Graduate School Form 30
Updated 1/15/2015

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Wei Peng

Entitled

ON SEVERAL PROBLEMS REGARDING THE APPLICATION OF OPPORTUNISTIC PROXIMATE LINKS IN SMARTPHONE NETWORKS

For the degree of Doctor of Philosophy

Is approved by the final examining committee:

Xukai Zou
Chair

Ninghui Li

Feng Li

Dongyan Xu

To the best of my knowledge and as understood by the student in the Thesis/Dissertation Agreement, Publication Delay, and Certification Disclaimer (Graduate School Form 32), this thesis/dissertation adheres to the provisions of Purdue University's "Policy of Integrity in Research" and the use of copyright material.

Approved by Major Professor(s): Xukai Zou and Ninghui Li

Approved by: Sunil Prabhakar 04/06/2015
Head of the Departmental Graduate Program Date

ON SEVERAL PROBLEMS REGARDING THE APPLICATION OF OPPORTUNISTIC PROXIMATE LINKS IN SMARTPHONE NETWORKS

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Wei Peng

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

May 2015

Purdue University

West Lafayette, Indiana

To Mom, Dad, and Q: *you* are the why.

ACKNOWLEDGMENTS

Four years have passed since I was last in this position, writing acknowledgments for my Master's thesis. A lot of things have changed during that time, but my gratitude to my mentors, Dr. Feng Li and Dr. Xukai Zou, remain the same. Again, I am grateful for you:

- taking me onboard when I was wandering;
- initiating me into the joys and pains of scientific research;
- putting yourselves in my shoes and supporting me;
- making a pitch for me beyond your duty;
- trusting and encouraging me when I was in doubt;
- not giving up on me;
- and showing me life is, after all, larger than work.

I am grateful for Dr. Ninghui Li and Dr. Dongyan Xu, who kindly serve on my advisory committee. Beyond their exemplary scholarship, they are the most kind, considerate, and accommodating committee members that I could ask for. Thank you, Dr. Li and Dr. Xu.

Numerous kind souls have helped me in various ways along the journey. In particular, I have lost count how many times Nicole and Sheila navigate me through complicated processes, and Scott provides excellent advise on using university IT resources. Thank you.

To my dear wife, Min. I am grateful for your love, patience, support, and all the happiness that you bestow on me. Your companionship is what makes the past few years so wonderful; I am looking forward to our journey ahead, *together*.

To my soon-to-be-born baby: Mom and dad love you; be good.

TABLE OF CONTENTS

Page

Page

Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

Figure Page

Figure Page

ABSTRACT

Peng, Wei Ph.D., Purdue University, May 2015. On Several Problems Regarding the Application of Opportunistic Proximate Links in Smartphone Networks. Major Professors: Dr. Feng Li, Dr. Xukai Zou, and Dr. Ninghui Li.

A defining characteristic of smartphones is the availability of short-range radio transceivers (the proximate channel) such as Bluetooth, NFC, and Wi-Fi Direct, in addition to traditional long-range cellular telecommunication technologies (the cellular channel). Coupled with smartphones' portability and their human users' mobility, the proximate channel provides opportunistic proximate links as a supplement/alternative to the cellular channel's persistent infrastructural links for data communication.

Opportunistic proximate links have a diverse set of applications, with each application scenario bringing a unique set of often conflicting objectives to balance. This dissertation presents a study on several problems regarding the application of opportunistic proximate links in smartphone networks. The first part of this dissertation, which includes Chapter 2, 3, and 4, focuses on the cost-effective distribution of content using opportunistic proximate links, and examines several applications: 1. Chapter 2 is on the use of opportunistic proximate links in selecting a representative subset from a set of smartphones for prioritized defense deployment in a Bring-Your-Own-Device (BYOD) enterprise network environment. 2. Chapter 3 is on the use of opportunistic proximate links for offloading bounded-delay-tolerant topical content from cellular persistent infrastructural links. 3. Chapter 4 is on the use of opportunistic proximate links in a generalized scenario of content distribution in a smartphone network that is heterogeneous in the availability of cellular persistent infrastructural links.

The second part of this dissertation, which includes Chapter 5 and 6, considers the opposite problem of preventing the distribution of unwanted content (mobile malware) over opportunistic proximate links and the supplementary problem of detecting mobile malware. Chapter 5 considers a probabilistic behavioral malware detection framework for delay-tolerant smartphone networks that are connected by opportunistic proximate links. Solutions to several challenging problems that are unique to decentralized and opportunistic nature of such networks, including “balance between insufficient evidence and evidence collection risk,” “liars,” and “defectors” are proposed and evaluated. Based on the widely used Android mobile computing platform, Chapter 6 presents the design, implementation, and evaluation of a novel declarative approach to static binary analysis of Android apps, which underlies the problem of detecting malware on the Android platform. Real Android malware samples are analyzed, and techniques to robustly handle them are proposed and evaluated.

1 INTRODUCTION

The past two decades witness two significant transformations in the telecommunication industry: 1. the widespread consumer adoption of personal portable cellular phones (along with the telecommunication infrastructures that support them), 2. the evolution of such phones from earlier generations that only provide text-based user interface (UI) and basic voice/text service (with the retronyms “dumb phones” and “feature phones”) to the latest generation of smartphones that provide graphical UI and data-driven multimedia services. Ephemeral market statistics [84] of “estimated worldwide Android-based mobile device shipments reaching 1.4 billion units in 2015” aside, the significant growth of smartphone adoption in the consumer market is reflected by a surge of academic research on the application [183, 184, 25, 116, 152] and security [204, 69, 236, 149, 228, 43] of smartphone platforms.

A defining characteristic of smartphones is their diverse connectivity capabilities. In addition to long-range cellular telecommunication technologies (the cellular channel, e.g., voice channel and 3G/4G/LTE data channel), smartphones are often equipped with short-range radio transceivers (the proximate channel, e.g., Wi-Fi, Bluetooth, NFC [146], and Wi-Fi Direct [210]) with various effective communication distances from direct contact for NFC up to tens of meters in practice for recent Bluetooth and Wi-Fi Direct implementations. Although limited in coverage range comparing with the almost ubiquitous availability of the cellular channel, the proximate channel is free of data surcharges and usually has wider and more consistent bandwidth than the cellular channel. Coupled with smartphones’ portability and their human users’ mobility, the proximate channel provide *opportunistic proximate links* as a supplement/alternative to the cellular channel’s *persistent infrastructural links* for data communication.

Opportunistic proximate links have diverse applications, with each application having a unique set of often conflicting objectives to balance. The main subject of this dissertation is to examine several problems regarding such applications of opportunistic proximate links in smartphone networks.

- Chapter 2, "T-dominance: Prioritized Defense Deployment for BYOD Security" [153], is on the use of opportunistic proximate links in selecting a representative subset from a set of smartphones for prioritized defense deployment in a bring-your-own-device (BYOD) enterprise network environment.

- Chapter 3, "The Virtue of Patience: Offloading Topical Cellular Content through Opportunistic Links" [154], is on the use of opportunistic proximate links in offloading bounded-delay-tolerant topical content from cellular persistent infrastructural links.

- Chapter 4, "Temporal Coverage Based Content Distribution in Heterogeneous Smart Device Networks" [155], is on the use of opportunistic proximate links in a generalized scenario (from the above applications) of content distribution in a smartphone network that is heterogeneous in the availability of cellular persistent infrastructural links.

The common challenges of these applications are:

- The use of opportunistic proximate links is decentralized due to the high costs or unavailability of centralized coordination through cellular persistent infrastructural links.

- Content that needs to be distributed in these applications (e.g., vulnerability patches in Chapter 2 and user-subscribed content in Chapter 3) can often tolerate a bounded amount of delivery delay, in exchange for the reduced delivery cost of using free opportunistic proximate links instead of costly persistent infrastructural links.

