FORENSICS AND FORMALIZED PROTOCOL CUSTOMIZATION FOR ENHANCING NETWORKING SECURITY

by

Fei Wang

A Dissertation

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Doctor of Philosophy

Department of Computer Science

West Lafayette, Indiana

December 2021

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. Xiangyu Zhang, Chair

Department of Computer Science

Dr. Dongyan Xu

Department of Computer Science

Dr. Ninghui Li

Department of Computer Science

Dr. Bharat Bhargava

Department of Computer Science

Approved by:

Dr. Kihong Park

To my loving and beloved family members.

ACKNOWLEDGMENTS

First, my greatest thanks goes to my advisor Professor Xiangyu Zhang for his support of my PhD study and research. His patience, understanding, guidance, immense knowledge and research passion drive me here. He is a self-motivated researcher with great personal charisma, enlightening and inspiring his students. Learning from him has significantly improved my academic skills in multiple perspectives.

I would like to give my special thanks to Professor Dongyan Xu, my co-advisor. He has been consistently offering a critical eye on my research justification. I greatly appreciate his generous help and suggestions in my PhD career.

My sincere thanks goes to Professor Ninghui Li and Professor Bharat Bhargava, who also serve as my thesis committee, for their insightful comments and suggestions to improve my research and dissertation.

I dedicate this dissertation to my loving and beloved family members. Without their unconditional support and love, I could not get through the hard times in my life, especially for the COVID-19 pandemic. My appreciation and love to my parents, Hanping Wang and Ruidi Zhuang, my parents-in-law, Xingjiang Xiao and Liqin Yang, my wife and the love of my life, Wenfei Xiao, and my newborn baby, Yichen Wang are always beyond my words.

Last, it is indeed my fortune to work with great colleagues such as Yonghwi Kwon, Kexin Pei, Jianliang Wu, Weihang Wang, Shiqing Ma, Wen-chuan Lee, Yuhong Nan, Wei You, Yapeng Ye, Zhuo Zhang, Yousra Aafer, Hongjun Choi, I Luk Kim, Chung Hwan Kim and Rohit Bhatia, and my roommates Chengyuan Lin and Xilun Wu. They all helped me a lot in my past seven years in West Lafayette.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Comprehensive networking security is a goal to achieve for enterprise networks. In forensics, the traffic analysis, causality dependence in intricate program network flows is needed in flow-based attribution techniques. The provenance, the connection between stealthy advanced persistent threats (APTs) and the execution of loadable modules is stripped because loading a module does not guarantee an execution. The reports of common vulnerabilities and exposures (CVE) demonstrate that lots of vulnerabilities have been introduced in protocol engineering process, especially for the emerging Internet-of-Things (IoT) applications. A code generation framework targeting secure protocol implementations can substantially enhance security.

A novel automaton-based technique, NetCrop, to infer fine-grained program behavior by analyzing network traffic is proposed in this thesis. Based on network flow causality, it constructs automata that describe both the network behavior and the end-host behavior of a whole program to attribute individual packets to their belonging programs and fingerprint the high-level program behavior. A novel provenance-oriented library tracing system, Lprov, which enforces library tracing on top of existing syscall logging based provenance tracking approaches is investigated. With the dynamic library call stack, the provenance of implicit library function execution is revealed and correlated to system events, facilitating the locating and defense of malicious libraries. The thesis presents ProFactory, in which a protocol is modeled, checked and securely generated, averting common vulnerabilities residing in protocol implementations.

1. INTRODUCTION

Comprehensive networking security is one of the most critical application issues for enterprise environment. There is an increasing demand on more advanced forensic analysis techniques because of the evolving complexity of cyberattacks, such as APT attacks. Attackers can intrude giant companies or governments, and spread the malice by leveraging software vulnerabilities or well-crafted social engineering tricks. The attack can be stealthy, where malware may lurk in the network for several months or even years until it is triggered. Also, the attack can be launched from an insider who can bypass all the security gatekeepers configured to prevent intrusion from outsiders. Therefore, to detect or intercept those attacks at their entry points is particularly challenging. Even if a malicious action is immediately detected, it is still challenging to understand the root cause or the ramifications. Hence, in addition to the significant contribution manifested in attack detection, forensic analysis becomes an irreplaceable pillar in enterprise environment. The analysis can be done on the network gateway (traffic analysis) and the end host (system provenance). Existing traffic analysis techniques have accuracy concerns because their working largely relies on flow-based traffic recognition, which overlooks internal program semantics that are well reflected in causally correlated network flows. Mainstream provenance tracking tools always treat loadable modules as regular data-based system objects (e.g., files and sockets), which misses the execution provenance in loadable modules because module loading does not guarantee code execution.

Security in fundamental network connection is attracting more and more research attention as the emerging IoT applications propose and implement lots of new protocols every year. Recent CVE reports show that such fast-paced shipment of specifications and implementations have produced massive vulnerabilities, of which the majority are related to incorrect message parsing. Fixing those vulnerabilities is usually impossible after product shipment and they are concrete threats to individual users and enterprise networks. This motivates a code generation framework which formalizes protocol specifications/models and produces secure protocol implementations.

In my research, I work on solving system security problems [1]–[4]. I will present my work on fine-grained system forensics [5] and formalized protocol customization [6] to help enhance networking security.

1.1 Dissertation Statement

This dissertation addresses important issues in research of system forensics and protocol customization to comprehensively enhance networking security for enterprise environment. First, it proposes NetCrop, a new traffic analysis tool leveraging causal dependency between flows to improve the accuracy in inferring fine-grained program behaviors. Second, it proposes Lprov, a library-aware provenance system that combines library tracing and audit-based provenance tracking to identify malicious library execution. Last, it proposes ProFactory, a code generation framework which produces secure protocol implementations by DSL-based protocol customization and automated verification.

The thesis of this dissertation is as follows: comprehensive enhancement of networking security can be achieved by fine-grained system forensics, i.e., causality-aware behavior inference and library-aware provenance tracing, and formalized protocol customization.

1.2 Contributions

The contributions of this dissertation are as follows:

- We propose a novel causality-aware network-level program and program activity fingerprinting system NetCrop, without requiring any modification on end-user systems during production run. It runs an instruction-level program analysis tool to automatically infer program communication patterns which represent causality and dependency between network flows. It leverages a practical automaton extraction algorithm to construct a tree-like network automaton for each program from network traces. Further, we develop a trace partitioning and alignment method that maps program activities to automaton states. and a highly efficient graph-matching algorithm to parse network traffic with the automata on-the-fly, attributing network flows to program owners and disclosing corresponding end-host actions. The experiments show that NetCrop is

highly effective in popular network programs, outperforming existing traffic analysis tools.

- We propose an efficient provenance tracking system LPROV, combining library tracing in user space and syscall tracing in kernel space. Whenever a provenance-related syscall is made from a thread, its full library-level execution path is also unveiled. Equipped with the library provenance, causality is revealed not only between explicit value-based input and output system entities but also inside the implicit fine-grained execution-based shared libraries. It runs on a lightweight and efficient system-wide library tracing infrastructure. The evaluation demonstrates that it can precisely identify the provenance in malicious libraries which are missed by the state-of-the-art, and the performance overhead is competitive.

- We propose PROFACTORY, a novel system that realizes efficient and secure protocol customization. In PROFACTORY, developers formally and unambiguously model protocols in a DSL instead of natural languages. Symbolic model checking is performed to verify the model correctness. Then, the model is fed to the code generation engine to produce kernel-oriented protocol implementation which provides guarantees of being free from memory safety vulnerabilities in message parsing and from concurrency control vulnerabilities in message multiplexing. Such guarantees are provided by the automatically generated sanity checking code, such as bound checks and input validation checks, and by applying automated verification tools to the generated code. The experiments demonstrate that PROFACTORY can help to avert the majority of existing protocol vulnerabilities.

1.3 Dissertation Organization

This dissertation is organized as follows: Chapter 2 discusses the design, implementation and evaluation of NETCROP, which is a new causality-aware behavior inference system. Chapter 3 presents LPROV, which is a lightweight and effective library-aware provenance

tracking system. In Chapter 4, we will discuss ProFactory, a formalized protocol customization framework aiming to generate secure protocol implementations.

1.4 Dissertation Overview

1.4.1 Fine-grained Forensic Analysis in Enterprise Environment

In enterprise environment, forensic analysis can be executed on either the front line, the traffic analysis, or the back line, the end-host system provenance. Traffic analysis is less intrusive and it applies classifiers or clusters to infer and log suspicious behaviors for further investigation. End-host system provenance deploys intrusive logging tools, collecting system-wide execution context to perform backtracking on particular system events or objects. Existing traffic analysis and system provenance tools usually suffer from the low accuracy, and my techniques try to resolve this issue.

Causality-aware Behavior Inference

Substantial research efforts have been taken on traffic classification in the past decade. The majority of them rely on learning-based methods seeking recognizable patterns in packet sequences of individual flows, and apply classification or clustering techniques. However, the flow-based protocol disclosure alone would not be sufficient in handling many upcoming challenges in complex contemporary enterprise network environments. For example, accurate session discovery requires the analysis of programs' flow causality and semantics, rather than protocol identification. Hence, to improve situation awareness in traffic analysis, I propose NetCrop which infers fine-grained program activities by analyzing network flow dependency. It answers (1) what program generates the packets and (2) what activities those packets represent in the program. It utilizes dynamic analysis to infer causality between network flows, extracts annotated (by program activity/semantic) automaton model from programs' execution traces, partitions network traces to establish the mapping between automaton states and network flows, and attributes traffic to annotated automaton states to disclose program behaviors. The experimental results show that NetCrop is highly

effective with over 90% precision and recall in both traffic attribution and activity inference, outperforming existing techniques.

Library-aware Provenance Tracing

Most of existing provenance techniques are operating on system event auditing that discloses dependence relationships by scrutinizing syscall traces. Unfortunately, such auditing-based provenance is not able to track the causality of the shared libraries. Different from other data-only system entities like files and sockets, dynamic libraries are linked at runtime and may get executed, which poses new challenges in provenance tracking. For example, library provenance cannot be tracked by syscalls and mapping; whether a library function is called and how it is called within an execution context is invisible at syscall level; linking a library does not promise their execution at runtime. To address such challenges, I develop LPROV which combines library tracing and provenance tracking. Upon the beginning of a program execution, LPROV is loaded into process memory by a customized loader. It records the entrance and exit of library calls by manipulating symbol tables and maintains library call stacks for each thread. To be integrated with the audit logging techniques, LPROV also deploys a kernel module to collect syscall events. During production runs, when a syscall is made, its deriving path from the library perspective is disclosed by the library call trace on causality correlations. The experimental results show that it can successfully identify malicious libraries that are missed by the state-of-the-art with reasonable overhead.

1.4.2 Generating Secure Protocol Implementations

Existing research in protocol security reveals that the majority of disclosed protocol vulnerabilities are caused by incorrectly implemented message parsing and network state machine. Instead of testing and fixing those bugs after development, which is extremely expensive (especially for the emerging IoT devices), we would like to avert them upfront. For this purpose, I propose PROFACTORY which formally and unambiguously models a protocol, checks model correctness, and generates secure protocol implementation. In PROFACTORY, a protocol is modeled/customized in a protocol-oriented DSL to eliminate specification confu-

sion. The model is then formally verified and used to emit secure protocol implementations. The evaluation shows that it can help to avert 82 known vulnerabilities.

2. NETCROP: FINE-GRAINED PROGRAM ACTIVITY INFERENCE ASSISTED BY CASUAL DEPENDENCY ANALYSIS BETWEEN NETWORK FLOWS

The increasing complexity of today's Internet is continuously motivating the development of more intelligent network analysis techniques to reveal semantic information from such highly complex network traffic. We introduce NetCrop, a novel automaton-based technique to fingerprint end-host program activities by observing network traffic only. Automata are constructed through a program-analysis-assisted training phase in which both of network traffic and program activities are profiled and correlated. Each automaton depicts an enriched context associating a program's network communication patterns and activity characteristics, improving the network-level causality awareness which is missed in most of existing flow-based traffic analysis tools. During production runs, automata are deployed on network gateways, attributing traffic to individual programs and activities. In particular, program activities are automatically fingerprinted as part of the automata parsing. The evaluation on a set of common and popular networking programs including instant messaging, P2P sharing and email service shows that NetCrop is highly effective. It can attribute traffic with over 90% precision and recall. It can also fingerprint program activities with over 95% precision and recall. The experimental results on a large scale real-world enterprise data set show that NetCrop has more than 79% recall with less than 0.03% false positives.

2.1 Introduction

Attributing network traffic to programs and their activities is a critical task. It is an enabling technique for network intelligence that matters the most in many research areas such as overlay network, software defined network and next generation Internet. For many years, the demands of improving quality of network data collection have continued to increase, mainly due to the ever increasing complexity of modern network applications and their communication. In particular, such complex and sophisticated network environment motivates the development of a fine-grained network packet attribution technique that has

a wide range of applications in an industry setting. For Internet Service Providers (ISPs), traffic origin awareness greatly facilitates various network management tasks, such as Quality of Service (QoS) management, dynamically adjusting network routing strategies and maintaining network hierarchies [7]–[9]. Furthermore, it is also important for enterprise network administration as it allows better network status understanding, host behavior profiling, customized security monitoring and accurate network intervention. For instance, the classification technique deployed inside Cisco's NetFlow system serves as a fundamental tool for capacity planning, behavior monitoring and anomaly detection [8], [9].

There have been substantial research efforts on traffic classification in the past decade. The majority of them hinge on learning-based methods [10] that look for statistical patterns in packet sequences, and apply classification or clustering techniques to attribute traffic. Deep Packet Inspection (DPI) is an alternative approach which tries to search for specific keywords and strings in packet payloads [11], [12]. Such keywords and strings are utilized as unique signatures in distinguishing corresponding applications. These techniques can achieve high accuracy in recognizing commonly used application-layer protocols including HTTP, HTTPS, SMTP, POP3, FTP, SSH and even P2P [7], [12]–[14]. For some protocols that adopt port masquerade and payload encryption/encoding/obfuscation, they exploit intriguing flow features to reveal protocol usage [15]. However, flow-based protocol disclosure alone would not be sufficient in handling many upcoming challenges in complex contemporary enterprise network environments. For example, accurate session discovery requires the analysis of programs' flow causality and semantics, rather than isolated protocol identification. While a straightforward method is to deploy a comprehensive host logging system to record all the host events, such logging systems are often too expensive and impractical for production runs when considering their performance issues. For example, Linux Audit Logging incurs high runtime overhead and produces large sizes of logs [16]. Although Event Tracing of Windows (ETW) offers an efficient tracing infrastructure, it must be coupled with a sufficiently large buffer and a high-performance log consumer daemon to prevent event missing [17], which often exceeds the processing capacity of an end-host. Hence, to improve situation awareness without using intrusive approaches (e.g., host instrumentation), a new approach aiming to disclose traffic causality should be applied to perform fine-grained traffic attribution.

System Goals and Motivation Fine-grained traffic attribution is expected to answer **Q1:** *what program generates the packets* and **Q2:** *what activities those packets represent in the program*. Knowing the exact programs but not only the isolated protocols running is quite meaningful, especially for security concerns. Since some programs can have newly disclosed vulnerabilities, administrators should assist patching them timely. More importantly, it would also serve as a pillar component of anomaly detection, where examining all the connections equally would cause a large number of false alarms. Nevertheless, by precluding traffic generated by known/benign programs, it can assist narrowing down the detection scope to those highly suspicious ones, substantially improving the performance.

Inferring precise program activities from network traffic is beneficial for privacy awareness because it pre-informs program users of how many behavior details could be leaked in the sense of networking profiles. In particular, if such behaviors are related to sensitive information exchange like file transfer, they could also be leveraged for forensic inference when security incidents such as insider information leakage happen. Also, it allows the administrator of an enterprise network to understand if any suspicious program operations are being performed by an employee/guest such as disallowed file transfer on instant messaging applications, which enlightens the enforcement of fine-grained (per-activity per-program) enterprise network policies.

Note that there exist research efforts in addressing the issue of fine-grained program activity inference. However, due to the lack of traffic causality inference, they either work in a small set of programs [18] or suffer from unsatisfactory accuracy [19].

Our Solution To this end, we propose NetCrop that can attribute the traffic of a host to the belonging programs and fingerprint the specific program activities by monitoring *network traffic only*, where **Q1** and **Q2** are answered separately. Specifically, in addition to recognizing statistical patterns inside packet streams like what has been done in most previous works, NetCrop pays more attention to *orders and causal relations* among network flows and constructs *network traffic models* by correlating network traffic and end-host actions (during the training phase).

Our contributions are summarized in the following:

- We propose NetCrop, a novel network-level program and program activity fingerprinting system without requiring any modification on end-user systems during production run.
- We develop an instruction-level program analysis tool to automatically infer program communication patterns which represent causality and dependency between network flows.
- We develop a practical automaton extraction algorithm, constructing a tree-like network automaton for each program from network traces. Further, we develop a trace partitioning and alignment method that maps program activities to automaton states.
- We develop a highly efficient graph-matching algorithm to parse network traffic with the automata on-the-fly, attributing network flows to program owners and disclosing corresponding end-host actions.
- We evaluate NetCrop on 14 popular Windows applications including instant messaging, P2P sharing and e-mail service programs. The experimental results show that NetCrop is highly effective with over 90% precision and recall in both traffic attribution and activity inference. The experiment on a large scale real-world industry data set shows that NetCrop has over 79% recall with less than 0.03% false positives.

2.2 Approach Overview

Figure 2.1 presents the architecture of NetCrop. It consists of three phases: *Application Training*, *Automata Extraction*, and *Traffic Attribution*.

Application Training Phase (Offline) Given a subject program, NetCrop first executes the program with an analysis facility and a tracing tool. As such, we obtain multiple samples of syscall traces and network traces which include information of TCP/UDP types, local/remote host ports, remote IP addresses, remote domain names, packet lengths, packet payload bytes and inter-packet delays. Meanwhile, program's distinctive communication patterns are worked out by inter-flow causal dependency analysis.

Automata Extraction Phase (Offline) In this phase, NetCrop constructs automata from programs' network traces by leveraging syscall traces and communication patterns. Automaton inference in traces is usually formulated as recognizing regular grammars from strings in languages which is, unfortunately, a well-studied NP-complete problem [20]. Most efficient probabilistic algorithms, e.g., homing sequence and diversity-based inference [21]–[23], require the full set of input symbols and the access to the oracle of counterexamples. However, in our case, such requirement cannot be satisfied due to the non-enumerable network parameters such as domain names and the lack of counter-traces for reference. Besides, there have been many studies in specification inference from network traces [24]–[36]. Unfortunately, they mainly concentrate on message format inference, individual stateful protocols or simplex IoT applications hence they are not suitable for whole program models which are considered more intricate. In this paper, enlightened by [37], NetCrop extracts *approximate yet practical tree-structured automata* by repeatedly finding common sequences among traces rather than computing precise automata for programs. More details on the automata construction can be found in Section 2.3.1 and Section 2.3.2.

Once we obtain the automata, we identify the program activities associated with network flows by partitioning and aligning syscall traces. Then, we annotate the corresponding automaton states with high-level program activities. Note that while many efforts have been taken in identifying behaviors inside program execution or function call traces, which can be applied in anomaly and malware detection [38], [39], our goal is different as we map host-end program execution to a network automaton model which is more fine-grained. Specifically, during production run, when network traffic is parsed by the model, the corresponding program activity is also disclosed. The details are dissected in Section 2.3.2.

Traffic Attribution Phase (Online) During production run, the *annotated automata* of all the programs under monitoring are deployed on the network (e.g., on a gateway). Note that NetCrop does not require any additional instrumentation/modification/installation of applications on end hosts in this phase. NetCrop uses a highly efficient graph-matching algorithm to attribute packets to their owners. During the attribution, conflicts in state transitions during the packet attribution might happen due to the complexity of modern

network communication. NetCrop resolves the conflicts by defining additional transition restrictions and matching rules. Details can be found in Section 2.3.3.

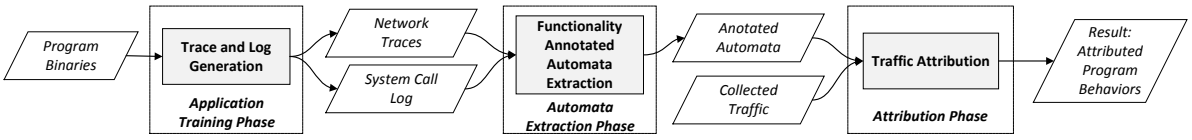

Figure 2.1. The architecture of NetCrop

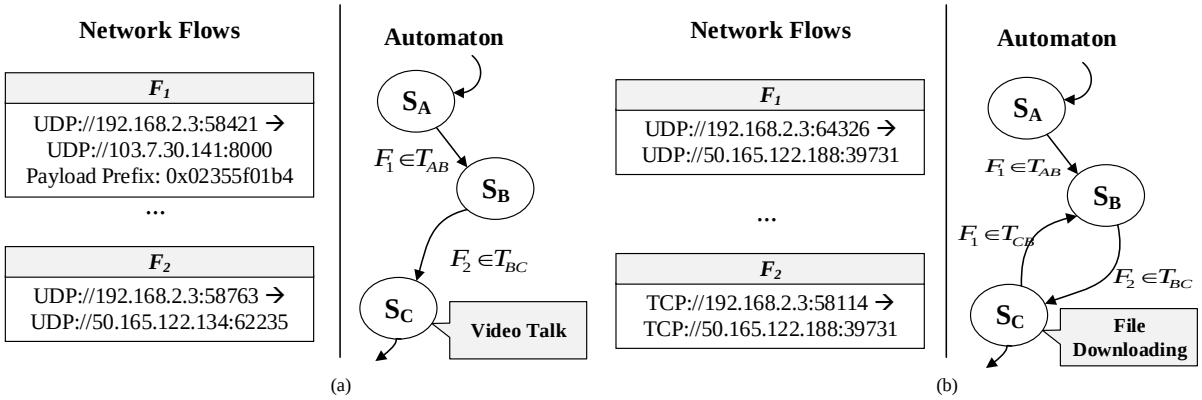

Figure 2.2. NetCrop working examples for QQ and uTorrent

Illustrative Examples Figure 2.2 (a) illustrates a communication segment captured on the local gateway. In particular, F_2 is a UDP flow between a local host and a remote IP. In this example, we want to answer **Q1** and **Q2** from F_2. When analyzed independently, as most existing flow-based classification approaches do, it is recognized as a P2P flow. However, such information is rather misleading and insufficient for our purpose as indeed the flow is generated by *Tencent QQ* which is not a typical P2P program. In contrast, NetCrop is capable to disclose a program which generated the flow as well as its particular activity via automata-based network analysis. Specifically, in the right side of Figure 2.2 (a), S_A, S_B and S_C are three states in the automaton of *Tencent QQ*, extracted by NetCrop through offline training and program analysis. Between the states, T_{AB} and T_{BC} are the acceptable driven condition sets for the corresponding transitions. The model specifies that if a UDP

flow F_1 (with a specific remote IP and binary payload prefix in its first packet) occurs before F_2, F_1 and F_2 can trigger the transitions $S_A \rightarrow S_B$ and $S_B \rightarrow S_C$ respectively. It is also the only transition path for the automata of all the currently running programs. Hence, we know the traffic must belong to *Tencent QQ*. Moreover, the annotation on S_C tells us that the local user is engaged in a video talk.

Figure 2.2 (b) shows the network communication in *uTorrent*, a P2P program, and part of its network automaton. The isolated TCP flow F_2 is identified as a P2P connection. We observe a UDP flow F_1 happen before F_2 and $S_A \rightarrow S_B \rightarrow S_C$ in the automaton of *uTorrent* is the only possible transition path at that moment, then we attribute the two flows to *uTorrent* and infer from the label on S_C that it is downloading a file. Note that the automata in this work are different from those of individual networking protocols such as FTP and HTTP. They are *global* automata that describe whole program activities which often consist of traffic in multiple connections and protocols.

2.3 System Design

In this section, we discuss the design of NetCrop in details. In the program training phase (Section 2.3.1), NetCrop bootstraps each target program, logs the network and syscall traces generated during execution, and recognizes communication patterns. In the automaton extraction phase (Section 2.3.2), the network traces are parsed to extract the fields of interest and denote them with symbols. Then, an automaton which intuitively describes the communication model of the whole program is derived from the symbols and their transitions, and then the automaton is annotated with program activities by syscall traces. In the traffic attribution phase (Section 2.3.3), the annotated automaton is used for on-the-fly network traffic parsing.

2.3.1 Application Training

Trace and Log Collection In offline, a subject application is launched on a machine set up beforehand, and the network and syscall (with arguments) traces produced during execution are collected. Then, we specify a set of predefined behaviors (triggered by user-

side input) which we want to fingerprint later such as instant messaging, video talk, and file transfer. To this end, we selectively trigger program behaviors through automated testing scripts [40], [41] or/and manual interactions. Due to load balancing (Content Delivery Network) or other non-deterministic factors, each time a program executes, it may interact with different remote servers. To tolerate such non-determinism, the program is tested for multiple times with various configurations (e.g., different user accounts and locations). We leverage a tool [41] that can record and replay user-inputs (e.g., keystrokes and mouse clicks) to automate the process.

Network Traces Preprocessing We use a network flow as a basic unit in our automata because we are focusing internal states of the whole-program execution but not a single protocol/flow. First, all the packets generated during the training (of a program) are abstracted to flows. In particular, a *network flow* comprises of one or more packets between a pair of hosts (e.g., local and remote hosts) on a pair of ports. As DNS traffic often includes important semantics related to program behaviors, a DNS request and its response are also considered a network flow. Specifically, all the DNS resolution requests and responses are recorded and the pairs of the domain name and the remote IP are extracted from them. Then, we order the flows by the timestamps of the first packets of the flows. Figure 2.3(a) shows traces from Skype, Outlook and uTorrent, where a TCP or UDP flow is denoted as a tuple {local port, remote IP, remote port, domain name (if applicable)}.

Next, we discuss how we annotate the fields of network flows based on the program execution context. The annotations below are directed by the communication patterns discovered in a prior automated dynamic inter-flow causal dependence analysis. They are highly scalable since we are only required to rerun the automated program analysis facility to enrich the pattern (annotation) set when new programs are added. The inter-flow dependence analysis is conducted in flows' four meta-fields, local port, remote IP, remote port and domain name. For each field in a flow, we perform backward reaching-definition analysis [42] just before the flow (socket) is actually created, figuring out the data-dependence roots. Following that, we also apply forward taint analysis [43] and constant propagation [44] on those computed roots. As such, the full set of inter-flow dependence and corresponding communication patterns can be revealed:

• *Protocol Type* There are three types of protocols: TCP, UDP, and DNS. TCP flows are annotated by `T` while UDP flows are annotated by `U`. Among UDP flows, DNS related flows are denoted by `D`. For DNS flows, we consider four different categories of DNS flows and append additional annotation to the flows: `A (AAAA)`, `SRV`, `MX` and `TXT`. Other DNS types are precluded since they are rarely used in practice. Interested readers may refer to related RFC documents [45], [46].

• *Domain Name* Depending on how the program chooses and makes use of the domains, we label domain names in DNS flows into four categories: `CONST`, `ANY`, `SIMILAR` and `DEPEND`. `CONST` means the program uses the same domain in different runs. Such domain names are often hard coded in program binaries. Some programs connect to the domain names plumbed through user input (e.g., FTP, SSH, and email client software) which may change from run to run. We hence use `ANY` for such cases. `SIMILAR` describes the domain names derived from user's input with shared postfix. For example, Outlook 2013 allows a user to manage his/her email account by typing in the account name. Then, Outlook will test the connection to the domain and also subdomains prepended with strings such as "imap", "pop3", "pop" and "mail", and the domain name leading to a successful connection will be used afterward. The Outlook example in Figure 2.3-Outlook(a) shows the flows for configuring Gmail account, in which *imap.gmail.com* is used after the 5 DNS requests. Therefore, the 5 DNS flows are grouped into a flow annotated with `SIMILAR` and the five prefixes (we use "VOID" if the prefix is an empty string). `DEPEND` describes the domains resolved from `SRV` or `MX` DNS flows. The `MX` and `SRV` records are used to describe the host name of mail server and other services (e.g., web service) under a requested domain name.

• *Local Port* In practice, local ports in most programs are dynamically determined at runtime. This is because programs need to support the adoption of network address and port translation (NAPT) techniques by routers and gateways. Also, by dynamically assigning local ports, program can operate correctly even if a particular local port is already occupied by another program. Hence, we abstract a local port to one of the following two types: `ANY` and `UPNP`. Mostly, a local port is randomly allocated by operating systems inside syscalls, and hence we use `ANY` to denote this type. For P2P applications, a port is propagated through a UPnP packet for peer discovery. In particular, a program reserves a unique port number

for both data sending and receiving, and the peers communicating through this port. For instance, Skype reserves a port 8820 for peer communication in Figure 2.3-Skype (a). We use `UPNP` to denote such type of port usage rather than a concrete port number. Note that when UPnP is prohibited, NetCrop adds additional states which represent configuration flows into the automata. More details are in Section 2.4.1.

• *Remote IP* We use five different types to annotate a remote IP in TCP and UDP flows: `D-CON`, `D-DYN`, `ANY`, `DEPEND` and `CONST`. For the first two types (`D-CON` and `D-DYN`), a remote IP is resolved from the prior DNS request regarding domain names. If the domain names are immutable (i.e., CONST type), we annotate the remote IP with `D-CON`. Otherwise, the domain names is determined by the user or the remote server (i.e., `ANY`, `SIMILAR` and `DEPEND` for the domain name field), and we annotate the remote IP with `D-DYN`. We use the `ANY` annotation in flows whose local port is `UPNP` or the IP is directly from user input (e.g. FTP and SSH client). `DEPEND` is used when the remote IP is assigned from (or propagated from the same root with) a previous flow whose remote IP is `ANY`. For instance, the initiated control connection is followed by data transmission connections at the same IP in FTP. `CONST` is used to annotate all the other flows, either propagated from binary constants or extracted from incoming payload (excluding UPnP).

• *Remote Port* A remote port is annotated with either `CONST`, `ANY` or `DEPEND`. The ports less than 1024 or defined by binary constant are `CONST`, and all the others are `ANY`. Further, if a subset of `ANY` ports are defined by the same dependence root, the following ports are `DEPEND` on the first one.

Summarizing Repeated Flows After field annotations, we further identify the flows that repeatedly occur and summarize them to make the annotated flow set more compact. We observe some applications issue the same set of network requests (in different protocols) multiple times to increase the chance of acquiring successful connections. One such example is uTorrent (also shown in Figure 2.3-uTorrent(a)). It attempts to first set up UDP connections to peers. It then tries to connect to the same remote IP and port under TCP. In such cases, we will see a flow "copies" the remote settings of the previous flow and we annotate this type of flow with the `COPY` annotation and the previous flow with `COPIED` (we differentiate this with the IP dependency in FTP). If several flows share the exactly same annotation in

every field, only one flow is retained, with the `MULTIPLE` annotation. Figure 2.3 illustrates the original flows and annotated flows for three applications: Outlook, Skype, and uTorrent.

Constant and Dependency The fields annotated by `CONST` or `D-CON` are also assigned the concrete constant strings. The order numbers of flows being depended are attached on the flows owning fields dependent on them (e.g. `DEPEND` and `D-DYN`). Note that from our experience in above field annotation strategy, no flows are dependent on more than one flow and no conflict of dependency relationship exists.

2.3.2 Automaton Extraction Phase

We derive the network automaton from the enriched network flows after the network trace preprocessing (Section 2.3.1). The information retained within a single flow (intra-flow features) and the transitions between flows (inter-flow ordering and causalities) are both leveraged to construct the automaton. We run a program multiple times to increase the coverage of traces related to network behaviors, which may also lead to uncommon network flows. Unfortunately, if the uncommon and common flows are treated equally, the chances of successful matching will be reduced. Hence, we identify the common flow sequences and make them the core of the automaton. Resolving this problem is similar to finding a solution for MLCS (multiple longest common sequence) problem, which is NP-complete. Thus, we apply the NBMAS approximate algorithm proposed in [47] for cost-effective computation.

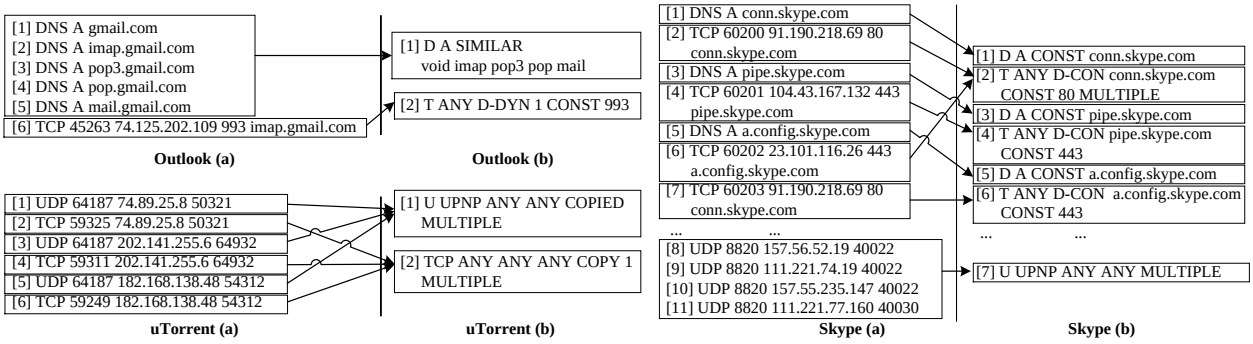

Figure 2.3. Samples of original and annotated flows.

Like a regular automaton, the network automaton we construct consists of *states*, *symbols* and *transition functions*, where a state by our definition is used for separating symbols and

describing program behavior. Different from regular automata, there is no accept state in our design since the application could terminate at any time. A symbol is a network flow and a state receiving one expected flow will proceed to the next state directed by the corresponding transition function. Below we elaborate the algorithm for constructing network automaton:

Automata Extraction Algorithm The outline of the algorithm is described in Algorithm 1 and it can be summarized in three steps:

Step 1. Initialize an automaton and add a start state q_1 into the set of states Q. The set of input symbols U and transition functions F are both set to ϕ.

Step 2. Compute the longest common sequence from multiple network traces collected from a program, where a formatted flow symbol could be treated as a character. For the longest common string $S_C = s_{c_1}s_{c_2}...s_{c_k}$, we add k states $q_{c_1}q_{c_2}...q_{c_k}$ into Q, add k input symbols $s_{c_1}s_{c_2}...s_{c_k}$ into U and add k transition functions $f_{c_1} : (q_1, s_{c_1}) \rightarrow q_{c_1}, f_{c_2} : (q_{c_1}, s_{c_2}) \rightarrow q_{c_2}, ..., f_{c_k} : (q_{c_{k-1}}, s_{c_k}) \rightarrow q_{c_k}$ into F. Delete the elements in S_C from all the traces. Repeat this step until there are no common flows among the remaining traces.

Step 3. For each flow s_r left in traces, we add a state q_r into Q, an input symbol s_r into U and a transition function $f_r : (q_1, s_r) \rightarrow q_r$ into F.

We take additional steps for several kinds of flows. During the automaton extraction, the flows depending on other flows are precluded and we introduce a branch for each of them on the corresponding states after the above steps are accomplished. For a `MULTIPLE` flow, we introduce a loop on its origin state, including the flows that depend on it. Figure 2.4 illustrates this algorithm. For simplicity, we use letters to denote different flows and only consider two traces. In the example, each longest common sequence (in circle) is appended to the start state as a chain and the remaining flows (uncommon ones) are appended to the start state directly, and each leads to a single state (in diamond). Since flow C is marked by `MULTIPLE`, a loop on Q_3 is added. Then, the `DEPEND` flow (M) is added and linked to the flow C on which its meta-fields depend. A new state Q_9 (in hexagon) is added as well.

