and Kevin Butler. Hi-fi: Collecting high-fidelity whole-system provenance. ACSAC '12.

[30] Kiran-Kumar Muniswamy-Reddy, David A Holland, Uri Braun, and Margo I Seltzer. Provenance-aware storage systems. Usenix ATC '06.

[31] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. NDSS'05.

[32] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. Bitblaze: A new approach to computer security via binary analysis. ICISS '08.

[33] Vasileios P. Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D. Keromytis. Libdft: Practical dynamic data flow tracking for commodity systems. VEE '12.

[34] James Clause, Wanchun Li, and Alessandro Orso. Dytan: A generic dynamic taint analysis framework. ISSTA '07.

[35] Erik Bosman, Asia Slowinska, and Herbert Bos. Minemu: The world's fastest taint tracker. RAID'11, 2011.

[36] Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou, and Youfeng Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. MICRO 39.

[37] Stephen McCamant and Michael D. Ernst. Quantitative information flow as network flow capacity. PLDI '08.

[38] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. Dta++: Dynamic taint analysis with targeted control-flow propagation. NDSS'11.

[39] Mingwu Zhang, Xiangyu Zhang, Xiang Zhang, and Sunil Prabhakar. Tracing lineage beyond relational operators. VLDB '07.

[40] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. Loggc: garbage collecting audit log. CCS '13.

[41] Wes Masri, Andy Podgurski, and David Leon. Detecting and debugging insecure information flows. ISSRE '04.

[42] Ashish Gehani and Dawood Tariq. Spade: Support for provenance auditing in distributed environments. Middleware '12.

[43] Tracepoints. `https://www.kernel.org/doc/Documentation/trace/tracepoints.txt`.

[44] Apache benchmark. `https://goo.gl/L7bGOK`.

[45] Proftp backdoor. `http://www.osvdb.org/69562`.

[46] Student hacker. `http://www.huffingtonpost.com/2014/03/05/student-hacking_n_4907344.html`.

[47] Irs phishing email. `https://www.spamstopshere.com/blog/spam-news/alert-irs-scam-email-links-malicious-code`.

[48] Samuel T King, Zhuoqing Morley Mao, Dominic G Lucchetti, and Peter M Chen. Enriching intrusion alerts through multi-host causality. NDSS'05.

[49] Taesoo Kim, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Intrusion recovery using selective re-execution. OSDI'10.

[50] Ningning Zhu and Tzi-cker Chiueh. Design, implementation, and evaluation of repairable file service. DSN'13.

[51] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding data lifetime via whole system simulation. USENIX SSYM'04.

[52] Uri Braun, Simson Garfinkel, David A Holland, Kiran-Kumar Muniswamy-Reddy, and Margo I Seltzer. Issues in automatic provenance collection. In *Provenance and annotation of data*.

[53] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazieres, Frans Kaashoek, and Robert Morris. Labels and event processes in the asbestos operating system. SOSP '05.

[54] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in histar. OSDI '06.

[55] Kangkook Jee, Georgios Portokalidis, Vasileios P. Kemerlis, Soumyadeep Ghosh, David I. August, and Angelos D. Keromytis. A general approach for effciently accelerating software-based dynamicdata flow tracking on commodity hardware. NSDI'12.

[56] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. CCS '07.

[57] Byung Chul Tak, Chunqiang Tang, Chun Zhang, Sriram Govindan, Bhuvan Urgaonkar, and Rong N. Chang. vpath: precise discovery of request processing paths from black-box observations of thread and network activities. USENIX ATC'09.

[58] Kiran-Kumar Muniswamy-Reddy, Uri Braun, David A. Holland, Peter Macko, Diana Maclean, Daniel Margo, Margo Seltzer, and Robin Smogor. Layering in provenance systems. USENIX ATC'09.

[59] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32, 2014.

[60] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. OSDI'10.

[61] Kui Xu, Huijun Xiong, Chehai Wu, Deian Stefan, and Danfeng Yao. Data-provenance verification for secure hosts. *Dependable and Secure Computing, IEEE Transactions on*, 9(2), 2012.

[62] Hao Zhang, Danfeng Daphne Yao, and Naren Ramakrishnan. Detection of stealthy malware activities with traffic causality and scalable triggering relation discovery. ACSAC'14.

[63] Yulai Xie, Kiran-Kumar Muniswamy-Reddy, Darrell DE Long, Ahmed Amer, Dan Feng, and Zhipeng Tan. Compressing provenance graphs. TaPP'11.