Each of the above chapters is dedicated to addressing the manifestation of these challenges in its respective application scenario.

Unlike the aforementioned chapters, which focus on facilitating cost-effective content distribution using opportunistic proximate links, Chapter 5, "Behavioral Malware Detection in Delay Tolerant Networks" [156, 151], considers the opposite problem of preventing the distribution of unwanted content over opportunistic proximate links. A probabilistic behavioral malware detection framework is considered for delay-tolerant smartphone networks that are connected by opportunistic proximate links, and solutions to several challenging problems that are unique to de-centralized and opportunistic nature of such networks, including "balance between insufficient evidence and evidence collection risk," "liars," and "defectors" are proposed and evaluated.

As a supplement to Chapter 5, Chapter 6, "Web of APKs (WoA): A Declarative Approach for Static Android PacKage (APK) Binary Analysis," takes a different angle to addressing the problem of detecting mobile malware. Based on a concrete target—the widely used Android mobile computing platform, Chapter 6 presents the design, implementation, and evaluation of a novel declarative approach to static binary analysis of Android apps, which underlies the problem of detecting malware on the Android platform. Real Android malware samples are analyzed in detail, and techniques to robustly handle them are designed and evaluated.

Each of the following chapters is self-contained and can be read independently.

2 *T*-DOMINANCE: PRIORITIZED DEFENSE DEPLOYMENT FOR BYOD SECURITY

The application of opportunistic proximate links considered in this chapter is prioritized defense deployment in enterprise BYOD smartphones. The essence of this chapter is to use individual smartphones' proximate channel encounter information to distributedly elect, *without central planning or coordination*, a subset of smartphones to *represent* the full set of smartphones. These representative smartphones are prioritized for deploying security mechanisms such as malware scanning or patching, and can then extend the reach of these security mechanisms to the full smartphone network through their opportunistic encounters.

This chapter is previously published as a conference paper [153] in IEEE Conference on Communications and Network Security (CNS), 2013.

2.1 Introduction

Bring Your Own Device (BYOD) is an enterprise information technology (IT) policy that encourages employees to use their own devices to access sensitive corporate data at work through the enterprise IT infrastructure. Employees' demand/satisfaction, decreased IT acquisition and support cost, and increased use of cloud/virtualization technologies in enterprise IT infrastructure are common justifications for adopting BYOD [166]. With the consumerization of smartphones and tablet computers (smartphones for brevity) in recent years, the demand for using personal smartphones in the workplace has brought BYOD to the attention of enterprise IT professionals as one of the "tech trends for 2013 [98]."

Despite the commonly cited benefits, BYOD presents significant security challenges. On the one hand, forwarding corporate e-mails to public Web mail services,

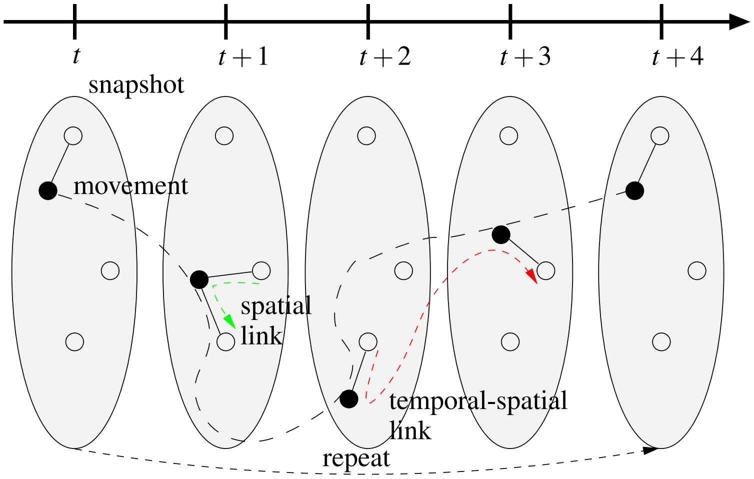


Figure 2.1.: T-dominance exploits temporal-spatial patterns of BYOD devices to implement prioritized defense deployment. The black node T-dominates the white ones for $T > 4$.

using public cloud-based storage services (e.g., Dropbox and Apple's iCloud) to store corporate documents, or even interacting with smartphones through voice in the workplace may leak sensitive corporate information assets [24]; moreover, employees may inadvertently or maliciously introduce malware to the enterprise network behind the firewalls through their own malware-infected smartphones. On the other hand, forcing employees to disable common applications such as Dropbox [24], though may be necessary security-wise, significantly worsen employees' BYOD experience; frequently auditing the use of employees' smartphones not only intrudes on their convenience, but is also costly to implement.

To address such tension, we propose *prioritized defense deployment*: Instead of employing the same costly and intrusive security measures on each BYOD smartphone, more stringent threat detection/mitigation mechanisms are deployed on those representative smartphones, each of which represents, security-wise, a group of smartphones in the whole BYOD device pool.

In this chapter, we interpret and measure security representativeness through the temporal-spatial pattern inherent in an enterprise environment: Those BYOD smartphones that connect with *many* other smartphones *often* are representative security-wise, because they are exposed to more attacks and have more severe consequences if compromised.

More specifically, we interpret and measure security representativeness with a novel temporal-spatial structural property and propose a distributed algorithm (running distributedly on individual smartphones) that robustly preserves that property. We name both the property and the algorithm *T-dominance*, in which T is a temporal bound. Each BYOD smartphone executes the T-dominance algorithm and, based on potentially outdated information from proximate smartphones (as briefly discussed in Section 2.2, such information is readily available on many consumer smartphones), estimates its security representativeness. If a smartphone considers itself as representative, it turns into an *agent*. The algorithm needs no central coordination, which reduces maintenance overhead for enterprise IT administration and is less intrusive to BYOD employees. After running the algorithm for awhile, the whole BYOD smartphone pool will be T-dominated by the agents: Each smartphone is either an agent, or is highly likely to be proximate to an agent with a delay not exceeding T. The idea of T-dominance is illustrated in Figure 2.1. A more intrusive and costly defense mechanism will be deployed on the agents.

Prioritized defense deployment based T-dominance provides an adjustable (through T) balance between security provision and mechanism intrusiveness/cost. We define the concept of T-dominance and present an algorithm to implement it (Section 2.3). We show the temporal robustness and the effectiveness of the proposed algorithm through analysis (Section 2.4) and trace-driven experiments (Section 2.5), and put our works in the context of previous research (Section 2.7).

In summary, we make the following contributions.

- We propose prioritized defense deployment based on security representativeness as a solution to the tension between the demand for BYOD security practices and the intrusiveness/cost of such practices.

- We propose a novel interpretation of security representativeness, based on the inherent temporal-spatial structures in an enterprise environment, and illustrate the application of the concept: strategic sampling to detect malware, prioritized patching to prevent or recover from damage.

- We propose a method, T-dominance, to capture the temporal-spatial dynamics of BYOD smartphone networks in a graph structure (Definition 1) and maintain such a structure with an algorithm that does not incur extra administration cost, and is less intrusive to employees (Section 2.3).

- We show the temporal robustness and the effectiveness of the proposed algorithm through analysis (Section 2.4) and trace-driven experiments (Section 2.5). The temporal robustness ensures that the T-dominance algorithm will maintain the T-dominance structural property on potentially outdated information, due to the absence of constant, central coordination.

2.2 Model

Due to the wide deployment of Wi-Fi infrastructure in enterprise networks and the wide availability of Wi-Fi co-location information on smartphones (to support, for example, location-based services), we consider a threat model that includes, besides the common drive-by download attack, smartphone malware that can infect Wi-Fi co-located smartphones through techniques such as ARP poisoning; we briefly discuss the feasibility and current state of such proximity malware attacks in Section 2.6.