Each chain reflects the partial order of the underlying network interactions of the fingerprinted program. Besides, we still need to establish the order of individual chains. For the example, there are three common sequences, $\{ACG\}$, $\{B\}$ and $\{D\}$, and they are all connected to the start state. According to the automaton, the start state should accept the

Algorithm 1: Automata Extraction

Input: m network flow sequences $S_{\mathrm{i}} = s_{\mathrm{i}_1} s_{\mathrm{i}_2} ... s_{\mathrm{i}_{\ell_{\mathrm{i}}}}, \ell_{\mathrm{i}} = Length(S_{\mathrm{i}}), 1 \leq \mathrm{i} \leq m$

Output: An automaton $\mathbf{A} = \{Q, U, q_1, F\}$, where Q is the set of states, U is the set of input symbols, q_1 is the initial state and $F : Q \times U \rightarrow Q$ is the set of transition functions

$\mathbf{A} = \{q_1\}$, $scnt \leftarrow 1$, $U \leftarrow \phi$, $F \leftarrow \phi$ //$scnt$ is the amount of states
Function `Update(`q`,`s`)`
  $scnt \leftarrow scnt + 1$
  $Q \leftarrow Q \cup \{q_{scnt}\}$
  $U \leftarrow U \cup \{s\}$
  $F \leftarrow F \cup \{f_{scnt} : (q, s) \rightarrow q_{scnt}\}$
end
while $\bigcap_{\mathrm{i}=1}^{m} S_{\mathrm{i}} \neq \phi$ **do**
  $S_c \leftarrow MLCS(S_1 S_2 ... S_m)$
  for k from 1 to ℓ_{S_c} **do**
    for t from 1 to m **do** Delete s_{c_k} from S_t
    if $k == 1$ **then** $Update(q_1, s_{c_k})$
    else $Update(q_{scnt}, s_{c_k})$
  end
end
for k from 1 to m **do**
  for each s in S_k **do** $Update(q_1, s)$
end
return A

symbols B and D, which is not true since A precedes D and $\{A, C\}$ precede B in the traces. In other words, Q_1 should not lead to Q_5 or Q_6 in one step. To correct the automaton, we introduce a *pre-set* and a *post-set* for each state to enforce such prerequisite transitions. A state can only be reached after all the states in its *pre-set* are reached and a state can no longer be visited after any state in its *post-set* is reached. The algorithm for *pre-set* and *post-set* computation is shown in Algorithm 2 and it has two steps which are summarized below:

Step 1. For each state q_t on the LCS that is not the start state, we assume that the flow leads to q_t is s. We traverse backward along each chain and examine every flow s' other than s. If s' precedes s in all collected traces that contain both s and s', we include state q that accepts s' into the *pre-set* of q_t.

Algorithm 2: *Pre-set* and *Post-set* Computation

Input: Any state $q_t \neq q_1$ in the state set Q, the set of LCS chains C and the m network traces S used in Algorithm 1

Output: The Pre-set Pre_{q_t} and Post-set $Post_{q_t}$ of q_t

Assume the flow triggering the transition to q_t is s $Pre_{q_t} \leftarrow \phi$,$Post_{q_t} \leftarrow \phi$
for each $C_{\mathrm{i}} = s_{\mathrm{i}_1} s_{\mathrm{i}_2} ... s_{\mathrm{i}_{\ell_{\mathrm{i}}}}$ *in* C **do**
  for j from ℓ_{i} to *1* **do**
    if The partial order $< s_{\mathrm{j}}, s > \in \forall S_k$, where $s_{\mathrm{j}} \in S_k$ and $s \in S_k$, $1 \leq k \leq m$ **then**
      Assume s_{j} triggers the transition to q $Pre_{q_t} \leftarrow Pre_{q_t} \cup \{q\}$
      break
    end
  end
  for j from *1* to ℓ_{i} **do**
    if The partial order $< s, s_{\mathrm{j}} > \in \forall S_k$, where $s_{\mathrm{j}} \in S_k$ and $s \in S_k$, $1 \leq k \leq m$ **then**
      Assume s_{j} triggers the transition to q $Post_{q_t} \leftarrow Post_{q_t} \cup \{q\}$
      break
    end
  end
end
return $Pre_{q_t}, Post_{q_t}$

Step 2. From each state q_t on the LCS, we traverse forward along each chain and examine every flow s'. If s precedes s' in all collected traces that contain both s and s', we include state q that accepts s' into the *post-set* of q_t.

Note that by constructing *pre-set* and *post-set*, and enforcing the associated rules, the order between states can be precisely determined. In addition, it reduces the chance of *transition conflicts* (i.e., one flow is observed from multiple programs) when multiple programs of interest run concurrently. To notice, *pre-set* or *post-set* only contain states in common sequence chains to prevent automata from getting stuck in endlessly waiting in the presence of rare flows.

Program Activity Annotation Once NetCrop obtains the automata, syscall traces are leveraged to denote relations between network interactions and program activities. During the annotation, we mainly focus on three kinds of syscalls: network-related, I/O-related and message-related. First, network-related syscalls (e.g., `send`) are utilized to align program

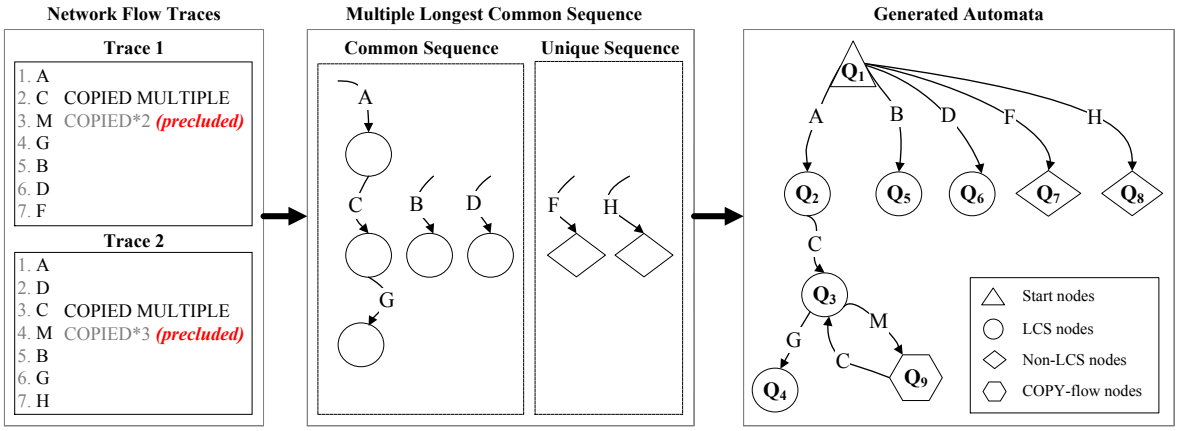

Figure 2.4. An example for automata extraction algorithm

execution with network flows. Secondly, I/O-related syscalls (e.g., `WriteFile`) are leveraged to record program operations on files (or devices) that we are interested in. Lastly, message-related syscalls (e.g., `GetMessage`) are for capturing user interactions such as mouse clicks and keystrokes. To annotate activities on the automata, basic information about program components such as actions of UI components and cache/storage paths must be available beforehand. Note that usually such information can be obtained from software documentations and configuration files. However, a syscall trace may contain thousands of I/O operations and message processing records while many of them are not particularly useful for our analysis. Hence, we first prune the two kinds of syscalls via keyword based filtering where the keywords are the specific program components we are interested in. Then, by aligning the network trace with the pruned syscall trace, network flows are partitioned into different segments by those selected I/O and message related syscalls. Note that those segments facilitate narrowing down the scope of network flows related to each program component and we obtain mappings between network flows and user inputs that generated the network flows. In particular, in each segment, we first search for the newly generated flow s_f leading to an automaton state q_f and s_f is the only flow that can reach q_f. Then, the state q_f is directly annotated with the corresponding activity. However, a segment may not always contain such a distinguishable s_f, but it can reuse established flows or emit the same flows shared

with other segments. Therefore, we leverage additional flow characteristics to assist activity differentiating: packet signature and interaction handshake.

• *Packet Signature* Some programs (e.g., IM software) often maintain a network connection to the server for a long time which may remain during the entire program's lifetime. In practice, such connection might be reused to achieve different activities such as message sending, file transferring and video talking. In such case, s_f does not exist in the corresponding segments. Fortunately, we observe that in order to inform the server of user actions, a unique operation code for each action is often embedded at the beginning of a synchronizing packet (the first packet in the segment) for the server in most cases. Such unique constants are automatically recognized by data-flow analysis in Section 2.3.1 and the activity is annotated with a *positive packet signature* feature on the state, where *positive* means the prefix only appears in the packet sent to the server. Note that such signature discovery is not limited in the long-running connection, but can also be applied in any emitted flows in the segment.

• *Interaction Handshake* Programs may encrypt the traffic to enhance communication security, voiding our efforts in signature analysis. Besides, a signature may not always exist in a communication segment. For such programs, NetCrop attempts to record the amount of interaction handshakes. Specifically, similar to the three handshakes in TCP, communication in any connection is also an operating protocol and each user activity must be realized by several interaction roundtrips. The amount of such roundtrips is learned from the successfully executed sending and receiving syscall pairs at the same connection in the segment. Then, if a unique amount of rountrips could be profiled in a segment, the activity is annotated with a *positive interaction handshake* feature on the state, where *positive* means the roundtrip is launched at the client side.

For program activities that are still indistinguishable, NetCrop leverages existing learning-based techniques [12], [19], [48] to perform a selective training in those communication segments, producing an intra-program classifier to conduct the activity identification.

2.3.3 Traffic Attribution

Once the annotated automata are generated, they are leveraged to identify programs and their activities from network traffic during production run. We formulate this process as a problem of matching path through graph traversing. In particular, the start states of all program automata are initially put into a set D. When a new flow f is recognized, NetCrop searches D for state d_s which has a transition edge e matching f. If d_s exists, e is removed to prevent re-matching. Then, we look into a state d_t following e. If $d_t \notin D$, then $D \leftarrow D \cup \{d_t\}$. Otherwise, all edges starting from d_t removed a priori are added back. A state is removed from D only when all the edges starting from it are pruned. During flow matching, the edges with fields dependent on other flows have the highest matching priority, `CONST` has the second highest, and `ANY` has the lowest.

Checking Pre-set and Post-set We define a validation factor v for each state, which is set to 0 initially. We also assign validation status for each state. All states are labeled as invalid at the beginning except the states with an empty *pre-set*, which are labeled as valid. When a state d_s is examined, we check if it is in the *pre-set* or *post-set* of another state d_t. If d_s is in the *pre-set* of d_t, v of d_t is increased by 1. If v equals to the size of *pre-set*, the status of d_t is changed to valid and d_t can be transited to. If d_s is in the *post-set* of d_t, d_t is set to invalid and it cannot be used further.

Resetting the Matching Process We define a reset signal for each program. The signal is sent when a constant IP connection or a constant domain resolution is detected. If the signal is caught, the automaton of the corresponding application will be reset to the initialization status. For example, `conn.skype.com`, which is the first domain contacted by Skype, is determined as the reset signal for Skype. This resetting mechanism greatly reduces the overhead of matching process.

Transition Conflict For automata that can compose an asynchronous automaton, there are existing solutions [49], [50] that can address the transition conflict problem. However, they are not applicable in our scenario since no concurrency restrictions exist between individual programs which are required in the asynchronous automaton model [49]. Fortunately, we were able to substantially reduce the possibility of runtime transition conflicts by leveraging

the precomputed *pre-set* and *post-set* as they enforce strict transition conditions for programs. Moreover, as we compressed a set of flows that share common annotations into a single `MULTIPLE` flow, conflicts *only happen between different programs*. When a conflict happens, NetCrop computes the subtraction set of states' *post-sets*. If it is empty, then the conflict is considered non-resolvable and NetCrop exploits existing recognition methods [12], [19], [48] to train an inter-program classifier. Otherwise, NetCrop temporarily proceeds on all the automata and when any state s in the subtraction set is reached, all the automata except the one containing s just trace back.

Activity Inference If a state is annotated with an activity, we record it when the state is matched. Note that this may require an online query sent to the intra-program or inter-program classifier. While all the field annotations discussed in Section 2.3.1 can be obtained from the first packet of the connection, NetCrop only selectively checks complete flows on activity-annotated states to be more efficient, skipping other packets on unannotated states during the attribution process.

2.4 Evaluation

2.4.1 Experiment Setup

Platform and Data Collection Windows was selected as our experimental platform but NetCrop is also portable to any other operating systems. We tested 14 programs, *QQ (Tencent QQ)*, *Skype*, *Slack*, *uTorrent*, *PPS*, *Popcorn (Popcorn Time)*, *BitTorrent*, *Foxmail*, *Outlook (Outlook 2013)*, *Clawsmail*, *Thunderbird (Mozilla Thunderbird)*, *iTunes*, *FZ (FileZilla)* and *Putty*. We deployed 8 different machines with Windows 10 64-bit installed, operated by 8 people located in different places above IPv4 configuration. The network is under full-cone NAT support, which is the most common configuration in enterprise networks. The data collection procedure has two stages. First, for collecting training data, users execute our testing scripts with network tracing, syscall tracing and analysis tools to run all the programs. Second, for collecting attribution testing data, users are required to concurrently run all the programs as their daily usage for two weeks, with network tracing and syscall tracing tools only. Note that in the second stage, the collected syscall traces are

only exploited to fetch the ground truth of flow attribution but they *do not* act as any input to automata in the attribution phase. We collected 100 traces for each target program to train annotated automata and used about 986k network flows (in different traces and 862k from target programs) for attribution testing.

Implementations We used Intel Pin [51] and MS Detour [52] to deploy program analysis facility, log the system calls and obtain the ground truth for network traffic attribution by mapping real flows to network-related syscalls via function arguments. For network-related syscalls, we mainly record DNS and socket functions such as `DNSQuery`, `getaddrinfo`, `send` and `recv`. Among I/O system calls, we are interested in file management functions and UI management functions including `CreateFile`, `ReadFile`, `WriteFile` and `CreateWindow`. We intercept messages sent to programs from the arguments in two messaging system calls `GetMessage` and `PeekMessage`. To copy and dissect network connections, we utilize windump [53] and Python's `dpkt` package [54].

Load Balancing and CDN In collection of network traces, we observed some distinctive patterns of load balancing and Content Delivery Network (CDN), reflected in the random allocation in a group of IP addresses and domains. For example, we observed "2-trouter-cus-a.drip.trouter.io" and "35-trouter-cus-a.drip.trouter.io" are used in *Skype* for different users or machines. Because our dynamic data-flow analysis can not cover all those constant strings, hence, when attributed, only the invariant letter string parts are matched, skipping all the digits which may vary across runs. We also observed a group of constant IP addresses from "103.7.30.139" to "103.7.30.169" for different users in *Tencent QQ*. Similarly, we only match the first 26 bits of the 32-bit IPv4 address as the first 26 bits do not vary across runs. Note that we automatically identify such invariants by tracing instruction addresses in the aforementioned data-flow analysis (Section 2.3.1) across multiple runs because any instance of those connections must go through the same call site of socket or DNS function in the execution path.

DNS Caching DNS resolution results could be cached in the client side. Therefore, to cover all the DNS resolutions from programs in monitoring/training, we flushed the DNS cache (*one time only*) before collecting traces for automata construction. Interestingly, however, from the results in our experiment, nearly all the domain resolutions had to go through the

network instead of the cache, even if we executed them in short intervals. This phenomenon is due to (1) programs (like many web apps) generally set the *TTL* of their servers' domains to less than 100 seconds and (2) Windows DNS cache only stores the first record in the resolution results. As those servers usually have multiple aliases, the first record of the response is always a *CNAME* record without any IP address. In short, the DNS cache does not pose challenges in practice.

UPnP Configuration Universal Plug and Play (UPnP) is a collection of protocols that allows network devices to discover and establish services between other devices. UPnP was enabled in our experiment as it is enabled by default in common network environments. With UPnP, the local port reservation on gateways is attempted by three steps in P2P programs. First, they always try to discover UPnP service on the multi-cast address 239.255.255.250 at port 1900 and request the port mapping on the gateway at port 80 by HTTP. This step provides the port-program pair information in an XML packet. Second, if the first step fails, then the program will launch an NAT port mapping request on the gateway at port 5351 or require the port binding and mapping through Session Traversal Utilities for NAT (STUN) service on an external server, where a local reserved port, say p is contained in the request packet. Last, if both steps fail, it will exploit public super-nodes (intermediate-nodes) using local port p to assist the establishment of peer connections. Note that the communication of last two steps has no intelligible information, hence the port owners cannot be directly recognized. More importantly, NetCrop also handles the networks with UPnP disabled via additional UPnP-oblivious states. In particular, we read the p in the port mapping request and regard it as a `UPNP` port which is attributed through automaton state transition.

Advertisement Embedding diverse advertisements is a main income source for free software. Large amounts of programs may share common advertisement domains such as "ad.doubleclick.net" and "ad.doubleclick.de", which may cause transition conflicts in our attribution. Considering that the advertisement loading behavior is a typical parasitic web behavior and it is *unrelated to the important actions of programs*, we use a public advertisement provider list [55] to prune AD-related traffic.

VPN For security reasons, third-party VPNs are generally disabled in enterprise networks via protocol filtering. Hence, NetCrop does not take VPN settings into consideration.

We argue that analyzing VPN or other tunneling traffic is an orthogonal research effort to NetCrop.

2.4.2 Case Study

QQ, Skype and Slack The three applications are widely used for instant messaging. The enterprise might be interested in identifying their specific activities (e.g., file transmission) instead of just knowing their existence. They all maintain a long-running connection to perform as a command channel, fulfilling their functionalities. While *QQ* realizes different user actions by inserting unique command codes at the packet head in plaintext: `0x2355f00cd`, `0x2355f01bd`, `0x2355f0346`, and `0x2355f01b4` for sending messages, sending files, receiving files, and video talking. In contrast, *Skype* and *Slack* encrypt all the traffic [56], [57] and hence we have to train intra-program classifiers to infer some of their end-host activities. Figure 2.5 shows an example of the long-running connection in *QQ* and *Skype*. We can observe the connection to the *QQ* server (103.7.30.141) lasts much longer than others. Note that the y-axis is the timeline in the unit of second, and the server "gateway.messenger.live.com" of *Skype* has a much longer duration in Figure 2.6 (note that the domain prefix "BN1MSGR1011310" is omitted in the figure) than the others.

In addition to the command channel, they also deploy a P2P-like component which is responsible for file transfer and real-time video service, where *Skype*, *QQ* and *Slack* register the local port through UPnP, port mapping and STUN respectively.

Outlook, Clawsmail, Foxmail and Thunderbird Identifying the existence of a particular third-party email client program currently in use is useful for system managers as some third-party programs may have software vulnerabilities. However, due to the similar functionalities among third-party email clients, such task is challenging. In this case study, we show how NetCrop can pinpoint an e-mail client by using network traces. While different third-party e-mail clients have similar functionalities, we observe that implementations of some functionalities often differ from each other, allowing NetCrop to identify an exact email client currently in use. For example, when users add an e-mail account, if they do not know the exact server name, they usually give program a rough domain name (e.g.,

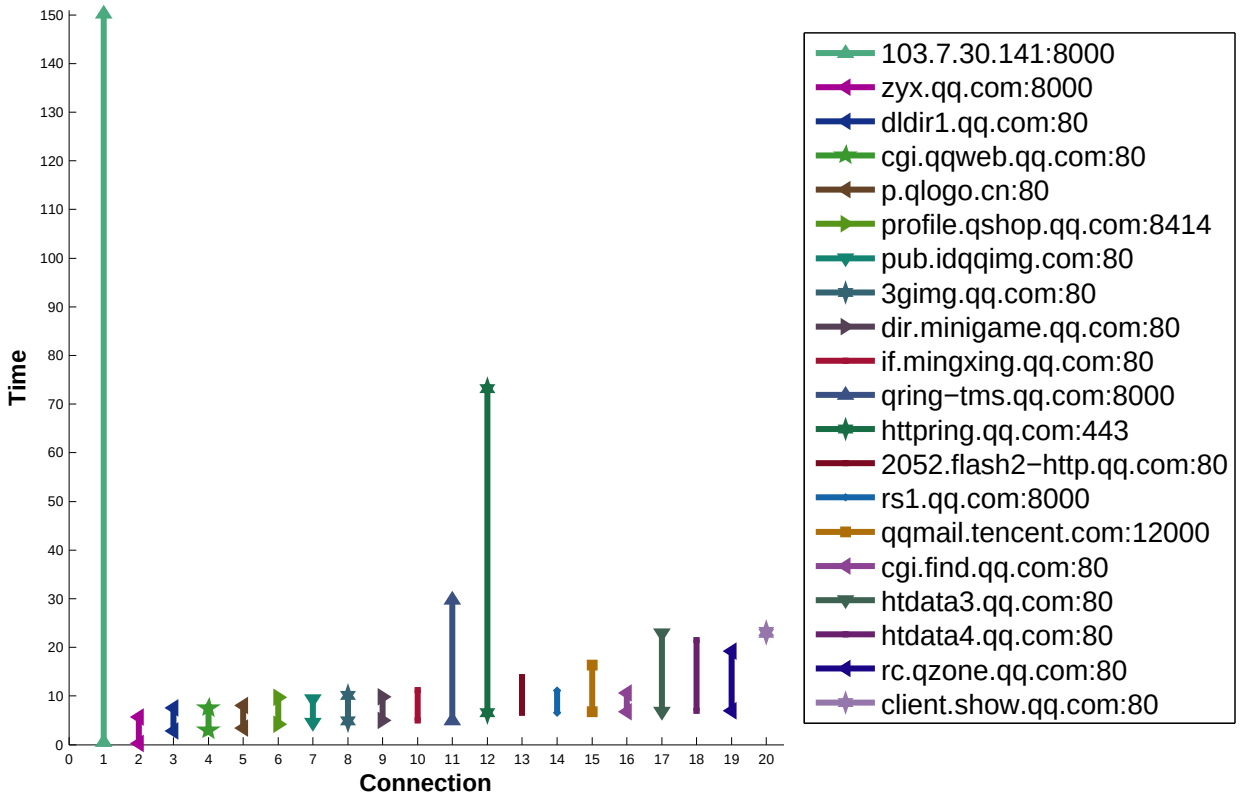

Figure 2.5. Duration of flows in Tencent QQ

`user@gmail.com`) and let the program identify the correct one (e.g., `imap.gmail.com`). The most convenient and reasonable way to search for the correct mail server is to launch the `MX` query for the postfix domain of the e-mail account. However, it turns out that programs prefer to rely on their own strategies since not all the e-mail servers actually register their `MX` records. As shown in Figure 2.3, *Outlook* attempts the five domains to discover the proper mail servers. If all the five trials fail, it will attempt a sixth (the last) domain prepended with "autodiscover". *Clawsmail* uses three SRV queries with strings "_pop3s._tcp.", "_submission._tcp." and "_pop3._tcp." as prefixes to obtain the email server host names whereas *Foxmail* connects to its own query server "addrapi.exmail.qq.com" to retrieve e-mail servers. In contrast, *Thunderbird* firstly tests a query on the "autoconfig" prefix, and then fetches the answer from its "mx.thunderbird.net" server if the query fails.

uTorrent and PPS *uTorrent* is a well-known program built on BT (bittorrent) protocol which is dominating the P2P implementation today and it resembles another two programs

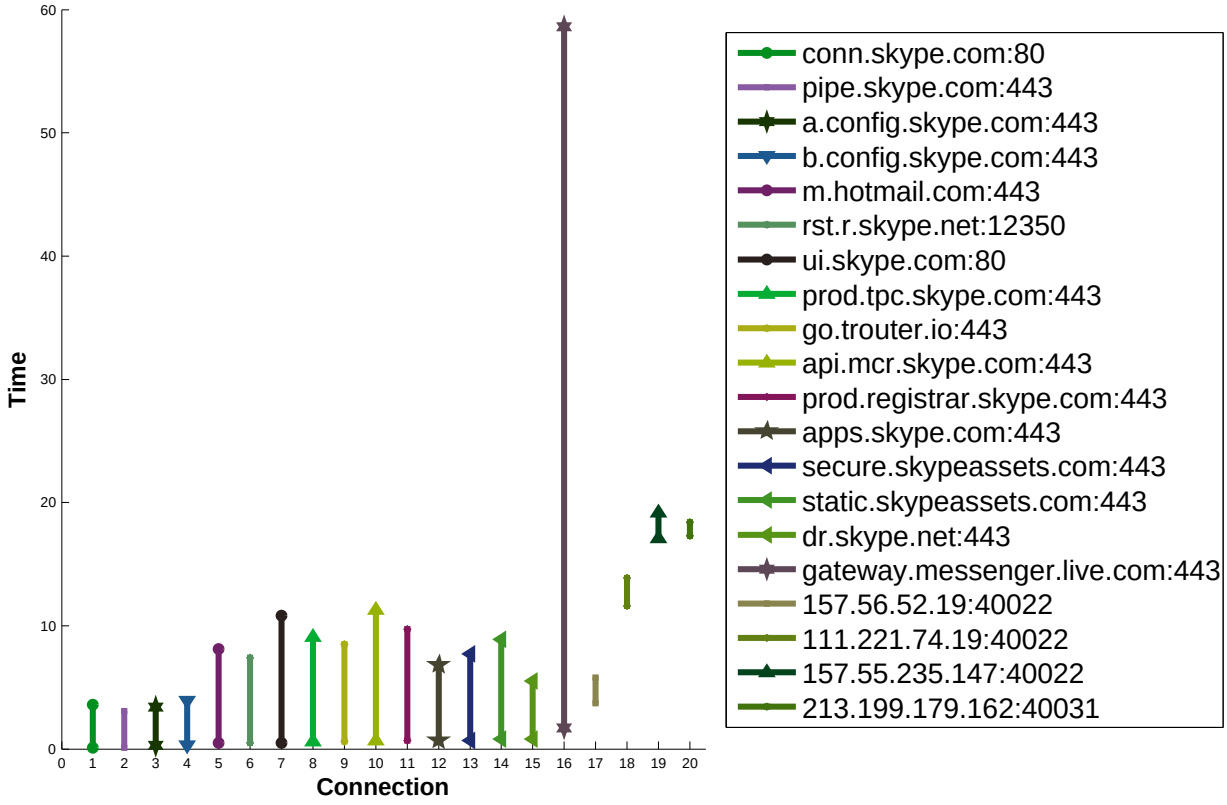

Figure 2.6. Duration of flows in Skype

Popcorn and *BitTorrent*. As explained in Section 2.3.1, when a P2P program built on BT protocol downloads a file from a peer, it will back up the downloading connection by duplicating a TCP flow copying remote peer IP and port from the UDP flow created before, as illustrated in Figure 2.3. *PPS* is a web-based P2P client for a video site *iQiYi*. It uses P2P techniques similar to *bittorrent* to accelerate the content delivery speed for video watching. When fetching movie segments from peers, the program exhibits the same flows shared with other segments but the amount of *interaction handshakes* can help distinguish its behaviors.

2.4.3 Performance Overhead

The overhead incurred by NetCrop is measured from two perspectives, resource (CPU and memory) consumption and network throughput. We deploy a gateway server (8 cores and 32GB memory) which also performs other local services such as Web and SSH, and 8 end-host machines actively running client-side applications including the 14 target programs

(all the end-host execution is automated by testing scripts and follows a pre-recorded one-day workload). In particular, the evaluation consists of three separated days, $d1$, $d2$ and $d3$, where the gateway's on-the-fly attribution is turned off in $d1$, kept running in $d2$ and only turned on (without conducting any attribution task) in $d3$. Note that the purpose of the evaluation in $d3$ is to demonstrate NetCrop's scalability during production runs (i.e., whether program accumulation would significantly increase attribution costs). We compute the median of per-second overhead for each hour and the results are collected in Figure 2.7. As illustrated, the overhead of NetCrop is actually negligible, as memory overhead is below 4.5%, and CPU/throughput overhead does not exceed 3.0%. In addition, compared with $d3$, $d2$ does not degrade the performance when the attribution is enforced, which indicates that NetCrop should be highly scalable in the perspective of the number of target programs. Here, to reflect the throughput loss caused by NetCrop, throughput overhead of $d2$ (the same for $d3$) is calculated as $\frac{out_1/in_1}{out_2/in_2} - 1$, where out_1 and in_1 are outgoing and incoming throughput of the gateway in $d1$, and out_2 and in_2 are that in $d2$.

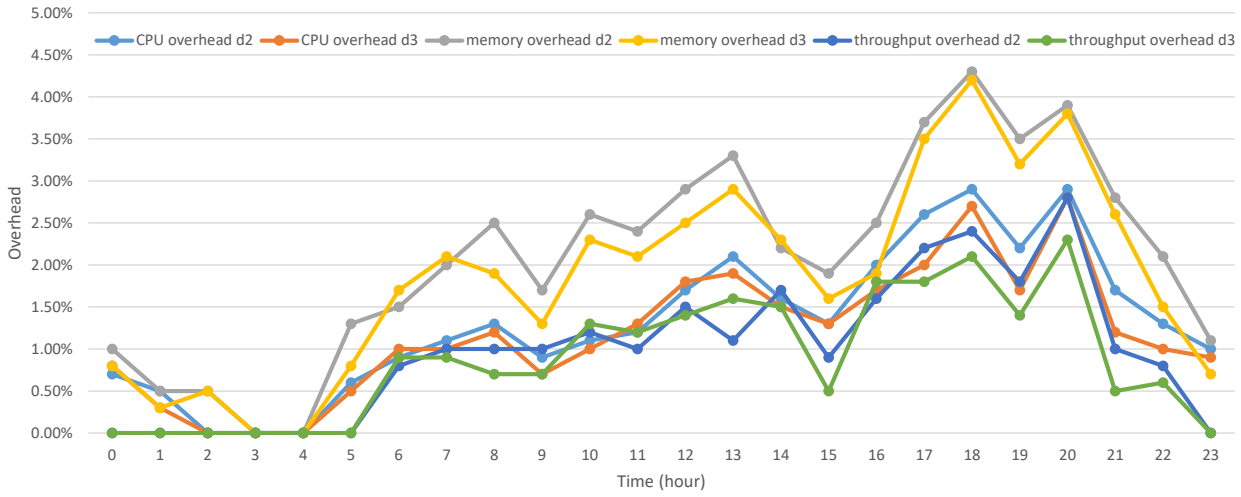

Figure 2.7. Performance overhead of NetCrop

2.4.4 Results

We compare our system with three existing learning-based classification techniques: A [12], B [48] and C [19] for traffic attribution. In particular, we leverage those tech-

niques to train our intra-program and inter-program classifiers, implement training models in WEKA [58] and perform a 10-fold cross validation to attenuate overfitting problem. For the technique involving the combination result of multiple classifiers, we just select the one with the best performance. The $F1$ scores are shown in Figure 2.8, where $F1 = 2 * Precision * Recall/(Precision + Recall)$, $Recall = TP/(TP + FN)$, $Precision = TP/(TP + FP)$. Note that NetCrop has a much better performance in most of the 14 programs. As stated above, automaton model expresses the flow causal dependency but learning-based techniques are well-tailored for flow-based recognition. NetCrop achieves high F1 scores (above 85%) in all programs except the four email clients. We inspect the cases and it turns out that except the account adding behaviors, other common functionalities such as e-mail sending/receiving are mostly similar between them. So, when attributing such flows, the conflicts force NetCrop to apply the inter-program classifier. Note that, however, NetCrop can still pinpoint an email client in use via other functionalities with distinctive implementations.

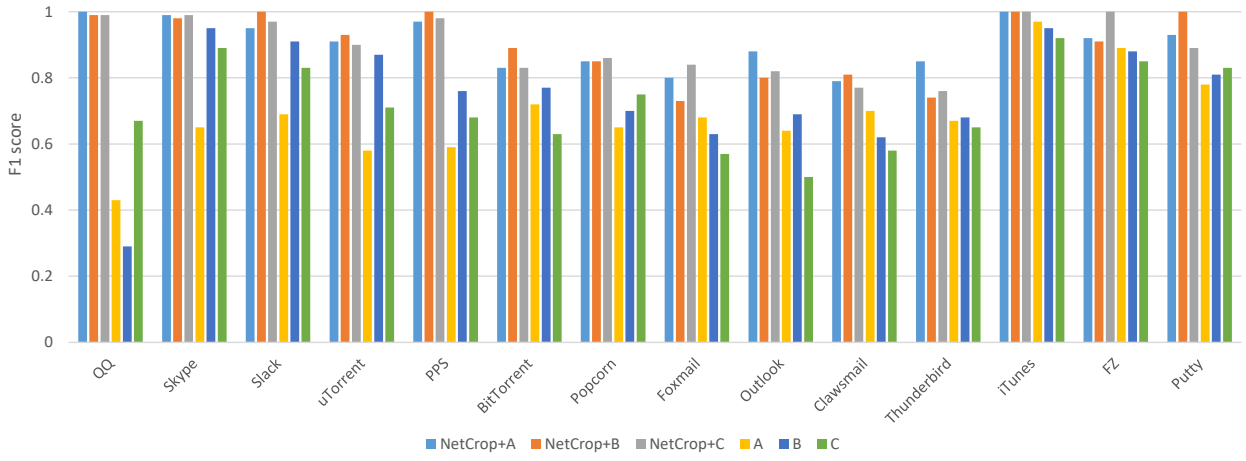

Figure 2.8. Comparison of F1 score with existing classification techniques

For further comparison, we evaluate the performance of NetCrop and existing classification techniques under different training sizes. Figure 2.9 collects the results, where the $F1$ score is the overall result from the 14 programs and all the techniques are evaluated under the training size of 20, 40, 60, 80 and 100 traces respectively. For each training size, we still perform the evaluation in a 10-fold cross validation. As the results indicate, the training set

of 100 or fewer traces for each program may be too small to achieve a satisfactory accuracy for learning-based classification techniques. This shows a significant advantage of NetCrop, which is that a small amount of training traces can still bring a relatively high accuracy. Moreover, in the classification stage (without behavior inference), NetCrop can quickly attribute a flow by examining the first packet of the flow while ML techniques need to wait for the whole flow statistics, which makes NetCrop outperform others in the attribution efficiency.

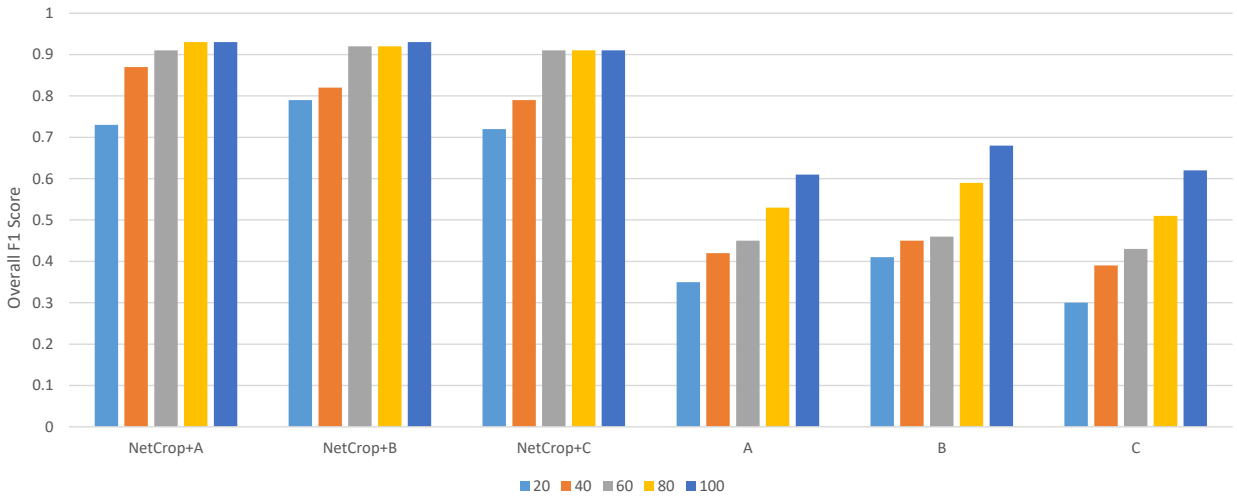

Figure 2.9. Comparison of overall F1 score under different training sizes

Web browsers Web browsers are not considered in our evaluation targets but they serve as unknown or "other" programs. All the unmatched web flows are supposed to be generated by unknown programs and may need further inspection. Note that modern browsers are highly complex and can be seen as OSes running various languages and web-applications. Behavior analysis inside browsers is partitioned on different websites but not separated apps. Profiling websites in web is not in the scope of current NetCrop, while one can implement NetCrop in browsers to do the profiling. Also, one can leverage website fingerprinting as well as web-based covert channel or malware analysis techniques [59]–[61], as complementary tools to examine the unmatched web flows.

For further analysis on traffic attribution of NetCrop, we compute the misclassification rate between programs. The misclassification rate (A is attributed to B) here is

Table 2.1. Misclassification between programs

	QQ	Skype	Slack	uTorrent	PPS	BitTorrent	Popcorn	Foxmail	Outlook	Clawsmail	Thunderbird	iTunes	FZ	Putty	Other
QQ	N/A	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
Skype	0%	N/A	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
Slack	0%	0%	N/A	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
uTorrent	0%	0%	0%	N/A	0%	2.78%	3.41%	0%	0%	0%	0%	0%	0%	0%	2.81%
PPS	0%	0%	0%	0%	N/A	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
BitTorrent	0%	0%	0%	3.27%	0%	N/A	2.47%	0%	0%	0%	0%	0%	0%	0%	2.20%
Popcorn	0%	0%	0%	3.79%	0%	4.54%	N/A	0%	0%	0%	0%	0%	0%	0%	2.67%
Foxmail	0%	0%	0%	0%	0%	0%	0%	N/A	8.8%	9.1%	10.7%	0%	0%	0%	0%
Outlook	0%	0%	0%	0%	0%	0%	0%	10.9%	N/A	8.3%	9.8%	0%	0%	0%	0%
Clawsmail	0%	0%	0%	0%	0%	0%	0%	11.2%	10.1%	N/A	9.5%	0%	0%	0%	0%
Thunderbird	0%	0%	0%	0%	0%	0%	0%	8.1%	8.4%	9.9%	N/A	0%	0%	0%	0%
iTunes	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	N/A	0%	0%	0%
FZ	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	N/A	0%	0%
Putty	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	N/A	0%
Other	0%	0.90%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	N/A

$Misclass_{A,B} = \frac{FN_A \cap FP_B}{FN_A + TP_A}$. We regard all the applications that are not in the 14 target programs as *Other*. The results are summarized in Table 2.1. We can observe that there is misclassification between P2P applications. It is mainly resulted from the transition conflicts of common resource trackers like "tracker.bittorrent.com" in their automata. Also, some packets of those P2P applications are identified as *Other*. This is due to (1) trackers in the P2P architecture can be dynamically updated and they were not completely observed during our automata extraction process and (2) programs often talk with bundle servers on port 80 or 443 to obtain the resource profile and those servers can be input from users' seed files. The misclassification between email clients is expected because they exhibit extremely similar network behaviors in regular e-mail sending and receiving hence difficult to completely differentiate them. Note that in practice, users also use other functionalities in the email clients, allowing us to differentiate them. Hence, this does not mean failure of NetCrop. *Skype* has embedded components that exchange data with websites such as "www.facebook.com" and "connect.facebook.net", which overlaps the normal web browsing, causing the misclassification from *Other*.