[64] Yulai Xie, Dan Feng, Zhipeng Tan, Lei Chen, Kiran-Kumar Muniswamy-Reddy, Yan Li, and Darrell DE Long. A hybrid approach for efficient provenance storage. CIKM '12.

[65] Yulai Xie, Kiran-Kumar Muniswamy-Reddy, Dan Feng, Yan Li, and Darrell DE Long. Evaluation of a hybrid approach for efficient provenance storage. *ACM Transactions on Storage (TOS)*, 9(4):14, 2013.

[66] Adam Bates, Kevin R.B. Butler, and Thomas Moyer. Take only what you need: Leveraging mandatory access control policy to reduce provenance storage costs. TaPP '15.

[67] Zhenyu Guo, Dong Zhou, Haoxiang Lin, Mao Yang, Fan Long, Chaoqiang Deng, Changshu Liu, and Lidong Zhou. G^2: A graph processing system for diagnosing distributed systems. USENIX ATC'11.

[68] Markus Jakobsson and Ari Juels. Server-side detection of malware infection. NSPW '09.

[69] Amit Vasudevan, Ning Qu, and Adrian Perrig. Xtrec: Secure real-time execution trace recording on commodity platforms. HICSS'11.

[70] Adam Bates, Dave (Jing) Tian, Kevin R.B. Butler, and Thomas Moyer. Trustworthy whole-system provenance for the linux kernel. Usenix Security'15.

[71] Swaminathan Sundararaman, Gopalan Sivathanu, and Erez Zadok. Selective versioning in a secure disk system. Usenix Security'08.

[72] Radu Sion. Strong worm. ICDCS'08.

[73] Watering hole attack. `https://goo.gl/aw1t9l`.

[74] Many watering holes, targets in hacks that netted facebook, twitter and apple. `https://goo.gl/NIg2Va`.

[75] More details on "operation aurora". `https://goo.gl/p76ovs`.

[76] Github hacked, millions of projects at risk of being modified or deleted. `https://goo.gl/EdguGO`.

[77] Chinese hacker arrested for leaking 6 million logins. `https://goo.gl/AO2Qlz`.

[78] The browser exploitation framework. `http://beefproject.com/`.

[79] Samuel T King, Zhuoqing Morley Mao, Dominic G Lucchetti, and Peter M Chen. Enriching intrusion alerts through multi-host causality. NDSS '05.

[80] Linux audit subsystem. `https://goo.gl/WSwnJB`.

[81] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. Linux security modules: General security support for the linux kernel. In *Proceedings of the 11th USENIX Security Symposium*, pages 17–31, Berkeley, CA, USA, 2002. USENIX Association.

[82] Zhang Xu, Zhenyu Wu, Zhichun Li, Kangkook Jee, Junghwan Rhee, Xusheng Xiao, Fengyuan Xu, Haining Wang, and Guofei Jiang. High fidelity data reduction for big data security dependency analysis. CCS '16.

[83] Event tracing for windows (etw). `http://msdn.microsoft.com/en-us/library/windows/desktop/aa363668(v=vs.85).aspx`.

[84] Windows event log. `https://msdn.microsoft.com/en-us/library/windows/desktop/aa385780(v=vs.85).aspx`.

[85] Vim document: windows. `https://goo.gl/Lqp9Gb`.

[86] Sloccount. `http://www.dwheeler.com/sloccount/`.

[87] Leaked data. `https://haveibeenpwned.com/`.

[88] The sony hack. `https://goo.gl/B4G7Pl`.

[89] Adam Cummings, Todd Lewellen, David McIntire, Andrew P Moore, and Randall Trzeciak. Insider threat study: Illicit cyber activity involving fraud in the us financial services sector. Technical report, DTIC Document, 2012.

[90] Marisa R Randazzo, Michelle Keeney, Eileen Kowalski, Dawn Cappelli, and Andrew Moore. Insider threat study: Illicit cyber activity in the banking and finance sector. Technical report, DTIC Document, 2005.

[91] Michelle Keeney, Eileen Kowalski, Dawn Cappelli, Andrew Moore, Timothy Shimeall, Stephanie Rogers, et al. Insider threat study: Computer system sabotage in critical infrastructure sectors. *US Secret Service and CERT Coordination Center/SEI*, 2005.