Each smartphone maintains a connectivity log of past access point associations, with entries in the form of $(ST = s, ET = e, APID = \text{AP}_i)$ indicating that the smartphone is associated with access point AP_i from time s to e. Connectivity logging is a standard feature on major mobile platforms, such as the consolidated.db in iOS's location-aware services [1].

Given the connectivity log of a pair of smartphones, u and v, we can find the maximal temporal intervals during which the two smartphones are co-located within the temporal window $[t-W, t]$ of size W[1]: $[s_1, e_1], [s_2, e_2], \ldots, [s_k, e_k]$. Let $s_{k+1} = s_1 + W$; we have $s_1 < e_1 < \ldots < s_i < e_i < \ldots < s_k < e_k \leq s_{k+1} = s_1 + W$.

At a particular moment m ($t - W \leq m \leq t$), the waiting time $g(m)$ before the next encounter between u and v is:

$$g(m) = \begin{cases} 0 & \exists i, \text{s.t. } s_i \leq m \leq e_i, \\ \min_{s_i \geq m}(s_i - m) & \text{otherwise.} \end{cases} \tag{2.1}$$

Thus, we define the expected delay $\mathbf{r}(u, v)$ till next encounter between u and v at time t as their *reachability*, computed by:

$$\mathbf{r}(u, v) = \frac{\int_{s_1}^{s_{k+1}} g(m)\mathrm{d}m}{W} = \frac{\sum_{i=1}^{k}(s_{i+1} - e_i)^2}{2W}. \tag{2.2}$$

As a special case, if the two smartphones are not co-located between $t - W$ and t (reflected by the lack of common intervals in l_1 and l_2 during that temporal window), their reachability is defined to be $+\infty$. The definition of reachability in Equation (2.2) has implications (Lemma 1) on our design (Section 2.4)[2].

Given a set of smartphones $P = \{u, v, w, \ldots\}$ along with their connectivity logs, we define the *reachability graph* $G(P)$ of P to be a weighted undirected graph with P as the vertices and $\mathbf{r}(u, v)$ as the weights on the edges between two smartphones u and v. Given a threshold T, we define the *filtered reachability graph* $G^T(P)$ to be the subgraph of $G(P)$ consisting of all the vertices along with those edges with weights no greater than T.

[1] Temporal window is used in the definition of reachability to phase out old information that may be outdated. An example is that, for an employee who transferred from one department to another two days ago, a temporal window W of 2 days will exclude the information before the transfer when computing reachability.

[2] In Equation (2.2), we use $\int_{e_1}^{s_{k+1}} g(m)\mathrm{d}m$, instead of $\int_{t-W}^{t} g(m)\mathrm{d}m$, as the numerator; effectively, we cut the temporal interval $[t-W, s_1]$ and paste it to the right of $[t-W, t]$; then we take an interval of length W from the right to form the interval $[s_1, s_1 + W]$, i.e., $[s_1, s_{k+1}]$. This ensures the temporal robustness of the reachability metric in Theorem 1 (more specifically, in Lemma 1).

2.3 Design

2.3.1 Motivation: Prioritized Defense Deployment

Threat detection/mitigation in an enterprise network is an ongoing, rather than a one-shot, process. Threat detection/mitigation mechanisms, such as malware detection and vulnerability patching, need to be deployed on BYOD smartphones and regularly updated to defend against evolving and emerging threats. Doing so constantly on all BYOD smartphones is costly for the enterprise, and intrusive to the employees. *Random sampling* is less costly and intrusive, but is oblivious to the temporal diversity of BYOD employees' connectivity patterns and, thus, presents challenges such as how many and how often devices shall be checked for security vulnerabilities and receive updates, as well as how to quantify the security provision.

Prioritized defense deployment addresses these challenges by assigning each BYOD smartphone one of two mutually exclusive roles, *agents* and *non-agents*, according to its security representativeness, and prioritizing the agents for defense mechanism deployment. The use of the neutral terms (agent and non-agent) to differentiate the security representativeness brings forth the essence of such distinction without confining prioritized defense deployment to one narrow scenario. For example, in the context of proximity malware attacks, prioritized defense deployment can support *strategic sampling* for detecting malware, and *prioritized patching* for preventing/recovering from malware attacks.

- In strategic sampling, the agents resemble traditional Internet *honeypots* for intrusion detection [165]: They attract and expose propagating malware. The agents are periodically checked for malware infection by enterprise IT security staff. Prioritized defense deployment will choose those security-wise representative smartphones as agents, and hence, provide a quantifiable security provision for detecting malware.

- In prioritized patching, the agents resemble the high-risk population (prior to their immunization) and vaccine depot (after their immunization) in human epidemiology: They are high-risk target of malware (prior to being patched against the malware) due to their temporal-spatial importance in connecting the network; they are also good deliverers of the security patches (after being patched) for the same reason.

Strategic sampling is reactive, and prioritized patching is proactive: Whereas an agent in the former waits for a co-located smartphone to infect it, an agent in the latter actively distributes patches to co-located smartphones. Nevertheless, in both applications of prioritized defense deployment, a smaller number of agents lowers the sampling/patching cost for enterprise IT management, and reduces intrusiveness to employees; it is, therefore, more desirable.

In this chapter, we propose T-dominance as an approach to implement prioritized defense deployment. In the rest of the section, we define the concept of T-dominance (Section 2.3.2) and design a localized and temporally robust algorithm for electing a T-dominating agent set in a BYOD network for prioritized defense deployment (Section 2.3.3).

2.3.2 T-dominance: The Concept

The concept of T-dominance is defined on the filtered reachability graph $G^T(P)$ (Section 2.2) over a network of smartphones P as follows.

Definition 1 (T-dominance) *Let P be a set of smartphones and A be a subset of P called the* agents. *We say that the agents A T-dominates the smartphones P at moment t if, for any $u \in G^T(P)$, either $u \in A$ or u is a neighbor of an agent $a \in A$ in $G^T(P)$.*

By definition, P trivially T-dominates itself. We are interested in a non-trivial A that T-dominates P. For prioritized defense deployment based on T-dominance, a small A is desirable.

T-dominance quantifies the security provision in both strategic sampling and prioritized patching (Section 2.3.1). For example, consider that a Wi-Fi co-location-based epidemic malware starts to propagate at the moment t.

In strategic sampling, if the agents T-dominate the network, it is highly likely that one of the (T-dominating) agents will co-locate with an infected smartphone, and thus, be infected before $t+T$. Thereafter, the infection will be detected the next time the infected agent is checked. If the periodic check is scheduled at a cycle of T, the epidemic is highly likely to be detected before $t + 2T$, which is controllable by the choice of T. Comparing to both constant monitoring and random sampling, strategic sampling through the T-dominating agents provides control over the trade-off between cost/intrusiveness, in terms of the scale and frequency of the sampling, and the security provision, in terms of the maximal detection delay.

In prioritized patching, when a piece of smartphone malware is detected or a system vulnerability is uncovered, patches for preventing further exploitation can be first issued to the agents at the moment t. The agents will then become immune to this particular threat and will, therefore, slow down the malware's epidemic propagation. Furthermore, the agents can distribute the patches to their co-located smartphones. T-dominance ensures that most BYOD smartphones will receive the patches by $t + T$. Like in strategic sampling, prioritized patching through T-dominating agents provides control over the trade-off between cost/intrusiveness, in terms of the scale and frequency of the initial patching, and the security provision, in terms of maximal patching delay.

2.3.3 T-dominance: The Algorithm

Now we have seen that the T-dominating agents serve a specific role in prioritized defense deployment. In this section, we present a local algorithm that runs on individual smartphones to elect agents without central coordination. The algorithm consists of two decision processes: activation and deactivation. We present deactivation before activation, because activation contains deactivation as a sub-process.