Effectiveness of Activity Inference Table 2.2 shows the result of activity inference inside the 14 programs. The overall results are promising and most of the activities we focus on are captured accurately. The account adding operations in email clients are all through programs' automatic server settings. If a user manually sets his email servers without the automatic server setting, it turns out all the programs do not generate any network packets for the action, hence is invisible in network traces. The same as that in traffic attribution, the

Table 2.2. Activity inference result

Program	Activity	#Total	TP	FP	FN	Program	Activity	#Total	TP	FP	FN
QQ	Open Chat Window	269	269	0	0	uTorrent	Download File	224	224	0	0
	Send Message	1103	1103	0	0		Share File	129	129	0	0
	Send File	33	33	0	0	PPS	Watch Video	187	187	0	0
	Receive File	19	19	0	0		Share video	42	42	0	0
	Voice/Video Call	34	34	0	0	BitTorrent	Download File	193	193	0	0
Skype	Send Message	105	105	53	0		Share File	97	97	0	0
	Send File	17	17	0	0	Popcorn	Watch Video	271	271	0	0
	Receive File	11	9	0	2		Share video	85	85	0	0
	Voice/Video Call	71	67	0	4	FZ	Download File	59	55	9	4
	Open Chat Window	29	22	6	7		Upload File	51	50	0	1
	Add Credit	8	8	0	0	Putty	Download File	53	53	31	0
Foxmail	Add Account	24	24	0	0		Upload File	76	70	0	6
	Send Email	82	67	19	15	Outlook	Add Account	28	28	0	0
	Retrieve Email	139	102	34	37		Send Email	151	117	39	34
Clawsmail	Add Account	17	17	0	0		Retrieve Email	282	240	44	40
	Send Email	53	41	8	12	Slack	Send Message	197	197	39	0
	Retrieve Email	113	87	28	26		Send File	21	21	0	0
Thunderbird	Add Account	15	15	0	0		Receive File	14	14	0	0
	Send Email	35	25	3	8		Voice/Video Call	18	18	0	0
	Retrieve Email	49	34	7	10		Open Chat Window	31	29	8	2
iTunes	Download App	47	47	0	0	iTunes	Listen to Music	68	68	0	0

high FP and FN in email sending and receiving activities are due to the forced inter-program classifier of NetCrop in conflicting automata matching. However, we observe that the majority of them can still be correctly attributed and it indicates that the implementations of those email-client programs can still emit statistically different flows even in email exchange. The false positive cases in the message sending inference for *Skype* and *Slack* were expected because the long-running flow with the command server is also responsible for other usage such as status update, where the attribution is trapped into the intra-program classifier, and the same situation also happens to "Open Chat Window". NetCrop fails to recognize 4 voice/video calls. We manually inspect these cases and it turns out that three calls failed to get through, so there is no interaction between the two people. The other call is too short and no words were spoken during the call. We notice that there are several false negative instances for file transferring activities in *Skype*, *FZ* and *Putty*. We found that they are due to the transferring of small-size files which are misclassified by the intra-program classifier. Similarly, the false positive cases for file downloading identification in *FZ* and *Putty* are due to the execution of commands with heavy feedback.

2.4.5 Traffic Attribution in Enterprise Network

The prior evaluation result suggests NetCrop is able to attribute network traffic to applications and their behaviors with high accuracy. We also test NetCrop on traffic data collected within a real enterprise to evaluate its effectiveness for the real-world settings.

Dataset We were granted with access to the logs collected by host agent installed on tens of thousands machines within the enterprise.[1] The host agent monitors the operations of processes running on the deployed machine, including file access and network communications, and transmits the logs to a central database for host provenance. For scalability, the events of the same type (e.g., connection to a domain by one process) are aggregated and only the first and last event timestamp are kept. Besides, the logs are periodically cleaned to save storage space. Due to the limited quota assigned to us, we query and download 6-week logs related to 3 applications of sufficient installation number (QQ, Skype, and iTunes) to compute recall of application fingerprinting[2] and 12-hour logs of all applications to compute false-positive rate (FPR). We only fingerprint applications but not specific behaviors because the host agents do not collect any behavior information and hence we do not have the ground truth. Note that fingerprinting applications alone is very meaningful for enterprises.

Changes on traffic attribution We adjust the traffic attribution method for this dataset. First, as the exact timestamp for network connection is missing, we match the network flows and the automaton without ordering. Second, to handle missing network logs, we consider a machine running the tested application if its network flows match at least half of the states along one path of the automaton.

Results Table 2.3 shows the result for recall and FPR. For recall result, we count the number of machines installing the application and machines detected by NetCrop. Even given the complexity of the enterprise environment (different versions of the application are running concurrently, and the hosts are distributed globally) and the issue of missing data, NetCrop is still able to catch most instances (79.4% at least for iTunes). The FPR is negligible for application fingerprinting. We count the number of machines not running the

[1]↑The enterprise deploys agents on end-hosts and proxies at network perimeters. We use the logs from host agents as ground-truth. NetCrop does not require the host logs when deployed by enterprise.
[2]↑Outlook is not considered for testing as the enterprise did not disclose their internal configurations.

Table 2.3. Recall and False-positive Rate (FPR) on enterprise dataset.

Program	Recall (Matched/Total)	FPR (Matched/Total)
QQ	95.0% (19/20)	0.04% (3/8334)
Skype	82.1% (288/351)	0.22% (18/8289)
iTunes	79.4% (923/1163)	0.04% (3/8100)

applications in the 12-hour dataset and the machines alarmed by NetCrop and calculate FPR. FPR for QQ and iTunes are only 0.04%, while FPR for Skype is 0.22%. We inspect the logs and found that 17 machines (out of 18) are captured as they all run a program `SkypeC2CAutoUpdateSvc.exe` which yields similar network flows as Skype. In fact, as it updates Skype automatically, we believe there is a high chance that the 17 hosts run Skype. The results show that NetCrop is highly effective in handling real-world network traffic.

2.5 Limitation and Discussion

Activity Selection Currently, NetCrop mainly focuses on commonly used functionalities such as instant messaging, file transferring and video talking. However, the design of NetCrop is general and not limited to these activities. It can also infer other activities as long as they (1) involve explicit user-side input (e.g. keyboard/mouse events and file operations), (2) generate distinctive network segments (program behaviors does not generate network packets are not considered). Some behaviors such as pushing notification to clients are dominated by the server and they can be completed without explicit user-side actions. To support those behaviors, NetCrop is required to record and analyze additional syscalls which can reflect the user-side reactions (e.g., text changes in GUI components).

Attribution for Similar Programs As shown in our technical design, traffic attribution and activity inference between similar programs hinge on the implementation diversity inside program components. However, it is not a panacea and such diversity may not exist in some similar program sets. For instance, the performance of NetCrop in distinguishing email sending and receiving behaviors from various clients is unsatisfactory. However, while NetCrop could not resolve this merely from the perspective of automaton annotation

and flow-based classification, integrating NetCrop with lightweight host-end tracing may mitigate the problem.

Training Coverage In training phase, understanding program components involves some manual efforts while program testing on common use cases may not cover all the execution paths. Due to the limited coverage, some communication patterns may not be derived during the training. However, it can be alleviated by combining more sophisticated program analysis and testing techniques [62]–[64].

Automaton Model The automata derived by NetCrop are irregular. Therefore, the large body of well-studied automata processing techniques, such as transition conflict handling, may not be directly applicable. If one can obtain regular automata for programs, existing automata processing techniques can be applied. However, from our analysis and observations on modern network communication, program models (like those in NetCrop) are hardly regular due to the complexity of program semantics.

Dynamic Flows It is possible that a program can dynamically change connection parameters (e.g. remote IP and remote port) at runtime (e.g., the parameters may be dynamically set by previous flows which may also be encrypted). Note that this is different from CDN, where servers are still a set of constant addresses. In such case, NetCrop might have difficulties in fingerprinting programs and activities. However, due to the complexity and runtime cost of such functionalities, they are rarely observed in practice. Indeed, our evaluation result shows that NetCrop can work on most categories of programs.

2.6 Related Work

The issue of traffic classification has been attracting researchers' continuous attentions. In the beginning, such task is fulfilled through simple port examination. In TCP/UDP headers, port numbers can be mapped to their application-layer protocols defined in IANA [12], but the port recognition method is not helpful when traffic is tunneled in popular protocols or dominated by P2P applications. Considering the flexible usage of network ports today, this port-based technique is facing increasing limitations [7], [13], [65], [66]. To understand what protocol packets are really talking about, following proposed tools try to scan packet

payloads and inspect interesting signatures [11]. However, despite the high accuracy such DPI-based tools can achieve, they could be thwarted by possible challenges. For example, the searching process may incur high runtime overhead and encrypted packets may cause unexpected errors. To overcome the problems in port-based and DPI-based techniques, statistical machine learning based approaches have been proposed. Most of the proposed mechanisms implement classification utilizing flow statistics like communication duration, inter-packet delay, packet size and partial transmission ratio [10], [12], [66]–[68].

Many supervised machine learning tools have been applied in traffic classification. NBC (Naive Bayes classifier), nearest neighbors, decision trees and SVM (support vector machines) [7], [69], [70] are proved to be useful. In [7], about 250 flow-based statistical features are collected for the classification in NBC. Utilizing correlation-based filter in dimension reduction and kernel density estimation in distribution computation, the approach performs well in distinguishing applications of web, email, file exchanging, database and multimedia. Semi-supervised machine learning techniques such as expectation maximization and K-means [12], [71] are also explored. Three cluster algorithms, K-means, Gaussian mixture model and spectral clustering are used in [12]. With the characteristics collected from the sizes of first several packets in each flow, the method achieves promising results in identifying most well-known application-layer protocols. In [72], combination of diverse classifiers are applied to achieve a high classification accuracy, even for private application protocols such as Skype. Besides the flow-based statistics, some end-host behaviors are involved in multi-level classification models [13], [73]. In [13], a novel three-level clustering is equipped. Adding host-end connection counting, the advanced model can achieve above 90% accuracy and report unknown protocols. Although those machine learning algorithms and models could achieve highly accurate detection rates in certain network circumstances, they have limitations [10], [67], [74]. Most of them handle flows separately and hence, they are not suitable for traffic attribution and behavior inference in the granularity of whole program, which requires the information of flow causal dependency.

Network traffic analysis is also used for mitigating threats from adversaries. For example, Borders et al. [59] proposed a system called `Webtap` to detect anomalous outbound traffic going through HTTP tunnel (e.g., potential data exfiltration attempts). The goal of our

approach is different in that it aims to accurately fingerprint programs and their behaviors. Our approach could also complement these existing works in network provenance.

3. LPROV: PRACTICAL LIBRARY-AWARE PROVENANCE TRACING

With the continuing evolution of sophisticated APT attacks, provenance tracking is becoming an important technique for efficient attack investigation in enterprise networks. Most of existing provenance techniques are operating on system event auditing that discloses dependence relationships by scrutinizing syscall traces. Unfortunately, such auditing-based provenance is not able to track the causality of another important dimension in provenance, the shared libraries. Different from other data-only system entities like files and sockets, dynamic libraries are linked at runtime and may get executed, which poses new challenges in provenance tracking. For example, library provenance cannot be tracked by syscalls and mapping; whether a library function is called and how it is called within an execution context is invisible at syscall level; linking a library does not promise their execution at runtime. Addressing these challenges is critical to tracking sophisticated attacks leveraging libraries. To facilitate fine-grained investigation inside the execution of library binaries, we develop LPROV, a novel provenance tracking system which combines library tracing and syscall tracing. Upon a syscall, LPROV identifies the library calls together with the stack which induces it so that the library execution provenance can be accurately revealed. Our evaluation shows that LPROV can precisely identify attack provenance involving libraries, including malicious library attack and library vulnerability exploitation, while syscall-based provenance tools fail to identify. It only incurs 7.0% (in geometric mean) runtime overhead and consumes 3 times less storage space of a state-of-the-art provenance tool.

3.1 Introduction

APT (advanced persistent threat) attacks are eternal enemies to cybersecurity communities and contemporary enterprise networks are suffering the most among all the network environments. Incented by tremendous economic interests in commercial espionage, attackers are taking persistent efforts in penetrating enterprise networks from diverse vectors, which motivates the increasing demands in cyber attack investigation. Detecting or intercepting an

APT attack at its entry point is particularly challenging due to their advanced and stealthy attack techniques. For example, backdoor implantation scheme allows attackers to inject malicious code into a benign program in order to disguise the malicious behaviors as normal benign behaviors. Rather than downloading and executing obvious malicious programs, they leverage existing benign applications and services to conduct malicious behaviors such as downloading or opening attachments from well social-engineered emails, and clicking URL links in luring advertisements [75]. Hence, in recent years, in addition to the significant contribution manifested in attack detection, provenance tracking becomes an irreplaceable pillar in APT analysis and defense. Given a target system entity or object (e.g., compromised file, socket, or process), provenance tracking systems analyze it from multiple aspects, and figure out the entity's root (or origin) as well as deriving path [3], [75]–[77]. The root contains all the external entities (e.g., an IP address) affecting the status or value of the target entity; while the deriving path is an organized causal graph illustrating how the entity is eventually influenced from the root. Such tracking information can facilitate locating the attack and prevent repeated infection.

Existing provenance tracking techniques can be divided into three categories: non-unit provenance, tainting-based provenance, and unit-based provenance. Most provenance tools leverage event logging to trace system events (e.g., syscalls) and associate them for further offline attack tracking and investigation [78]–[85]. While there are various system events such as network communication, memory operations and syscalls, most provenance tools focus on logging and analyzing syscall events as syscall logging (e.g., audit logging) is widely used (included in most Linux distributions by default) and practical (low overhead).

Non-unit provenance techniques Non-unit provenance tools have a conservative assumption: a process is causally related to all the system entities (e.g., files and sockets) it has accessed so far. In such conservative causality correlating models, any output object (e.g., files and sockets) of a process has causal relations to all the preceding input objects, resulting in many bogus causality relations and confusing the following attack investigation. We call such problem the dependence explosion problem [75], [78], [86], [87] and the problem becomes particularly severe in complex long-running programs such as *firefox* and *Apache*. For instance, consider a case where a user opens *firefox* and browses 10 websites. Then, the

user carelessly downloads a malware binary on one of the website, namely *xxx.com*. Later, the malware is detected and an investigator wants to identify which website downloads the malware. When non-unit provenance techniques are used, they report all of the 10 websites as roots while only the website *xxx.com* is the true root cause and other 9 websites are not.

Tainting-based provenance Tainting-based provenance techniques [77], [88]–[96] assign tags to multiple tainting sources (e.g., receiving sockets and input files) and propagate them by monitoring executed instructions. When those tags reach sinks (e.g., sending sockets and output files), they detect information flow between sources and sinks, revealing the roots of the sink entities (e.g., IP addresses). However, since these techniques work on the instruction level, most of them cause significant runtime overhead and they are rarely used in production runs. Note that while the latest instruction-level tracking system may perform replay-based provenance analysis in low overhead (3.22%), its resource pruning and selective tainting in replay require pre-recorded instruction-level execution logs [77], hence it is not applicable in our context. Moreover, taint analysis suffers from over/under-approximation as it has difficulty handling implicit information flow through control dependencies (e.g., data compression and table lookup). Besides, taint set operations in instructions are error-prone since pointers, arrays, syscalls, third-party libraries and language-specific features should be carefully processed.

Unit provenance techniques. Unit-based provenance such as BEEP [75], [97] and Protracer [76] is the state-of-the-art in provenance tracking. They are based on an observation from a study [75]: in diverse open-source software including both of client-side and server-side, most programs are designed in input-triggered loops which dominate the event handling. Hence, unit-based provenance tracking techniques profile such loops from program binaries and partition programs' execution into units. Those units are semantically autonomous and they are usually responsible for independent input events. By partitioning a long-running execution into multiple small execution units, they significantly mitigate the dependence explosion problem. An output object is considered causally correlated to an input object only if they belong to the same unit. However, a single unit may not cover the whole execution subroutine for one input event, such as the asynchronous unit cooperation in message queue processing. Hence, memory dependency between units is also tracked to detect inter-unit

dependencies. With such design, the unit-based provenance tracking techniques can accurately identify the root (the *xxx.com* website in the previous example) pruning out other bogus dependencies.

Unfortunately, while the unit-based provenance techniques mitigate the dependence explosion problem, they fail to consider another important aspect of provenance, the shared libraries. An integrated executable binary consists of a main module and several depending shared libraries. For example, the binary of *vim* links 14 shared libraries and *firefox* has more than 20. In most of provenance tracking systems, the main module and those libraries are usually handled as one process/program entity but they are not analyzed separately. Since the library loading phase is in the program initialization but outside any event handling loop, the output of any unit has no dependence on any input library in unit-based tracking techniques. As a result, the unit-based techniques are not able to track provenance in libraries. Note that although Protracer [76] correlates all the loaded libraries in generating the causal graph, it is still coarse-grained since they are the same input for all the units, which is considered no causality in the logic of dependence analysis. Unlike other input sources such as files and sockets which are value-based hence can be tracked by monitoring I/O syscalls, the provenance inside libraries is execution-based and the correlation cannot be simply tracked by causality deduction in I/O syscalls. Specifically, mapping or linking a library does not promise any execution instance, and whether a specific library is executed or how it is executed within a unit cannot be answered in the syscall granularity. Note that there have been proposed user-space tracing tools to perform provenance tracking [98]–[100], but they are too coarse-grained and the tracing requires repeated manual efforts in event/-causality definition or program instrumentation because stealthy attacks could act as normal behaviors (See Section 3.2).

Our solution To this end, we develop a user-space library tracing technique and merge it into existing syscall-based provenance tracking systems in order to improve the visibility of library execution in provenance tracking. We propose a novel provenance tracking system LPROV which performs on the granularity of library functions other than those on syscalls. It aims at addressing the obstinate wart, the absence of user-space library provenance, in syscall-based auditing systems.

It works as follows. Upon the beginning of a program execution, LPROV is loaded into process memory by a customized loader. It records the entrance and exit of library calls by manipulating symbol tables and maintains library call stacks for each thread. To be integrated with the audit logging techniques, LPROV also deploys a kernel module to collect syscall events. Only when trapped into a syscall, a process's library call stack is retrieved for output. To ensure the efficient processing in kernel, a daemon process in user space is designed to take over log delivery and optimization (e.g., reduction of redundant or duplicate logs). During production runs, when a syscall is made, its deriving path from the library perspective is disclosed by the library call stack on causality correlations.

Our contributions are summarized as follows.

• We propose an efficient provenance tracking system LPROV, combining library tracing in user space and syscall tracing in Kernel space. Whenever a provenance-related syscall is made from a thread, its full library-level execution path is also unveiled. Equipped with the library provenance, causality is revealed not only between explicit value-based input and output system entities but also inside the implicit fine-grained execution-based shared libraries.

• We devise a lightweight and efficient system-wide library tracing infrastructure. The tracing provides a friendly running environment for heavy-threaded programs and all the thread properties (e.g., concurrency) are well preserved.

• We evaluate our prototype and the results are promising. LPROV can precisely identify the provenance in malicious library, and it incurs only 7.0% runtime overhead (in geometric mean) and consumes 3 times less storage space (29.7% of the space for provenance data by BEEP [75]).

3.2 Motivating Example

The Linux Ebury attack in ssh service [101] motivates the importance of library aware provenance tracking. This attack leverages a stealthy backdoor to implant subsequent malicious binaries such as ssh clients or servers. The first version of Ebury attempts to replace ssh-related binaries such as *sshd*, *ssh* and *scp* by carefully crafted malicious binaries. How-

ever, the crafted programs are too obvious and attack ramification could be easily exposed to existing provenance tracking systems. Hence, Ebury evolves into exploiting well-camouflaged shared libraries rather than directly intruding the program bodies. In this paper, we reproduce a version of the library-base Ebury attack to show the effectiveness of LPROV.

Library-based Ebury To make the attack stealthy, library-based Ebury carefully chooses a particular library, *libkeyutils.so*, which is one of the libraries for Kerberos authentication. Specifically, Kerberos authentication is a widely used identity authentication protocol between ssh clients and servers and most Linux versions support it by default. In Linux, it is implemented by 4 libraries, *libkcrypto.so*, *libkrb.so*, *libkrbsupport.so* and *libgssapi_krb.so*. Among these libraries, the key management library *libkeyutils.so* is only called by `krb_get_notification_message` in *libkrb.so* which is, in fact, never called in the current Kerberos implementation. In other words, *libkeyutils.so* is a "dangling" library in ssh programs. Moreover, we observe that in most Linux versions, no other programs are using *libkeyutils.so* except the ssh service. Hence, the attacker chooses *libkeyutils.so* as it would only affect the ssh program without attracting attentions from users and security administrators.

Attack Scenario This is an exfiltration attack that aims at stealing users' private keys. An administrator is maintaining several servers and he generates public/private key pairs (one pair for one server or one pair for multiple servers) for remote login, which is protected by the ssh public key authentication. Considering the flexibility of server configuration, such as location changing, service switching and load balancing, those servers are managed by some dynamic domain name service (e.g., No-IP) where IP addresses are not fixed. Through an unverified package update, the *libkeyutils.so* library in administrator's laptop is replaced by a malicious one containing a backdoor in the library's *constructor*, transferring hosts' private keys to a remote attacker-controlled site `y.y.y.y`. Note that it is possible to make the program load the malicious library without physically replacing the library file. Specifically, the library can be placed into directories with higher search priority in the loading phase, without changing the original library file. After a few months, the administrator logs into a server and notices that the system has been compromised. He then realizes that his private key was leaked since the attacker successfully got through the login authentication.

Provenance Analysis The causal graphs tracking from the private key file `id_rsa.1` generated by BEEP [75] and LPROV are in Fig. 3.1. Note that the graph contains about thirty different remote server nodes and many file object nodes but we only analyze a small part of it (related to the provenance analysis). Since the ssh client is not a long-running program with an input handling loop, the unit partition of BEEP cannot take any effect here but all the objects are dependent on the whole process. Hence the network connection established for exfiltration in the library's *constructor* is causally correlated to *ssh* in the same way as other remote servers. When a user accesses any remote server through the ssh client, the corresponding public/private key is always read for identity authentication. Therefore, without extra evidence, it is impossible to differentiate the attacker's site `y.y.y.y` from other server addresses (e.g., `x.x.x.x`, `z.z.z.z`, and so on). Accomplishing the file transmission, the library's *constructor* outputs connection error messages and calls `exit` to terminate. Therefore, the ssh client would not connect to any server after the communication with `y.y.y.y`, preventing the attack being readily caught in the light of a suspicious ssh process associated with two different remote servers. From the administrator's perspective, the program termination is just regarded as a normal connection failure and no anomalies could be perceived. Moreover, the attack is conditionally triggered with a certain probability, hence most ssh connections launched from the client are benign ones. If the administrator is fortunate enough to obtain the whole and correct server-side logs such as login and DHCP, the suspicious connection to `y.y.y.y` could be identified. However, it is challenging to understand the root cause and internal profound details of the malicious library file.

In contrast to BEEP, LPROV's causal graph shows the execution-based provenance for the attack. In this clear context of execution paths, we can figure out that the private key was exfiltrated to the remote server `y.y.y.y` through the *constructor* function of *libkeyutils.so* in trivial efforts. Note that to simplify and clean the illustration, Figure 3.1 omits the provenance of the library file, but this will be detailed in Section 3.5.2.

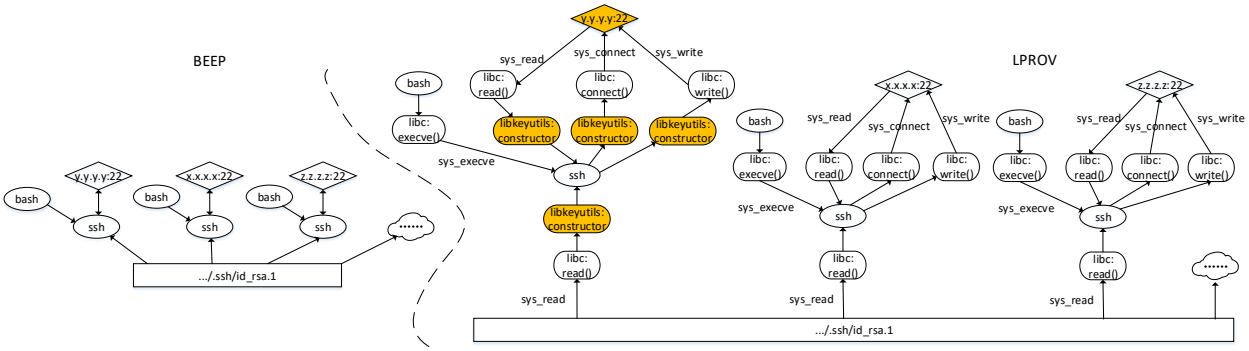

Figure 3.1. Causal graphs generated in provenance tracking for Ebury exfiltration attack. BEEP can not distinguish the attacker address but LPROV gives a clear attack context.

3.3 System Overview

LPROV leverages the same program unit instrumentation scheme from BEEP and ProTracer [75], [76], assuaging the concerns of dependence explosion. Fig. 3.2 illustrates the architecture of LPROV. Specifically, the customized loader mandatorily preloads the tracing library *lprov.so* into processes' memory at the program bootstrap and monitors the initialization procedure of libraries upon loading. The library *lprov.so* takes charge of tracing and storing library call events into a memory chunk shared with kernel. The kernel module and the user-space daemon are largely inherited from ProTracer [76] and we augment them to accommodate library-level events, but our contribution mainly lies in improving the library awareness of auditing-based provenance tracking by efficient library tracing. Our kernel module is responsible for (1) recording syscalls and (2) copying associated library call stacks into a circular buffer shared with the daemon process. The daemon pulls log entities from the shared buffer and outputs them to the log file for further provenance analysis. Note that to minimize the attack surface (i.e., to prevent hijacking from compromised libraries), external library codes are statically linked to the customized loader, *lprov.so* and the user-space daemon.

A shared library could be loaded through process initialization (by system loader) or calling `dlopen` on demand. In either way, when a library is loaded, *lprov.so* alters the offsets of exported function symbols in the library ELF header and redirects them to injected

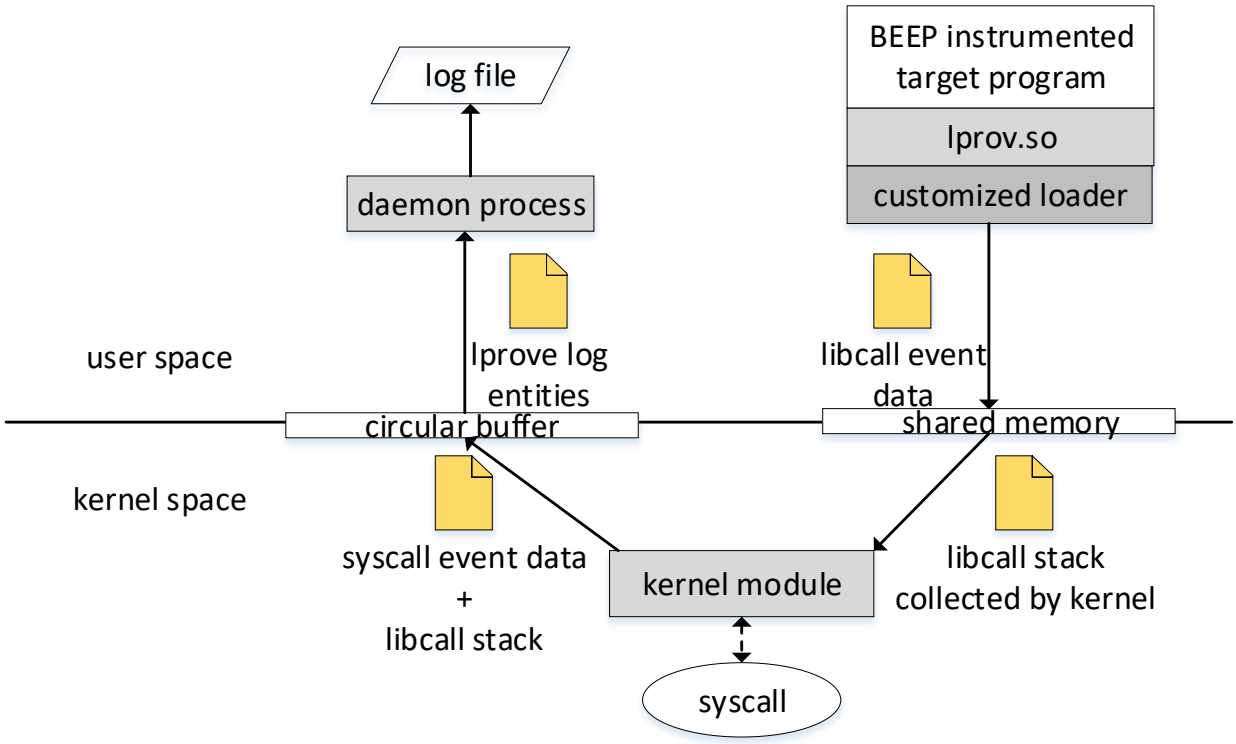

Figure 3.2. The architecture of LPROV: dashed lines denote control flow and solid lines denote data flow.

wrappers (i.e., entrance wrappers). In the entrance wrapper routine, the return address of a library call is then instrumented with another wrapper (i.e., exit wrapper) to catch function exit. During program execution, the two wrappers record the enter and exit sequences of library calls to the shared buffer which is read by kernel later. Note that such a library tracing procedure hinges on dynamic symbol resolution and its limitations will be discussed in Section 3.6. Details of library call tracing are discussed in Section 3.4.1. Compared with syscalls involving low-level kernel object operations, user-space library functions are called much more frequently. For instance, more than 18.2 million library functions are executed when a user opens *firefox*, visits the homepage of *New York Times*, and clicks the headline. Meanwhile, only about 753 thousand syscalls are invoked. However, many of those library function calls are less significant (e.g. `tolower, toupper`) for provenance analysis. Hence, instead of recording all the call sites of library functions, LPROV focuses on a small fraction of important library calls (e.g., `read, write, constructor`) and builds a concise,

yet comprehensive (regarding provenance tracking) library call graph. Only if a library call terminates at a syscall, its full call stack is considered for output. Other cases which do not interact with kernel are pruned . The design of kernel module is elaborated in Section 3.4.2.

The daemon fetches log data from the circular buffer as a consumer. A log entity is a syscall event annotated by a library call stack. Among candidate entities, the daemon only outputs the ones which are not duplicating any already-output log record. This part is dissected in Section 3.4.3.

Assumption In this paper, like other existing provenance systems built on audit logging, we trust the kernel and the user-space processing daemon. Apart from that, LPROV also assumes the integrity of the library tracing component in user space. The limitation of such assumptions will be discussed in Section 3.4.1 and Section 3.6.

3.4 Design and Implementation

3.4.1 Library Call Tracing

In this section, we discuss the detailed design of LPROV and compare it with *ltrace* (the most widely used library call tracing tool in Linux).

Design

The tracing functionality of LPROV is encapsulated as a library *lprov.so* loaded into the runtime memory of the tracee. It realizes logging library calls with three cooperative components: dynamic symbol redirector, entrance/exit wrapper and customized loader.

Dynamic Symbol Redirector This component serves as the pillar of *lprov.so*. It is composed of an overrider of `dlopen` and a constructor function (of *lprov.so*).

When importing a module, the loader typically runs in a lazy binding mode, where the resolution of external function symbols is not executed until the program reaches their first reference. The system resolver relies on the symbol table of library's exported functions to parse entrance addresses of external functions.

Hence, the overrider manipulates specific fields of the table to direct the resolution results to our injected wrapper routines (introduced in the next two paragraphs). Specifically, the

overrider first calls the original `dlopen` and obtains the library handle. Then, it locates the addresses of dynamic function symbols (i.e., `DYNSYM` section which contains the list of exported functions). Finally, the attributes of `st_value` and `st_info` inside `Elf_Sym` structure are modified for each symbol. The `st_value` attribute of a dynamic function symbol implies the offset of the function's entrance to the library handle. We change the values of `st_value` in order to interpose the functions.

However, if a library function is compiled as `IFUNC` whose implementation is bound during runtime, changing the value of `st_value` will cause the program to crash when the function is called since `st_value` of an indirect function indicates the address of the runtime resolver instead of the function entrance. Since these functions are specified as `IFUNC` and the types are stored in `st_info`, we also alter this value to make the system resolver handle indirect functions as non-indirect ones. In addition, the original address of an indirect function could be pre-fetched by `dlsym`. Hence, the constructor function of *lprov.so* interposes `dlopen` on all the libraries (including dependent libraries) in the program's ELF header, while the libraries loaded by `dlopen` during runtime are automatically instrumented by the overrider.

Entrance/Exit Wrapper The manipulated symbol table dispatches library calls to the corresponding function entrance wrappers. A library function's entrance routine (1) stores functions' original return addresses (stored at `(%esp)`) and stores programs' states, (2) logs library call entrance events, (3) redirects return addresses to exit wrappers, and (4) restores programs' states and resumes library function execution.

An exit wrapper (1) fetches functions' original address and storing programs' states and (2) pops out the last library call entrance events and resumes library function returns.

Customized Loader To guarantee that the library call tracing functionality is deployed before the target program executes any library function, the library *lprov.so* must be loaded before all others. In Linux, generally, this could be done by enabling the `LD_PRELOAD` environment variable. However, many anti-debugging and self-protecting techniques probe `LD_PRELOAD` and then refuse to execute when the variable is used. Moreover, according to Linux security policy, `LD_PRELOAD` referring to any untrusted path is ignored in loading phase when binaries' `setuid` or `setgid` bit is set. To this end, rather than using the `LD_PRELOAD`, we modify the loader to enforce the pre-loading of *lprov.so* anyway through `do_preload`.

Apart from all the exported functions, a library's constructor plays a critical role in the library loading mechanism. It is called by the loader upon the library importing procedure, before the library handle is actually returned. Therefore, we customize the loader further to monitor the execution of libraries' constructors. Specifically, in the loader's initializer `_dl_init`, at the execution points where `call_init` is called and returns, the constructor entrance event is pushed and popped as regular library calls.

Design Choices

In this section, we elaborate our design choices comparing with the design choices of *ltrace*, and highlight advantages of LPROV. In particular, *ltrace* leverages breakpoints and debugging mode in order to interpose library calls, which results in high overhead and missing library call events. In contrast, LPROV uses an dynamic instrumentation approach which outperforms *ltrace* in various aspects. In the following, we elaborate the advantages from five different perspectives.

Interposing Nested Library Calls To interpose library calls, *ltrace* leverages `ptrace` which imposes high overhead. In addition, *ltrace* inserts software breakpoints (`INT 3`) at the PLT (Procedure Linkage Table) trampoline entries of library functions. Note that software breakpoints incur significant overhead as the entire process is stopped when a breakpoint is reached. More importantly, the resolutions of external function symbols in different libraries (modules) are independent and they all hold their local PLTs. The breakpoint insertion of *ltrace* is confined within the PLT trampoline entries of the program (the main module) but all the libraries' PLT segments are not instrumented. Hence, if library calls are nested, *ltrace* is unable to trace the inner ones.

Unlike *ltrace*, LPROV focuses on the symbol resolution phase inside the loader. Hence, it can trace nested library calls with significantly lower overhead compared with *ltrace*.

Interposing Non-PLT Library Calls Functions in a library can be called through indirect calls using function pointers. Such calls do not go through PLT trampolines hence *ltrace* is not able to interpose such library calls[1].

[1]↑https://linux.die.net/man/1/ltrace, the latest document claims that *ltrace* can handle the `dlopen` case but the latest software version 0.7.3 still does not have this feature.

Since LPROV operates through dynamic symbol tables, all library calls by function pointers derived during symbol resolution can be traced in LPROV.