[92] Eileen Kowalski, Tara Conway, Susan Keverline, Megan Williams, Dawn Cappelli, Bradford Willke, and Andrew Moore. Insider threat study: Illicit cyber activity in the government sector. *US Department of Homeland Security, US Secret Service, CERT, and the Software Engineering Institute (Carnegie Mellon University), Tech. Rep*, 2008.

[93] Tuxpaint. `www.tuxpaint.org`.

[94] Extensions to the c language family. `https://goo.gl/evrruW`.

[95] Extensions to the c++ language. `https://goo.gl/pn19Np`.

[96] Clang language extensions. `https://goo.gl/UpniZC`.

[97] Dave (Jing) Tian, Adam Bates, Kevin R.B. Butler, and Raju Rangaswami. Provusb: Block-level provenance-based data protection for usb storage devices. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 242–253, New York, NY, USA, 2016. ACM.

[98] David Devecsery, Michael Chow, Xianzheng Dou, Jason Flinn, and Peter M Chen. Eidetic systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 525–540, 2014.

[99] Simon Miles, Paul Groth, Steve Munroe, and Luc Moreau. Prime: A methodology for developing provenance-aware applications. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(3):8, 2011.

[100] Michelle A. Borkin, Chelsea S. Yeh, Madelaine Boyd, Peter Macko, Krzysztof Z. Gajos, Margo Seltzer, and Hanspeter Pfister. Evaluation of filesystem provenance visualization tools. *IEEE Transactions on Visualization and Computer Graphics*, 19(12):2476–2485, December 2013.

[101] Charles F Bevan and R Michael Young. Planning Attack Graphs. In *ACSAC*, 2011.

[102] Vaibhav Mehta, Constantinos Bartzis, Haifeng Zhu, Edmund Clarke, and Jeannette Wing. Ranking Attack Graphs. *9th International Symposium on Recent Advances in Intrusion Detection (RAID'06)*, 4219:127–144, 2006.

[103] Xinming Ou, Wayne F. Boyer, and Miles a. McQueen. A scalable approach to attack graph generation. In *Proceedings of the 13th ACM conference on Computer and communications security - CCS '06*, page 336, 2006.

[104] Reginald E. Sawilla and Xinming Ou. Identifying critical attack assets in dependency attack graphs. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 5283 LNCS, pages 18–34, 2008.

[105] Xinming Ou, Sudhakar Govindavajhala, and Aw Appel. MulVAL: A logic-based network security analyzer. *14th USENIX Security . . .* , (August):8, 2005.

[106] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J. M. Wing. Automated generation and analysis of attack graphs. In *Proceedings - IEEE Symposium on Security and Privacy*, volume 2002-January, pages 273–284, 2002.

[107] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiao-yong Zhou, and XiaoFeng Wang. Effective and efficient malware detection at the end host. USENIX'09.

[108] Z. Gu, K. Pei, Q. Wang, L. Si, X. Zhang, and D. Xu. Leaps: Detecting camouflaged attacks with statistical learning guided by program analysis. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 57–68, June 2015.

[109] Clemens Kolbitsch, Engin Kirda, and Christopher Kruegel. The power of procrastination: Detection and mitigation of execution-stalling malicious code. CCS '11. ACM.

[110] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, and Konrad Rieck. Drebin: Effective and explainable detection of android malware in your pocket. NDSS'14.

[111] Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D Ernst. Leveraging existing instrumentation to automatically infer invariant-constrained models. ESEC/FSE'11.

[112] Karthik Nagaraj, Charles Killian, and Jennifer Neville. Structured comparative analysis of systems logs to diagnose performance problems. NSDI'12.

[113] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I Jordan. Detecting large-scale system problems by mining console logs. SOSP'09.

[114] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[115] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.

[116] George E Dahl, Jack W Stokes, Li Deng, and Dong Yu. Large-scale malware classification using random projections and neural networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 3422–3426. IEEE, 2013.

[117] Yisroel Mirsky, Tomer Doitshman, Yuval Elovici, and Asaf Shabtai. Kitsune: An ensemble of autoencoders for online network intrusion detection. *arXiv preprint arXiv:1802.09089*, 2018.

[118] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

[119] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[120] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. Adversarial examples in the physical world. *arXiv preprint arXiv:1607.02533*, 2016.

[121] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z Berkay Celik, and Ananthram Swami. The limitations of deep learning in adversarial settings. In *Security and Privacy (EuroS&P), 2016 IEEE European Symposium on*, pages 372–387. IEEE, 2016.