2.3.3.1 Agents vs. Non-agents

Agents and non-agents differ in the amount of auxiliary information they maintain: An agent keeps track of other smartphones it shares co-location opportunities with, while a non-agent does not. The auxiliary information helps smartphones make informed activation/deactivation decisions without central coordination; the differentiation in the amount of maintained auxiliary information reduces prioritized security deployment's overhead for those non-agent smartphones. The auxiliary information maintained by the agents includes co-located smartphones' IDs, agent/non-agent status, and connectivity logs; each record is time-stamped for later consolidation.

When two smartphones are co-located (or, meet) and at least one of them is an agent, the agent will collect information from the other smartphone. When an agent u meets another smartphone v, there are two scenarios.

- When agent u meets non-agent v, since a non-agent only maintains its own auxiliary information, u can only obtain v's own information from v. After the meeting, u's auxiliary information expands to include v.
- When agent u meets another agent v, they share information on other smartphones they directly met with. After the meeting, u's auxiliary information expands to include v and v's direct acquaintance, as illustrated in Figure 2.2.

In both cases, agent u forms a filtered reachability graph $G^T(P)$ from all the phones P within its expanded scope, and takes the largest connected component containing itself, $G_D(u)$, as its *domination graph.* Later operations will be conducted on this domination graph $G_D(u)$.

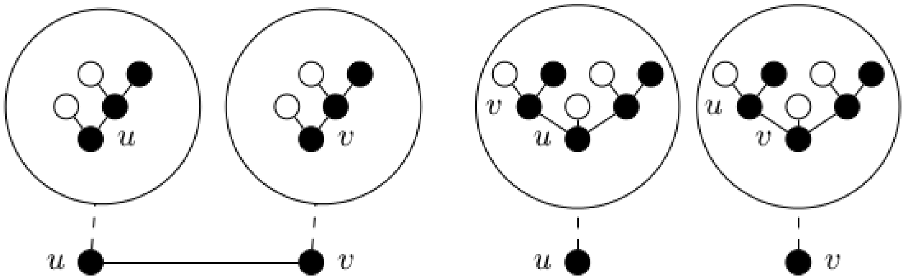


Figure 2.2.: After exchanging auxiliary information during their encounter, agent u's scope expands to include another agent v's direct acquaintance and vice versa.

2.3.3.2 Deactivation

Each agent first collects at least a time window's intelligence before it is eligible for deactivation. When an agent u meets another agent v, and *only after u has been an agent for at least a temporal window W's time*[3], u makes a decision of whether it will deactivate itself: A deactivated agent changes into a non-agent. Deactivation reduces the number of agents and, hence, the overall sampling/patching cost and intrusiveness of the deployed defense mechanism.

u makes its deactivation decision based on its domination graph $G_D(u)$. Let $N(w)$ be the neighbors of a vertex w in $G_D(u)$ and $N[w] = N(w) \cup \{w\}$ be the closed neighbor set of w. Depending on the security context/corporate policy, u may choose to be either aggressive or conservative in deactivating itself. Accordingly, we propose two alternative rules that u can follow to decide whether to deactivate itself:

- **Individual.** u deactivates itself if there exists an agent w with higher priority in $G_D(u)$ so that $N[u] \subseteq N[w]$.
- **Group.** u deactivates itself if there exists a connected set of agents U in $G_D(u)$, *each* of which has a higher priority than u, so that $N[u] \subseteq \bigcup_{w \in U} N[w]$. Such a U is said to be a replacement of u.

[3] The intuitive explanation for this is to let the agent be well informed before making a decision. We provide a technical justification in Section 2.4.

The definition of the two rules implies that agents under the Group rule are more aggressive in deactivation that those under the Individual rule. The two rules provide a trade-off between cost (in terms of number of agents) and responsiveness (in terms of delay between malware infection and detection in strategic sampling, or between patch release and distribution in prioritized patching) in a prioritized defense deployment scheme. The requirement for connectedness in Group is to enlist the bridging nodes in the BYOD smartphone network (such as an inter-departmental courier on a large corporate site) for security defense due to their critical role in connecting the network and, hence, higher chances of being attacked.

To complete the previous rules, u needs to decide whether w has a higher priority than itself. Again, u can be either aggressive or conservative: There are two alternative criteria that u can apply to decide whether w has a higher priority than itself (let $N_\cap = N(u) \cap N(w)$):

- **Strong.** w has a priority higher than u if 1) $N_\cap \neq \emptyset$; 2) $\exists x \in N_\cap, \mathbf{r}(x, w) < \mathbf{r}(x, u)$; 3) $\forall x \in N_\cap, \mathbf{r}(x, w) \leq \mathbf{r}(x, u)$.
- **Weak.** w has higher priority than u if 1) $N_\cap \neq \emptyset$; 2) $\sum_{x \in N_\cap} \mathbf{r}(x, w) < \sum_{x \in N_\cap} \mathbf{r}(x, u)$.

By definition, if an agent decides that one of its peers has a higher priority than itself under the Strong rule, it will reach the same conclusion under the Weak rule. Similar to Individual and Group, Strong and Weak provide a trade-off between cost and responsiveness in a prioritized defense deployment scheme. The absence of equivalence in the second clauses in both criteria is to eliminate the case that a pair of agents (wrongfully) assume that the other party will take over the responsibility for their dominated nodes and, hence, deactivate themselves during the same encounter.

2.3.3.3 Activation

When an agent u meets a non-agent v, u makes the decision of whether it should activate v.

One possible strategy is to activate every co-located non-agent. Given enough contact opportunities, such a strategy leads to an epidemic activation: Every smartphone gets activated at least once, no matter whether it is representative. However, since some of the agents are to be deactivated later, a more discreet strategy is desirable to avoid thrashing, i.e., employees' smartphones get repeatedly activated and deactivated in cycle, which consumes computational and energy resources on the smartphones without much security benefits.

The insight is that a non-agent should be activated *unless it is highly likely to be deactivated later*. Thus, the activation decision process comes down to measuring the likelihood of the non-agent being deactivated *later* if it is activated *now*.

Let us consider how an agent u can decide whether to activate a non-agent v. The activation process consists of two consecutive stages, *deactiviablity* and *coverage*.

Deactiviability. u computes a filtered reachability graph on its scope *along with that of* v, and invokes the deactivation strategy *for* v on the graph (in other words, u assumes v's perspective and decides, if v is to make the deactivation decision, whether v will deactivate itself). We say v is *deactivable* if the result (computed by Agent u) turns out to be that v will deactivate itself.

If v is *not* deactivable, u will activate v and terminate the activation decision process.

Otherwise, if u is deactivable, Agent v will proceed to the next stage.

Coverage. Let $A(u)$ be the set of *agents* that u knows of (including u itself). Agent u estimates v's *unique* coverage contribution to $A(u)$ and activates v with a corresponding probability.

The unique coverage contribution of v to $A(u)$ is those periods of time (within the temporal window) that none of the agents in $A(u)$ covers, but only v does.

Let the total length of v's *unique* coverage be $c(v\backslash A(u))$, and let the total length of $A(u)$'s coverage be $c(A(u))$. u activates v with a probability:

$$1 - \exp(-\frac{c(v\backslash A(u))}{c(A(u))}). \tag{2.3}$$

Thus, the probability is close to 0 if v contributes little unique coverage ($c(v \backslash A(u)) \to 0$) and is close to 1 if v contributes significant unique coverage ($\frac{c(v \backslash A(u))}{c(A(u))} \to \infty$). In other words, the more unique coverage v contributes, the more likely it will be activated by u. The unique temporal coverage contribution of the newly activated agent may help expose malware infection in strategic sampling or deliver patches in prioritized patching.