Constructor Function While constructor function initializes variables and states, it can also execute any code upon library loading. Hence, without the awareness of constructor function, distinguishing library calls inside it from the ones outside is particularly challenging. Unfortunately, *ltrace* is not capable of tracing the constructor functions as it focuses on PLT trampolines.

The customized loader of LPROV makes sure that LPROV traces all the constructor functions as well as library function calls within the constructor functions.

Thread Support *ltrace* leverages software breakpoints (i.e., `INT 3`) to interpose library calls. Unfortunately, when a program halts on `INT 3`, it enters a SIGTRAP status where all the threads are suspended. In addition, to be informed of function returns, *ltrace* inserts breakpoints at the return addresses, which would trap the execution of other threads into unexpected breakpoint handling routine when text segments are shared in a thread group. As a result, *ltrace* hurts thread concurrency, eventually incurring additional overhead in multi-threaded applications.

As LPROV does not rely on software breakpoints, it does not have any of the aforementioned issues. It offers a thread-friendly tracing environment, where thread concurrency properties are preserved well.

Runtime Overhead Compared to LPROV, the breakpoint scheme of *ltrace* incurs much higher overhead due to context switch and the limited thread support. Table 3.1 shows the runtime overhead of the two tracing tools in the same workloads of four server-side programs (*httpd, simplehttpd, proftpd, sshd*), eight client-side programs (*firefox, filezilla, lynx, links, w3m, wget, ssh, pine*) and three editors/readers (*vim, emacs, xpdf*). The Apache Benchmark [102] tool is used to measure the two web service programs, *Apache httpd* and *simplehttpd*, and *ftpbench* [103] is used to measure *proftpd*. For *firefox*, we use the standard browser benchmarking tool *SunSpider* [104], and we use corresponding program scripts for all the other programs. In the columns of *ltrace*, `N/A` in *httpd* and *firefox* indicates that programs do not terminate in a reasonable time limit (e.g., meaning that it incurs more than 1000% overhead) or just crash. This is because *ltrace* incurs particularly high overhead in multi-threaded programs. Note that to measure the performance of *ltrace*, we develop a

Table 3.1. LPROV has much lower runtime overhead than *ltrace*

Program	ltrace		LPROV		#Gap
	# of calls	overhead	# of calls	overhead	
httpd	N/A	N/A	8764.7K	53%	N/A
simplehttpd	946.4K	563%	965.5K	28%	19.1K
proftpd	1407.8K	622%	1494.3K	36%	86.5K
sshd	4987.0K	769%	5019.8K	44%	32.8K
ssh	3668.6K	813%	3741.6K	30%	73.0K
firefox	N/A	N/A	394461.6K	126%	N/A
filezilla	7311.5K	988%	7383.2K	47%	71.7K
lynx	912.6K	672%	933.0K	34%	20.4K
links	659.7K	565%	697.8K	29%	38.1K
w3m	757.2K	622%	785.3K	41%	28.1K
wget	448.8K	390%	453.3K	30%	4.5K
pine	1092.8K	522%	1103.6K	34%	10.8K
vim	8276.0K	734%	8336.9K	37%	60.9K
emacs	3053.2K	658%	3104.4K	31%	51.2K
xpdf	781.5K	742%	803.0K	40%	21.5K

simple library hooking module that leverages the same techniques used in LPROV without any additional provenance related components. Specifically, it only records entrances and exits of library calls.

The primary results show that LPROV outperforms *ltrace*. The overhead incurred by *ltrace* is 10-20 times more than that of LPROV. Moreover, LPROV can trace more library calls as *ltrace* is not capable of tracing nested and Non-PLT library calls. Observe there exists a gap between the amounts of recorded library calls.

Data Integrity

As malicious libraries and recorded library call stacks reside in the same memory space, the tracing data might be compromised during runtime. To mitigate the issues, we can leverage complementary address space (re)randomization techniques [105]–[107] such as ASLR (Address Space Layout Randomization). Specifically, randomization techniques make sure that the library of LPROV is loaded into a random address. Moreover, we can leverage these techniques to randomize addresses of our data structures. Also, hardware features can be leveraged to ensure data integrity [108] as well.

3.4.2 LPROV Kernel Module

Kernel Event Tracing A syscall interrupt (INT 80) traps program execution into kernel mode and the kernel module steps in at this moment. Linux offers several convenient and stable tracing facilities in kernel, LSM [109] (Linux security module), Tracepoints [110] and KProbes [111], [112] to register customized kernel event handlers. Specifically, LSM is often applied to implement MAC (mandatory access control) around kernel objects such as *inode*. It operates at a finer granularity than syscalls, which can incur additional overhead when a syscall accesses an object more than once, and miss customized syscall events introduced by unit partition in BEEP and Protracer [75], [76]. Tracepoint is a lightweight kernel tracing infrastructure which is adopted by many high-performance tracing tools such as Linux perf and audit logging. Unlike LSM, tracepoint allows tracing kernel work flows at different granularities (either object or function) through pre-defined tracing events. It statically embeds global event-tracing placeholders into kernel source code and users can then register effective probe functions on those tracepoint instances. Complementary to tracepoint, kprobe is a dynamic tracing infrastructure for kernel debugging. Among the three techniques, kprobe has the finest kernel tracing granularity since it provides the interface which can insert callbacks at any kernel-space instruction address, however, as a result of its purely dynamic design operating on software breakpoints, it has higher overhead than tracepoint. To this end, we select tracepoint as the underlying event handling mechanism to provide the workaround for syscall tracing inside the kernel module.

Tracing Target For provenance purpose, only the syscalls related to explicit causality deduction are considered in kernel tracing. The set of tracing targets is the same as Protracer [76], including syscalls on basic file read/write operations, file redirection syscalls, IPC syscalls, process management syscalls and customized syscalls to mark the unit in/outs. In particular, LPROV enlarges the set by another category of syscalls, memory permission manipulating syscalls. Change of memory access right is a critical clue for provenance inference, especially in user-space library tracing. For example, a function in library A alters the permission of pages allocated to library B and modifies the image of B dynamically. BEEP and Protracer are oblivious on such dependence because the inter-unit memory read event

defined in them is an explicit value-based read that cannot model an implicit *execution-based read*. Hence, LPROV additionally collects syscall events of `mprotect`[2] and `mmap`. When library A invokes `mprotect` to set the `PROT_WRITE` or `PROT_EXEC` flag to a memory chunk mapped to library B, then the corresponding memory section of B has runtime dependence on A. In addition, syscall `mmap` can be utilized to obtain a memory chunk with specific permissions and this is the default way library binaries are loaded into process memory by the program loader. Library tracing is accomplished by *lprov.so*, and hence it is unnecessary to additionally handle the general library mapping here. Specially, if files (including libraries) are non-anonymously mapped, they will be considered either input or output object of the process according to the permissions and opening modes of file descriptors. For all the other anonymously (i.e. `MAP_ANONYMOUS`) mapped executable pages, we regard them as a part of the main module and then the main module is considered dynamically dependent on the `mmap` invoker.

Event Collecting For a syscall instance, in addition to the standard syscall event which is recorded by existing systems built on audit logging, LPROV also correlates the thread's library execution path to this event. The kernel module retrieves library call stacks from the buffer shared with user-space applications, packs syscall events with the stacks into log entries and delegates the log outputting task to the user-space daemon through the circular buffer. By doing so, kernel does not need to wait for the previous events to be completely processed. The size of the buffer can be also configured so that kernel would not wait when the buffer is full. It can be also configured to drop events when the buffer is full. To accelerate the library call processing inside kernel, the buffer maintaining programs' library call stack is a per-thread buffer indexed by thread id so that kernel can access the memory efficiently. To prevent expensive dynamic memory management from weighing in, we choose to pre-map the per-thread buffer with a fixed size when the kernel module is loaded.

[2]↑The more efficient syscall `pkey_mprotect` is not included here since it is not supported until Linux-4.9 kernel and it also requires specific hardware assistance.

3.4.3 LPROV Daemon Process and Log Analysis

Daemon Process The daemon keeps reading log events from kernel through the producer-consumer buffer. Since there are only one producer (producer does not overwrite data) and one consumer, the circular buffer is implemented in lock-free mode. Plumbed from Protracer [76], the daemon marshals a thread array to perform log processing and logs from one process should be tackled by only one thread to maintain processes' consistent and complete execution context. The on-the-fly log processing phase aims at reducing duplicate events on the same system object within one unit. For example, downloading of a large file inside browsers is fulfilled by thousands of socket-reading and file-writing operations, but only one of them needs to be recorded for provenance purpose. In addition, the processing also connects redirected files (`dup` function group) to prevent causality loss. Note that we do not apply the log reduction scheme playing with taint propagation in Protracer [76] since the taint cannot clarify the implicit execution path from the source to target hence not applicable to our purpose. Moreover, if we combine LPROV with the tainting technique, a taint must be spawned for each library call stack per syscall event, which does not take any advantage over the direct logging. Eventually, the filtered log entities are delegated to the log outputting thread to generate the log file on disk.

Log Analysis LPROV provenance tracking is expected to answer how a system object is affected and how it affects the system in the perspective of syscall and library events. Hence, we provide both of backward and forward tracking, disclosing the deriving path and the aftermath of objects in a directed provenance graph. The log analysis algorithm is similar to that in BEEP [75] and Protracer [76] but LPROV augments the provenance graph by handling additional events on memory permission manipulation and capturing library-level execution causalities.

Algorithm 3 describes the process of backward tracking in LPROV. It takes as input the reverse-ordered log and a designated system object, and then generates a causal graph by correlating system entities in LPROV events. In Algorithm 3, a tuple of *pid*, *uid* and *uinst* defines a unique runtime process unit (line 2). LPROV has seven types of *event*, where `SYS_R/SYS_W` is the system read/write event such as socket or file read/write, `SYS_PROC`

Algorithm 3: LPROV Log Analysis Algorithm

Input : Log_{rev} - LPROV log in reverse event order
: obj_t - designated tracking object
Output: $Graph$ - LPROV provenance graph
Def. : $Object$ - set of objects causally related to obj_t
: $event = \{pid_e, uid_e, uinst_e, type_e, obj_e, para_e, lstack_e\}$ - a LPROV event in Log_{rev}
: pid - process id
: $uid, uinst$ - unit id and unit instance
: $type_e \in \{\texttt{SYS\_R/SYS\_W}, \texttt{SYS\_PROC}, \texttt{MEM\_R/MEM\_W}, \texttt{MEM\_PERM}, \texttt{MEM\_MAP}\}$ - event type
: obj_e - system object in an event
: $para_e$ - event parameter
: $lstack_e$ - libcall stack including the syscall
: $Memory[pid]$ - set of memory use instances in pid
: $rel[u]$ - whether unit u is causally related to obj_t
: $edge = (obj_a, obj_b, lstack)$ - a directed causality edge from obj_a to obj_b interconnected by $lstack$
Init. : $Object \leftarrow \{obj_t\}$
: $Graph \leftarrow \Phi$
: $Memory[pid] \leftarrow \Phi$
: $rel[u] \leftarrow False$

```
1  foreach event ∈ Log_rev do
2      unit ← (pid_e, uid_e, uinst_e)
3      if type_e ∈ {SYS_W, SYS_PROC, MEM_PERM} ∧ obj_e ∈ Object then
4          Graph ← Graph ∪ {edge(pid, obj_e, lstack_e)}
5          Object ← Object ∪ {pid_e}
6          rel[unit] ← True
7      end
8      if type_e == SYS_R ∧ rel[unit] == True then
9          Graph ← Graph ∪ {edge(obj_e, pid, lstack_e)}
10         Object ← Object ∪ {obj_e}
11     end
12     if type_e == MEM_R ∧ rel[unit] == True then
13         Memory[pid] ← Memory[pid] ∪ {para_e}
14     end
15     if type_e == MEM_W ∧ para_e ∈ Memory[pid] then
16         Memory[pid] ← Memory[pid] − {para_e}
17         rel[unit] ← True
18     end
19     if type_e == MEM_MAP then
20         if para_e ∈ {RDONLY, RDWR} ∧ rel[unit] == True then
21             Graph ← Graph ∪ {edge(obj_e, pid, lstack_e)}
22             Object ← Object ∪ {obj_e}
23         end
24         if para_e ∈ {WRONLY, RDWR} ∧ obj_e ∈ Object then
25             Graph ← Graph ∪ {edge(pid, obj_e, lstack_e)}
26             Object ← Object ∪ {pid_e}
27             rel[unit] ← True
28         end
29     end
30 end
```

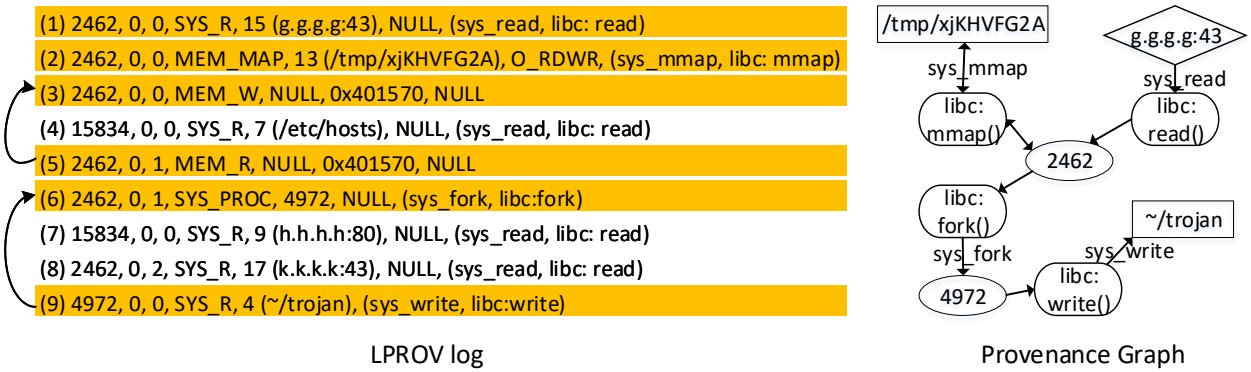

Figure 3.3. Log analysis example.

is the process creation event, `MEM_R/MEM_W` is the memory read/write event used to infer unit dependence, `MEM_PERM` is the memory permission event that makes executable memory pages writable (`mprotect` and anonymous `mmap`) and `MEM_MAP` is the non-anonymous memory mapping event. In a LPROV *event*, obj_e is file descriptor for `SYS_R/SYS_W` and `MEM_MAP`, *pid* for `SYS_PROC`, memory section of a module/library for `MEM_PERM` and invalid for other event types; $para_e$ is the memory address for `MEM_R/MEM_W`, file and memory permission (`RDONLY`, `WRONLY` or `RDWR`) for `MEM_MAP`, and invalid for other event types; *lstack* is the library call stack for each causality edge and it is invalid for `MEM_R/MEM_W` events. Specifically, the algorithm can be inducted into three causality rules: a process unit has causality to the tracking object if (1) an event within the unit writes/creates any system entity that has causality to the tracking object (line 3-7, line 24-28) or (2) the unit writes a memory chunk which is read by another unit that has causality to the tracking object (line 12-18); while a system entity has causality to the tracking object if (3) the entity is read by an event within a unit that has causality to the target object (line 8-11, line 20-23). Fig. 3.3 is a log analysis example, where log entries are in LPROV log format $event = \{pid_e, uid_e, uinst_e, type_e, obj_e, para_e, lstack_e\}$, and the target object `trojan` is casually correlated to the unit $(pid = 2462, uid = 0, uinst = 0)$. The forward tracking algorithm is just the reverse version of Algorithm 3, and hence it is omitted in discussion.

3.5 Evaluation

We set up four machines (machine A, B, C, and D) with similar hardware configuration (16GB RAM and Intel i7 CPU) in our evaluation experiments. In order to compare the overall performance, those machines are all deployed LPROV and BEEP.

We select 11 programs from Table 3.1 for measurement as some of them share the same functionalities. Not all of them are instrumented by BEEP as programs like *ssh* do not have an event handling loop and they are not designed to run for a long time.

We assign one machine (i.e., machine A) as the server (running both server-side and client-side programs) and the other three (e.g., machine B, C, and D) as pure client machines (running client-side programs only). We refer the anonymous users of these machines with the same alphabet letters (User A, B, C, and D).

User A configures, manages and maintains the server service. The server operates a tiny web server for a group project, an FTP server for file sharing and an SSH server for remote accessing. The other three users (Users B, C, and D) are required to actively communicate with the designated server and use the selected programs during the experiment. Besides the necessary communication with the server, the three users also have their own behavior profiles. User B mainly uses the machine to watch TV or movies online, visit social network sites, browse news and chat with friends. User C undertakes most of his project coding work on the assigned machine. He usually downloads and reads programming manuals or documents. User D is preparing some coming interviews. She mainly accesses her email account for personal communication, watches presentation videos and visits Q&A websites to collect interview-related materials.

3.5.1 Performance Overhead

Storage Overhead The performance experiment lasted two weeks and the results are shown in Table 3.3. From the results, we can observe that in spite of logging the library call stack, the storage consumption of LPROV is only 29.7% of BEEP. Since LPROV applies a on-the-fly reduction phase, the generated log has much less redundancy compared to the original audit event log. Note that because LPROV logs additional library-level events and

Table 3.2. Comparison of storage overhead between LPROV and ProTracer in a two-week performance experiment

User	ProTracer		LPROV		ProTracer/LPROV	
	#item(M)	size(GB)	#item(M)	size(GB)	item	size
A	17	11	59	45	28.8%	24.4%
B	28	16	62	41	45.2%	39.0%
C	14	7	42	26	33.3%	26.9%
D	12	6	33	20	36.4%	30.0%
Avg.	18	10	49	33	36.7%	30.3%
Prog.	ProTracer		LPROV		ProTracer/LPROV	
	#item(M)	size(GB)	#item(M)	size(GB)	item	size
httpd	5.3	3.0	13.9	8.3	38.1%	36.1%
proftpd	3.8	2.2	8.3	5.4	45.8%	40.7%
sshd	4.3	2.5	15.4	10.8	27.9%	23.1%
ssh	1.4	0.9	4.1	2.8	34.1%	32.1%
firefox	27.1	15.1	68.0	49.6	39.9%	30.4%
filezilla	1.8	1.0	2.7	1.6	66.7%	62.5%
w3m	0.5	0.3	1.2	0.9	41.7%	33.3%
wget	0.4	0.2	0.8	0.5	50.0%	40.0%
pine	0.3	0.2	1.0	0.8	30.0%	25.0%
vim	3.4	1.9	15.2	11.0	22.4%	17.3%
xpdf	2.0	1.1	6.0	3.9	33.3%	28.2%

the tainting scheme in Protracer [76] is not applicable in our context, it appears to be unavoidable that LPROV has greater storage overhead than Protracer. We apply the object tainting technique proposed in ProTracer to BEEP's logs, performing the comparison of storage overhead between LPROV and ProTracer. As Table 3.2 demonstrates, the storage consumption of ProTracer is 30.3% of LPROV. We argue that this is a tradeoff between cost and benefits.

Runtime Overhead As shown in Figure 3.4, while LPROV logs more user-space library information, its runtime performance is still competitive and acceptable. Specifically, the geometric mean and arithmetic mean of the programs' overhead in LPROV are 7.0% and 8.6%, compared to 8.4% and 9.8% in BEEP, 4.5% and 5.3% in ProTracer. We notice that the programs' performance in Figure 3.4 achieves significant improvement from Table 3.1. This is because the daemon process decouples the onerous log outputting task from the library tracing component. Note that in runtime overhead benchmarking, instead of imposing

Table 3.3. Comparison of storage overhead between LPROV and BEEP in a two-week performance experiment

User	BEEP		LPROV		LPROV/BEEP	
	#item(M)	size(GB)	#item(M)	size(GB)	item	size
A	139	116	59	45	42.4%	38.8%
B	185	146	62	41	33.5%	28.1%
C	119	91	42	26	35.3%	28.6%
D	109	90	33	20	30.3%	22.2%
Avg.	138	111	49	33	35.5%	29.7%
Prog.	BEEP		LPROV		LPROV/BEEP	
	#item(M)	size(GB)	#item(M)	size(GB)	item	size
httpd	40.9	32.3	13.9	8.3	33.9%	25.7%
proftpd	29.2	23.6	8.3	5.4	28.4%	22.9%
sshd	33.0	26.8	15.4	10.8	46.7 %	40.3%
ssh	10.5	8.1	4.1	2.8	39.0%	34.6%
firefox	207.7	168.9	68.0	49.6	32.7%	29.4%
filezilla	13.7	10.6	2.7	1.6	19.7%	15.1%
w3m	3.9	3.1	1.2	0.9	30.8%	29.0%
wget	2.6	2.0	0.8	0.5	30.8%	25.0%
pine	1.9	1.6	1.0	0.8	52.6%	50.0%
vim	26.4	21.0	15.2	11.0	57.6%	52.4%
xpdf	15.0	11.4	6.0	3.9	40.0%	34.2%

intensive and bursting tasks, we apply day-long program workloads from general use cases which we profiled from the four deployed machines/users. Such measurement reflects real-world scenarios where APT attacks happen (i.e., institute/organization/enterprise environment). It also reasonably amortizes the expensive GUI constructing/destroying computation at program opening/closing. There is an outlier. LPROV incurs high overhead (28.9%) on *firefox*. This is caused by the fact that *firefox* makes massive library calls (around 3 million per page loading in average). We believe that the problem could be attenuated by reducing the library logging scope as discussed in Section 3.6.

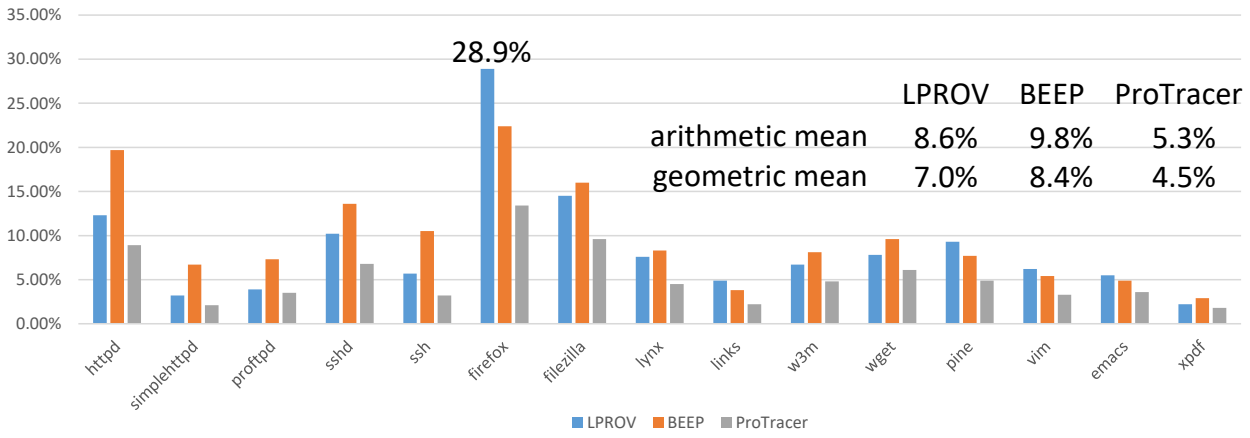

Figure 3.4. Comparison of runtime overhead of BEEP, ProTracer and LPROV.

3.5.2 Case Study

Ebury Variant Attack

We use another variant of Linux Ebury attack [101] to show the effectiveness of LPROV in provenance tracking. The attack leverages a malicious library to alter memory images of benign programs. The attacker steals a student's campus credential and leverages the credential to spread the malice. Note that universities usually assign each student one credential for all the campus resources and it would lead to serious privacy issues if the credential was stolen. Given a compromised credential on an end-host, LPROV can then assist identifying the attack source and preventing future attempts.

Attack Scenario Bob is a college student and he usually uses ssh service to reach lab computers on campus. He installed LPROV and BEEP on his laptop and enabled them in his daily application usage. One day, he was notified by the campus network administrator that his account was sending spam and phishing emails. After necessary investigation, Bob proved to be innocent but his account was temporarily deactivated and he was required to use the two-factor authentication in the future. Later, he wants to know how the attack happened as he did not ever tell anyone else his credential information, type in his credential on others' machines or open any phishing links.

Note that as ssh client stores all the public keys of known hosts to verify their identities, the program should have displayed a warning message (e.g., the public key mismatch) if the

attack happened on the connection to a known server. Unfortunately, Bob confirmed that he deleted the the `known_hosts` file as many websites suggest to do so (assuming that the server changed its key pair) [113]. Moreover, the latest variants of Linux Ebury can instrument the authentication part to suppress the mismatch warning [101].

Attack Investigation Bob did not store his student credential in any file. Hence, instead of concentrating on sensitive files (as we did in Section 3.2), he focused on analyzing logs from all the processes he recently used. Eventually, he obtained Fig. 3.5 (the graph is simplified) when dissecting an *ssh* process and the graph of LPROV contains the complete attack chain. As demonstrated in Fig. 3.5 (the red dash line is auxiliary for presentation), the PLT image permission of ssh process was manipulated by the constructor of *libkeyutils.so.1* and then the process connected to a remote address `b.b.b.b:22` which is a machine outside of the campus. Based on this result, we can infer that the constructor altered the process's PLT entries (probably the function path of `connect`) and redirected the connection to the attacker's server. Further, we apply backward tracking on the object of the library file. It reveals that the malicious library is downloaded by *firefox* from `a.a.a.a:43`. Then, it was implanted into a target directory through unverified package installation. Note that this attack cannot be uncovered in BEEP due to its lack of user-space library semantics. Specifically, in BEEP, even if backward tracking is performed on all the loaded libraries, the root cause of the attack would remain unexposed unless `a.a.a.a` is pre-known as malicious.

Library Vulnerability Exploitation

We use an infiltration attack reproduced from CVE-2015-7547 [114]–[116] to show the effectiveness of LPROV in the context of remote vulnerability exploitation. There is a buffer-overflow vulnerability in *libresolv.so* that can be triggered by maliciously crafted DNS responses. Specifically, when a client calls `getaddrinfo` with `AF_UNSPEC` to resolve a domain name, a pair of IPv4 and IPv6 requests would be sent by `send_dg` and `send_vc` in *libresolv.so*. If the response is larger than the pre-allocated 2048 bytes buffer, the two functions allocate additional heap buffers and the mismanagement of the buffers leads to the exploitation. Further details can be found on [117].

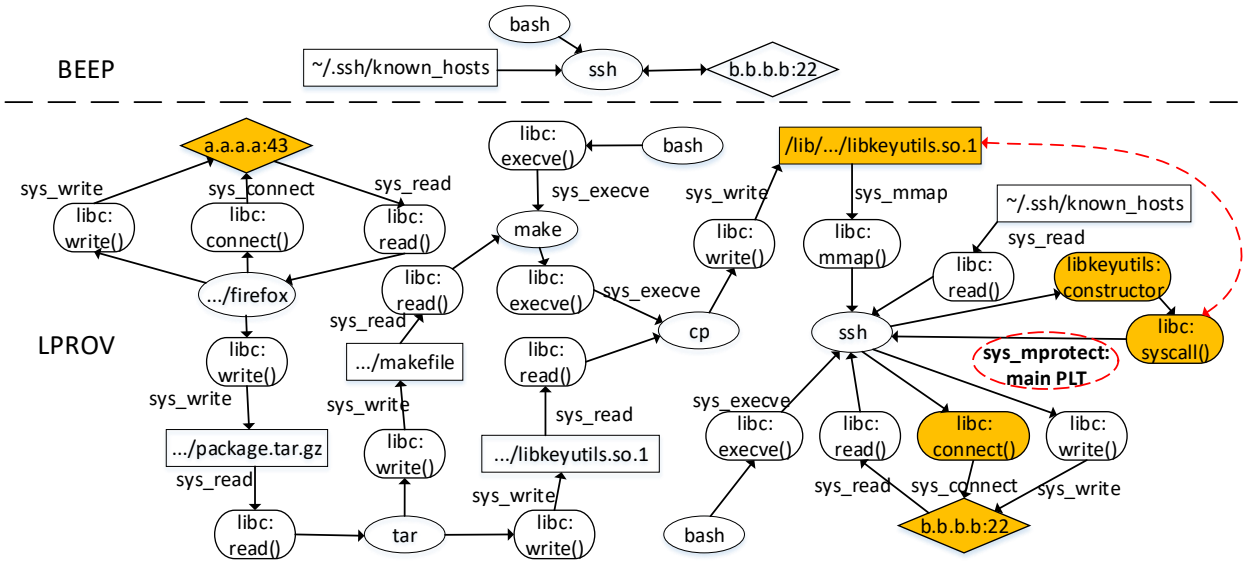

Figure 3.5. Provenance graphs generated by BEEP and LPROV for the student credential stealing attack.

Attack Scenario The attacker acts as a local DNS proxy and responds all the DNS requests from the victim client which conducts DNS resolution on `getaddrinfo`. When receiving requests from the victim client, the attacker sends crafted responses to implant a malicious payload [115], [116]. The payload launches a shell executing *wget* to download a file `secret.txt` that replaces the current secret file under the victim's home directory and then exits. Several days later, the file was found compromised.

Attack Investigation In Fig. 3.6, BEEP fails to obtain the origin of the attack rooting from the compromised library, while LPROV successfully captures the execution path of the exploitation when tracking from `secret.txt` (`name4_r`, `nsearch` and `sndmmsg` are shorts for `_nss_dns_gethostbyname4_r`, `__libc_res_nsearch` and `__sendmmsg`). The *bash* process is spawned inside *client* when executing `__libc_res_nsearch`. Further, by checking the library-level provenance on the exit event of `client`, we conclude that the attack is caused by a vulnerability exploitation because `__libc_res_nsearch` does not return when `client` terminates. Note that LPROV effectively identifies the attack provenance even though the library call stack does not include `send_dg` and `send_vc` which are static functions (Related discussions can be found in Section 3.6).

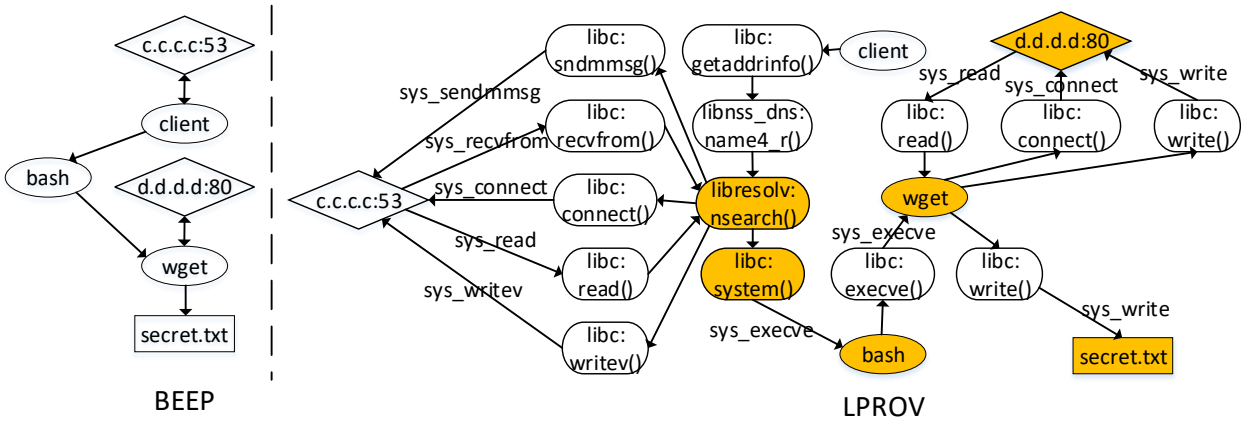

Figure 3.6. Provenance graphs generated by BEEP and LPROV for library vulnerability exploitation.

Library Loading Analysis

DARPA Transparent Computing program [118] investigates library loading processes in an effort to thwart advanced attacks that leverage malicious libraries. To demonstrate the effectiveness of LPROV, we test a sample library program provided by the red team of DARPA TC program. This case focuses on understanding internal details of library loading.

Scenario (1) Process *libloader A* is launched by *bash*. (2) *libloader A* spawns a thread B that prints out some text. (3) *libloader A* maps */dev/shm/testlib* into memory and reads *testlib.so* into the mapped memory. (4) *libloader A* dynamically loads */dev/shm/testlib* by `dlopen` and the library's constructor spawns another thread C that prints out some testing text. (5) *libloader A* calls 8 library functions `f1-f8` by `dlsym` and each function prints out some text. (6) *libloader A* prints out some text and exits.

Provenance Analysis Fig. 3.7 presents the provenance graphs tracking from the standard output device `/dev/stdout` and the two threads (B and C), where `testlib`, `libptd` and `ptd_create` are shorts for `/dev/shm/testlib`, `pthread_create` and `libpthread`. LPROV accurately correlates *testlib.so* and */dev/shm/testlib*, reveals the creation of thread C during */dev/shm/testload* loading and clearly distinguishes output behaviors from the 8 different library functions, while BEEP misses all of those details due to the mishandling on memory mapping and the oblivion on library events. Note that the graph has two *libloader A* nodes

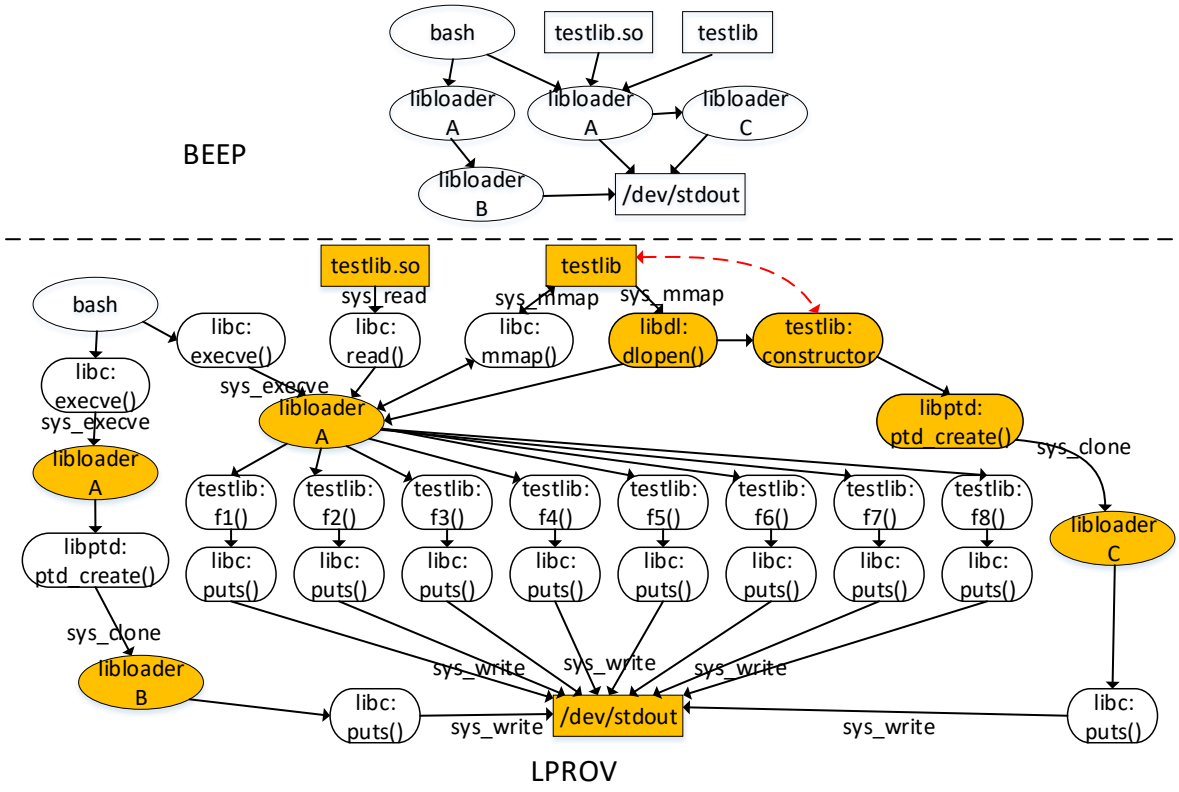

Figure 3.7. Provenance graphs generated by BEEP and LPROV for DARPA library loading case.

spawned by *bash* because thread B has no causality to the two files */dev/shm/testlib* and *testlib.so* and this would be clarified when tracking from thread B (Fig. 3.7 is a combination of the provenance graphs on the three tracking objects).

3.6 Discussion

Library Attestation Binary attestation is a widely used technique to ensure program integrity. For instance, operating-system-level virtualization techniques such as application container and enclave on trusted hardware can leverage software attestation to provide trusted computing platforms. While it can be used to detect and prevent possible compromise on libraries, local attestation can be thwarted by replacing the attestation measurement and remote attestation can be compromised if the attestation interface is undermined in ad-

vance. Moreover, LPROV can provide detailed contextual information of attacks (e.g., how the system is compromised and which files/processes are affected) while library attestation would simply detect and thwart attacks. To this end, we argue that monitoring internal library behaviors is a crucial primitive to enhance the security of libraries.