[122] Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 39–57. IEEE, 2017.

[123] Seyed Mohsen Moosavi Dezfooli, Alhussein Fawzi, and Pascal Frossard. Deepfool: a simple and accurate method to fool deep neural networks. In *Proceedings of 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, number EPFL-CONF-218057, 2016.

[124] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. Deepxplore: Automated whitebox testing of deep learning systems. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 1–18. ACM, 2017.

[125] Shixiang Gu and Luca Rigazio. Towards deep neural network architectures robust to adversarial examples. *arXiv preprint arXiv:1412.5068*, 2014.

[126] Dongyu Meng and Hao Chen. Magnet: a two-pronged defense against adversarial examples. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 135–147. ACM, 2017.

[127] Weilin Xu, David Evans, and Yanjun Qi. Feature Squeezing: Detecting Adversarial Examples in Deep Neural Networks. In *Proceedings of the 2018 Network and Distributed Systems Security Symposium (NDSS)*, 2018.

[128] Reuben Feinman, Ryan R Curtin, Saurabh Shintre, and Andrew B Gardner. Detecting adversarial samples from artifacts. *arXiv preprint arXiv:1703.00410*, 2017.

[129] Kathrin Grosse, Praveen Manoharan, Nicolas Papernot, Michael Backes, and Patrick McDaniel. On the (statistical) detection of adversarial examples. *arXiv preprint arXiv:1702.06280*, 2017.

[130] Jan Hendrik Metzen, Tim Genewein, Volker Fischer, and Bastian Bischoff. On detecting adversarial perturbations. *arXiv preprint arXiv:1702.04267*, 2017.

[131] Xingjun Ma, Bo Li, Yisen Wang, Sarah M Erfani, Sudanthi Wijewickrema, Michael E Houle, Grant Schoenebeck, Dawn Song, and James Bailey. Characterizing adversarial subspaces using local intrinsic dimensionality. 2018.

[132] Fangzhou Liao, Ming Liang, Yinpeng Dong, Tianyu Pang, Jun Zhu, and Xiaolin Hu. Defense against adversarial attacks using high-level representation guided denoiser. *arXiv preprint arXiv:1712.02976*, 2017.

[133] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[134] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. 2009.

[135] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[136] Gary B. Huang, Manu Ramesh, Tamara Berg, and Erik Learned-Miller. Labeled faces in the wild: A database for studying face recognition in unconstrained environments. Technical Report 07-49, University of Massachusetts, Amherst, October 2007.

[137] DNNInvariant. Dnninvariant/nninvariant. `https://github.com/DNNInvariant/nninvariant`, 2018.

[138] Nicolas Papernot, Patrick McDaniel, Xi Wu, Somesh Jha, and Ananthram Swami. Distillation as a defense to adversarial perturbations against deep neural networks. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 582–597. IEEE, 2016.

[139] Tom B Brown, Dandelion Mané, Aurko Roy, Martín Abadi, and Justin Gilmer. Adversarial patch. *arXiv preprint arXiv:1712.09665*, 2017.

[140] Ivan Evtimov, Kevin Eykholt, Earlence Fernandes, Tadayoshi Kohno, Bo Li, Atul Prakash, Amir Rahmati, and Dawn Song. Robust physical-world attacks on deep learning models. *arXiv preprint arXiv:1707.08945*, 1, 2017.

[141] Battista Biggio and Fabio Roli. Wild patterns: Ten years after the rise of adversarial machine learning. *arXiv preprint arXiv:1712.03141*, 2017.

[142] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[143] Yang Song, Taesup Kim, Sebastian Nowozin, Stefano Ermon, and Nate Kushman. Pixeldefend: Leveraging generative models to understand and defend against adversarial examples. 2018.

[144] Cihang Xie, Jianyu Wang, Zhishuai Zhang, Zhou Ren, and Alan Yuille. Mitigating adversarial effects through randomization. 2018.

[145] Nicolas Papernot, Patrick McDaniel, Arunesh Sinha, and Michael Wellman. Towards the science of security and privacy in machine learning. *arXiv preprint arXiv:1611.03814*, 2016.

[146] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. In *International Conference on Learning Representations*, 2018.

[147] Nicholas Carlini and David Wagner. Defensive distillation is not robust to adversarial examples. *arXiv preprint arXiv:1607.04311*, 2016.