2.3.3.4 T-dominance-based Prioritized Defense Deployment

By the activation and deactivation processes, a subset of the whole BYOD smartphone pool are elected as agents, and the rest are non-agents. Since the enterprise has much less central control over employees' BYOD devices than traditionally enterprise-issued ones, the T-dominating agent set allows security measures to be prioritized for those security-wise representative devices in order to reduce security mechanisms' cost/intrusiveness under a quantified security provision. For example, during each round of strategic sampling, an agent will have a higher probability of being sampled than a non-agent; similarly, in prioritized patching, when a new vulnerability is found, an agent will have a higher priority of being patched early than a non-agent. Thus, prioritized defense deployment can be formalized as follows.

Prioritized defense deployment. In deploying a repeatedly executed/upgraded defense mechanism, let the *priority* of, or the *probability* of deploying a security mechanism in *one round* on, the agents and non-agents be p and q respectively. *Prioritized defense deployment* is to have $p > q$.

A relatively small q reduces security overheads for those devices that have less contacts with others and, therefore, are less likely to spread or be infected with malware, while at the same time not completely neglecting the security of these relatively reclusive devices.

As a special case, $p = 1$ and $q = 0$: All the agents, and only the agents, are sampled in each round in strategic sampling, or patched (directly by corporate IT security staff) against each new vulnerability in prioritized patching. Suppose there is a bounded maximal rate of sampling or patching (due to technological, economic, or organization-political restriction), T-dominance-based prioritized defense deployment provides an ordering that favors more security-wise representative devices in the sampling/patching request queue. In Section 2.5.2.2, we simulate this scenario over real Wi-Fi association traces.

2.4 Analysis

In this section, we show that the algorithm presented in Section 2.3 satisfies a few desirable properties.

A desirable algorithm should maintain the T-dominance structural property on a BYOD smartphone network, or, in other words, be correct: The effectiveness of strategic sampling and prioritized patching is contingent on the premise that the delay (as estimated by the reachability metric) to reach most smartphones from the agents through co-location is bounded by T.

Property 1 (Correctness) *The T-dominance structural property is maintained by the algorithm.*

If an algorithm that implements T-dominance-based prioritized defense deploy requires employees to forfeit their co-location records for centralized planning, the algorithm will still be costly for the enterprise (due to the collection and central planning) and intrusive to the employees (due to the forfeiture). A distinction of a BYOD enterprise network, in comparison with a traditional enterprise-issued device network, is that it is more costly for the enterprise to provide security support the diverse set of devices and that the employees are more reluctant to intrusive security measures initiated by the enterprise (since, by definition, they belong to the employees). A key idea of prioritized security deployment is the observation that what matters to the enterprise is that which BYOD smartphones are representative security-wise (so that they will be prioritized for security mechanism deployment), rather than the detailed co-location information; employees may prefer not going through the chore of periodically updating with the enterprise about their whereabouts, but only sharing the information locally with co-located smartphones when needed. Thus, a desirable algorithm should be localized.

Property 2 (Localization) *An agent makes its activation/deactivation decisions based on its own status and the connectivity logs from other smartphones it co-locates with.*

While localization (Property 2) decentralizes information collection process among opportunistically co-located smartphones, the information collected by agents about its past co-located neighbors may be outdated, and the reachability computed from such information may be different from the actual one at that moment. Requiring employees to constantly or on-demandly update such information with their neighbors induces great overheads and, therefore, negates the benefits of decentralization. Thus, a desirable algorithm should be able to handle outdated information while electing agents for prioritized defense deployment.

Property 3 (Temporal robustness) *Property 1 is achieved even if the connectivity logs obtained from other smartphones during Wi-Fi co-location is outdated.*

In the rest of the section, we will show that the algorithm presented in Section 2.3 satisfies the Properties 1–3. Because only Deactivation (Section 2.3.3.2) may violate the properties, and Group-Weak is the most aggressive deactivation rule, we prove, in Theorem 1, that all three properties are satisfied by the design under the Group-Weak rule; the cases for other less aggressive deactivation rules are corollaries to Theorem 1. In addition, we complement our analysis here with simulations on real AP-association traces in Section 2.5.

Theorem 1 *If an agent a deactivates itself in its local (and potentially outdated) view at the moment t, then, in the global (and updated) view, each of the smartphones T-dominated by a, including a itself, is still T-dominated by some agent at t.*

We break the proof of Theorem 1 down to a series of lemmas. Before proceeding, we need to make some extension to the notation to be more precise. The reachability metric, as defined in Equation (2.2) for two smartphones u and v, are actually defined on snapshots of u and v connectivity logs l_u and l_v, respectively. Therefore, we make this explicit by writing $\mathbf{r}(l_u, l_v)$ in place of $\mathbf{r}(u, v)$.

Lemma 1 is a property of the reachability metric defined in Equation (2.2).

Lemma 1 *Let l_u and l'_u (l_v and l'_v) be two snapshots of the connectivity log of smartphone u (v). If the common intervals of l'_u and l'_v are all contained in those of l_u and l_v in the temporal window $[t-W, t]$, then:*

$$\boldsymbol{r}(l_u, l_v) \leq \boldsymbol{r}(l'_u, l'_v).$$

Proof In the same notation in Equation (2.2), let the common intervals of l_u and l_v within the window $[t-W, t]$ be $[s_1, e_1], \ldots, [s_k, e_k]$; $s_{k+1} = s_1 + w$. By Equation (2.2), $\mathbf{r}(l_u, l_v) = \sum_{i=1}^{k}(s_{i+1} - e_i)^2 \Big/ 2W$.

Since the common intervals of l'_u and l'_v are all contained in the common intervals of l_u and l_v in the temporal window $[t-W, t]$, the common intervals of l'_u and l'_v within the window $[t-W, t]$ can be represented as $[s_i, e_i], [s_{i+1}, e_{i+1}], \ldots, [s_j, e_j]$ for some $1 \leq i \leq j \leq k$. By Equation (2.2), $\mathbf{r}(l'_u, l'_v) = \sum_{n=i}^{j}(s'_{n+1} - e'_n)^2 \Big/ 2W$.

We have:

$$\begin{aligned} \mathbf{r}(l'_u, l'_v) - \mathbf{r}(l_u, l_v) = \Big[&(s_i + W - e_j)^2 - \\ &\sum_{n=1}^{i-1}(s_{n+1} - e_n)^2 - \sum_{n=j}^{k}(s_{n+1} - e_n)^2\Big] \Big/ 2W. \end{aligned} \tag{2.4}$$

Since $s_1 < e_1 < \ldots < s_i < e_i < \ldots < s_j < e_j < \ldots < s_k < e_k < s_{k+1} = s_1 + W$,

$$\begin{aligned}
&\sum_{n=1}^{i-1}(s_{n+1}-e_n)^2 + \sum_{n=j}^{k}(s_{n+1}-e_n)^2 \\
\leq&\left[\sum_{n=1}^{i-1}(s_{n+1}-e_n)\right]^2 + \left[\sum_{n=j}^{k}(s_{n+1}-e_n)\right]^2 \\
\leq&(s_i-e_1)^2+(s_{k+1}-e_j)^2 \\
\leq&(s_i-e_1+s_{k+1}-e_j)^2 = (s_i-e_1+W+s_1-e_j)^2 \\
\leq&(s_i-s_1+W+s_1-e_j)^2 = (s_i+W-e_j)^2.
\end{aligned} \tag{2.5}$$

By Equations (2.4) and (2.5), $\mathbf{r}(l_u, l_v) \leq \mathbf{r}(l'_u, l'_v)$. ■

In the following discussion, we use $l^t_{u(v)}$ to denote the snapshot of smartphone v's connectivity log stored on an agent u (only agents store other smartphones' connectivity logs) at time t, or in other words, u's local view of v at t. By definition, $l^t_{u(u)}$ is u's latest connectivity log at t, which is exactly u's connectivity log at t from the global view; therefore, we write $l^t_{u(u)}$ simply as l^t_u. We use $l^t_{u(u)}$ and l^t_u in different contexts to emphasize the different perspectives: The former is from u's local view, and the latter is from the global view.