Tracing Protection Like other approaches, LPROV assumes the kernel and user-space tracing facilities are benign and uncompromised. Note that it is a general assumption shared among various tracing systems. This assumption can be relaxed if the implementation of LPROV submerges into hypervisor level [119]. However, the hypervisor tracing has limited portability and it suffers from the high overhead. Guarding user-space tracing tools is a knotty problem as they can be subverted by other user-space programs. However, the *initial tampering* with the tracing infrastructure can be accurately recorded and users can revoke the trust on the logs from that time point. The same strategy is also applicable in tracee's runtime, where the library tracing is trusted as long as the memory image of *lprov.so* or the customized loader is not altered via `mprotect`. Note that if malicious libraries residing in the tracee can locate and overwrite the shared memory containing tracee's library events (e.g., via memory disclosure vulnerabilities), the initial tampering would not be captured since no syscalls are triggered. Nevertheless, we argue that hardening memory operations is an orthogonal research effort to LPROV.

Tracing Coverage LPROV traces library calls through the standard symbol resolution in `dynsym` table. Hence, LPROV might be less effective if malicious libraries opt for customized loading, static linking, or dynamic binary generation. For instance, executable binaries can be encoded into a library, then decoded and mapped at runtime. The following function execution (through hardcoded offsets) inside the mapped memory images is invisible to LPROV. Also, exported functions inside the same library can have execution dependence and it can be handled in either static linking or dynamic symbol resolution. For example, string-related functions like `strcpy` and `strlen` are frequently used inside other *glibc* functions, such as `getenv` and `setenv`. However, instead of inserting PLT trampolines at the library's ELF header to resolve them during runtime, *glibc* opts to statically link those functions as local ones with the `hidden_def` attribute. Unfortunately, the current implementation of LPROV is not able to identify such nested library calls (e.g., `strcpy` in `getenv`) as the inner

ones are not resolved through the symbol resolver. However, note that we can still capture the outer functions (e.g., `getenv`) which are sufficient in practice. We argue that even with these limitations, it is still effective in the perspective of provenance investigation.

System Overhead As shown in evaluation, LPROV still has non-trivial overhead on *firefox* and it is actually due to the program's extremely heavy execution dependence on library functions. To mitigate this, LPROV can opt to whitelist several libraries and only trace recently changed ones, for example, libraries updated in the past four weeks. However, memory images of libraries cannot be trusted due to the manipulation of memory access permission from `mprotect`. LPROV hence needs extra communication channel between kernel and user space to dynamically deploy tracing on libraries whose memory image is contaminated during runtime.

3.7 Related Work

Non-unit audit logging In recent years, significant efforts have been taken on the system-level audit logging [78], [79], [81]–[83], [86], [87], [120]–[123]. They track system objects to infer mutual dependence for provenance investigation. Equipped with the generated system logs, root causes of attacks are revealed by backward analysis [79], [80], [124] and the aftermath of attacks is identified by forward tracking [78], [86], [87], [96], [121], [123]. Nevertheless, most of them regard the whole lifecycle of a process as a single system entity, which incurs dependence explosion problem. To assuage such concerns, researchers search for other system resources to complement syscall events. File offset is additionally leveraged in [84] to handle file-related syscalls to provide fine-grained tracing. However, it is only a specific optimization on file objects but not generic for other entities. In [125], memory operations between pages is utilized to establish low-level object dependence. But due to its lack of program semantics, the system is not sufficiently effective.

Unit-based audit logging Unit-based event tracing is the state-of-art in system-level logging and it follows the line of work that has been done in audit logging. Improved from those coarse-grained techniques, unit-based schemes provide fine-grained provenance inference by program partitioning. In BEEP [75], relying on the observation that long-

running programs usually respond user inputs by event handling loops, programs are trained by dynamic program analysis to extract those loops. Then the loops are instrumented by customized syscall events at both of enter and exit. Based on this technique, the execution of programs is partitioned into multiple loop instances and each instance is named a program unit. Therefore, dependence between input and output objects is efficiently disambiguated by confining causality correlation within the same unit. ProTracer [76] improves BEEP by applying dynamic object tainting to unit processing and decoupling high-overhead audit logging from event tracing. It performs between system tainting and audit logging to lower runtime and storage overhead. Each accessed object is assigned a unique taint and the taints are propagated upon object read. An event is considered for logging only if it is an output event. Furthermore, in the offline processing, the units sharing the same taint set are collapsed to make the generated graph concise. In MPI [3], the unit is refined to data structure instances but it requires efforts in developer annotation. Note that LPROV leverages existing unit-based provenance techniques.

Hybrid approaches In [126], to infer the causality relationship between syscalls, dependence analysis on low-level instructions is entailed. Such technique requires instruction instrumentation and incurs non-trivial runtime overhead. Since it works on deep and fine-grained program analysis, it involves much low-level memory dependence such as dynamic memory management which has no importance in causality inference. Barham et al. proposed the system Magpie [100] which monitors user-space events, kernel events, middleware and system resource usage for each application input by system-level instrumentation. It is a tool chain designed for system workload extraction under real-world conditions in operating systems. Although the system achieves high performance with relatively low overhead, it needs an application-specific schema to parse and correlate those system events, which restricts the system scalability. In [127], authors devised a logging and analysis system targeting intrusion detection, allowing users to specify the operating granularities, however, it requires users to provide the causality definition beforehand. Therefore, although the system offers implementation flexibility, it involves lots of human efforts and limits the system practicability. In RAIN [77], dynamic information flow tracking is applied in a replay-based provenance system to infer fine-grained causality. It leverages syscall events to minimize the

analysis scope and performs refinable attack investigation by selective tainting, and it only incurs 3.22% overhead. Nevertheless, the tainting optimization in replay still requires pre-recorded instruction-level execution logs, and hence it is not applicable in LPROV's scenario.

Information flow and object tainting The inference of information flow between system objects is adopted by lots of existing forensic systems to enhance malware detection and analysis [128]–[130]. Yin et al. proposed the memory-level tracking system Panorama [130] to disclose the information flow between malware and sensitive data. The system can accurately capture malicious data access and processing launched by malware. But due to its high runtime overhead in low-level dependence analysis, it requires special support in hardware-level tracking. In [129], researchers invented a event tracing system VPath that can work either in OS kernel or virtual machine manager. It monitors thread behaviors and network activities to deduce system-wide causalities. Vpath can reveal precise information flow at low overhead, however, it needs carefully pre-defined patterns for system activities. Dynamic taint analysis in system entities is a widely used technique to perform provenance tracking. It can capture sensitive information leakage and system input causality through fine-grained data propagation [91], [96], [128]–[136]. This area has been well studied in multiple perspectives including file, socket and low-level kernel objects on various operating systems [131], [135], [137]. In spite of the high accuracy in provenance analysis, dynamic tainting system has limited usability due to its high overhead caused by heavy instrumentation. Besides, tainting can only tell whether there exists causality between two end objects but cannot offer a clear deriving path from the source to the sink.

Log protection Protecting log integrity is important. Jacobsson et al. proposed a lightweight logging-based malware detection system which operates in a real-time client/server audit mode [138]. The end-host under auditing delivers fresh runtime log data (encrypted and signed) to a remote server for further verification of system infection. However, it relies on the network communication and the connection between client and server becomes the weakest link. In [139], the authors proposed XRec, a primitive system offering integrity of execution trace on instruction level by branch trace messages. It can verify whether a specific code segment has ever been executed. However, it incurs 200%-400% performance overhead as it works in a system debug mode. In [140], researchers devised LPM

(Linux provenance module), the first generic framework to build secure provenance-aware systems. It leverages LSM to create a trusted provenance-aware execution platform, collecting whole-system provenance with low overhead. Sundararaman et al. presented a secure disk system, SVSDS, that performs transparent versioning of data in the disk-level [140]. By enforcing specific data constraints, the system can protect executables and system log files. The latest secure logging systems are implemented on the trusted hardware execution of Intel SGX. In [141], the syslog is ported onto the boundary of trusted enclaves and application logs are generated through enclave *ocalls*. To secure the log storage, all the log data are encrypted on the disk using the SGX secrecy sealing/unsealing functionality. Note that they are complementary to LPROV.

4. PROFACTORY: IMPROVING IOT SECURITY VIA FORMALIZED PROTOCOL CUSTOMIZATION

As IoT applications gain widespread adoption, it becomes important to design and implement IoT protocols with security. Existing research in protocol security reveals that the majority of disclosed protocol vulnerabilities are caused by incorrectly implemented message parsing and network state machines. Instead of testing and fixing those bugs after development, which is extremely expensive, we would like to avert them upfront. For this purpose, we propose ProFactory which formally and unambiguously models a protocol, checks model correctness, and generates secure protocol implementation. Meanwhile, it can also realize protocol diversity through automated randomization. We leverage ProFactory to generate a group of IoT protocols in the Bluetooth and Zigbee families and the evaluation demonstrates that 82 known vulnerabilities are averted. and high implementation entropy is achieved.

4.1 Introduction

As a pillar for smart living, the scale of IoT (Internet of Things) has recently experienced unprecedented growth in both the number of devices and their complexity. According to reports from Statista [142] and Forbes [143], about 26.6 billion IoT devices were installed and connected worldwide in 2019 and they project to exceed 42 billion devices in 2022, lifting the global IoT market to more than 1.2 trillion dollars. Such a growth prospect and tremendous investment are continuously motivating efforts to develop innovative IoT wireless techniques, such as Bluetooth which has effective connectivity, low hardware cost and well-maintained development community [144], [145]. In particular, Bluetooth has enabled bridging interfaces between applications and wireless peripherals including keyboards/mice, headsets/speakers, smart watches, fitness trackers, medical recorders, smart home appliances, and hands-free systems [144], [145]. It was reported that 4.2 billion Bluetooth devices were shipped in 2019, notching a 8.8% compound annual growth rate (CAGR) over 5 years [146]. Zigbee is another

popular IoT protocol which also witnesses a 8.0% CAGR in market growth, and its market value is expected to reach 4.3 billion by 2023 [147].

However, accompanied with the ubiquitous adoption, security has inevitably become a critical issue in IoT protocols. According to the latest Bluetooth market update [146], there were 14 member groups working on 80 active protocol projects during 2019. Actually, most of these protocol projects were built from scratch [146], [148]–[150] and this repeated procedure is considered onerous, tedious, and error-prone [151]–[156], leading to lots of vulnerabilities in protocol implementations. As reported by previous research of protocol security [157], [158], the majority of disclosed protocol vulnerabilities are due to incorrectly-implemented message parsing, where parsing errors can result from the lack of sanity checks (especially for the sizes of message fields) and parsing ambiguities (caused by diverting understanding of specification across developers). Bluetooth implementations are such examples, as reflected by the newly reported Bluetooth-related CVEs (Common Vulnerabilities and Exposures) [148]. In addition, protocol implementation vulnerabilities are also present in state machine components [159]–[161]. Although protocol-specific fuzzing techniques [155], [162], [163] are helpful in exposing those implementation bugs, difficulties in stateful fuzzing, encoding protocol specifications, and achieving complete coverage make exposing all defects infeasible in practice. Therefore, rather than imposing a time-consuming postmortem bug finding procedure in protocol engineering, we propose to address the problem at the very beginning of the development life-cycle. Specifically, protocol developers ought to be relieved from the error-prone low-level implementation efforts. Instead, they should focus on the design of essential pieces of protocol specifications (e.g., message format and finite state machine) and the corresponding protocol implementations will be automatically generated in a manner that ensures security.

To achieve this goal, protocol rewriting is a plausible option, where protocol specifications are extracted from existing implementations, customized, and re-implemented/reinforced by developers to produce new (hardened) protocols and/or implementations. However, previous protocol reverse-engineering work observes that due to the flexibility of protocol design, recovering comprehensive and accurate specifications from protocol implementations is extremely challenging, especially for state machines [154]. Even if design details were known a

priori, there are no reliable existing methods to faithfully project specifications to low-level implementations, as in complex protocols, various components are tightly entangled and interwoven. Hence, without well-documented, well-structured, and well-annotated source code, rewriting-based protocol customization is error-prone. For example, recent research in TLS (Transport Layer Security) discloses that protocol customization on existing implementations can produce a composite state machine which enables communication peers to accept unexpected messages [164].

The recent research shows that formalizing protocol specifications is a promising approach to addressing this issue [157], [165]–[169]. In particular, message formats are customized in a DSL (Domain Specific Language) and secure protocol implementations are emitted accordingly. For example, EverParse [165] devises a DSL describing tag-length-value message formats to produce zero-copy parsers and corresponding serializers. In contrast with existing data serialization/deserialization tools (e.g., Protobuf [170] and Thrift [171]), parsers generated by EverParse are formally verified and their security is guaranteed. In [157], a similar DSL is proposed for generating additional sanity checks to harden USB (Universal Serial Bus) message parsers. Nevertheless, we observe that existing protocol formalization efforts mostly focus on message formats but many fall short in modeling some dynamic functionalities such as multiplexing and state transitions. Consequently, these DSLs require substantial extension to fit low-level and kernel-oriented IoT protocols.

Our Solution To this end, we select to develop a new unified DSL whose syntax can specify both protocol message formats and dynamic behaviors. We propose ProFactory to automatically generate secure low-level protocol implementations for Linux kernels from protocol specifications written in the DSL. It aims at facilitating protocol development, and eliminating message-parsing vulnerabilities and fundamental state-transition errors in protocol implementations. Currently, ProFactory targets code generation for IoT protocols in the Linux kernel including those in the Bluetooth and Zigbee families. It can be easily adapted for other protocols or production kernels as long as the platform-dependent interfaces and settings are available.

Specifically, ProFactory works as follows. First, a protocol is modeled/customized in our DSL. Then, symbolic model checking is performed to verify the model correctness

(e.g., network state transitions are not vulnerable). After passing the model checking, the customized protocol model is fed to the code generation engine to produce kernel-oriented protocol implementation which provides guarantees of being free from memory safety vulnerabilities (i.e., buffer overflow, invalid pointer dereference, memory leakage, use after free and double free) in message parsing and from concurrency control vulnerabilities (i.e., race and deadlock) in message multiplexing. Such guarantees are provided by the automatically generated sanity checking code, such as bound checks and input validation checks, and by applying automated verification tools to the generated code. Only if the protocol passes the model and implementation verification, should the implementation be integrated into the production kernel on both of the communication peers. Finally, the peers communicate through the customized protocol. We highlight our contributions in the following.

• We propose ProFactory, a novel system that realizes efficient and secure protocol customization. In ProFactory, developers formally and unambiguously model protocols in a DSL instead of natural languages and the models lead to the production of vulnerability-free implementations.

• We develop a type-based DSL that is closely coupled with protocol semantics. In our DSL, various protocol specifications such as message format, finite state machine and connection multiplexing can be well expressed by a number of abstract types.

• We develop a code generation engine, emitting kernel code without message-parsing errors according to the DSL-defined protocol model, where concurrency correctness and memory access safety of generated C codes are formally verified using VCC [172], [173] and Frama-C [174]. Also, the engine allows the randomization of message formats to offer convenient connectivity isolation and prevent attack propagation.

• We develop a symbolic model checker to capture potential bugs residing in protocol state transitions. Those bugs are abstracted as protocol property violations.

• We build a prototype of ProFactory and generate 8 protocols in Bluetooth and Zigbee. The evaluation demonstrates that ProFactory can offer high implementation diversity and help to avert 82 known vulnerabilities with low overhead in generated implementations. Also, the generated implementations have low overhead.

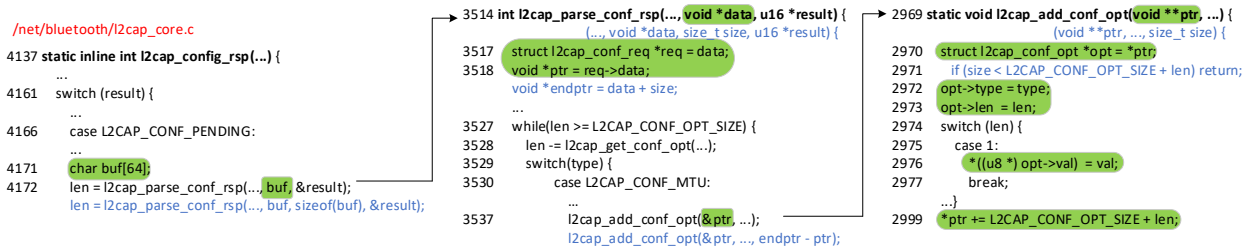

Figure 4.1. L2CAP configuration buffer overflow in BlueZ implementation

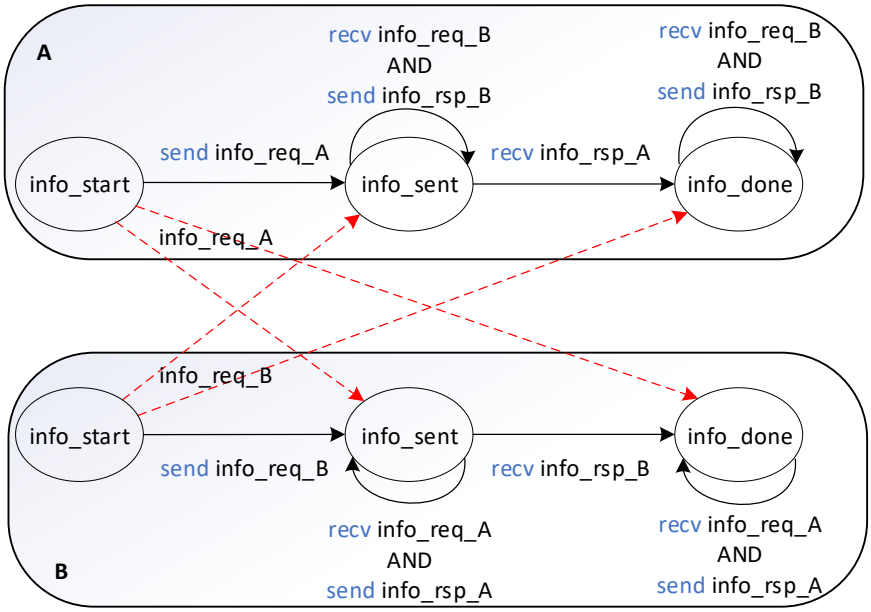

Figure 4.2. Message loss in L2CAP information exchange

4.2 Motivation

The increasing number of security issues in IoT protocol implementation motivates secure protocol customization. Generally, those issues can be divided into two categories, message-parsing vulnerabilities and state-transition errors. Next, we use two examples (one for each category) to illustrate how ProFactory can help avert them.

Motivating Example One. This vulnerability (CVE-2017-1000251) [175] resides in the L2CAP implementation of Linux BlueZ (kernel 4.13.1 and older) and it was disclosed in the Blueborne report [148], allowing a malicious Bluetooth user to launch a denial-of-service or remote-execution attack. In L2CAP, before data transmission, the two peers are required to

negotiate a group of connection options (or parameters) and the negotiation is accomplished by exchanging two kinds of messages, *configuration request* and *configuration response*. If a configuration request from a peer cannot be accepted, the peer has to send a second request based on the response contents (e.g., copying the configurations indicated in the response to the second request).

Figure 4.1 elaborates the buggy code and its official patch (in blue) [176], where response events and response parsing are handled by `l2cap_config_rsp` and `l2cap_parse_conf_rsp` At line 4171, a local 64-byte buffer `buf` is allocated to hold the second request (as the previous request was pending, i.e., not accepted) and a loop (lines 3527-3537) is used to extract information from the response and emit the request body, where the parser invokes `l2cap_add_conf_opt` to append a request option to `buf`. Before patching, since neither the response size nor the buffer boundary is checked, an attacker can craft a long response such that the buffer is overflowed when the large number of configuration options in the response are copied. From the perspective of protocol specification, the reason for such a buffer-overflow vulnerability is that the protocol is actually under-specified. In particular, the developer allocates 64 bytes (a magic number of bytes) for the request, indicating that there must exist an upper bound for the response size which is not explicitly respected by the emission loop, causing the vulnerability. Fortunately, such specification confusions can be eliminated by ProFactory in the protocol modeling stage.

In our DSL, the option group is modeled as a parameter list (see modeling details in Section 4.4), for which the maximum size must be specified. Hence, we have the upper-bound check for the option group size. In addition, instead of manipulating bare buffers, ProFactory performs all the message-related operations on the well protected socket buffer data structure `sk_buff`. This data structure allows convenient field appending or truncating through `skb_pull` and `skb_put`/`skb_push`, and the kernel intrinsically performs all the needed boundary checks. Furthermore, the structure always maintains the current data length and hence the size of each message segment (e.g., the option group) can be strictly validated when unpacked. Overall, ProFactory rejects any unspecified/invalid messages and offers secure message parsing/construction.

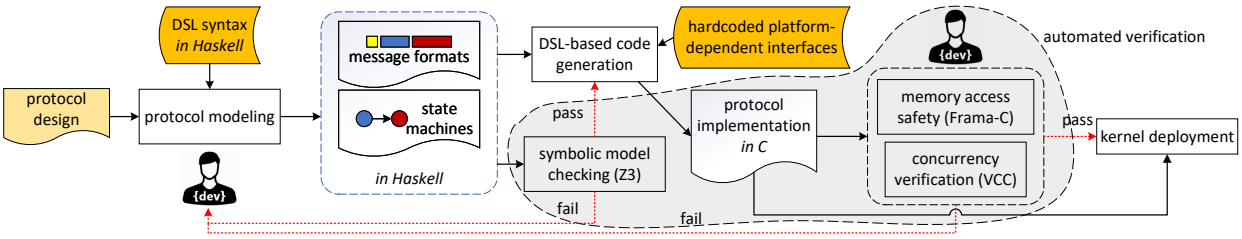

Figure 4.3. The overall workflow of ProFactory (black arrow denotes data flow and red arrow denotes control flow)

Motivating Example Two. This example is to demonstrate a typical transition error in an asynchronous state machine. In L2CAP, at the beginning of a connection, each peer can request information from the remote side in order to set up a connection with the supported functionalities. The textual specification of L2CAP simply mentions "*L2CAP implementations shall respond to a valid information request with an information response*" [177] without much detail. Figure 4.2 presents two buggy asynchronous state machines for two communicating devices A and B, respectively. In the state machines, the solid edges denote state transitions and the dashed edges denote messages (to the other party). The problem lies in that a device may undesirably drop requests when it is at the `info_start` state. The buggy state machine requires the device to first send a request before it can properly receive requests from the other party. Initially, assume both devices are in the `info_start` state and they both send out a request in the same time. However, the two sends form a race. It is hence possible that when the request from a remote peer arrives, the local device is not in the next state that can receive the message. This causes undesirable loss of messages. The user may encounter difficulties in establishing a Bluetooth connection due to the bug. This error is averted by performing the race-free property checking in ProFactory (Section 4.6). The fix is to allow devices to receive messages in `info_start`.

4.3 Approach Overview

Figure 4.3 presents the overall workflow of ProFactory. ProFactory executes in three main phases: *Protocol Modeling*, *Code Generation* and *Automated Verification*.

Protocol Modeling Phase. When creating a customized protocol, developers first need to model the protocol in ProFactory's DSL as the system input. The DSL, in a sense, is de-facto a translator of protocol specifications. In this DSL, all the protocol elements are abstracted as hierarchical data types and each of those types would instruct ProFactory to emit a unique set of concrete data structures and code blocks in the following code generation phase. Therefore, modeling a protocol in ProFactory corresponds to assembling definition instances of those types. To enrich the language to support various protocols, the design of the DSL syntax is closely coupled with protocol semantics including message format, FSM (finite state machine) and other protocol-specific features. The DSL is embedded in Haskell, which offers an easy-to-use development environment that allows manipulating its own syntax and facilitates implementing a DSL. We will discuss the complete DSL design in Section 4.4.

Code Generation Phase. In this phase, ProFactory automatically produces C code for a production kernel, according to the input protocol model. The generated code consists of type-based code blocks (i.e., protocol message data structure definitions and message parsing/construction procedures) and kernel shims which can assist seamless code insertion or replacement in a production kernel. In particular, the type-based code blocks perform both message recognition and state transition, while kernel shims prepare standard socket interfaces and the accesses to the underlying platform or lower-layer protocols, requiring the hardcoded platform-dependent interfaces. As demonstrated in the first motivating example, the lack of sanity checks is an important contributor to protocol vulnerabilities, ProFactory hence enforces validity verification on corresponding message fields when emitting code for message parsing/building. Besides, ProFactory also offers an option for developers to randomize message formats in the generated implementation. As emphasized in [149], [178], universal connectivity of IoT devices has raised severe privacy issues. Hence, the randomization can assist generating protocol dialects which can offer connectivity isolation and prevent arbitrary attack propagation. The details of code generation are elaborated in Section 4.5.

Automated Verification Phase. Protocol security issues include not only vulnerabilities incurred by field mishandling in message parsing and problematic concurrent accesses to shared information, but also correctness problems in the underlying FSM. Therefore, Pro-

FACTORY verifies the generated implementation through VCC [172], [173] (free from race and deadlock) and Frama-C [174] (free from buffer overflow, invalid pointer dereference, memory leakage, use after free and double free). In Frama-C verification, limited manual intervention may be required to assist constructing proofs. In addition, PROFACTORY also performs model checking for the protocol state machine. We encode the protocol model and perform property checking using the Z3 SMT solver [179]. We devise a set of general properties (i.e., transition, security and customization properties) that should be respected by a protocol state machine and validate them against protocols written in our DSL. Protocol models failing to pass any of the verifications should be remodeled by developers. There have been research efforts in protocol model checking [160], [180]–[184], but the majority of them target proving cryptographical correctness for AKA (authentication and key agreement) protocols or components of specific protocols (e.g., TCP/IP), while the model checking in PROFACTORY focuses on generic functional correctness (e.g., correct state transitions). Our work is therefore orthogonal and complements existing work. This phase will be explained in Section 4.6.

4.4 Protocol Modeling

In protocol specifications, message format and FSM are the two building blocks. Hence, our DSL focuses on describing these two perspectives. To facilitate precise discussion, we simplify and summarize the synatx of DSL in Figure 4.4 and semantics in Figure 4.5. As illustrated in Figure 4.4, a protocol can be represented by a set of abstract data types related to message format, socket, and state machine. These abstract types are further instantiated to concrete types when describing individual protocols. Low level implementation is automatically generated from the DSL specification. In the following, we first explain the syntax and then the semantics of the DSL, which are followed by an example.

4.4.1 DSL Syntax

Message Format. Most network messages have hierarchical structure, meaning that a network message m is often the raw data field of a message at a lower layer. It often has

MESSAGE FORMAT RELATED ABSTRACT TYPES

Fix f	::=	$(n_{size},\ n_{low},\ n_{high})$
Var v	::=	$(n_{low},\ n_{high})$
Hdr h	::=	$f^{+} \cdot f_{len}$
Para p	::=	$(n_{key},\ f_{key} \cdot f_{val})$
Plist ℓ	::=	$(n_{size},\ \mathcal{P}(p))$
Msg m	::=	$v \mid p \mid f^{+} \mid \ell \mid h \cdot m_{sub} \mid h \cdot (f_{type} = n_{type}\ ?:\ m_{sub})^{+}$

SOCKET RELATED ABSTRACT TYPES

Chan ch	::=	(n^{*}_{key})
Conn cn	::=	(n^{*}_{key})

STATE MACHINE RELATED ABSTRACT TYPES

State s	::=	$n_{timeout}$
Recv r	::=	$(m_{in},\ (e, s_{from},\ s_{to},\ (m_{out} \mid \varepsilon),\ \{S\})^{+})$
Send d	::=	$n_{act} \cdot (m_{out}, (e,\ s_{from},\ s_{to},\ \{S\})^{+})$

STATEMENT

S	::=	$S_1; S_2 \quad \mid \quad x := e$
	$\mid$	**if** (e) $\{S_t\}$ **else** $\{S_f\}$
	$\mid$	**for** $(x$ **from** e **to** $e)$ $\{S\} \quad \mid$ **iter** $(\ell, \{S\}) \mid \ldots$

EXPRESSION

e	::=	$n \mid string \mid bool \mid x \mid \ominus e \mid e \oplus e \mid \ldots$

Figure 4.4. PROFACTORY DSL Syntax (n denotes an unsigned integer constant and operator $\cdot$ denotes field concatenation)

its own structure too, encapsulating some message(s) at a higher layer. Note that even the messages of a same protocol are organized in multiple layers. For example, L2CAP command messages are encapsulated in generic L2CAP messages. With such layered structure, the corresponding network message parsing code largely follows a fixed pattern, namely, the parser for a message m of a particular layer unfolds the structure at that level, checks some field that determines the (inner) message type of the raw data field, and further invokes the corresponding parser(s) of the inner message(s). This regularity allows us to abstract network messages and message parsing to a set of general abstract types that have hierarchical relations. An example of abstracting L2CAP messages using our DSL can be found later in the section.

We introduce 6 abstract types to describe (hierarchical) message formats, two basic field types Fix and Var denoting a fixed-sized field and a variable-sized field, respectively, and four other types: Header Hdr, Parameter $Para$, Parameter List $Plist$, and Message Format

Msg that are built on the two basic types. Besides, we provide two socket related abstract types $Conn$ and $Chan$ to model network connections. We explicitly model connection types to allow easy code generation for the interface with the kernel.

Fixed-sized Field (Fix) A fixed-sized field consists of three integer attributes, describing its size (n_{size}) and range (n_{low} and n_{high}). The size is measured in bytes. The range attributes (can be *nil*) specify the lower and upper bounds of the field. This could be further extended to support other value constraints. The attributes allow safe code generation (with bound checks and validity checks).

Variable-sized Field (Var) A variable-sized field represents a byte sequence, with its size range specified by n_{low} and n_{high}, which enable mandatory bound checks in code generation.

Header (Hdr) A message header is a sequence of fields followed by a length field, which is a dedicated fixed-sized field describing the length of the following message content. Note that we are defining an abstract type, which is further instantiated to concrete types for a specific protocol. In general, the fields in a header describe the meta information of a message and instruct the parser to correctly extract and process sub-messages. For example, a message header $f_{type} \cdot f_{id} \cdot f_{len}$ consists of three fields describing the type of the message (that determines how the message body is interpreted), the connection ID and the length of the body. Note that the operator $\cdot$ denotes field concatenation. The length field is for automatic generation of validity check.

Parameter (Para) A parameter (in message) consists of two fields, a key field f_{key} and a value field f_{val}. A parameter abstract type consists of a constant n_{key} that uniquely identifies the parameter kind and the specifications of the two fields. Parameters denote configurations that can be negotiated across the peers of a connection. Intuitively, if the value of the key field (at runtime) matches the constant n_{key}, the parameter is of the corresponding type. For instance, $(2,\ (4,2,2) \cdot (4,0,1024))$ is an MTU (Maximum Transmission Unit) parameter type, with a static value $n_{key} = 2$ denoting the type, the first key field 4 bytes long with a fixed value of 2, and the second value field 4 bytes long with a value in [0,1024]. At runtime, a concatenation of two fields (in a message) is considered an MTU parameter when the first field has the value of 2.

Parameter List (Plist) A parameter list denotes a variable set of parameters (whose f_{key} sizes must be the same) with n_{size} specifying the maximum number of parameters in it.

Message (Msg) A message could be a variable-sized field, parameter, sequence of fixed-sized fields, parameter list, header followed by a sub-message, or header followed by multiple possible sub-messages. The last two alternatives describe the hierarchical structure of network messages. Specifically, in the last alternative, field f_{type} is a field in the header h. When it has the value of n_{type}, the sub-message is of the m_{sub} type. Note that a concrete message type may have multiple sub-message branches. ProFactory automatically generates parsing code based on the specified hierarchy. An example can be found later in the section.

Socket Related Types. In a production kernel, a protocol has a socket-like interface that serves the applications. The interface includes a number of socket peripheral data types such as protocol connection *Conn* and protocol channel *Chan*. A protocol may have multiple channels sharing an end-to-end connection (for multiplexing). The abstract type of a connection/channel consists of a list of static values n_{key} that uniquely identify the parameters for the connection/channel. The key values are a subset of the key values in parameter type definitions. Sample parameters include device type for a connection and MTU for a channel in L2CAP. Different from other abstract types, ProFactory only allows one instantiation for *Conn* and *Chan*, meaning that all the connections and channels in a protocol have to be homogeneous.

State Machine. Protocol execution is largely driven by state machines. In particular, besides message parsing, the other focus of protocol implementation is to properly update state machines. Upon receiving a message, protocol implementation parses the message, updates some state variable(s), composes and sends a response message if needed. In some cases, it performs side-effect operations such as logging critical events and collecting statistics. Although most protocols follow the same execution model, their low level implementations have substantial diversity. For example, they may or may not have explicit state variable(s); some protocols update state variables before sending response whereas some others the opposite. Our DSL leverages the inherent regularity of the execution model to produce uniform implementation, enabling security and easy verification. Our DSL allows developers to specify protocol state machines, through three abstract types *State*, *Recv* and *Send*.

Protocol State (State) A state type is defined by a constant denoting the timeout of the state, which specifies the maximum time a state must be retained before a connection is terminated if no legitimate connection activity is observed, in case of idle/crashed applications and lost connections. To specify a concrete protocol, the developer often needs to define multiple states. ProFactory pre-defines a number of them including the start state s_{init}, and the end state s_t.

State Transition (Recv, Send) *Recv* specifies the state transition and the associated operations that are triggered by a received message and *Send* specifies transitions and operations triggered by a message-sending request. In *Recv*, m_{in} is the received message, followed by a group of transition options. Each option is guarded by an expression e. If the expression is evaluated to true, a (compound) statement S is executed, the state is updated from s_{from} to s_{to}. Meanwhile, a response message m_{out} may be sent. S typically includes invocation of the receive function of the sub-message of m_{in}, creating a channel/connection, setting/getting a parameter value, constructing m_{out}, and collecting statistics. The syntax of *Send* is similar with m_{out} the message to send, but it specifies an action type n_{act} to express one of the only three socket operations, i.e., connection establishment, message delivery and connection shutdown, that perform active message sending. Note that socket errors (e.g., a message was not successfully received/sent) get automatically handled by the state timeout (see "Timer and Counter" in Section 4.5).

Statement and Expression. The syntax of statements and expressions in our DSL largely follow the C language. Statements and expressions are mostly used in type definitions. Different from a program in mainstream programming languages that often has a *main()* function that specifies the main logic of a program. Protocol code is event-driven, for instance, by connection, send, and receive events. As such, in ProFactory the main logic of a protocol is directly derived from the type definitions, in the form of a list of event handlers and functions called by these handlers. Statements and expressions are merely part of the type definitions such as e and S in *Recv* and *Send*. ProFactory supports assignment, `if` − `else`, `for` loop, constants, variables, common unary or binary operations. In addition, it provides a number of statements convenient for network protocols. For instance, i*ter* executes statement S on each element in a parameter list ℓ.