[148] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z Berkay Celik, and Ananthram Swami. Practical black-box attacks against machine learning. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 506–519. ACM, 2017.

[149] Anish Athalye, Nicholas Carlini, and David Wagner. Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples. *arXiv preprint arXiv:1802.00420*, 2018.

[150] Anh Nguyen, Jason Yosinski, and Jeff Clune. Deep neural networks are easily fooled: High confidence predictions for unrecognizable images. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 427–436, 2015.

[151] Arjun Nitin Bhagoji, Daniel Cullina, and Prateek Mittal. Dimensionality reduction as a defense against evasion attacks on machine learning classifiers. *arXiv preprint arXiv:1704.02654*, 2017.

[152] Zhitao Gong, Wenlu Wang, and Wei-Shinn Ku. Adversarial and clean data are not twins. *arXiv preprint arXiv:1704.04960*, 2017.

[153] Dan Hendrycks and Kevin Gimpel. Early methods for detecting adversarial images. 2017.

[154] Xin Li and Fuxin Li. Adversarial examples detection in deep networks with convolutional filter statistics. *CoRR, abs/1612.07767*, 7, 2016.

[155] Nicolas Papernot and Patrick McDaniel. Deep k-nearest neighbors: Towards confident, interpretable and robust deep learning. *arXiv preprint arXiv:1803.04765*, 2018.

[156] Bita Rouhani, Mohammad Samragh, Tara Javidi, and Farinaz Koushanfar. Safe machine learning and defeating adversarial attacks. *To appear in IEEE Security and Privacy (S&P) Magazine*, 2018.

[157] Nicholas Carlini and David Wagner. Adversarial examples are not easily detected: Bypassing ten detection methods. In *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*, AISec '17, pages 3–14, New York, NY, USA, 2017. ACM.

[158] Jiajun Lu, Theerasit Issaranon, and David Forsyth. Safetynet: Detecting and rejecting adversarial examples robustly. In *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 446–454. IEEE, 2017.

[159] Pei-Hsuan Lu, Pin-Yu Chen, and Chia-Mu Yu. On the limitation of local intrinsic dimensionality for characterizing the subspaces of adversarial examples. *arXiv preprint arXiv:1803.09638*, 2018.

[160] Pouya Samangouei, Maya Kabkab, and Rama Chellappa. Defense-gan: Protecting classifiers against adversarial attacks using generative models. In *International Conference on Learning Representations*, 2018.

[161] Anish Athalye and Nicholas Carlini. On the robustness of the cvpr 2018 white-box adversarial example defenses. *arXiv preprint arXiv:1804.03286*, 2018.

[162] Chuan Guo, Mayank Rana, Moustapha Cissé, and Laurens van der Maaten. Countering adversarial images using input transformations. 2018.

[163] Guneet S Dhillon, Kamyar Azizzadenesheli, Zachary C Lipton, Jeremy Bernstein, Jean Kossaifi, Aran Khanna, and Anima Anandkumar. Stochastic activation pruning for robust adversarial defense. 2018.

[164] Guanhong Tao, Shiqing Ma, Yingqi Liu, and Xiangyu Zhang. Attacks meet interpretability: Attribute-steered detection of adversarial samples. In *Advances in Neural Information Processing Systems*, pages 7727–7738, 2018.

[165] Ari S Morcos, David GT Barrett, Neil C Rabinowitz, and Matthew Botvinick. On the importance of single directions for generalization. *arXiv preprint arXiv:1803.06959*, 2018.

[166] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 340–353. ACM, 2005.

[167] Mei-Chen Hsueh, Timothy K Tsai, and Ravishankar K Iyer. Fault injection techniques and tools. *Computer*, 30(4):75–82, 1997.

[168] David MJ Tax and Robert PW Duin. Data domain description using support vectors. In *ESANN*, volume 99, pages 251–256, 1999.

[169] David MJ Tax and Robert PW Duin. Support vector domain description. *Pattern recognition letters*, 20(11-13):1191–1199, 1999.

[170] Pramuditha Perera and Vishal M Patel. Learning deep features for one-class classification. *arXiv preprint arXiv:1801.05365*, 2018.

[171] Marco AF Pimentel, David A Clifton, Lei Clifton, and Lionel Tarassenko. A review of novelty detection. *Signal Processing*, 99:215–249, 2014.

[172] Florian Tramèr, Nicolas Papernot, Ian Goodfellow, Dan Boneh, and Patrick McDaniel. The space of transferable adversarial examples. *arXiv preprint arXiv:1704.03453*, 2017.