Lemma 2 shows that, after collecting a window's intelligence, an agent's local view on the set of smartphones that is T-dominated by it agrees with the global view. This is the technical justification for requiring an agent to collect a window's intelligence before deactivating itself.

Lemma 2 *Suppose a is an agent during $[t-W, t]$. For each smartphone u with $\boldsymbol{r}(l^t_a, l^t_u) < +\infty$, we have*

$$\boldsymbol{r}(l^t_{a(a)}, l^t_{a(u)}) = \boldsymbol{r}(l^t_a, l^t_u).$$

Proof Since $\mathbf{r}(l^t_a, l^t_u) < +\infty$, by Definition (2.2), a has met u at least once during $[t-W, t]$; since a is an agent during $[t-W, t]$, $l^t_a = l^t_{a(a)}$ includes a record on the last meeting between a and u. Thus, the common intervals of $l^t_{a(a)}$ and $l^t_{a(u)}$ are exactly the same with those of l^t_a and l^t_u in the temporal window $[t - W, t]$. By Lemma 1, $\mathbf{r}(l^t_{a(a)}, l^t_{a(u)}) \leq \mathbf{r}(l^t_a, l^t_u)$ and $\mathbf{r}(l^t_{a(a)}, l^t_{a(u)}) \geq \mathbf{r}(l^t_a, l^t_u)$. Hence $\mathbf{r}(l^t_{a(a)}, l^t_{a(u)}) = \mathbf{r}(l^t_a, l^t_u)$. ■

Proof [Proof of Theorem 1] a deactivates itself at t if a is an agent during $[t-W, t]$ and finds, in its local view, a group of agents A with higher priorities, so that each smartphone T-dominated by a (including a itself) is T-dominated by at least one agent from A.

By Lemma 2, a's local view on the set of smartphones T-dominated by itself agrees with the global view. Hence, we only need to show that a non-agent u, which is T-dominated by both a and another agent $v \in A$ at t in a's local view, is actually T-dominated by some agent at t in the global view.

The proof is concluded if a's local view on v agrees with the global view. However, two possible discrepancies between a's local view and the global view demands further discussion: connectivity log and agent status of v.

The first case is straightforward to resolve. Suppose v is still an agent at t in the global view. Since $l^t_{a(u)}$ and $l_{a(v)}$ are past snapshots of l^t_u and l^t_v respectively, the common intervals of $l^t_{a(u)}$ and $l_{a(v)}$ are all contained in l^t_u and l^t_v in the temporal window; by Lemma 1, $\mathbf{r}(l^t_u, l^t_v) \leq \mathbf{r}(l^t_{a(u)}, l^t_{a(v)})$. Since u is T-dominated by v in a's local view, we have $\mathbf{r}(l^t_{a(u)}, l^t_{a(v)}) \leq T$. Thus, $\mathbf{r}(l^t_u, l^t_v) \leq T$: u is T-dominated by the agent v at the moment t in the global view.

The latter case is more involved. Suppose v is no longer an agent at t in the global view. v must have deactivated itself and delegated the dominance of u to a replacement w at an earlier time $t' < t$ *after the last encounter between a and v* (w must be an agent at the moment t' for this to happen). Thus, $l^t_{a(v)}$ is a past snapshot of $l^{t'}_{v(v)}$. Since $l^{t'}_{v(u)}$ contains all the encounters between u and v up to the moment t', the common intervals of $l^t_{a(u)}$ and $l^t_{a(v)}$ are all contained in those of $l^{t'}_{v(u)}$ and $l^{t'}_{v(v)}$, thus $\mathbf{r}(l^{t'}_{v(u)}, l^{t'}_{v(v)}) \leq \mathbf{r}(l^t_{a(u)}, l^t_{a(v)}) \leq T$ by Lemma 1.

Since v deactivated itself at t' for w, $\mathbf{r}(l^{t'}_{v(u)}, l^{t'}_{v(w)}) \leq \mathbf{r}(l^{t'}_{v(u)}, l^{t'}_{v(v)}) \leq T$. Since $l^{t'}_{v(u)}$ and $l^{t'}_{v(w)}$ are both past snapshots of l^t_u and l^t_w, the common intervals of $l^{t'}_{v(u)}$ and $l^{t'}_{v(w)}$ are contained in those of l^t_u and l^t_w, thus $\mathbf{r}(l^t_u, l^t_w) \leq \mathbf{r}(l^{t'}_{v(u)}, l^{t'}_{v(w)}) \leq T$ by Lemma 1.

That is to say, even though v may be deactivated at t, u is still T-dominated by the agent w delegated by v.

Thus, by the same argument on v's replacement w at t', even if v has deactivated itself by t, either the replacement w actually T-dominates u at t, or it has further delegated u to other agents at an earlier time. By tracing back this chain of delegation, we can eventually find, in the global view, an agent that T-dominates u at t.

We now show that there is no loop in the chain of delegation. Along with the fact $T \geq \mathbf{r}(l^t_{a(a)}, l^t_{a(u)}) \geq \mathbf{r}(l^t_{a(u)}, l^t_{a(v)}) \geq \mathbf{r}(l^{t'}_{v(u)}, l^{t'}_{v(v)}) \geq \mathbf{r}(l^{t'}_{v(u)}, l^{t'}_{v(w)}) \geq \dots$ we have just proved, the non-equality requirement in the priority comparison rule ensures that there is no loop in the chain of delegation. ■

2.5 Experiment

We complement our analysis on T-dominance with simulations driven by real-world collected datasets.

2.5.1 Dataset and Methodology

The dataset is from the Wireless Topology Discovery (WTD) project [139]. The dataset consists of traces collected from over 190 UC San Diego freshmen using hand-held mobile devices for an 11-week period. Periodic Wi-Fi AP scanning and association results were recorded every 20 seconds. The students participating in this experiment, though coming from different majors, resided in the same university housing facility. This setting resembles the arrangement in a large enterprise site, with employees working in their designated office spaces (corresponding to students' dormitories). The traces capture the mobility and connectivity patterns of a group of users in a relatively short period of time [139].

Given the frequency of data recording (once every 20 seconds), we transformed the periodic records into a series of sessions (a session is defined as a device associating with an AP during a period of time) by the following method: Consecutive records of the same device associating with the same AP within 20 seconds were combined to form a single session.

The transformed traces were then fed into an event-driven simulator implemented in Perl. Each session in the transformed traces triggers two events along the time line: an association event and a de-association event. We took the first 200 thousands entries in the records and used the first 30% of the data for the 190 nodes to accumulate connectivity logs, which allowed them to simulate the agent election process later. Then, some nodes were randomly selected as initial agents; the agents made activation/deactivation decisions, based on the algorithm specified in Section 2.3. In the following scenarios, the simulation process was repeated with different psuedo-random number generator (PRNG) seeds to obtain the means and quartiles.

2.5.2 Scenario and Results

2.5.2.1 T-dominating Agent Election

We simulated the agent election process under the different T-dominating strategies (Group-Strong, Group-Weak, Individual-Strong, and Individual-Weak) with different numbers of initial agents and T. Since for a given set of initial agents, no proximity-based activation strategy can activate more agents than the epidemic one (the epidemic strategy is one in which agents unconditionally activate their co-located neighbors), we normalized the results with the epidemic strategy to make them comparable: At a particular moment, the number of agents elected by a T-dominance strategy is divided by the number of agents activated by the epidemic strategy, to obtain the normalized agent set size. We computed the means of the results from multiple rounds of simulation, with different PRNG seeds, to account for the potential bias introduced by a peculiar initial setting. Figure 2.3 shows a representative result with 5, 10, and 15 initial agents (out of the 190 nodes) with $T = 18,000s$ (5 hours). The following are a few notes on Figure 2.3:

- In terms of agent set reduction through self-deactivation, Group-Weak is the most aggressive strategy, and Individual-Strong is the most conservative one, while the other two come in between, and are comparable. This confirms our design intuition in Section 2.3.