Data Structures and Auxiliary Functions

σ : store that maps key(s) to a value; $\mathbb{F}$: functions defined; $\mathbb{P}$: symbolic constraints; $\mathbf{Ch_{cur}}$: the current channel; $\mathbf{s_{init}}$: the initial connection state

newId(): acquire a new connection or channel id; **newMsgHandler**(): acquire a message handler (like a file handler)

Semantic Rules

1. f:=**Fix** (n_s, n_l, n_h)	$\mathbb{F}+=\left\{\begin{array}{l}\textbf{def } f.parse(M,V)\ \{\sigma[M][f]=V[0,n_s];\},\\ \textbf{def } f.compose(V,x)\ \{V[0,f.len-1]=\sigma[x];\}\end{array}\right\}$ $\mathbb{P}+=\left\{\begin{array}{l}f.len=n_s, f.val\le n_h,\\ f.val\ge n_l\end{array}\right\}$
2. v:=**Var** (n_l, n_h)	$\mathbb{F}+=\left\{\begin{array}{l}\textbf{def } v.parse(M,V,x_l)\ \{\sigma[M][v]=V[0,\sigma[x_l]-1];\},\\ \textbf{def } v.compose(V,x_l,x_v)\ \{V[0,\sigma[x_l]-1]=\sigma[x_v];\}\end{array}\right\}$ $\mathbb{P}+=\{v.len\le n_u,\ v.len\ge n_l\}$
3. h:=**Hdr** $f_1\cdot f_{len}$	$\mathbb{F}+=\left\{\begin{array}{ll}\textbf{def } h.parse(M,V)\ \{ & \textbf{def } h.compose(V,x_1,x_{len})\ \{\\ \quad f_1.parse(M,V); & \quad V[0,f_1.len-1]=\sigma[x_1];\\ \quad f_{len}.parse(M,V[f_1.len,\ ]);\} & \quad V[f_1.len,f_1.len+f_{len}.len-1]=\sigma[x_{len}];\}\end{array}\right\}$ $\mathbb{P}+=\{h.len=f_1.len+f_{len}.len\}$
4. p:=**Para** $(n_{key}, f_{key}\cdot f_{val})$	$\mathbb{F}+=\left\{\begin{array}{ll}\textbf{def } p.parse(M,V)\ \{ & \textbf{def } setPara(x_c,x_k,x_v)\ \{\\ \quad f_{key}.parse(M,V); & \quad \textbf{switch}(x_k)\ \{\\ \quad \textbf{assert}(\sigma[M][f_{key}]\equiv n_{key}); & \quad\quad \textbf{case } n_{key}:\ \sigma[\sigma[x_c]][n_{key}]=\sigma[x_v];\\ \quad f_{val}.parse(M,V[f_{key}.len,\ ]);\}, & \quad\quad ...\}\\ & \quad\}\end{array}\right\}$ $\mathbb{P}+=\{n_{key}=f_{key}.val,\ p.len=f_{key}.len+f_{val}.len\}$
5. l := **Plist**$(n_s, \mathcal{P}(\{$ $p_1:=(n_{k1}, f_{k1}\cdot f_{v1}),$ $p_2:=(n_{k2}, f_{k2}\cdot f_{v2})\})$	$\mathbb{F}+=\left\{\begin{array}{l}\textbf{def } l.parse(M,V)\ \{\\ \quad \textbf{while}(i<n_s\ \&\&\ j<strlen(V))\ \{\\ \quad\quad \textbf{switch}(V[j,\ j+f_{k1}.len-1])\ \{\\ \quad\quad\quad \textbf{case } n_{k1}:\ p_1.parse(M,V[j,]);\ j+=p_1.len\\ \quad\quad\quad ...\\ \quad\quad\},\end{array}\right\}$ $\mathbb{P}+=\{l.len=p_1.len+p_2.len,\ f_{k1}.len=f_{k2}.len\}$
6. m := **Msg**$(h:=f_t\cdot f_{len})\cdot$ $(f_t=n_{t1}\ ?:\ m_{t1}\ \mid$ $f_t=n_{t2}\ ?:\ m_{t2})$	$\mathbb{F}+=\left\{\begin{array}{ll}\textbf{def } m.parse(M,V)\ \{ & \textbf{def } m.compose(V,x_t,V_{sub},x_l)\ \{\\ \quad h.parse(M,V); & \quad \textbf{if}(\sigma[x_t]\equiv n_{t1})\ \{\\ \quad \textbf{if}(\sigma[M][f_t]\equiv n_{t1}) & \quad\quad h.compose(V,n_{t1},x_l);\\ \quad\quad m_{t1}.parse(M,V[h.len,\ ]); & \quad\quad V[h.len,h.len+x_l-1]=V_{sub};\\ \quad \textbf{if}(\sigma[M][f_t]\equiv n_{t2}); & \quad\}\\ \quad\quad m_{t2}.parse(M,V[h.len,\ ]);\}, & \quad ...\end{array}\right\}$ $\mathbb{P}+=\{n_{t1}=f_t.val\vee n_{t2}=f_t.val,\ f_t.val=n_{t1}\rightarrow(m.len=h.len+m_{t1}.len\wedge f_{len}.val=m_{t1}.len),$ $f_t.val=n_{t2}\rightarrow(m.len=h.len+m_{t2}.len\wedge f_{len}.val=m_{t2}.len)\}$
7. ch:=**Chan** (n_1, n_2)	$\mathbb{F}+=\left\{\begin{array}{l}\textbf{def } createChan(x_c,x_{addr},x_1,x_2)\ \{\\ \quad id=\textbf{newId}();\ \sigma[id][addr]=\sigma[x_{addr}];\ \sigma[id][n_1]=\sigma[x_1];\ \sigma[id][n_2]=\sigma[x_2];\\ \quad \sigma[id][state]=\mathbf{s_{init}};\ \sigma[\sigma[x_c]][channels]+=\{id\};\ \textbf{ret } id;\}\\ \textbf{def } findChanByPara(x_c,x_k,x_v)\ \{\\ \quad \textbf{foreach}\ (id\in\sigma[\sigma[x_c]][channels])\\ \quad\quad \textbf{if}\ (\sigma[id][\sigma[x_k]]\equiv\sigma[x_v])\ \ \textbf{ret } id;\}\end{array}\right\}$
8. cn:=**Conn** (n_1, n_2)	$\mathbb{F}+=\left\{\begin{array}{l}\textbf{def } createConn(x_1,x_2)\ \{\\ \quad id=\textbf{newId}();\ \sigma[id][channels]=\{\};\ \sigma[id][n_1]=\sigma[x_1];\ \sigma[id][n_2]=\sigma[x_2];\\ \quad \textbf{ret } id;\}\end{array}\right\}$
9. r := **Recv** $(m_{in},\ (e_1, s_{from},$ $s_{to},\ m_{out},\ \{S\}), (e_2, ...))$	$\mathbb{F}+=\left\{\begin{array}{l}\textbf{def } r.receive(V)\ \{\\ \quad M=\textbf{newMsgHandler}();\\ \quad m_{in}.parse(M,V);\\ \quad \textbf{if}\ (e_1)\ \{\ S;\ \textbf{assert}(\sigma[\mathbf{Ch_{cur}}][state]\equiv s_{from});\ \sigma[\mathbf{Ch_{cur}}][state]=s_{to};\\ \quad\quad\quad m_{out}.compose(V,...);\ \textbf{send}(V);\}\\ \quad \textbf{if}\ (e_2)\ ...\ \}\end{array}\right\}$ $\mathbb{P}+=\{(e_1\wedge state=s_{from})\rightarrow state=s_{to},\ e_2...\}$

Figure 4.5. ProFactory DSL Semantics

4.4.2 DSL Semantics

Figure 4.5 presents the semantics of a subset of DSL specifications, with the data structures and auxiliary functions used, followed by the rules. Different from a regular programming language, in which each statement has concrete semantics, our DSL is mainly for type definitions. As such, we define semantic rules for *concrete* type definitions. In the semantic rules section of Figure 4.5, the first column shows concrete type definitions and the corresponding semantic rules in the second column show a list of functions and symbolic constraints derived from the definition. The functions define a list of operations for a concrete type. Some of the functions are event handlers that constitute the interface of protocol. The symbolic constraints are used for symbolic modeling checking that validates the correctness of protocol specifications.

Specifically, for a type definition, ProFactory updates $\mathbb{F}$, which denotes the list of functions defined, and $\mathbb{P}$, which denotes the list of symbolic constraints derived. Inside a function, the semantics is described using C-like statements, many of which update a store σ that is similar to a store in classic programming language semantics. Intuitively, one can consider it as a hash-map that projects a key or a number of keys to some value. Here, key can be a name, a value, or a variable.

Rule 1 specifies that for a definition of fixed-sized field, two functions are introduced, with $f.parse(M, V)$ parses the field and $f.compose(V, x)$ composes the field (as part of whole message composition). In the parser function, parameter M is a handler for the message (kind of id for the message). One can intuitively consider each incoming message (to parse) has its unique handler; V is a buffer passed in from the kernel, containing the message (or part of a message). The function copies n_s (i.e., the size of the field) bytes from V to the store. Note that the concrete field value is indexed by the handler M and the symbolic field name f. This is because f is a field *type* instead of a concrete field. In the composition function, variable x denotes the value used to compose the field and V is the buffer storing the message to compose. V will be passed on to the kernel to send a message after composing the whole message. In addition to the functions, three symbolic constraints are added to $\mathbb{P}$ dictating the (symbolic) length of the field equals to n_s, and the (symbolic) value of the

field must be in between n_l and n_u. These constraints will be used for model checking. The semantics for a variable-sized field definition is similar.

In Rule 3 for a header consisting of two fields f_1 and f_{len} (for the length of the message body), the parser function parses the two fields in order by invoking their parser functions. This implies that the two field types need to be defined. The expression $V[f_1.len,]$ means that a sub-buffer starting at offset $f_1.len$ of V. The composition functions copies the two variables x_1 and x_{len} to the result buffer. The symbolic constraint dictates that the length of header be the sum of the lengths of the two fields.

In Rule 4 for a parameter, the parser function parses the two fields and asserts that the value of the key field must equal to the specified key value n_{key}. A global function, i.e., a function not specific to a definition, $setPara(x_c, x_k, x_v)$ is also introduced to set a parameter, with x_c denoting the connection/channel, x_k the key, and x_v the value. It sets the parameter denoted by the value of x_k to the value of x_v. It will be invoked when connections/channels are created/configured.

In Rule 5 for a parameter list, the parser function traverses through the buffer V and parses individual parameters until it reaches the end of V or the number of parameters reaches the upper bound n_s. The last symbolic constraint requires the parameters (in the list) have the same key field size. Note that $strlen(V)$ means the dynamic length of buffer V which is not null-terminated.

In Rule 6 for a message with two possible sub-message formats, the parser function first parses the header. It then checks the value of the type field f_t in the header and invokes the parser of the corresponding sub-message ("?:" is similar to "switch-case"). The composition function is symmetrically defined, with x_t the type of the sub-message, V_{sub} a buffer containing the sub-message composed before-hand, and x_l the length of the sub-message. The symbolic constraints ensure that (1) the value of the type field must be n_{t1} or n_{t2}; (2) if it is n_{t1}/n_{t2}, the message length is the sum of the header and the sub-message m_{t1}/m_{t2} and the value of the length field f_{len} in the header must match the length of m_{t1}/m_{t2}.

In Rule 7 for a channel definition, two global functions are introduced. The first one is to create a new channel, with x_c denoting the connection to which the channel belongs, x_{addr} the address of the channel, and x_1, x_2 the values for the channel parameters n_1 and

n_2. Inside the function, a new local channel id is created to uniquely represent the new channel. The state of the channel is set to s_{init} and the list of channels for the connection is updated. Some protocols explicitly specify channel id in their messages so that they can be properly attributed. However, there are protocols that implicitly encode channel id in some parameter(s). For example, L2CAP may encode channel id in PSM (Protocol Service Multiplexer). As such, we provide a *findChanByPara()* function to help look up a channel in connection x_c, using the parameter key x_k and value x_v. It returns the reference to the found channel or `NULL`. In Rule 8, the creation function of connection is similarly defined. The list of channels is initialized to empty.

Rule 9 specifies the semantics for the definition of a *Recv* state transition, which leads to the definition of a *receive*() function. If the message m_{in} is a top level message, the function is invoked by another protocol at the lower layer. Otherwise, it is invoked by the receive functions of higher level messages. Inside the function, a handler is first allocated to denote the message. One can intuitively consider it as an id. Message m_{in} is then parsed. If the expression e_1 is satisfied, statement S is executed; the state is updated from s_{from} to s_{to}; a response message is composed and sent. Similarly, if the expression e_2 is satisfied, a different transition is performed. The symbolic constraint specifies the possible state transitions. The semantics for *Send* is similarly defined and elided.

4.4.3 A Real-world Example

In the Bluetooth protocol stack, L2CAP is one of the most critical protocols, responsible for protocol multiplexing and data delivery between applications and the protocol stack. It sits on top of the HCI (Host Controller Interface) layer (i.e., a link layer) and serves a large number of upper layers such as RFCOMM (Radio Frequency Communication), HIDP (Human Interface Device Profile), and BNEP (Bluetooth Network Encapsulation Protocol). Figure 4.6 (a) shows a few simplified code snippets from a Linux Bluetooth 5.0 implementation. They are to handle L2CAP command messages.

Function `l2cap_sig_channel` is invoked by a callback from the lower HCI layer to process a L2CAP command. Depending on the command type (line 5335) in the command

Figure 4.6. A running example of modeling a subset of Bluetooth L2CAP specifications

header (line 5702), it invokes function `l2cap_connect_req` to process a connection request or `l2cap_config_req` to process a configuration request. Inside `l2cap_connect`, lines 3777-3820 leverage the PSM (protocol service multiplexer, like port for TCP) field to look up a parent channel listening to this kind of service request. If there is such a channel and no existing channel is using the requested source channel ID (lines 3771 and 3798), it spawns and initializes a new L2CAP channel (the channel is initialized to the initial `BT_START` state, which is not explicitly shown in the snippets). It then sets the channel to `BT_CONFIG` state, and uses `l2cap_send_cmd` which is a wrapper API of the lower HCI layer to send a response message (lines 3841-3870). Inside `l2cap_parse_conf_req`, the loop in lines 3316-3325 traverses a list of configuration options. For example, it sets the MTU of the channel if an mtu option is included (line 3323). Inside `l2cap_config_req`, a configuration request cannot be accepted/parsed unless the channel is at `BT_CONFIG` state (line 4052). After parsing, it sends a response message and sets the channel to the ready state `BT_CONNECTED` (if the configuration is successful, lines 1251-1261).

Using our DSL, we can rewrite the complex implementation (hundreds of LOC) to 31 LOC in DSL as shown in Figure 4.6 (b). We use yellow, blue and green in (a) and (b) to mark artifacts related to L2CAP commands, connection requests, and configuration requests,

respectively. We also use circled annotations in (a) to associate concrete variables and operations to abstract types in (b). For example, `u8` $*$ `data` at line 5700 is abstracted to line T21, meaning that it is a L2CAP command message whose header directs the parsing to different commands. The corresponding message formats and state machine are shown in Figure 4.6 (b.i, b.ii) for easy understanding.

4.5 Code Generation

ProFactory automatically generates C code from the DSL specification. The semantic rules in Figure 4.5 specify the functions we need to generate and the semantics of these functions. However, those functions are still abstract. In this subsection, we discuss how the concrete C code is generated. Specifically, each type definition in a protocol specification leads to a C data structure. For example, a header type definition $h :=$ **Hdr** $f_{type} \cdot f_{len}$ leads to a C data structure definition "*typedef struct* {... f_{type} ...; f_{len} ...;} h;". Functions like those described in the semantic rule of the type definition are generated. For the above header example, the two functions in Rule 3 in Figure 4.5 are generated. Hash-map operations through the store, e.g., $\sigma[M][f_{type}]$, are compiled to the corresponding data structure field accesses.

Sanity Checks An important advantage of ProFactory is that it ensures parser security by inserting bound checks and input validity checks. Note that the length of each field, header, message is clearly specified in our DSL. If necessary, value ranges are also specified. Runtime checks like those in the symbolic constraints set $\mathbb{P}$ (derived while compiling the protocol specification) are automatically inserted.

Multiplexing ProFactory supports multiplexing which entails concurrent connections. To avoid races, ProFactory automatically adds mutexes to guard accesses to data structures involved in multiplexing, such as the channel list field in a connection data structure, the current connection/channel variable, and connection/channel parameter data structures.

Packet Fragmentation Developers are oblivious to the implementation details of packet fragmentation. They only need to specify how the MTU value is negotiated as part of the protocol specification. To support fragmentation, ProFactory automatically inserts an additional

fragmentation header (including fragment ID, offset and continuation flag) into the header of each message and the fragmentation logic is injected in message send/receive functions.

Timer and Counter ProFactory does not customize timer or counter constructs. Instead, it leverages the generic kernel socket timer *sk_sndtimeo* (40s is a reasonable timeout value) for message sending, and the delayed callback registration *schedule_delayed_work*() (value is set by the timeout attribute of *State*) for message receiving, where the timeouts trigger the close of sockets. They are transparent to developers, and they are automatically generated for *Recv* and *Send*.

Cryptographic Operations Modeling cipher suites is out of the scope of our DSL, while cryptographic constructs are packed into prepared interfaces. Specifically, ProFactory offers two expressions *setSec*(*int*) and *getSec*() (omitted in Figure 4.4 for brevity) for security level setting and fetching. *setSec*(*int*) sets the security level to a predefined integer value, which indicates whether the protocol performs encryption/decryption. The lower-layer protocol checks the setting, establishes the corresponding lower-layer connection and delivers messages. The lower-layer of the other communication peer updates this security level after the lower-layer connection is established, but this is oblivious to the upper-layer. If the other peer wants to know what security level the communication operates on, it should explicitly check it by using *getSec*(). Developers are oblivious to the implementation details of those interfaces but only regard them as cryptographic delegating pipes of the lower-layer. For instance, L2CAP encloses a security level *sec_level* in the channel data structure, and the lower-layer (HCI) accesses this value for encryption jurisdiction when performing message delivery. NWK maintains *nwkSecurityLevel* in NIB (Network Information Base), and the lower-layer (MAC) accesses this value to apply corresponding security strategies.

Interfacing with Application, Kernel and Other Protocols While our DSL is platform-independent, allowing to specify the main logic of network protocols, the generated code has to be platform dependent, interfacing with four parties: *user space applications*, *the underlying kernel*, *lower layer protocol* and *upper layer protocol(s)*. ProFactory currently supports Linux. The generated implementation for a protocol is packaged as a Linux kernel module. In the following, we explain the four interfaces. Then we present an example of L2CAP.

All network protocols in Linux interface with user applications through a fixed socket interface, which includes socket data structure and a number of API functions such as *bind*(), *listen*(), *accept*(), *connect*(), *send*() and *recv*(). To setup the interface (with the user space), the kernel module initialization needs to register the protocol by providing the protocol name such as "`L2CAP`" and the socket data structure of the protocol (containing a reference to a channel data structure). It also registers a list of functions that implement the aforementioned APIs, e.g., *l2cap_sock_sendmsg*() is registered for *send*() and *sendmsg*(). According to the action type n_{act}, *connect*(), *send*()/*sendmsg*() and *shutdown*() are connected to the message sending functions emitted through *Send* instances. This is transparent to developers. Also, the user space *send*() and *recv*() functions are merely sending and receiving raw data such that the underlying protocol is complete transparent to them.

The generated code interfaces with the upper layer protocols through a provided callback function. Theoretically, such functions can be registered. However, the current Linux protocol stack implementation hardcodes them. For example, the callback function provided by the Bluetooth family to L2CAP for raw data delivery has a fixed name *l2cap_data_recv*(), which is invoked inside the body of m_{data}.*recv*() that receives an L2CAP data message. Note that invoking the actual callback is transparent to developers but they only need to write a *deliver* expression (omitted in Figure 4.4 for brevity) to fulfill this. The interface with the lower layer protocol is similar. Upon receiving a data message in the lower layer (addressing our protocol), a fixed function is invoked that further invokes the various parser functions generated from DSL. Developers are also oblivious to the connection between the callback and the generated parser functions. The generated code also makes use of kernel functions such as socket allocation *sk_alloc*() when creating new channels, to which the developers are oblivious.

Figure 4.7 illustrates the platform-dependent interfaces of L2CAP, with the user space on the left, the lower layer protocol (HCI) on the right, the box with green header in the center generated from the DSL and the boxes with headers of other colors the interfaces. Specifically, module initialization (the orange box in the middle of the second column) registers the protocol structure (the top orange box in the third column) and L2CAP socket operations that are invoked by user space (the bottom orange box). The registered socket operations

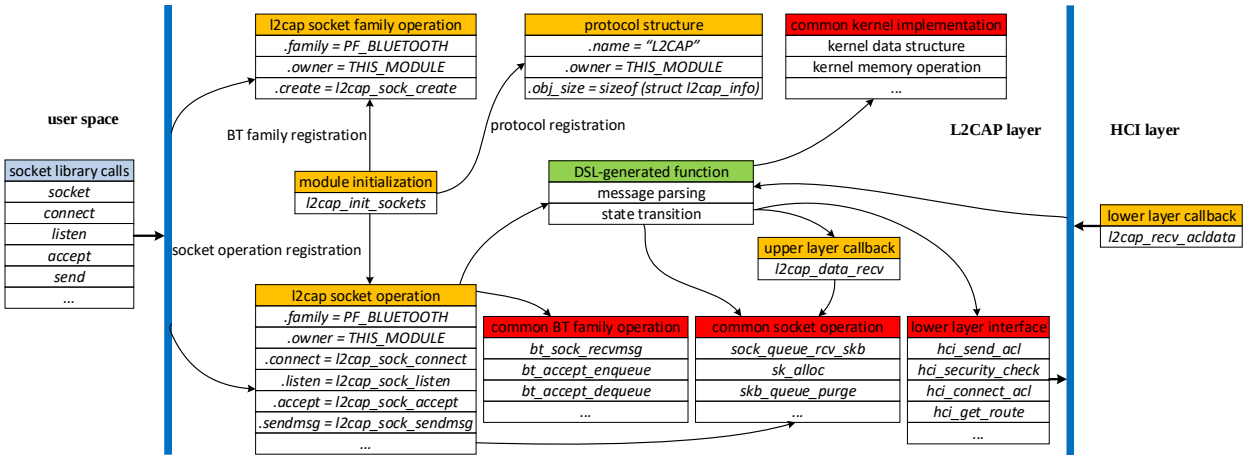

Figure 4.7. The platform-dependent interfaces of L2CAP

need to invoke the functions generated from DSL (in the center) to fulfill message parsing and state transition. Inside those functions, common operations of Bluetooth family, kernel socket, kernel data structure, and kernel memory are invoked (through the interfaces in the red boxes). In particular, L2CAP needs to invoke these interfaces to accomplish data delivery and connection establishment. The lower layer protocol HCI (Host Controller Interface) invokes a callback *l2cap_recv_acldata* to inform L2CAP upon receiving a message. L2CAP invokes a callback *l2cap_data_recv* after unwrapping the raw data.

Code Generation Algorithm The algorithm of code generation is simplified and summarized in Figure 4.8. First, data structures are emitted to compose a header file (lines 1-8). Note that the emission of some fields (e.g., timer, state, linked list, mutex and lock) in channel and connection structure is transparent to developers, and ProFactory leverages those fields to fulfill state transitions and multiplexing. Then, $findChanByPara$ function is generated for each of the parameters defined in the channel structure (lines 9-11), and *createChan* and *createConn* are also emitted (lines 12-13), where concurrency control operations are included. For each message containing a header, a parser is generated to extract (*skb_pull*) the header (lines 16-17), perform sanity checking (line 18) and invoke an inner parser (line 19), composing a hierarchical parsing tree. Similarly, a hierarchical message constructor (line 23-24) is generated to set header fields (e.g., length field). In contrast, a receiving transition defines the parser of a base message (leaf parsing node) without a header.

Def. : H, P, M, R, D - sets of defined headers, defined parameters, defined messages, defined receiving transitions and defined sending transitions
: $structGen$ - generate data structures
: $localGen$ - generate local pointers
: $extractGen$ - refer pointers to socket buffer data
: $checkGen$ - generate sanity check block
: $allocGen$ - generate socket buffer allocation block
: $baseMsg$ - get message base of a layered message
: $timerGen$ - generate timer updating block
: $sendGen$ - generate message delivery block
: $codeGen$ - generate type-specific block

foreach $h \in H$ **do**
  $structGen(h)$ ▷ packed data structure
end
foreach $p \in P$ **do**
  $structGen(p)$ ▷ packed data structure
end
$structGen(ch)$
$structGen(cn)$ ▷ end of header file generation
foreach $p \in ch$ **do**
  $codeGen(p)$ ▷ generate *findChanByPara*
end
$codeGen(ch)$ ▷ generate *createChan*
$codeGen(cn)$ ▷ generate *createConn*
foreach $m = h(f = n_1 ? : m_1) ... \in M$ **do**
  $codeGen(m) =$ **define** $parse\_m(ch, cn, skb\_in)\{$
  $localGen(h)$
  $extractGen(h)$
  $checkGen(h)$ ▷ failure is directed to drop
  if$(f == n_1)$ **return** $parse\_m_1(ch, cn, [h], skb\_in)$;
  ...
  else goto drop;
  drop: $kfree\_skb(skb\_in)$; **return** *error*;$\}$
  $+$ **define** $compose\_m(ch, cn, skb\_out)\{$
  $compose\_m_1(ch, cn, skb\_out); ...\}$ ▷ details omitted
end
foreach $r \in R$ **do**
  $codeGen(r) =$ **define**
    $parse\_baseMsg(m_{in})(ch, cn, h_{prev}, skb\_in)\{$
  $localGen(baseMsg(m_{in}))$
  $extractGen(baseMsg(m_{in}))$
  $checkGen(baseMsg(m_{in}))$ ▷ failure is directed to drop
  if$(codeGen(e_1) \wedge ch \rightarrow state == s_{f_1})\{$
  $allocGen(m_{out})$ ▷ failure is directed to drop
  $localGen(m_{out})$
  $extractGen(m_{out})$
  $codeGen(S)$
  $compose\_m_{out}(ch, cn, skb\_out)$;
  $(ch \rightarrow state) \leftarrow s_{t_1}$
  $timerGen(s_{t_1})$
  $sendGen(m_{out})\}$
  ...
  else goto drop;
  return *success*;
  drop: $kfree\_skb(skb\_in)$; $kfree\_skb(skb\_out)$; **return** *error*;$\}$
end
foreach $d \in D$ **do**
  $codeGen(d) =$ **define** $send\_m_{out}(ch, cn, data, len)\{$
  $allocGen(m_{out})$ ▷ failure is directed to drop
  $localGen(m_{out})$
  $extractGen(m_{out})$
  if$(codeGen(e_1) \wedge ch \rightarrow state == s_{f_1})\{$
  $codeGen(S)$
  $(ch \rightarrow state) \leftarrow s_{t_1}$
  $timerGen(s_{t_1})\}$
  ...
  else goto drop;
  if$(data \wedge len) memcpy(skb\_put(skb\_out, len), data, len)$;
  $compose\_m_{out}(ch, cn, skb\_out)$;
  $sendGen(m_{out})$; **return** *success*;
  drop: $kfree\_skb(skb\_out)$; **return** *error*;$\}$
end

Figure 4.8. Code Generation Algorithm

Similarly, it extracts all the fields (lines 28-29) and performs sanity checking (line 30). In addition, it allocates memory (line 32), prepares (*skb_put*) local references (lines 33-34), conducts the concrete state transition (lines 37-38), and performs packet delivery (line 39) for the outgoing message (if applicable). In particular, the optional header argument [h] in line 19 only applies to the parser of base message, which is reflected in line 27. This design aims to handle message formats where the innermost content parsing involves the information of the adjacent header. Finally, a sending transition defines a message serializer which is almost the same as the construction of the outgoing message in a receiving transition, but the only difference is that it is allowed to carry a user-space data chunk passed through a socket sending function (line 46, 56). Note that the recursive code generation of expressions/statements, and the channel/connection unlocking operations at the end of parsing are omitted for brevity.

Protocol Randomization. As the DSL specification only specifies the logic of a protocol, ProFactory can easily randomize the generated implementation, supporting different concrete message formats. For instance, the order of fields can be shuffled and random paddings can be added in between fields. As such, ProFactory can create lots of dialects (of the same protocol). Note that a same randomized implementation will be deployed on the peers of a connection and hence will not cause any incompatibility issues. Currently, ProFactory supports shuffling fields in headers and padding them with random bytes.

Implementation Entropy Randomization improves protocol security, especially in the confidentiality aspect. We quantify such improvement by studying the implementation entropy. For a peer (in a connection), we define a complete protocol execution path as

$$p = s_0, \mathbf{m_0}, s_1, \mathbf{m_1}, ..., s_{t-1}, \mathbf{m_{t-1}}, s_t$$

where s_0 is the initial state, $\mathbf{m_i} = \overrightarrow{m_i} | \mathbf{m_i} = \overleftarrow{m_i}$ means an outgoing or incoming message, $s_i, 1 \leq i < t$ denotes an intermediate state, and s_t is the end state. Correspondingly, the complete execution path of the other peer must be

$$p' = s'_0, \neg\mathbf{m_0}, s'_1, \neg\mathbf{m_1}, ..., s'_{t-1}, \neg\mathbf{m_{t-1}}, s'_t$$

where $\neg\mathbf{m_i}, 0 \leq \mathrm{i} < t$ denotes the m_i message in the opposite direction. Further assume all the exchanged messages along the path(s) have n unique headers $Hdr_1, ..., Hdr_n$ and the number of fields in Hdr_i is h_i. Since ProFactory can arbitrarily reorder all fields in a header and add k-padding bytes (per header) at arbitrary locations, according to permutation theory, we have the following entropy of p: $E(p) = \log_2[\prod_{\mathrm{i}=1}^{n} h_\mathrm{i}! \times \prod_{\mathrm{i}=1}^{n}(h_\mathrm{i}+1)^k]$. When $n = 2$, $k = 8$, and $\forall 1 \leq \mathrm{i} \leq n, h_\mathrm{i} = 3$, $E(p)$ is as large as 37.17. Note that here we consider the entropy of a full communication path because we assume the attacker needs to reverse engineer a path in order to exploit some valid functionality of the protocol. Also note that our computation assumes the fields are independent, which may not be satisfied in practice. If there is dependence between fields, the actual entropy would be smaller. To further improve security, it is also possible to randomize state machines by duplicating and splitting states, or adding redundant states. We leave it to our future work. Note that the improvement by randomization is orthogonal to that by encryption.

4.6 Automated Verification

An important goal of ProFactory is to achieve correctness and security by construction. The goal is validated by automated verification, which includes the following three aspects: verifying *concurrency control correctness using VCC [172], [173]*, *memory safety using Frama-C [174]*, and *state-machine correctness using Z3 [179]*. Note that bugs in any of these aspects could lead to security exploits. We use different tools for the three aspects as they have different focuses. For example, VCC was designed to prove concurrency correctness and has limited support for type/pointer casting, whereas Frama-C was designed to prove memory safety and can hardly reason about concurrent program behaviors. Z3 is a general reasoning engine, and hence very suitable for reasoning high-level state machine behaviors. The inputs to the first two are C-like functions that can be automatically emitted by ProFactory. However, as in most verification systems, the proof process may require developers' manual intervention, e.g., providing a small number of additional pre-conditions and/or loop invariants. The input to Z3 is a set of symbolic constraints derived by interpreting the DSL specification (i.e., $\mathbb{P}$ in Figure 4.5).

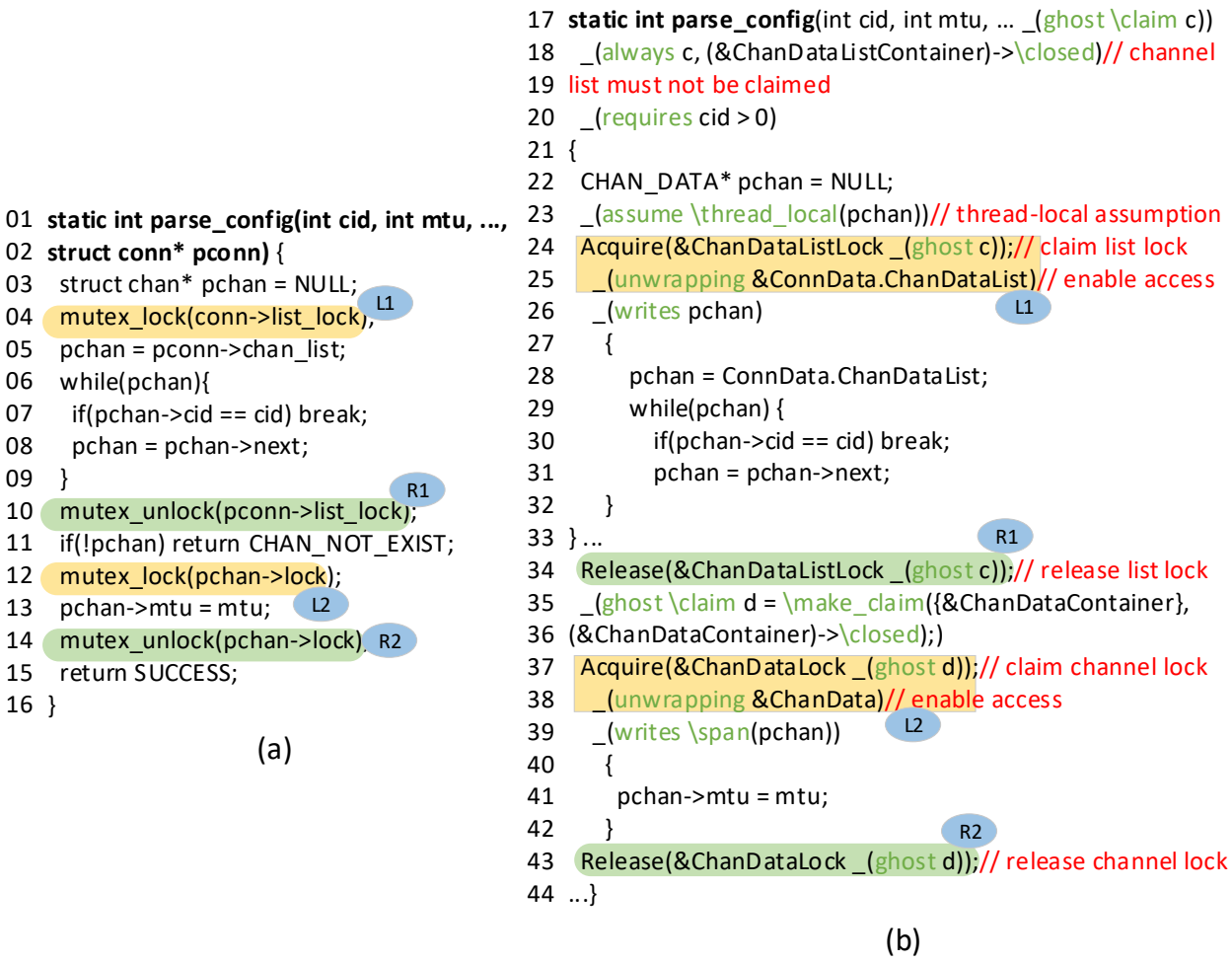

Figure 4.9. Concurrency correctness for updating a channel

Concurrency Control Correctness. In generated code, concurrency control takes place in connection multiplexing operations, where multiple channels share common information. The verification aims to ensure accesses to such common information do not cause races or deadlocks. During code generation, ProFactory also emits code that is amenable for VCC. Particularly, it makes the code self-contained by providing mock data structures and API functions, and replaces mutex operations with VCC-specific lock acquisition and release. With the annotations of the shared data structures, VCC automatically determines concurrency correctness. Figure 4.9 illustrates an example of this procedure. The code snippet in (a) presents a function generated by ProFactory that updates the `mtu` field of a channel indexed by a channel id `cid`. Observe that a lock is acquired at line 4 before

```
01 #define H_CONF_SIZE 8
02 #define P_OPT_SIZE 4
03 #define PL_MAX_CONF_SIZE 8
04 struct sk_buff {
05     int len;
06     char* data;
07     ...
08 };
09 struct conf {
10     int conf_type;
11     unsigned int len;
12 }__attribute__((packed));
13 struct opt {
14     unsigned int optVal;
15 }__attribute__((packed));
16 struct chan {
17     int mtu;
18     int fcs;
19     ...
20 };
21 /*@ requires ArgReq: \valid(pchan) &&
22 \valid(skb_in);
23   @ requires SkbReq: skb_in->len >= 0 && \
24 valid(skb_in->data + (0..skb_in->len));
25   @ requires SeparationReq: \
26 separated((char*)pchan + (0..sizeof(struct chan)),
27 (char*)skb_in + (0..sizeof(struct sk_buff)), skb_in-
28 >data + (0..skb_in->len));*/
29 static int parse_config(struct chan* pchan, struct
30 sk_buff* skb_in) {
31     int iter_cnt = 0;
32     struct conf* conf_hdr = 0;
33     struct opt* opt_para = 0;
34     /*@ loop invariant \valid(skb_in);
35       @ loop invariant \valid(pchan);
36       @ loop invariant skb_in->len >= 0;
37       @ loop invariant \valid(skb_in->data + (0..skb_in-
38 >len));
39       @ loop invariant iter_cnt <= PL_MAX_CONF_SIZE;
40       @ loop invariant \separated((char*)pchan +
41 (0..sizeof(struct chan)), (char*)skb_in +
42 (0..sizeof(struct sk_buff)), skb_req->data + (0..skb_in-
43 >len));
44       @ loop variant skb_in->len;*/
45     while(skb_in->len > H_CONF_SIZE && iter_cnt <
46 PL_MAX_CONF_SIZE) {
47         conf_hdr = (void*)skb_in->data;
48         skb_in->data += H_CONF_SIZE;
49         skb_in->len -= H_CONF_SIZE;
50         if(conf_hdr->len > skb_in->len || conf_hdr-
51 >len != P_OPT_SIZE) goto drop;
52         opt_para = (void*)skb_in->data
53         if(con_hdr->conf_type == 1) {
54             chan->mtu = opt_para->optVal;}
55         else if(con_hdr->conf_type == 2) {
56             chan->fcs = opt_para->optVal;}
57         ...
58         skb_in->len -= P_OPT_SIZE;
59         skb_req->data += P_OPT_SIZE;
60         iter_cnt++;
61         }
62   return 1;
63 drop:
64     ...
65     return 0;
66 }
```

Figure 4.10. Memory safety for iterating a parameter list

accessing the list of channels and then released at line 10. The channel is further locked at line 12 and then released at line 14 after updating the mtu field. The snippet in (b) shows the corresponding version for VCC verification with those in green being VCC-specific keywords declaring shared objects and auxiliary objects. The tags show the correspondences of the mutex operations in (a) and (b). VCC then proves that the code in (b) is race-free and deadlock-free. Details of VCC are not the focus of our work and hence elided. Interested readers are referred to [173].