[173] Nicolas Papernot, Patrick McDaniel, and Ian Goodfellow. Transferability in machine learning: from phenomena to black-box attacks using adversarial samples. *arXiv preprint arXiv:1605.07277*, 2016.

[174] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, Omar Fawzi, and Pascal Frossard. Universal adversarial perturbations. *arXiv preprint*, 2017.

[175] Yanpei Liu, Xinyun Chen, Chang Liu, and Dawn Song. Delving into transferable adversarial examples and black-box attacks. *arXiv preprint arXiv:1611.02770*, 2016.

[176] T. Gehr, M. Mirman, D. Drachsler-Cohen, P. Tsankov, S. Chaudhuri, and M. Vechev. Ai2: Safety and robustness certification of neural networks with abstract interpretation. In *2018 IEEE Symposium on Security and Privacy (SP)*, volume 00, pages 948–963. IEEE, 2018.

[177] Warren He, James Wei, Xinyun Chen, Nicholas Carlini, and Dawn Song. Adversarial example defense: Ensembles of weak defenses are not strong. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, Vancouver, BC, 2017. USENIX Association.

[178] Justin Gilmer, Luke Metz, Fartash Faghri, Samuel S Schoenholz, Maithra Raghu, Martin Wattenberg, and Ian Goodfellow. Adversarial spheres. *arXiv preprint arXiv:1801.02774*, 2018.

[179] ImageNet. Imagenet large scale visual recognition competition 2012 (ilsvrc2012). `http://www.image-net.org/challenges/LSVRC/2012/`, 2018.

[180] Nicolas Papernot, Nicholas Carlini, Ian Goodfellow, Reuben Feinman, Fartash Faghri, Alexander Matyasko, Karen Hambardzumyan, Yi-Lin Juang, Alexey Kurakin, Ryan Sheatsley, et al. cleverhans v2.0.0: an adversarial machine learning library. *arXiv preprint arXiv:1610.00768*, 2016.

[181] DenseNet. Densenet implementation in keras, 2018.

[182] Gao Huang, Zhuang Liu, Kilian Q Weinberger, and Laurens van der Maaten. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, volume 1, page 3, 2017.

[183] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[184] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[185] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2818–2826, 2016.

[186] MobileNet. Keras implementation of mobile networks, 2018.

[187] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[188] Mahmood Sharif, Sruti Bhagavatula, Lujo Bauer, and Michael K Reiter. Adversarial generative nets: Neural network attacks on state-of-the-art face recognition. *arXiv preprint arXiv:1801.00349*, 2017.

[189] Adam Smith, Abhradeep Thakurta, and Jalaj Upadhyay. Is interaction necessary for distributed private learning? In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 58–77. IEEE, 2017.

[190] Briland Hitaj, Giuseppe Ateniese, and Fernando Pérez-Cruz. Deep models under the gan: information leakage from collaborative deep learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 603–618. ACM, 2017.

[191] Congzheng Song, Thomas Ristenpart, and Vitaly Shmatikov. Machine learning models that remember too much. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 587–601. ACM, 2017.

[192] Chaowei Xiao, Jun-Yan Zhu, Bo Li, Warren He, Mingyan Liu, and Dawn Song. Spatially transformed adversarial examples. 2018.

[193] Warren He, Bo Li, and Dawn Song. Decision boundary analysis of adversarial examples. 2018.

[194] Qi Liu, Tao Liu, Zihao Liu, Yanzhi Wang, Yier Jin, and Wujie Wen. Security analysis and enhancement of model compressed deep learning systems under adversarial attacks. In *Proceedings of the 23rd Asia and South Pacific Design Automation Conference*, pages 721–726. IEEE Press, 2018.

[195] Chaowei Xiao, Bo Li, Jun-Yan Zhu, Warren He, Mingyan Liu, and Dawn Song. Generating adversarial examples with adversarial networks. *arXiv preprint arXiv:1801.02610*, 2018.

[196] Matthew Jagielski, Alina Oprea, Battista Biggio, Chang Liu, Cristina Nita-Rotaru, and Bo Li. Manipulating machine learning: Poisoning attacks and countermeasures for regression learning. *arXiv preprint arXiv:1804.00308*, 2018.