- The size of the initial agent set has little influence on the size of the agent set eventually elected. The small differences are mostly at the beginning of the process and are difficult to notice without zooming in. One explanation is that this dataset, like in many closed-world networks such as in an enterprise, is well connected: Except for maybe a few peculiar cases, an agent election process originating from a small set of agents spreads to the whole network quickly; the activation/deactivation process since that moment will then converge.

- An agent election process election process consists of two consecutive phases. The first phase (0 to around 5000 seconds in Figure 2.3) corresponds to the general trend of decreases (with occasional small increases) in normalized agent set size (NASS), and reflects the overwhelming effect of deactivation in search of the T-dominating agent set. The second phase (after 5000 seconds in Figure 2.3) is characterized by the dynamic balance between activation and deactivation when the T-dominanting agent set has been activated.

2.5.2.2 Prioritized Defense Deployment

We simulated prioritized defense deployment based on the T-dominance-elected agents. We consider the following scenario. Following the election of T-dominating agents as in Figure 2.3, the elected agents were periodically checked for malware infection; once an infection is detected, the infected agent would enroll itself, along

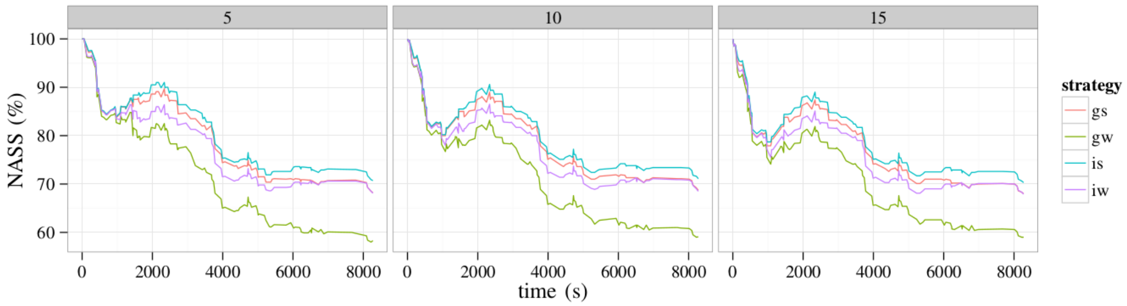


Figure 2.3.: A representative T-dominating agent election process with 5, 10, and 15 initial agents (out of the 190 nodes) and $T = 18,000s$ (5 hours). Agent set size is normalized by epidemic activation strategy: the y-axis is shown in normalized agent set size (NASS). Strategy notations: gs (Group-Strong), gw (Group-Weak), is (Indivdual-Strong), iw (Individual Weak).

with the non-agents T-dominated by it, in a malware patching pool. For simplicity, we considered the case in which there was no delay in detecting agent infection: The agents were constantly monitored for malware infection.

Independently, a smartphone was randomly selected from the patching pool at the rate of once every ten seconds and, if the smartphone was indeed infected, it would be patched. Patched smartphones would then become immune to malware infection.

We compared the T-dominance-based prioritized defense deployment, instantiated by this strategic sampling/patching (strategic s/p) strategy, with a random sampling/patching (random s/p) strategy. The latter periodically selected a smartphone randomly for malware infection checking, at the same rate as in the prioritized defense deployment (i.e., once every ten seconds). If the selected smartphone was indeed infected, it would be patched immediately.

We considered both epidemic and static malware models, which correspond to proximity malware attacks and drive-by download attacks, respectively. We assumed that an agent could detect malware infection in co-located smartphones, and, if malware infection was detected, would enroll its T-dominating smartphones in the malware patching pool.

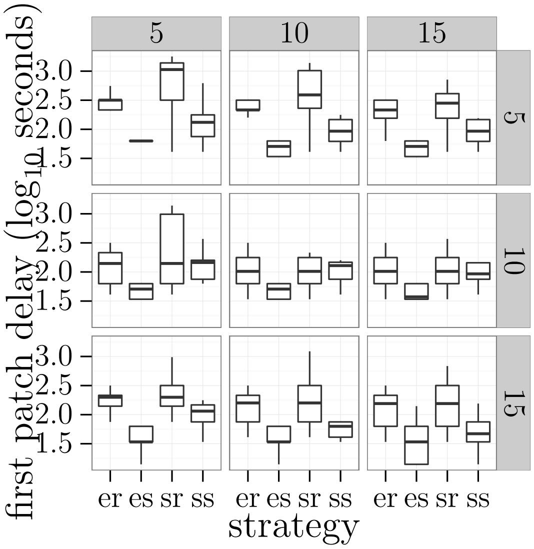


Figure 2.4.: Delay from the malware breakout to the first patching of a malware-infected smartphone. The patching rate is once per ten seconds. The row heading shows initial agent number *before* malware election; the column heading shows the number of malware-infected smartphone at the malware breakout. Strategy notation: er (epidemic malware, random sampling/patching), es (epidemic malware, strategic sampling/patching), sr (static malware, random sampling/patching), ss (static malware, strategic sampling/patching). The y-axis is shown in a $\log_{10}$ scale.

Boxplots of the results[4] are shown in Figures 2.4 and 2.5 for different numbers of initial agents (corresponds to the value in Figure 2.3) and initial malware-infected smartphones.

Figure 2.4 shows the delay between the initial malware outbreak and the first patching of a malware-infected smartphone. A few notes on Figure 2.4:

- Strategic s/p has a shorter delay than random s/p. In other words, the former is more responsive to malware infection than the latter. This justifies the adoption of T-dominance for prioritized defense deployment: By having an agent set that T-dominants the whole smartphone pool to serve as sampling points, malware outbreaks will be detected more promptly.

[4] Boxplots [138] show the max/min, 75%/25% quartiles, and the median of a group of observations.

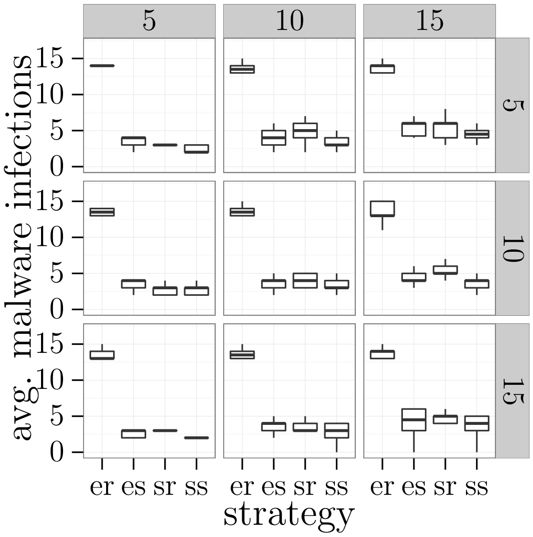


Figure 2.5.: Average malware number. The notations are the same as in Figure 2.4.

- The delay under the static malware model with small numbers of initial malware-infected smartphones is relatively long; the delay under the epidemic malware model with large numbers of initial malware-infected smartphones is relatively short. The explanation is that more smartphones will be infected by the malware shortly in the latter case, so initial sampling and ensuing patching will take less time than the former case.

Figure 2.5 shows the number of malware-infected smartphones averaged through the whole infection process (from the malware outbreak to the moment that all malware-infected smartphones were patched). A few notes on Figure 2.5:

- Under the epidemic malware model, T-dominance-based strategic s/p has significant less average malware infections than that of random s/p: A typical number of average infections is 3 for strategic s/p and 13 for random s/p. The difference is even more pronounced when there are more initial malware infections as in the upper rightest column with 15 initial infections.