Memory Safety. Frama-C verification requires per-function code annotations written in its own ACSL specification language [174]. Most such annotations (e.g., pointer validation, memory span validation, memory span separation, loop variant and loop invariant) can be generated by ProFactory, but due to the difficulty in automatically producing the complete set of annotations, developers may need to manually insert additional (very limited) preconditions and/or loop invariants to assist the verification process. All the kernel data structures (e.g., `sk_buff`) and their operations are manually pre-simplified (one-time effort) as they are not supported. The verification excludes memory access vulnerabilities, i.e., buffer overflow, invalid pointer dereference, memory leakage, use after free and double free. In particular, being free from buffer overflow and invalid pointer dereference are deductively verified, where intermediate targets such as being free of infinite loop, integer overflow/underflow and dividing by zero are also guaranteed, while Frama-C guarantees being free of memory leakage, use after free and double free by tracking memory allocation/deallocation operations. Figure 4.10 showcases a generated function with Frama-C annotations. The function iterates a parameter list stored in `skb_in`, where the lines highlighted in green are the annotations. It was successfully verified by Frama-C. Specifically, the annotations consists of two parts, *function annotations* (lines 21-28) and *loop annotations* (Line 34-44). The former denotes the function preconditions (which need to be verified at each callsite of the function). For instance, the execution of `l2cap_conf_parse` requires valid pointers (lines 21 and 22), valid socket buffer size (lines 23 and 24) and *separated argument spans* (lines 25-28), meaning they do not overlap. In contrast, loop annotations assist proving loop termination and iterative memory access safety. It usually includes a loop variant that changes across iterations and hence is related to loop termination (e.g., line 44, length of remaining data in the socket buffer), and a list of loop invariants that specify predicates that must hold across iterations and substantially facilitate the proof procedure (e.g., lines 34-43). Internally, the invariants are auxiliary lemmas which help the proof and must also be deductively verified from function annotations. More details about Frama-C verification can be found in [174].

State Machine Verification by Symbolic Model Checking. To verify state machine correctness, we translate the DSL specification to symbolic constraints and check if the

model satisfies a number of general properties. ProFactory rewrites DSL specification to the SMT-LIB [185] representation of Z3, a well-known theorem prover. Z3 supports reasoning of constraints in a large number of theories such as integer, string, array, and bit-vector theories. Specifically, symbolic variables are introduced to describe attributes of abstract data types, such as value of a field f, denoted as $f.val$ in $\mathbb{P}$ of Rule 1 in Figure 4.5, and variables such as channel state. Operations in $\mathbb{P}$ such as additions and multiplications are translated to Z3 operations in the corresponding theories. Logical operations, such as conjunctions, disjunctions, and implications (e.g., those in $\mathbb{P}$ of Rule 6 in Figure 4.5 describing the different possible state transitions guarded by different conditions) are directly supported by Z3. General statements such as assignments (whose semantics are standard and elided from Figure 4.5) are also translated to symbolic constraints. The translation of these statements is standard [186] and elided. Loops are unrolled with the unroll bound of 10, which is practically sufficient for the protocols we model. Note that while at runtime there are multiple channels, we do not have to model these instances during symbolic model checking as we are interested in state transition properties. Variables that can be defined by the user-space, kernel, and remote requests are largely free variables. That is, they are only constrained by range specifications if there are any. We say a property is satisfied (SAT) if Z3 can find a value assignment to all these free variables that can satisfy the property. We validate a number of general properties of state machine behaviors.

State Reachability The first property asserts that a destination state s_1 is reachable from the initial state s_0, denoted as $reachable(s_0, s_1)$. Let the symbolic encoding of the protocol be M, which includes the symbolic constraint encodings of all the protocol specifications and statements, and the state variable be $state$. We use $reachableInOne(s_0, s_1)$ to denote that s_1 can be reached from s_0 by one step transition. It is hence defined as $M \wedge state = s_0 \rightarrow state_1 = s_1$. Note that we have to rename $state$ to $state_1$ to denote the new state. Intuitively, it is SAT if Z3 can find a valuation to free variables (e.g., a message) that induces the state to change from s_0 to s_1 in one step. Constraint $reachableInTwo(s_0, s_1)$ is defined as $reachableInOne(s_0, s_k) \wedge reachableInOne(s_k, s_1)$. Note that the M encodings in $reachableInOne(s_0, s_k)$ and $reachableInOne(s_k, s_1)$ need to be re-

named as well since they need to be resolved independently (representing different messages). Therefore, $reachable(s_0, s_1)$ is defined as follows.

$$reachableInOne(s_0, s_1) \vee reachableInTwo(s_0, s_1) \vee ...$$

Currently, our reasoning is bounded at 15 steps, that is, the maximum transition path has 15 steps.

Transition Coverage This property dictates that any transition defined in the protocol is feasible, meaning that it can be triggered by some message sequence(s). Assume the condition guarding a transition from s_1 to s_2 is e, we assert $reachable(s_0, s_1) \wedge \mathrm{e}$. Intuitively, we assert that s_1 is reachable from the initial state and e is satisfiable.

Absence of Transition Conflict This property states that if a message can trigger two or more transitions, there are not two of them satisfiable simultaneously. Assume s can lead to s_1, s_2, ..., s_k, guarded by e_1, e_2, ..., e_k, respectively. For any $\mathrm{i}, \mathrm{j} \in [1, k]$ and $\mathrm{i} \neq \mathrm{j}$, we assert $reachable(s_0, s) \wedge \mathrm{e_i} \wedge \mathrm{e_j}$. Any SAT result indicates the protocol is buggy. If all are UNSAT, the property holds.

Race-free Message Sends This is the property illustrated in Section 4.2. When two peers are both in some state that is expected to send out a message, the protocol may be trapped into an asynchronous sending race that may lead to message loss. Suppose we have a state s_1^a for device A and a state s_1^b for device B and they have transitions to states s_2^a and s_2^b respectively, which are both triggered by a message send event, with A sending m_a and B sending m_b. As both devices are in a sending race, A may stay at s_1^a or reach s_2^a when m_b arrives. Correspondingly, B may stay at s_1^b or reach s_2^b when m_a arrives. Therefore, message loss may happen when (1) s_1^a or s_2^a does not accept m_b, or (2) s_1^b or s_2^b does not accept m_a, since the message m_a or m_b can be dropped. Validating this property requires coupling the reasonings of both sides of a connection. In the following, we define a constraint $coReachInOne(s_1^a, s_1^b, s_2^a, s_2^b)$ that states that by exchanging a message, device A can reach s_2^a (from s_1^a) and device B can reach s_2^b (from s_1^b).

$$(M_a \wedge s_1^a \rightarrow s_2^a) \wedge (M_b \wedge s_1^b \rightarrow s_2^b) \wedge (\overleftarrow{m}_a = \overrightarrow{m}_b \vee \overleftarrow{m}_b = \overrightarrow{m}_a)$$

The first two clauses assert there are messages that induce the state transitions and the last asserts that the incoming message at A must be the outgoing message at B or vice-versa. Note that the messages (e.g., $\overleftarrow{m}_a$ and $\overrightarrow{m}_a$) are essentially a subset of symbolic variables in the model M_a. They are instantiated when Z3 resolves M_a. We can define $coReach(s_1^a, s_1^b, s_2^a, s_2^b)$ to dictate that starting from s_1^a and s_1^b, the two devices can reach s_2^a and s_2^b, respectively, after exchanging a sequence of messages, in a way similar to defining $reachable()$ from $reachableInOne()$. For all state pairs s_1^a and s_1^b that can both send messages, guarded by conditions e_a and e_b. We assert $coReach(s_0^a, s_0^b, s_1^a, s_1^b) \wedge e_a \wedge e_b$. If none is SAT, the property holds. Otherwise, it is vulnerable.

Deadlock-free Message Receives The property states that at any time when two peers are both expecting an incoming message at some state, the protocol may get stuck in a receiving deadlock. Therefore, given a pair of states s_1^a and s_1^b at the two peers, respectively, we assert the following.

$$coReach(s_0^a, s_0^b, s_1^a, s_1^b) \rightarrow ((M_a \wedge s_1^a \rightarrow s_2^a \ \wedge \ \overrightarrow{m}_a \neq nil) \vee$$
$$(M_b \wedge s_1^b \rightarrow s_2^b \ \wedge \ \overrightarrow{m}_b \neq nil))$$

The antecedent is the reachability of the two states. The consequent is to say either one can move forward with a message send, meaning that the peer does not have to wait for an incoming message. Any pair that yields UNSAT indicates a deadlock problem.

Consistent Security Level In Linux each protocol maintains a security level variable that varies during the lifetime of a connection, depending on the states of authentication and encryption. ProFactory supports the mechanism although we do not model it in the DSL syntax/semantics for brevity. Given a state s, we use $sec(s)$ to denote the security level at the state. The security consistency property dictates that if a state can be reached through different message sequences, it must have the same security level. We assert the following.

$$reachable(s_0, s) \rightarrow sec(s) \ \wedge \ reachable(s_0', s') \rightarrow sec(s')$$
$$\wedge \ sec(s) \neq sec(s')$$

Here, s'_0 and s' denote a renamed version of s_0 and s, respectively, indicating that they are considered different symbolic variables internally although they have the same meaning. This is to allow Z3 to resolve them independently. Any SAT result suggests an inconsistency problem. In Section 4.7, we would exhibit a violation of the property.

For the above discussion, one can observe that the model checking is performed in two modes: *isolated* and *coupled*. In the former, we only consider one side of the connection and assume messages from the other side can be anything, even corrupted intentionally by the adversary. In the latter, we reason both sides together and trust the connection to deliver messages properly such that messages on the two sides can be coupled (e.g., in the race-free and deadlock-free properties).

4.7 Evaluation

We implement a prototype of ProFactory on Haskell. We use it to customize 8 IoT protocols, including various Bluetooth (v5.0) protocols for Linux 4.10 kernel and Zigbee (v1.0) NWK layer for ZBOSS simulator [187]. Note that ProFactory can also be ported to other protocol stacks or systems if corresponding platform-dependent interfaces are provided. Currently, ProFactory does not fully support code generation for the Android kernel. Hence, for the evaluation purposes, we generate core communication components of Bluetooth protocols and manually adapt and integrate them into Android to obtain a customized Bluedroid (or Fluoride) [188].

According to original Bluetooth specifications, we write SDP (Service Discovery Protocol), PAN (Personal Area Network), BNEP, HIDP, RFCOMM and L2CAP in our DSL, and generate codes for Linux. Note that BlueZ operates SDP and PAN in the user space but we generate kernel versions for them. Those implementations all pass the verification and they work properly when communicating with real devices (or simulator in the case of Zigbee). We customize RFCOMM and L2CAP, only focusing on the functionality of connection-oriented data delivery, and separating L2CAP classic and L2CAP BLE (Bluetooth Low Energy). Table 4.1 shows the lines of code in DSL, in the original implementation, and in the gener-

Table 4.1. LoC comparison between original BlueZ implementations and codes generated by ProFactory

Protocol	Lines of Codes		
	Model Definition	Original	Generated
SDP	971	5500	3478
PAN	183	1023	635
BNEP	590	1447	1162
HIDP	578	1966	1580
RFCOMM	738	4547	2465
L2CAP-CLA	1148	10328	3419
L2CAP-BLE	1247		3868
Zigbee-NWK	782	2373	1471

ated implementation for each protocol. Compared to the original implementation, the DSL specifications are much more succinct. Even the generated code is of smaller size.

4.7.1 System Performance

With the generated protocol implementation, we deploy two Raspberry Pi 3 devices, two desktop computers and two Android phones (Google Pixel 2) to measure the performance efficiency for paired communication. Specifically, Raspberry Pi 3 devices and desktop computers load our customized L2CAP and RFCOMM, while we manually replace communication functions with our generated codes (adapted for Android) for L2CAP and RFCOMM in Bluedroid. We perform file transfer (object exchange, using RFCOMM and L2CAP) of a 20MB file for both of the original Bluetooth implementations (BlueZ and Bluedroid), our customized ones and randomized (8-byte padding) versions. The experiment is repeated 10 times and we collect the geometric mean of time costs in Figure 4.11. As illustrated, the customized implementation is about 4% less efficient. The efficiency loss in customization is mainly caused by the sanity checks enforced on fields, and the lack of data structure layout optimization in type-based code generation. Meanwhile, the memory footprints of the original bluetooth module are 536KB, 536KB and 439KB for desktop, Raspberry Pi and Phone, while the ones of the customized module are 533KB, 533KB and 438KB. The difference is negligible, and the customized module consumes slightly less because we trimmed unused

components in customization. Figure 4.12 shows that the randomization incurs an additional 2% overhead which is due to transmitting extra bytes and assigning random values to padded bytes for each message.

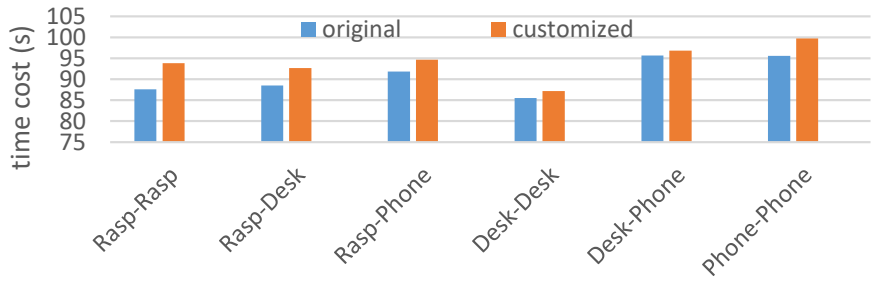

Figure 4.11. Comparison of time costs in paired file transfer

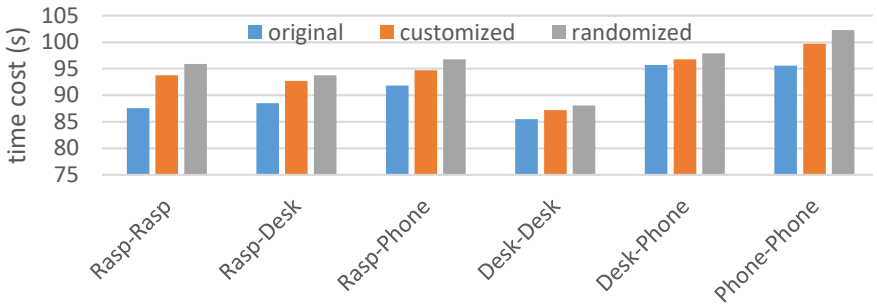

Figure 4.12. Extended comparison of time costs

We compute the implementation entropies (the minimum entropy among all the state transition paths from start to terminal) of the 8 customized protocols in Table 4.2. As the results demonstrate, the format randomization in ProFactory can achieve high implementation entropy (near 60-bit entropy for L2CAP-BLE). The reason for the low entropy of PAN and HIDP is that their messages only have one header which contains two fields. In addition, we also use Netzob [189] to perform message format inference for the randomized protocols. Specifically, we generate 20 randomized versions of each protocol and produces about 200 messages for each version. Then we mix the messages belonging to the same protocol to compose the evaluation dataset. As illustrated in Table 4.2, the format clustering precision of Netzob for randomized implementations is low (less than 10% for most cases) and the

Table 4.2. Implementation entropy of customized protocols

Protocol	padding			Netzob clustering precision		
	0-byte	4-byte	8-byte	0-byte	4-byte	8-byte
SDP	2.58	10.58	18.58	14.9%	9.6%	7.2%
PAN	1.00	7.34	13.68	45.2%	17.1%	8.8%
BNEP	9.75	24.09	38.44	10.3%	7.5%	4.8%
HIDP	1.00	7.34	13.68	43.3%	15.6%	8.0%
RFCOMM	12.68	29.7	46.71	5.4%	4.9%	4.3%
L2CAP-CLA	15.75	33.85	51.94	6.2%	5.0%	3.1%
L2CAP-BLE	21.68	39.52	57.35	4.4%	3.9%	2.6%
Zigbee-NWK	18.47	31.76	45.04	5.1%	4.5%	3.9%

precision decreases with higher entropies. According to the results, an entropy of 10 may be a reasonable threshold as it lowers the clustering precision to less than 10%.

We also collect the measurements of verification runtime, as Table 4.3 shows. In symbolic model checking, we set the timeout of Z3 to 20 seconds and we do not observe any timeout. For Frama-C, we set the timeout to 5s and all the function proofs are accomplished in 3s. Except the time cost, we also collect the number of concurrency control instances for VCC verification, the numbers of annotated functions, loops, and the total emitted annotations for Frama-C annotation, and the number of verified paths (either *Reachable* or *coReach*) for symbolic model checking. As demonstrated, the cost of verification for protocol implementations in practice are affordable and acceptable, where Frama-C verification of any protocol is finished within 80 seconds and symbolic model checking of any protocol is finished within 375 seconds.

Zigbee Evaluation It is challenging to obtain devices with a programmable Zigbee stack. Therefore, we select the Zigbee simulator ZBOSS to conduct our evaluation. Note that this is a reasonable option because mainstream manufacturers are using ZBOSS to test Zigbee implementations before product shipment. In Zigbee NWK, we model the complete NLDE (Network Layer Data Entity) which is responsible for data delivery, consisting of request (for data sending), confirm (for confirming receipt) and indication (for updating link quality). In particular, we customize NLDE by removing the alias fields which are rarely used. For NLME (Network Layer Management Entity), we only model the message

Table 4.3. Runtime internals of ProFactory in verification

Protocol	VCC verification		Frama-C verification				symbolic model checking	
	#concur.	cost (s)	#func.	#loop	#anno.	cost (s)	#path	cost (s)
SDP	0	0.00	12	2	171	33.17	26	85.82
PAN	1	1.71	8	2	115	19.20	49	179.55
BNEP	3	6.93	18	2	234	48.53	63	302.13
HIDP	2	3.08	16	3	242	39.08	25	64.20
RFCOMM	3	5.46	14	0	163	34.21	85	243.86
L2CAP-CLA	5	9.82	32	4	528	75.32	107	373.04
L2CAP-BLE	3	7.14	26	2	427	61.45	62	205.32
Zigbee-NWK	2	4.26	22	0	192	53.85	95	326.50

formats and generate the primary parsers, but all the payload processing is delegated to the original implementation in ZBOSS. NLME is hardware-specific, highly coupled with lots of hardware physical features and routing operations. Without introducing additional specifications to describe those semantics, ProFactory is not able to correctly express the whole procedure. Message formats of Zigbee are flatter than that of Bluetooth and a header tailing with a variable-sized field is sufficient to depict all the messages. Because Zigbee establishes a connectionless ad-hoc network, it does not have multiplexing, while all the shared connection information is stored in the NIB data structure for concurrent access. Also, since it does not maintain connections, timer operations are excluded (router nodes require timers to repeatedly send out broadcast messages, but our evaluation only focuses on message delivery of end nodes). ZBOSS is not a kernel-oriented implementation and hence the generation of standard socket interfaces are excluded. ZBOSS maintains a buffer pool, where a message buffer is allocated by `ZB_BUFF_FROM_REF` and released by `zb_free_buf`. Correspondingly, kernel socket buffer operations are shifted to the two functions in code generation for Zigbee. Between layers, data are delegated through a ring buffer (kernel simply passes the socket buffer). Ring buffer operations are wrapped by ZBOSS, and we can directly generate codes to invoke them. Note that different from kernel, the lower/upper layer in ZBOSS is not responsible to release the buffer, hence invoking `zb_free_buf` is always generated after writing to the ring buffer.

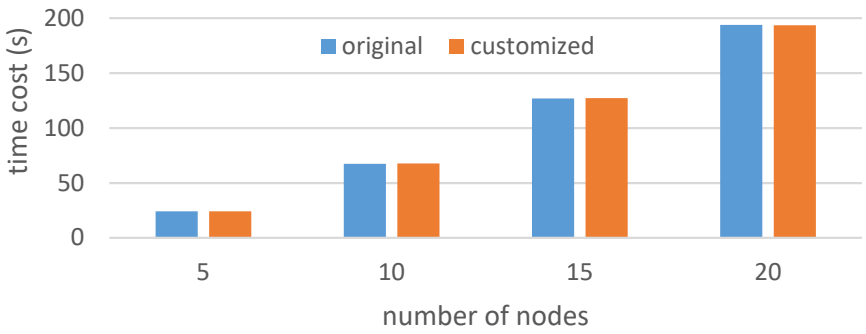

Figure 4.13. Comparison of time costs in Zigbee data transfer

In evaluation, we create 5, 10, 15 and 20 nodes to measure the communication time costs. One node communicates with each of the other nodes to transfer 1MB application data. In addition to those nodes, we also create a forwarding node serving as a router to prevent noises caused by pairwise communication. Results are collected in Figure 4.13. As demonstrated, no significant overhead could be observed. Meanwhile, the memory footprints of the original ZBOSS are 16.9KB, 17.8KB, 19.5KB and 22.4KB, while the ones of the customized version are 16.9KB, 17.7KB, 19.7KB and 22.4KB. The difference is also negligible.

4.7.2 Vulnerability Averting

Table 4.4 lists 81 (excluding the one in Section 4.2) recently-released CVEs that could have been averted if protocols had been defined and generated in ProFactory. Specifically, we searched for target CVEs by keyword "Bluetooth" (from 2017) and "Zigbee", and collected the ones related to field boundary checking and security downgrading. These represent the CVEs that we can find from public sources for the protocols we consider. We do no intend to claim the list is complete as there may be vulnerabilities that we are not aware of. When PoCs are available, we checked if they can attack our generated code. Otherwise, we manually checked the CVE codes/patches and compared with our generated code to ensure the triggering conditions are precluded. The avertable vulnerabilities are caused by either the lack of boundary checks in message parsing or the possible transition paths leading to security downgrading (i.e., inconsistent security levels). Further, we also find a representative vulnerability example that ProFactory cannot avert, cracking the protocol

session key. Since current version of ProFactory cannot model cryptographical interactions, any vulnerability residing in the key negotiation procedure of a protocol cannot be averted. The symbolic model checking does not disclose state-machine bugs in most of the protocols rewritten in our DSL. This is expected because these are widely-used protocols which are considered well specified and maintained. However, we do find a known state-machine vulnerability in PAN that could have been averted. Next, we showcase two of the 81 vulnerabilities (one for secure message parsing and one for state machine verification) in details. In addition, an avertable state-machine vulnerability in mainstream IoT clouds discovered in [161], [190] will also be illustrated.

Table 4.4. Averting IoT vulnerabilities

CVE No.	Protocol	Type	Avertable	CVE No.	Protocol	Type	Avertable
CVE-2020-0022	Bluetooth	missing boundary check	✓	CVE-2019-9474	Bluetooth	missing boundary check	✓
CVE-2019-9473	Bluetooth	missing boundary check	✓	CVE-2019-9462	Bluetooth	missing boundary check	✓
CVE-2019-9435	Bluetooth	missing boundary check	✓	CVE-2019-9434	Bluetooth	missing boundary check	✓
CVE-2019-9426	Bluetooth	missing boundary check	✓	CVE-2019-9425	Bluetooth	missing boundary check	✓
CVE-2019-9422	Bluetooth	missing boundary check	✓	CVE-2019-9419	Bluetooth	missing boundary check	✓
CVE-2019-9417	Bluetooth	missing boundary check	✓	CVE-2019-9413	Bluetooth	missing boundary check	✓
CVE-2019-9404	Bluetooth	missing boundary check	✓	CVE-2019-9402	Bluetooth	missing boundary check	✓
CVE-2019-9401	Bluetooth	missing boundary check	✓	CVE-2019-9398	Bluetooth	missing boundary check	✓
CVE-2019-9397	Bluetooth	missing boundary check	✓	CVE-2019-9396	Bluetooth	missing boundary check	✓
CVE-2019-9395	Bluetooth	missing boundary check	✓	CVE-2019-9394	Bluetooth	missing boundary check	✓
CVE-2019-9393	Bluetooth	missing boundary check	✓	CVE-2019-9390	Bluetooth	missing boundary check	✓
CVE-2019-9389	Bluetooth	missing boundary check	✓	CVE-2019-9388	Bluetooth	missing boundary check	✓
CVE-2019-9387	Bluetooth	missing boundary check	✓	CVE-2019-9368	Bluetooth	missing boundary check	✓
CVE-2019-9367	Bluetooth	missing boundary check	✓	CVE-2019-9363	Bluetooth	missing boundary check	✓
CVE-2019-9355	Bluetooth	missing boundary check	✓	CVE-2019-9353	Bluetooth	missing boundary check	✓
CVE-2019-9343	Bluetooth	missing boundary check	✓	CVE-2019-9342	Bluetooth	missing boundary check	✓
CVE-2019-9341	Bluetooth	missing boundary check	✓	CVE-2019-9333	Bluetooth	missing boundary check	✓
CVE-2019-9332	Bluetooth	missing boundary check	✓	CVE-2019-9331	Bluetooth	missing boundary check	✓
CVE-2019-9330	Bluetooth	missing boundary check	✓	CVE-2019-9328	Bluetooth	missing boundary check	✓
CVE-2019-9327	Bluetooth	missing boundary check	✓	CVE-2019-9326	Bluetooth	missing boundary check	✓
CVE-2019-9312	Bluetooth	missing boundary check	✓	CVE-2019-9289	Bluetooth	missing boundary check	✓
CVE-2019-9287	Bluetooth	missing boundary check	✓	CVE-2019-9286	Bluetooth	missing boundary check	✓
CVE-2019-9285	Bluetooth	missing boundary check	✓	CVE-2019-9284	Bluetooth	missing boundary check	✓
CVE-2019-9265	Bluetooth	missing boundary check	✓	CVE-2019-9260	Bluetooth	missing boundary check	✓
CVE-2019-9250	Bluetooth	missing boundary check	✓	CVE-2019-9249	Bluetooth	missing boundary check	✓
CVE-2019-9241	Bluetooth	missing boundary check	✓	CVE-2019-9237	Bluetooth	missing boundary check	✓
CVE-2019-2009	Bluetooth	missing boundary check	✓	CVE-2019-1996	Bluetooth	missing boundary check	✓
CVE-2018-9588	Bluetooth	missing boundary check	✓	CVE-2018-9583	Bluetooth	missing boundary check	✓
CVE-2018-9566	Bluetooth	missing boundary check	✓	CVE-2018-9560	Bluetooth	missing boundary check	✓
CVE-2018-9555	Bluetooth	missing boundary check	✓	CVE-2018-9544	Bluetooth	missing boundary check	✓
CVE-2018-9541	Bluetooth	missing boundary check	✓	CVE-2018-9540	Bluetooth	missing boundary check	✓
CVE-2018-9510	Bluetooth	missing boundary check	✓	CVE-2018-9509	Bluetooth	missing boundary check	✓
CVE-2018-9508	Bluetooth	missing boundary check	✓	CVE-2018-9507	Bluetooth	missing boundary check	✓
CVE-2018-9506	Bluetooth	missing boundary check	✓	CVE-2018-9505	Bluetooth	missing boundary check	✓
CVE-2018-9504	Bluetooth	missing boundary check	✓	CVE-2018-9502	Bluetooth	missing boundary check	✓
CVE-2018-9363	Bluetooth	missing boundary check	✓	CVE-2018-9358	Bluetooth	missing boundary check	✓
CVE-2017-0785	Bluetooth	missing boundary check	✓	CVE-2017-13283	Bluetooth	missing boundary check	✓
CVE-2017-1000250	Bluetooth	missing boundary check	✓	CVE-2020-0379	Bluetooth	downgrading security level	✓
CVE-2020-9770	Bluetooth	downgrading security level	✓	CVE-2019-2225	Bluetooth	downgrading security level	✓
CVE-2017-0783	Bluetooth	downgrading security level	✓	CVE-2015-8732	Zigbee	missing boundary check	✓
CVE-2015-6244	Zigbee	missing boundary check	✓	CVE-2020-15802	Bluetooth	cracking session key	✗

```
/sdpd-request.c
722 sdp_buf_t *pCache = sdp_get_cached_rsp(cstate); ...
726 if (pCache) {
      if (pCache && cstate->cStateValue.maxBytesSent < pCache->data_size) // patch
727     short sent = MIN(max_rsp_size, pCache->data_size - cstate->cStateValue.maxBytesSent);
728     pResponse = pCache->data;
729     memcpy(buf->data, pResponse + cstate->cStateValue.maxBytesSent, sent);
```

Figure 4.14. Code and patch of CVE-2017-1000250

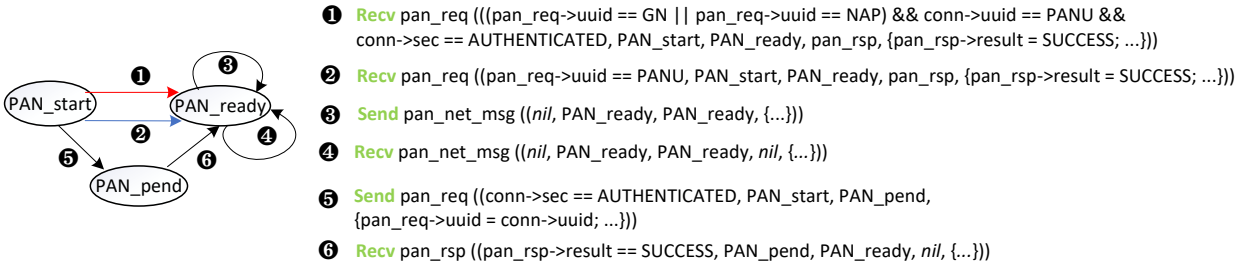

Figure 4.15. PAN state machine of CVE-2017-0783

CVE-2017-1000250. This vulnerability [148] can cause information leak. It resides in the SDP implementation of Linux BlueZ (version 5.46 or earlier). When BlueZ responds to an SDP request, the response message size may exceed the MTU of L2CAP and is dropped by L2CAP. Hence SDP must realize its own fragmentation mechanism. Specifically, a sender peer marks a field to notify a receiver peer that the current packet has continuous fragments. Accordingly, the receiver peer responds with an offset denoting the bytes that have been received so far. Then, the sender peer continues sending the next fragment from the exact offset. Figure 4.14 illustrates the buggy code and its original patch [191]. As demonstrated, before patching, the value of offset `maxBytesSent` is not checked and `sent` can be overflowed. Therefore, out-of-bound bytes can be copied to `buf` → `data` and sent to the remote peer. Note that automatically fixing the bug on the existing implementation (i.e., generating the illustrated patch) is extremely difficult without developers' intervention as it is very hard for an automatic analysis to infer the needed data-flow relations. In contrast, since we automatically generate secure packet fragmentation/assembly code all together and perform code verification with Frama-C, the vulnerability is avoided in the first place.

CVE-2017-0783. This vulnerability [192] allows a man-in-the-middle attack and it resides in Android/Windows's PAN implementation. PAN offers the service of network proxy via Bluetooth devices. In the protocol, a device can serve as any of the three roles, GN (Group Ad-hoc Network), NAP (Network Access Point) and PANU (PAN User). Among them, PANU acts as a network client user and GN/NAP represents a proxy/router/bridge. When a PANU device connects to a PANU/GN/NAP device, neither peer checks the security level. In contrast, when a GN/NAP device connects to a PANU device, the PANU device must perform the security check because an unauthorized GN/NAP device can redirect connections to malicious targets. In the vulnerable Bluedroid implementation [193], a PANU device is allowed to act as a GN/NAP device after it connects to a PANU device, bypassing the security check and performing reverse tethering. Figure 4.15 illustrates the problematic state machine, where a PAN device can send/receive wrapped network messages at `PAN_ready` state. The security issue lies in that a device has two transition paths (❶ and ❷) from `PAN_start` to `PAN_ready` but they have inconsistent security levels (❶ requires `AUTHENTICATED` but ❷ does not). The bug is detected when we model check the *consistent security level* property. Note that we integrate the `uuid` which is exchanged in ATT (Attribute Protocol) into PAN and omit the failure processing branches for PAN messages to simplify our modeling and demonstration. The official patch [193] roughly prohibits any connection to/from remote GN/NAP devices when an Android device performs as PANU, while splitting the `PAN_ready` state to deal with ❶ and ❷ separately may be a better solution.

Device Interaction in IoT Clouds. Unauthenticated interaction is a common vulnerability in mainstream IoT clouds [161], [190], which can enable various attacks such as device hijacking and denial of service. Such vulnerabilities have a similar nature, which can be modeled as buggy state transition and exposed by our model checking (if the protocol is specified in our DSL). In Figure 4.16, we model the (buggy) IoT cloud protocol. Specifically, a device is bound with a user after IoT cloud processes a binding request from the user-side mobile application (❶). When bound, the device is able to log in/out IoT could (❸ and ❹), receive commands (❺) and update status (❻). To unbind a device, a request could be sent from either the user-side mobile application (❷ and ❸) or the device (❼). The security issue lies in that all the device interactions are not `AUTHENTICATED`, and an attacker can craft

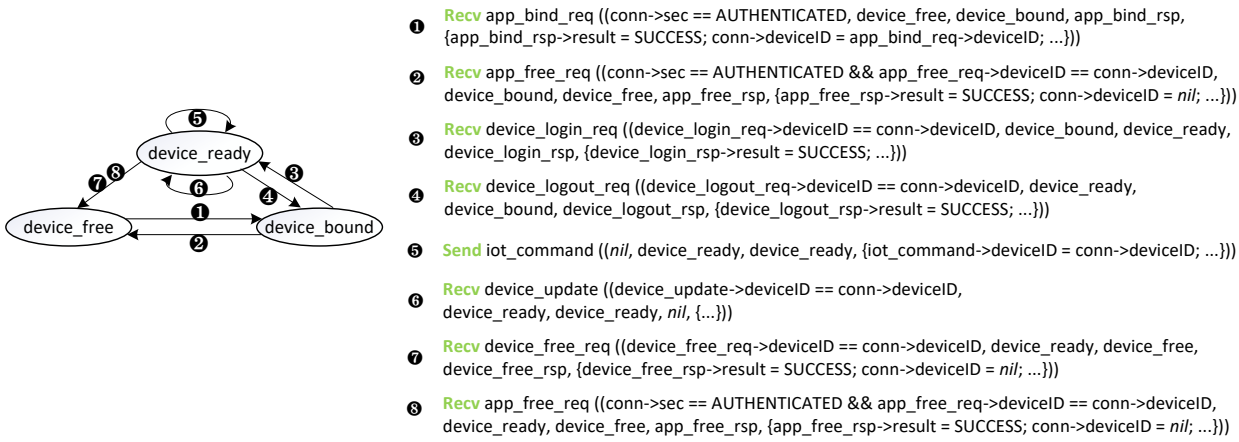

Figure 4.16. Unauthenticated device interaction in IoT clouds

a phantom device with pre-fetched device ID to logout, unbind and hijack the real device. Note that this is a global state machine associated with multiple parties, where the IoT cloud still maintains device states when the other parties are offline, and state transitions are triggered by messages transmitted through on-demand connections. Therefore, different from a single-connection protocol, the security level of a state cannot be inherited from the prior one. For example, `device_free` can transit to itself via ❶❸❼. If the transitions take place in a single connection, since the authentication of ❶ is asserted, the following ❸❼ must also be `AUTHENTICATED`. However, this is not true for the global state machine in Figure 4.16 because ❶❸❼ (can) happen in independent connections, among which security levels cannot be transmitted. Hence, when verifying the *consistent security level* property for such models, we determine the security level of a state only by the assertions on its immediate transition (❼ for `device_free` in this example). Accordingly, we could figure out two buggy states, `device_free` and `device_bound`, as their immediate transitions have different assertions of security levels. Note that `device_ready` cannot be identified as a buggy state even if we know all the unauthenticated transitions to it are insecure. This could be the limitation of our verification as we only focus on the inconsistency between security levels.

4.8 Discussion

Flow Control. ProFactory currently does not support modeling flow control algorithms. Specifying flow-control is challenging due to the lack of regular design of the various flow-control algorithms. However in the context of IoT, due to resource constraints, existing protocols rarely have complex flow control. In fact, early BlueZ implementations did not have it at all. Regularity may be abstracted out of popular light-weight flow control algorithms and modeled in our DSL. We will leave it to our future work.

Specification Flaws. The working of ProFactory largely relies on the robustness of the specification (or DSL). If any part of the specification was further found flawed (e.g., a protocol model specified by ProFactory cannot be securely handled in the generated implementation), the claimed security guarantees should be degraded.

Security Properties. Cryptographic constructs are not covered in current specification set, but all the cipher operations are delegated to a pre-established lower layer. This leads to the lack of security guarantees for cryptographic properties (e.g., authentication and secrecy) in ProFactory. The future work of integrating existing cryptographic modeling/verification tools into ProFactory can bridge the gap.

Firmware Deployment. We foresee two modes of deployment in IoT firmware. The first one is in-production customization, in which the manufacturers make use of ProFactory to generate secure and correct implementation and customize their networking dialects before shipping the products. The second is post-deployment customization. Through the update interface of a firmware, hardened and customized networking code can be uploaded to deployed products.

Platform Dependence. As discussed in Section 4.5, the code generation is heavily dependent on the underlying platforms because they can have diverse underlying protocol implementation primitives. This degrades the portability of ProFactory. Nevertheless, if we target code generation for a particular kernel, this seems to be an inevitable issue. Adding virtualization layers may potentially mitigate the problem.

Semantic Preservation. Currently, semantic-preservation correctness of ProFactory code translation has not been comprehensively verified. This will be part of our future

work for compiler verification. However, the post-modeling verification still offers security guarantees.