[197] Octavian Suciu, Radu Mărginean, Yiğitcan Kaya, Hal Daumé III, and Tudor Dumitraş. When does machine learning fail? generalized transferability for evasion and poisoning attacks. *arXiv preprint arXiv:1803.06975*, 2018.

[198] Chang Liu, Bo Li, Yevgeniy Vorobeychik, and Alina Oprea. Robust linear regression against training data poisoning. In *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*, pages 91–102. ACM, 2017.

[199] Ali Shafahi, W Ronny Huang, Mahyar Najibi, Octavian Suciu, Christoph Studer, Tudor Dumitras, and Tom Goldstein. Poison frogs! targeted clean-label poisoning attacks on neural networks. *arXiv preprint arXiv:1804.00792*, 2018.

[200] Binghui Wang and Neil Zhenqiang Gong. Stealing hyperparameters in machine learning. *arXiv preprint arXiv:1802.05351*, 2018.

[201] Florian Tramèr, Fan Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Stealing machine learning models via prediction apis. In *USENIX Security Symposium*, pages 601–618, 2016.

[202] Yunhui Long, Vincent Bindschaedler, Lei Wang, Diyue Bu, Xiaofeng Wang, Haixu Tang, Carl A Gunter, and Kai Chen. Understanding membership inferences on well-generalized learning models. *arXiv preprint arXiv:1802.04889*, 2018.

[203] Jinyuan Jia and Neil Zhenqiang Gong. Attriguard: A practical defense against attribute inference attacks via adversarial machine learning. *arXiv preprint arXiv:1805.04810*, 2018.

[204] Milad Nasr, Reza Shokri, and Amir Houmansadr. Machine learning with membership privacy using adversarial regularization. *arXiv preprint arXiv:1807.05852*, 2018.

[205] M Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M Songhori, Thomas Schneider, and Farinaz Koushanfar. Chameleon: A hybrid secure computation framework for machine learning applications. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pages 707–721. ACM, 2018.

[206] Matthew Fredrikson, Eric Lantz, Somesh Jha, Simon Lin, David Page, and Thomas Ristenpart. Privacy in pharmacogenetics: An end-to-end case study of personalized warfarin dosing. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 17–32, San Diego, CA, 2014. USENIX Association.

[207] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. Model inversion attacks that exploit confidence information and basic countermeasures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1322–1333. ACM, 2015.

[208] Mahmood Sharif, Sruti Bhagavatula, Lujo Bauer, and Michael K. Reiter. Accessorize to a crime: Real and stealthy attacks on state-of-the-art face recognition. In *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security*, October 2016.

[209] Guy Katz, Clark Barrett, David L Dill, Kyle Julian, and Mykel J Kochenderfer. Reluplex: An efficient smt solver for verifying deep neural networks. In *International Conference on Computer Aided Verification*, pages 97–117. Springer, 2017.

[210] Bernhard Schölkopf, Robert C Williamson, Alex J Smola, John Shawe-Taylor, and John C Platt. Support vector method for novelty detection. In *Advances in neural information processing systems*, pages 582–588, 2000.

[211] Fabien Letouzey, François Denis, and Rémi Gilleron. Learning from positive and unlabeled examples. In *International Conference on Algorithmic Learning Theory*, pages 71–85. Springer, 2000.

[212] George G Cabral, Adriano LI Oliveira, and Carlos BG Cahu. A novel method for one-class classification based on the nearest neighbor data description and structural risk minimization. In *Neural Networks, 2007. IJCNN 2007. International Joint Conference on*, pages 1976–1981. IEEE, 2007.

[213] George G Cabral, Adriano LI Oliveira, and Carlos BG Cahú. Combining nearest neighbor data description and structural risk minimization for one-class classification. *Neural Computing and Applications*, 18(2):175–183, 2009.

VITA

Shiqing Ma attended Purdue University for his Ph.D. study under the guidance of his advisor Professor Xiangyu Zhang and co-advisor Professor Dongyan Xu from Fall 2013 to Summer 2019. Before that, he earned his Bachelor of Engineering in Software Engineering from the Shanghai Jiao Tong University in Shanghai, China. His research interests lie in computer system security especially system and software security and artificial intelligence security, programming languages and software engineering. His work has been awarded the Distinguished Paper Awards at USENIX Security 2017 and ISOC NDSS 2016. His Ph.D. research has been partially funded via the Purdue Bilsland Dissertation Fellowship in the academic year of 2018-2019. In the Fall of 2019, he will join the Department of Computer Science at Rutgers University to be an Assistant Professor.