4.9 Related Work

Protocol Modeling Lots of tools towards secure parser generation have been proposed in recent years [157], [165]–[169]. In EverParse [165] researchers devise a compiler transforming tag-length-value message formats to low-level F* code that calls a library of parser combinators which are formally verified in F*. In this way, the security of parsers is guaranteed and it proves to be effective in averting existing TLS vulnerabilities. In [157] a USB-specific message DSL is proposed to emit a hardening suite, which is then be integrated into a production kernel to avert USB parsing vulnerabilities. PADS [167], Spicy [166], Hammer [168] and Nail [169] are all message formalization tools that produce robust parsers from customized message specifications, covering common text or binary protocols across different languages. In particular, PADS is more data-oriented, which offers auxiliary tools to convert data in XML and XQuery formats to formalized specifications. However, as aforementioned, those tools largely focus on messages, and some of them are not able to describe non-context-free formats, we hence develop ProFactory to realize comprehensive customization for low-level protocols.

Protocol Verification Verifying implementation correctness is a persistent research effort in protocol security area [160], [164], [180]–[184], [194]–[197]. In ProVerif [197] security protocols are represented by Horn clauses to prove (strong) secrecy, authentication and process equivalence. It is applied in [196] to verify the security of symbolic TLS1.3 models. Tamarin [195] specifies protocol actions taken by agents in different roles, using an expressive DSL based on multiset rewriting rules, to automatically construct proofs for security protocols. It is applied in [180], [181] to perform formal analysis of 5G AKA protocols. AGVI [194] applies iterative deepening to perform cost-constraint searching in order to generate a near-optimal security protocol, with the lowest cost, satisfying all the security requirements that are encoded in a DSL. In [164] authors resolves the TLS composite state machine issue (unexpectedly accepting invalid handshakes due to state machine fusion) by writing a secure

TLS implementation that is verified by Frama-C. In [160] researchers leverage adversarial testing technique to disclose an authentication vulnerability in 4G LTE. Those techniques are targeting security protocols that are considered orthogonal and complementary to ProFactory. In [182]–[184] various components of TCP implementation are verified through different symbolic modeling techniques, however, ProFactory aims to resolve protocol security issues at the beginning, generating secure protocol implementations.

Protocol Reverse Engineering and Fuzzing Reverse-engineering protocol specifications from network traces and/or program execution has been well studied in the past decade [2], [151]–[154], [189], [198]–[205]. In Discoverer [199] and ScriptGen [201], protocol communication pattern is heuristically learned to infer message formats. Hence, in [189], [200], researchers improve the learning by performing message clustering based on protocol context and semantic information. In [151]–[153], [202], researchers extract message formats in a different direction, leveraging dynamic tainting to monitor how a message is processed on the receiver side. In Prospex [154], apart from message formats, an approximate but meaningful state machine can also be extracted based on an augmented prefix tree acceptor. These efforts are complimentary to ProFactory as the reverse engineered protocol specification can be formalized with our DSL. Protocol fuzzing [155], [162], [163], [206] mutates network messages and network states to disclose vulnerabilities in protocols. They often require protocol specifications to operate. Our DSL provides a way to formulate such specifications.

5. CONCLUSION

Comprehensive networking security is a critical issue in contemporary application scenarios for enterprise environment. Due to the developing complexity of application technologies and the increasing diversity of peripheral devices, it becomes more and more challenging to achieve a satisfactory accuracy in detecting, investigating, and preventing cyberattacks. In network traffic forensics, evolving applications emit intricate networking flows, which calls for new analysis techniques to improve traffic attribution accuracy. In end-host system provenance, recently disclosed APT attacks are more persistent, stealthy and sophisticated, which motivates effective and efficient methods to perform fine-grained attack investigation. In IoT applications, the majority of vulnerabilities are caused by protocol engineering glitches, which desires code generation tools to obtain secure protocol implementations. In this dissertation, I propose new techniques and tools to assist fine-grained forensic analysis and generating secure protocol implementations.

In particular, we develop NetCrop, a novel automaton-based technique to infer fine-grained program behavior by analyzing network traffic only. It constructs automata that describe both the network behavior and the end-host behavior of a whole program to attribute individual packets to their belonging programs and fingerprint the high-level program behavior. Our evaluation results show that NetCrop is highly effective, attributing packets and fingerprinting program behavior with over 90% and 95% precision and recall respectively.

Also, we found that existing provenance tools cannot catch library attacks because library loading does not promise execution. Hence, we propose Lprov, a novel provenance-oriented library tracing system which enforces library tracing on top of existing syscall logging based provenance tracking approaches. It is lightweight and efficient, offering much better support for heavy-threaded programs than existing tools such as *ltrace*. With the dynamic library call stack, the provenance of implicit library function execution is revealed and correlated to system events. The fine-grained provenance on library functions facilitates the locating and defense of malicious libraries (e.g. Linux Ebury). In experiments, our system prototype can precisely identify the provenance inside malicious libraries with highly competitive overhead.

Last, we propose ProFactory, in which a protocol could be modeled, checked and securely generated, averting common vulnerabilities residing in protocol implementations. Meanwhile, it can also realize implementation diversity. We leverage ProFactory to generate Bluetooth and Zigbee protocols and the evaluation shows that ProFactory can help to avert 82 known CVEs.

REFERENCES

[1] Y. Kwon, F. Wang, W. Wang, K. H. Lee, W.-C. Lee, S. Ma, X. Zhang, D. Xu, S. Jha, G. F. Ciocarlie, A. Gehani, and V. Yegneswaran, "MCI : Modeling-based causality inference in audit logging for attack investigation," in *Proc. of NDSS '18*, 2018.

[2] Y. Ye, Z. Zhang, F. Wang, X. Zhang, and D. Xu, "Netplier: Probabilistic network protocol reverse engineering from message traces," in *NDSS'21*, The Internet Society, 2021.

[3] S. Ma, J. Zhai, F. Wang, K. H. Lee, X. Zhang, and D. Xu, "MPI: Multiple perspective attack investigation with semantic aware execution partitioning," in *USENIX Security'17*, 2017.

[4] K. Pei, Z. Gu, B. Saltaformaggio, S. Ma, F. Wang, Z. Zhang, L. Si, X. Zhang, and D. Xu, "HERCULE: attack story reconstruction via community discovery on correlated log graph," in *Proc. of ACSAC '16*, 2016.

[5] F. Wang, Y. Kwon, S. Ma, X. Zhang, and D. Xu, "Lprov: Practical library-aware provenance tracing," in *Proc. of ACSAC '18*, 2018.

[6] F. Wang, J. Wu, Y. Nan, Y. Aafer, X. Zhang, D. Xu, and M. Payer, "Profactory: Improving iot security via formalized protocol customization," in *Proc. of USENIX Security '22*, 2022.

[7] A. W. Moore and D. Zuev, "Internet traffic classification using bayesian analysis techniques," in *Proc. SIGMETRICS'05*, 2005.

[8] *Cisco guide*, https://goo.gl/Dk6JQz, 2019.

[9] *Cisco netflow case study*, https://goo.gl/8c9H5j, 2019.

[10] T. Nguyen and G. Armitage, "A survey of techniques for internet traffic classification using machine learning," *Communications Surveys Tutorials, IEEE*, vol. 10, no. 4, pp. 56–76, 2008.

[11] J. Ma, K. Levchenko, C. Kreibich, S. Savage, and G. M. Voelker, "Unexpected means of protocol inference," in *Proc. IMC'06*, 2006.

[12] L. Bernaille, R. Teixeira, and K. Salamatian, "Early application identification," in *Proc. CoNEXT'06*, 2006.

[13] T. Karagiannis, K. Papagiannaki, and M. Faloutsos, "Blinc: Multilevel traffic classification in the dark," in *Proc. SIGCOMM'05*, 2005.

[14] B. Rahbarinia, R. Perdisci, A. Lanzi, and K. Li, "Peerrush: Mining for unwanted P2P traffic," in *Proc. DIMVA'13*, 2013.

[15] D. Herrmann, C. Banse, and H. Federrath, "Behavior-based tracking: Exploiting characteristic patterns in dns traffic," Nov. 2013.

[16] *Linux audit*, https://goo.gl/Gyekms, 2019.

[17] *Event tracing for windows*, https://goo.gl/AYG25V, 2019.

[18] M. Conti, L. V. Mancini, R. Spolaor, and N. V. Verde, "Analyzing android encrypted network traffic to identify user actions," *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 1, pp. 114–125, 2016.

[19] B. Saltaformaggio, H. Choi, K. Johnson, Y. Kwon, Q. Zhang, X. Zhang, D. Xu, and J. Qian, "Eavesdropping on fine-grained user activities within smartphone apps over encrypted network traffic," in *Proc. WOOT'16*, 2016.

[20] E. M. Gold, "Complexity of automaton identification from given data," *Inform. Control*, vol. 37, pp. 302–320, 1978.

[21] R. L. Rivest and R. E. Schapire, "Diversity-based inference of finite automata," in *Foundations of Computer Science, 1987., 28th Annual Symposium on*, 1987.

[22] R. L. Rivest and R. E. Schapire, "Inference of finite automata using homing sequences," in *Proc. STOC'89*, 1989.

[23] C. L. Giles, C. B. Miller, D. Chen, H. H. Chen, G. Z. Sun, and Y. C. Lee, "Learning and extracting finite state automata with second-order recurrent neural networks," *Neural Comput.*, vol. 4, no. 3, 1992.

[24] J. Antunes, N. Neves, and P. Verissimo, "Reverse engineering of protocols from network traces," in *Proc. WCRE'11*, 2011.

[25] G. Bossert, F. Guihéry, and G. Hiet, "Towards automated protocol reverse engineering using semantic information," in *Proc. CCS'14*, 2014.

[26] C. Leita, M. Dacier, and F. Massicotte, "Automatic handling of protocol dependencies and reaction to 0-day attacks with scriptgen based honeypots," in *Proc. RAID*, 2006.

[27] T. Krueger, H. Gascon, N. Krämer, and K. Rieck, "Learning stateful models for network honeypots," in *Proc. AISec'12*, 2012.

[28] W. Cui, J. Kannan, and H. J. Wang, "Discoverer: Automatic protocol reverse engineering from network traces," in *Proc. USENIX Security'07*, 2007.

[29] Z. Lin, X. Jiang, D. Xu, and X. Zhang, "Automatic protocol format reverse engineering through context-aware monitored execution," in *Proc. NDSS'08*, 2008.

[30] W. Cui, V. Paxson, N. Weaver, and R. H. Katz, "Protocol-independent adaptive replay of application dialog," in *Proc. NDSS'06*, 2006.

[31] P. M. Comparetti, G. Wondracek, C. Kruegel, and E. Kirda, "Prospex: Protocol specification extraction," in *Proc. IEEE S&P'09*, 2009.

[32] H. Zhang, D. (Yao, and N. Ramakrishnan, "Detection of stealthy malware activities with traffic causality and scalable triggering relation discovery," in *Proc. ASIACCS'14*, 2014.

[33] Y. Kwon, F. Peng, D. Kim, K. Kim, X. Zhang, D. Xu, V. Yegneswaran, and J. Qian, "P2c: Understanding output data files via on-the-fly transformation from producer to consumer executions," in *Proc. NDSS'15*, 2015.

[34] S. Jero, H. Lee, and C. Nita-Rotaru, "Leveraging state information for automated attack discovery in transport protocol implementations," in *Proc. DSN'15*, 2015.

[35] W. Zhang, Y. Meng, Y. Liu, X. Zhang, Y. Zhang, and H. Zhu, "Homonit: Monitoring smart home apps from encrypted traffic," in *Proc. CCS'18*, 2018.

[36] R. Shi and Y. Wang, "Cheap and available state machine replication," in *Proc. of USENIX ATC '16*, 2016.

[37] F. C. Freiling and S. Schinzel, "Detecting hidden storage side channel vulnerabilities in networked applications," in *Proc. SEC'11*, 2011.

[38] K. Xu, K. Tian, D. Yao, and B. G. Ryder, "A sharper sense of self: Probabilistic reasoning of program behaviors for anomaly detection with context sensitivity," in *Proc. DSN'16*, 2016.

[39] X. Shu, D. Yao, and N. Ramakrishnan, "Unearthing stealthy program attacks buried in extremely long execution paths," in *Proc. CCS'15*, 2015.

[40] *Autoit*, https://www.autoitscript.com/site/, 2019.

[41] *Ghostmouse - ghost mouse recorder*, http://www.ghost-mouse.com/, 2019.

[42] *Reach-def analysis*, https://en.wikipedia.org/wiki/Reaching_definition, 2019.

[43] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *Proc. S&P'10*, 2010.

[44] *Constant propagation*, https://en.wikipedia.org/wiki/Constant_folding, 2019.

[45] *Rfc 2782*, https://www.ietf.org/rfc/rfc2782.txt, 2019.

[46] *Rfc 1035*, https://www.ietf.org/rfc/rfc1035.txt, 2019.

[47] K.-s. Huang, C.-b. Yang, and K.-t. Tseng, *Fast algorithms for finding the common subsequence of multiple sequences*, 2014.

[48] *Tie: Traffic identification engine*, http://tie.comics.unina.it/, 2019.

[49] W. Zielonka, "Notes on finite asynchronous automata," *ITA*, vol. 21, no. 2, 1987.

[50] N. Klarlund, M. Mukund, and M. A. Sohoni, "Determinizing asynchronous automata," in *Proc. ICALP'94*, 1994.

[51] *Intel pin*, https://goo.gl/GBchfb, 2019.

[52] G. Hunt and D. Brubacher, "Detours: Binary interception of win32 functions," in *Proceedings of the 3rd Conference on USENIX Windows NT Symposium - Volume 3*, 1999.

[53] *Windump*, https://www.winpcap.org/windump/, 2019.

[54] *Dpkt 1.8.7: Python package index*, https://pypi.python.org/pypi/dpkt, 2019.

[55] *Ad blocking*, https://pgl.yoyo.org/as/, 2019.

[56] S. Baset and H. Schulzrinne, "An analysis of the skype peer-to-peer internet telephony protocol," in *Proc. IEEE INFOCOM'06*, 2006.

[57] Y. Xu, C. Yu, J. Li, and Y. Liu, "Video telephony for end-consumers: Measurement study of google+, ichat, and skype," in *Proc. IMC'12*, 2012.

[58] *Weka 3: Data mining in java*, http://www.cs.waikato.ac.nz/ml/weka/, 2019.

[59] K. Borders and A. Prakash, "Web tap: Detecting covert web traffic," in *Proc. CCS'04*, 2004.

[60] M. Liberatore and B. N. Levine, "Inferring the source of encrypted http connections," in *Proc. CCS'06*, 2006.

[61] R. Perdisci, D. Ariu, and G. Giacinto, "Scalable fine-grained behavioral clustering of http-based malware," *Computer Networks*, vol. 57, no. 2, pp. 487–500, 2013.

[62] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. OSDI'08*, 2018.

[63] G. Li, I. Ghosh, and S. P. Rajan, "Klover: A symbolic execution and automatic test generation tool for c++ programs," in *Proc. CAV'11*, 2011.

[64] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," in *Proc. PLDI'05*, 2005.

[65] M. Roughan, S. Sen, O. Spatscheck, and N. Duffield, "Class-of-service mapping for qos: A statistical signature-based approach to ip traffic classification," in *Proc. IMC'04*, 2004.

[66] C. V. Wright, F. Monrose, and G. M. Masson, "On inferring application protocol behaviors in encrypted network traffic," *J. Mach. Learn. Res.*, vol. 7, 2006.

[67] G. Xie, M. Iliofotou, R. Keralapura, M. Faloutsos, and A. Nucci, "Subflow: Towards practical flow-level traffic classification," in *Proc. IEEE INFOCOM'12*, 2012.

[68] A. McGregor, M. Hall, P. Lorier, and J. Brunskill, "Flow clustering using machine learning techniques," in *Proc. PAM'04*, 2004.

[69] M. Pietrzyk, J.-L. Costeux, G. Urvoy-Keller, and T. En-Najjary, "Challenging statistical classification for operational usage: The adsl case," in *Proc. IMC'09*, 2009.

[70] H. Kim, K. Claffy, M. Fomenkov, D. Barman, M. Faloutsos, and K. Lee, "Internet traffic classification demystified: Myths, caveats, and the best practices," in *Proc. CoNEXT'08*, 2008.

[71] J. Erman, A. Mahanti, M. F. Arlitt, I. Cohen, and C. L. Williamson, "Semi-supervised network traffic classification," in *Proc. SIGMETRICS'07*, 2007.

[72] W. D. Donato, A. Pescape, and A. Dainotti, "Traffic identification engine: An open platform for traffic classification," *IEEE Network*, vol. 28, no. 2, pp. 56–64, 2014.

[73] K. Xu, Z. Zhang, and S. Bhattacharyya, "Profiling internet backbone traffic: Behavior models and applications," in *Proc. SIGCOMM'05*, 2005.

[74] A. Dainotti, A. Pescape, and K. Claffy, "Issues and future directions in traffic classification," *Netwrk. Mag. of Global Internet.*, vol. 26, no. 1, 2012.

[75] K. H. Lee, X. Zhang, and D. Xu, "High accuracy attack provenance via binary-based execution partition," in *NDSS'13*, 2013.

[76] S. Ma, X. Zhang, and D. Xu, "Protracer: Towards practical provenance tracing by alternating between logging and tainting," in *NDSS'16*, 2016.

[77] Y. Ji, S. Lee, E. Downing, W. Wang, M. Fazzini, T. Kim, A. Orso, and W. Lee, "Rain: Refinable attack investigation with on-demand inter-process information flow tracking," in *CCS'17*, 2017.

[78] A. Goel, K. Po, K. Farhadi, Z. Li, and E. de Lara, "The taser intrusion recovery system," in *SOSP'05*, 2005.

[79] X. Jiang, A. Walters, D. Xu, E. H. Spafford, F. P. Buchholz, and Y.-M. Wang, "Provenance-aware tracing of worm break-in and contaminations: A process coloring approach," in *ICDCS'06*, 2006.

[80] S. T. King and P. M. Chen, "Backtracking intrusions," in *SOSP'03*, 2003.

[81] S. Ma, K. H. Lee, C. H. Kim, J. Rhee, X. Zhang, and D. Xu, "Accurate, low cost and instrumentation-free security audit logging for windows," in *ACSAC'15*, 2015.

[82] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. Seltzer, "Provenance-aware storage systems," in *USENIX ATC'06*, 2006.

[83] D. J. Pohly, S. McLaughlin, P. McDaniel, and K. Butler, "Hi-fi: Collecting high-fidelity whole-system provenance," in *ACSAC'12*, 2012.

[84] S. Sitaraman and S. Venkatesan, "Forensic analysis of file system intrusions using improved backtracking," in *IWIA'05*, 2005.

[85] Y. Ji, S. Lee, and W. Lee, "Recprov: Towards provenance-aware user space record and replay," in *IPAW'16*, 2016.

[86] T. Kim, X. Wang, N. Zeldovich, and M. F. Kaashoek, "Intrusion recovery using selective re-execution," in *OSDI'10*, 2010.

[87] S. T. King, Z. M. Mao, D. G. Lucchetti, and P. M. Chen, "Enriching intrusion alerts through multi-host causality," in *NDSS'05*, 2005.

[88] E. Bosman, A. Slowinska, and H. Bos, "Minemu: The world's fastest taint tracker," in *RAID'11*, 2011.

[89] J. Clause, W. Li, and A. Orso, "Dytan: A generic dynamic taint analysis framework," in *ISSTA'07*, 2007.

[90] M. G. Kang, S. McCamant, P. Poosankam, and D. Song, "DTA++: dynamic taint analysis with targeted control-flow propagation," in *NDSS'11*, 2011.

[91] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis, "Libdft: Practical dynamic data flow tracking for commodity systems," in *VEE'12*, 2012.

[92] S. McCamant and M. D. Ernst, "Quantitative information flow as network flow capacity," in *PLDI'08*, 2008.

[93] F. Qin, C. Wang, Z. Li, H.-s. Kim, Y. Zhou, and Y. Wu, "Lift: A low-overhead practical information flow tracking system for detecting security attacks," in *MICRO'06*, 2006.

[94] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "Bitblaze: A new approach to computer security via binary analysis," in *ICISS'08*, 2008.

[95] M. Zhang, X. Zhang, X. Zhang, and S. Prabhakar, "Tracing lineage beyond relational operators," in *VLDB'07*, 2007.

[96] J. Newsome and D. X. Song, "Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software," in *NDSS'05*, 2005.

[97] K. H. Lee, X. Zhang, and D. Xu, "Loggc: Garbage collecting audit log," in *CCS'13*, 2013.

[98] C. Yang, G. Yang, A. Gehani, V. Yegneswaran, D. Tariq, and G. Gu, "Using provenance patterns to vet sensitive behaviors in android apps," in *Security and Privacy in Communication Networks'15*, B. Thuraisingham, X. Wang, and V. Yegneswaran, Eds., 2015.

[99] D. Tariq, M. Ali, and A. Gehani, "Towards automated collection of application-level data provenance," in *TaPP'12*, 2012.

[100] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier, "Using magpie for request extraction and workload modelling," in *OSDI'04*, 2004.

[101] *Linux ebury*, https://goo.gl/T677bv, 2018.

[102] *Apache benchmark*, https://httpd.apache.org/docs/2.2/programs/ab.html, 2018.

[103] *Ftpbench*, https://github.com/selectel/ftpbench, 2018.

[104] *Sunspider*, https://webkit.org/perf/sunspider/sunspider.html, 2018.

[105] D. Bigelow, T. Hobson, R. Rudd, W. Streilein, and H. Okhravi, "Timely rerandomization for mitigating memory disclosures," in *CCS'15*, 2015.

[106] D. Williams-King, G. Gobieski, K. Williams-King, J. P. Blake, X. Yuan, P. Colp, M. Zheng, V. P. Kemerlis, J. Yang, and W. Aiello, "Shuffler: Fast and deployable continuous code re-randomization," in *OSDI'16*, 2016.

[107] W. Hawkins, A. Nguyen-Tuong, J. D. Hiser, M. Co, and J. W. Davidson, "Mixr: Flexible runtime rerandomization for binaries," in *Proceedings of the 2017 Workshop on Moving Target Defense*, 2017.

[108] K. Koning, X. Chen, H. Bos, C. Giuffrida, and E. Athanasopoulos, "No need to hide: Protecting safe regions on commodity hardware," in *EuroSys '17*, 2017.

[109] *Linux security module*, https://goo.gl/gW2ykd, 2018.

[110] *Tracepoints*, https://goo.gl/TB6cas, 2018.

[111] *Kprobes*, https://goo.gl/SH2s4r, 2018.

[112] *An introduction to kprobes*, https://lwn.net/Articles/132196/, 2018.

[113] *Updating host keys*, https://goo.gl/ztY8QT, 2018.

[114] *Cve-2015-7547*, https://goo.gl/MTpo3V, 2018.

[115] *Cve-2015-7547 google security blog*, https://goo.gl/rHw1C5, 2018.

[116] *Cve-2015-7547 exploit-db*, https://www.exploit-db.com/exploits/39454/, 2018.

[117] *Cve-2015-7547 patch*, https://goo.gl/6ZhooX, 2018.

[118] *Darpa transparent computing*, https://goo.gl/EA77zv, 2018.

[119] Z. Deng, D. Xu, X. Zhang, and X. Jiang, "Introlib: Efficient and transparent library call introspection for malware forensics," in *DFRWS'12*, 2012.

[120] U. Braun, S. Garfinkel, D. A. Holland, K.-K. Muniswamy-Reddy, and M. I. Seltzer, "Issues in automatic provenance collection," in *IPAW'06*, 2006.

[121] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum, "Understanding data lifetime via whole system simulation," in *SSYM'04*, 2004.

[122] A. Gehani and D. Tariq, "Spade: Support for provenance auditing in distributed environments," in *Middleware'12*, 2012.

[123] N. Zhu and T.-C. Chiueh, "Design, implementation, and evaluation of repairable file service," in *DSN'13*, 2013.

[124] P. Ammann, S. Jajodia, and P. Liu, "Recovery from malicious transactions," *IEEE Trans. on Knowl. and Data Eng.*, vol. 14, no. 5, 2002.

[125] S. Krishnan, K. Z. Snow, and F. Monrose, "Trail of bytes: Efficient support for forensic analysis," in *CCS'10*, 2010.

[126] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu, "Behavior based software theft detection," in *CCS'09*, 2009.

[127] M. Fredrikson, M. Christodorescu, J. T. Giffin, and S. Jha, "A declarative framework for intrusion analysis," in *Cyber Situational Awareness - Issues and Research*, 2010, pp. 179–200.

[128] K.-K. Muniswamy-Reddy, U. Braun, D. A. Holland, P. Macko, D. Maclean, D. Margo, M. Seltzer, and R. Smogor, "Layering in provenance systems," in *USENIX ATC'09*, 2009.

[129] B. C. Tak, C. Tang, C. Zhang, S. Govindan, B. Urgaonkar, and R. N. Chang, "Vpath: Precise discovery of request processing paths from black-box observations of thread and network activities," in *USENIX ATC'09*, 2009.

[130] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: Capturing system-wide information flow for malware detection and analysis," in *CCS'07*, 2007.

[131] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris, "Labels and event processes in the asbestos operating system," in *SOSP'05*, 2005.

[132] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. D. McDaniel, and A. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *OSDI'10*, 2010.

[133] K. Jee, G. Portokalidis, V. P. Kemerlis, S. Ghosh, D. I. August, and A. D. Keromytis, "A general approach for efficiently accelerating software-based dynamic data flow tracking on commodity hardware," in *NDSS'12*, 2012.

[134] K. Xu, H. Xiong, C. Wu, D. Stefan, and D. Yao, "Data-provenance verification for secure hosts," *IEEE Transactions on Dependable and Secure Computing*, vol. 9, no. 2, 2012.

[135] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières, "Making information flow explicit in histar," in *OSDI'06*, 2006.

[136] H. Zhang, D. D. Yao, and N. Ramakrishnan, "Detection of stealthy malware activities with traffic causality and scalable triggering relation discovery," in *ASIA CCS'14*, 2014.

[137] Y. Kwon, D. Kim, W. N. Sumner, K. Kim, B. Saltaformaggio, X. Zhang, and D. Xu, "Ldx: Causality inference by lightweight dual execution," in *ASPLOS '16*, 2016.

[138] M. Jakobsson and A. Juels, "Server-side detection of malware infection," in *NSPW'09*, 2009.

[139] A. Vasudevan, N. Qu, and A. Perrig, "Xtrec: Secure real-time execution trace recording on commodity platforms," in *HICSS'11*, 2011.

[140] A. Bates, D. (Tian, K. R. Butler, and T. Moyer, "Trustworthy whole-system provenance for the linux kernel," in *USENIX Security'15)*, 2015.

[141] V. Karande, E. Bauman, Z. Lin, and L. Khan, "Sgx-log: Securing system logs with sgx," in *ASIA CCS'17*, 2017.

[142] *Statista IoT*, https://goo.gl/mqt3T7, 2018.

[143] *Forbes Roundup of IoT Market*, https://goo.gl/UPpmkn, 2018.

[144] W. Albazrqaoe, J. Huang, and G. Xing, "Practical bluetooth traffic sniffing: Systems and privacy implications," in *Proc. of MobiSys'16*, 2016.

[145] A. A. Levy, J. Hong, L. Riliskis, P. Levis, and K. Winstein, "Beetle: Flexible communication for bluetooth low energy," in *Proc. of MobiSys'16*, 2016.

[146] *Bluetooth Market Report*, https://bit.ly/3fRXkAv, 2020.

[147] *Zigbee Market*, https://bit.ly/3kdNmvr, 2019.

[148] B. Seri and G. Vishnepolsky, *BlueBorne*, https://armis.com/blueborne/, 2017.

[149] F. Xu, W. Diao, Z. Li, J. Chen, and K. Zhang, "Badbluetooth: Breaking android security mechanisms via malicious bluetooth peripherals," in *Proc. of NDSS'19*, 2019.

[150] *SIG Bluetooth Specifications*, https://bit.ly/2Vw9Y1Q, 2019.

[151] J. Caballero, H. Yin, Z. Liang, and D. Song, "Polyglot: Automatic extraction of protocol message format using dynamic binary analysis," in *Proc. of CCS'07*, 2007.

[152] Z. Lin, X. Jiang, D. Xu, and X. Zhang, "Automatic protocol format reverse engineering through context-aware monitored execution," in *Proc. of NDSS'08*, 2008.

[153] G. Wondracek, P. M. Comparetti, C. Krügel, and E. Kirda, "Automatic network protocol analysis," in *Proc. of NDSS'08*, 2008.

[154] P. M. Comparetti, G. Wondracek, C. Kruegel, and E. Kirda, "Prospex: Protocol specification extraction," in *Proc. of IEEE S&P'09*, 2009.

[155] O. Udrea and C. Lumezanu, "Rule-based static analysis of network protocol implementations," in *Proc. of USENIX Security'06*, 2006.

[156] D. Brumley, J. Caballero, Z. Liang, J. Newsome, and D. Song, "Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation," in *Proc. of USENIX Security'07*, 2007.

[157] P. C. Johnson, S. Bratus, and S. W. Smith, "Protecting against malicious bits on the wire: Automatically generating a usb protocol parser for a production kernel," in *Proc. of ACSAC'17*, 2017.

[158] J. A. Crain and S. Bratus, "Bolt-on security extensions for industrial control system protocols: A case study of dnp3 sav5," *IEEE Security Privacy*, vol. 13, no. 3, pp. 74–79, 2015.

[159] M. Vanhoef and F. Piessens, "Key reinstallation attacks: Forcing nonce reuse in wpa2," in *Proc. of CCS'17*, 2017.

[160] S. R. Hussain, O. Chowdhury, S. Mehnaz, and E. Bertino, "Lteinspector: A systematic approach for adversarial testing of 4g LTE," in *Proc. of NDSS'18*, 2018.

[161] J. Chen, C. Zuo, W. Diao, S. Dong, Q. Zhao, M. Sun, Z. Lin, Y. Zhang, and K. Zhang, "Your iots are (not) mine: On the remote binding between iot devices and users," in *Proc. of DSN'19*.

[162] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, and G. Vigna, "Snooze: Toward a stateful network protocol fuzzer," in *Proc. of ISC'06*, 2006.

[163] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang, and K. Zhang, "Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing," in *Proc. of NDSS'18*, 2018.

[164] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P. Strub, and J. K. Zinzindohoue, "A messy state of the union: Taming the composite state machines of tls," in *Proc. of IEEE S&P'15*, 2015.

[165] T. Ramananandro, A. Delignat-Lavaud, C. Fournet, N. Swamy, T. Chajed, N. Kobeissi, and J. Protzenko, "Everparse: Verified secure zero-copy parsers for authenticated message formats," in *USENIX Security'19*, 2019.

[166] R. Sommer, J. Amann, and S. Hall, "Spicy: A unified deep packet inspection framework for safely dissecting all your data," in *ACSAC'16*, 2016.

[167] K. Fisher and D. Walker, "The pads project: An overview," in *ICDT'11*, 2011.

[168] *Hammer*, https://bit.ly/3s3bMfI, 2019.

[169] J. Bangert and N. Zeldovich, "Nail: A practical tool for parsing and generating data formats," in *OSDI'14*, 2014.

[170] *Protobuf*, https://github.com/protocolbuffers, 2021.

[171] A. Agarwal, M. Slee, and M. Kwiatkowski, "Thrift: Scalable cross-language services implementation," Tech. Rep., 2007.

[172] M. Moskal, W. Schulte, E. Cohen, and S. Tobies, "A practical verification methodology for concurrent programs," Tech. Rep. MSR-TR-2009-2019, 2009. [Online]. Available: https://bit.ly/2ksmj5G.

[173] *VCC: A Verifier for Concurrent C*, https://bit.ly/2m4fCHt, 2008.

[174] *Frama-C*, https://frama-c.com, 2021.

[175] *CVE-2017-1000251*, https://bit.ly/2xymD8a, 2017.

[176] *CVE-2017-1000251 Patch*, https://bit.ly/2WKLkqw, 2017.

[177] *Bluetooth Core Specification*, https://bit.ly/2YwR4VD, 2020.

[178] K. Fawaz, K.-H. Kim, and K. G. Shin, "Protecting privacy of BLE device users," in *Proc. of USENIX Security'16*, 2016.

[179] L. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds., 2008.

[180] C. Cremers and M. Dehnel-Wild, "Component-based formal analysis of 5g-aka: Channel assumptions and session confusion," in *Proc. of NDSS'19*, 2019.

[181] D. Basin, J. Dreier, L. Hirschi, S. Radomirovic, R. Sasse, and V. Stettler, "A formal analysis of 5g authentication," in *Proc. of CCS'18*, 2018.

[182] M. Musuvathi and D. R. Engler, "Model checking large network protocol implementations," in *Proc. of NSDI'04*, 2004.

[183] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough, "Engineering with logic: Hol specification and symbolic-evaluation testing for tcp implementations," in *Proc. of POPL'06*, 2006.

[184] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough, "Rigorous specification and conformance testing techniques for network protocols, as applied to tcp, udp, and sockets," in *Proc. of SIGCOMM'05*, 2005.

[185] *SMT-LIB*, http://smtlib.cs.uiowa.edu/, 2021.

[186] N. Bjørner, L. de Moura, L. Nachmanson, and C. Wintersteiger, *Programming Z3*, https://stanford.io/2GS004D, 2017.

[187] *Zigbee*, https://bit.ly/3lGtrp1, 2011.

[188] *Bluedroid*, https://bit.ly/2JUPydq, 2015.

[189] G. Bossert, F. Guihéry, and G. Hiet, "Towards automated protocol reverse engineering using semantic information," in *Proc. AsiaCCS'14*, 2014.

[190] W. Zhou, Y. Jia, Y. Yao, L. Zhu, L. Guan, Y. Mao, P. Liu, and Y. Zhang, "Discovering and understanding the security hazards in the interactions between iot devices, mobile apps, and clouds on smart home platforms," in *USENIX Security'19*, 2019.

[191] *CVE-2017-1000250 Patch*, https://bit.ly/2FR7WzN, 2017.

[192] *CVE-2017-0783*, https://bit.ly/2UmQ3Rn, 2017.

[193] *CVE-2017-0783 Patch*, https://bit.ly/2YH8Shr, 2017.

[194] D. Song, A. Perrig, and D. Phan, "Agvi — automatic generation, verification, and implementation of security protocols," in *CAV'01*, G. Berry, H. Comon, and A. Finkel, Eds., 2001.

[195] S. Meier, B. Schmidt, C. Cremers, and D. Basin, "The tamarin prover for the symbolic analysis of security protocols," in *CAV'13*, 2013.

[196] K. Bhargavan, B. Blanchet, and N. Kobeissi, "Verified models and reference implementations for the TLS 1.3 standard candidate," in *IEEE S&P'17*, 2017.

[197] B. Blanchet, "Modeling and verifying security protocols with the applied pi calculus and ProVerif," *Foundations and Trends in Privacy and Security*, vol. 1, no. 1–2, pp. 1–135, Oct. 2016.

[198] Z. Wang, X. Jiang, W. Cui, X. Wang, and M. Grace, "Reformat: Automatic reverse engineering of encrypted messages," in *Proc. of ESORICS'09*, 2009.

[199] W. Cui, J. Kannan, and H. J. Wang, "Discoverer: Automatic protocol reverse engineering from network traces," in *Proc. of USENIX Security'07*, 2007.

[200] T. Krueger, N. Krämer, and K. Rieck, "Asap: Automatic semantics-aware analysis of network payloads," in *Proc. of PSDML'10*, 2010.

[201] C. Leita, K. Mermoud, and M. Dacier, "Scriptgen: An automated script generation tool for honeyd," in *Proc. of ACSAC'05*.

[202] W. Cui, M. Peinado, K. Chen, H. J. Wang, and L. Irun-Briz, "Tupni: Automatic reverse engineering of input formats," in *Proc. of CCS'08*, 2008.

[203] J. Newsome, D. Brumley, J. Franklin, and D. Song, "Replayer: Automatic protocol replay by binary analysis," in *Proc. of CCS'06*, 2006.

[204] W. Cui, V. Paxson, N. Weaver, and R. H. Katz, "Protocol-independent adaptive replay of application dialog," in *Proc. of NDSS'06*, 2006.

[205] T. Krueger, H. Gascon, N. Krämer, and K. Rieck, "Learning stateful models for network honeypots," in *Proc. of AISec'12*, 2012.

[206] J. Somorovsky, "Systematic fuzzing and testing of tls libraries," in *Proc. of CCS'16*, 2016.

VITA

Fei Wang 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 2014 to Fall 2021. Before that, he earned his Bachelor of Engineering in Network Engineering from Sichuan University in Chengdu, Sichuan, China, and his Master of Science in Information Security from University of Science and Technology of China in Hefei, Anhui, China. His research interests lie in computer security especially system and software security, protocol security, IoT security and software engineering. His work has been published in ISOC NDSS, USENIX Security and ACSAC. In the Fall of 2021, he will join Facebook to serve as a research scientist.