- Even under the static malware model, where the malware would not propagate from infected smartphones to others and, hence, the average number of malware-infected smartphones over time shall be less than the initial number, the strategic s/p based on the T-dominating agent set has 1 to 3 less average infections than the random s/p.

Results shown in Figures 2.4 and 2.5 collectively show the responsiveness and effectiveness of T-dominance-based prioritized defense deployment, as instantiated by the strategic s/p, in detecting and mitigating BYOD smartphone malware.

2.6 Extended Discussion

Currently, we are not aware of any real-world report of smartphone malware propagating through Wi-Fi co-location. However, this does not mean that the attack model is purely hypothetical or impractical. For example, a report [122] on hijacking hotel Wi-Fi hotspots for drive-by malware attacks on laptops comes close to what we have in mind; practical man-in-the-middle attacks against Wi-Fi co-located devices was demonstrated in a recent BlackHat security conference [198]. We note that enabling environments and techniques for Wi-Fi co-location-based smartphone malware are already in place.

- Given the complexity of the commercial smartphone software platforms and the diversity of security awareness and experience of application developers, it is reasonable to believe that remote exploitable vulnerabilities will be discovered and exploited.

- The privilege authorization frameworks on smartphone platforms, which supposedly prevent the malware from obtaining unwarranted privilege, are often ignored for convenience, or circumvented for customization by the users. Rootkits, like iOS Jailbreak[5], are routinely used by users for installing third-party applications, whose trustworthiness is often assumed, but not verified.

[5] `http://www.jailbreakme.com/`

- Commercially-available Wi-Fi honeypots like Wi-Fi Pineapple[6] enable DNS spoofing, ARP poisoning, and man-in-the-middle attacks.
- The concentration of mobile application development on the two major smartphone platforms (iOS and Android) greatly reduces device heterogeneity, and thus, makes malware epidemics possible.

Given these considerations, Wi-Fi-co-location-based smartphone malware is likely to emerge in the near future; even worse, such malware may have already been deployed in the real world. This makes the study in mitigating it for a comprehensive BYOD network security model relevant and worthwhile.

2.7 Related Work

Although BYOD features numerous recent IT industry analyses and news reports as one prominent enterprise IT trend in the coming years [98, 166, 24], academic studies on the security implications of BYOD are scarce and still at an early stage [141, 195, 159]. One explanation is that while it is agreed that BYOD brings many benefits as well as management/security challenges, approaches to modeling and resolving the challenges are still being explored. In this chapter, we identify the tension between security provision and employee intrusiveness/security mechanism deployment cost as one of the challenges for BYOD security and propose prioritized defense deployment as a solution.

Proximity malware has been studied previously in the context of sensor, ad hoc, P2P, or mobile networks, with a focus on either identifying the critical point for long-term malware survival/extinction under various epidemiological models [42, 201, 160], or extracting and exploiting mobility pattern and community structure for malware mitigation [83, 126, 125]. Studies on Android, one of the dominating smartphone software platforms, show that many mobile applications are vulnerable to attacks

[6] `http://hakshop.myshopify.com/products/wifi-pineapple`

and malware on the Android smartphone software platform [74, 92, 198] and that malware is rampant [236].

The fascinating topic of capturing temporal dynamics in complex networks is studied by many previous works, often in the context of human mobility patterns captured by telecommunication service traces [190, 193, 118]. T-dominance is our attempt to capture the temporal dynamics in the BYOD enterprise environment. The exploitation of temporal dynamics for mitigating BYOD malware threat is novel.

The T-dominance algorithm is inspired by previous works on the Connected Dominating Set (CDS) problem of topology and routing in ad hoc and sensor networks [213, 220, 177]. However, the interpretation of CDS for temporal dynamics, the application in electing security-wise representative nodes in a BYOD network, and the issue of temporal robustness are all novel.

2.8 Summary and Future Work

Evidence indicates that many enterprises have adopted or are considering adopting a BYOD IT policy. However, research on BYOD enterprise network security is still at an early stage; many issues are yet to be clearly identified. In this chapter, in the context of smartphone malware attacks and widely deployed enterprise Wi-Fi infrastructures, the tension between security provision and intrusiveness/cost is identified as one such issue; prioritized defense deployment based on security representativeness is one approach to address the tension; prioritization by temporal-spatial structure through T-dominance is one interpretation of security representativeness. Other issues/approaches/interpretations are to be explored.

Independent from the application of T-dominance in prioritizing defense deployment, we briefly discuss the possibility of abusing T-dominance in making BYOD malware attacks stealthy. This shows, from another perspective, the importance of understanding the temporal-spatial structure for BYOD enterprise network security. For example, a study on the competition between a strategic sampling/prioritized patching scheme and an instance of stealthy malware, running the T-dominance algorithm with different T, would be interesting.

3 THE VIRTUE OF PATIENCE: OFFLOADING TOPICAL CELLULAR CONTENT THROUGH OPPORTUNISTIC PROXIMATE LINKS

The application of opportunistic proximate links considered in this chapter is to offload cellular data traffic through proximate channels. The motivation behind this application is the observation that the cellular channel is often congested and costly in the current telecommunication service economic model, whereas proximate links are often under-utilized. This chapter considers a *topical* content distribution model with bounded content delivery delay tolerance, in which users subscribe to various topics of content, and the objective is to deliver a piece of content to its subscribers before its expiration. The challenge is to achieving the objective cost-effectively without costly central coordination/planning over cellular channel. The essence of this chapter is for each individual smartphone to aggregate its user and the user's opportunistic neighborhood's interest profile over proximate channel into a time-dependent metric called "patience," and locally follows a probabilistic cellular download algorithm periodically using the patience metric and according to the situation whether it has received the content or not.

This chapter is previously published as a conference paper [154] in IEEE International Conference on Mobile Ad-hoc and Sensor Systems (MASS), 2013.

3.1 Introduction

The cellular infrastructure is overloaded by an expanding user base and increasing bandwidth demand from smartphone applications. Indeed, driven primarily by smartphones, AT&T's wireless data traffic has grown 200 folds over the five years between 2007 and 2011 [64].

Mobile data offloading, or mobile cellular traffic offloading, exploits alternative communication technologies on smartphones, and user mobility, to deliver information originally scheduled for transmission over the cellular networks. Previous works [94, 95, 96] demonstrate the feasibility of offloading cellular traffic by peer-to-peer assisted forwarding through Bluetooth. Recent developments in communication technology, embodied in the smart mobile devices (e.g., Google Nexus 7 [27] and the iPhone 5 [174]) that support NFC [146] and Wi-Fi Direct [210], makes spontaneous bulk data transfers between proximate users a reality. Furthermore, the current data usage cap and tiered pricing model [212] incentivizes smartphone users to offload their cellular data. These developments make further research in mobile data offloading relevant and worthwhile.

In this chapter, we study the problem of offloading cellular traffic through opportunistic proximate links such as Wi-Fi Direct. In our model, we include a factor that was missing in existing mobile data offloading models: users' interests in content. Users' interests are particularly relevant for large-scale networks: Nobody desires (or is able) to consume all generated content. This lies behind the quest for better search engines and the rise of social taxonomy, or folksonomy, in tagging content. Additionally, we consider bounded delivery-delay tolerance to model the general case where the content, though having no hard real-time requirement, still needs to be delivered before too long, lest it becomes stale.

Figure 3.1 illustrates the complication brought by users' interests: When a, b, and c meet through a proximate link and if, due to limited budget, one and only one of them will download a piece of content through the cellular link: Who, among them, should download? Though b has more acquaintances than a does and therefore, in some sense, is more socially important, few of b's acquaintances are interested in the content, when compared with those of a: It is more cost effective for a to download and carry the content than b. In another comparison, c is more socially important than a, and most of c's acquaintances are interested in the content: Though c is not interested in the content, if c downloads and carries the content, c can serve more

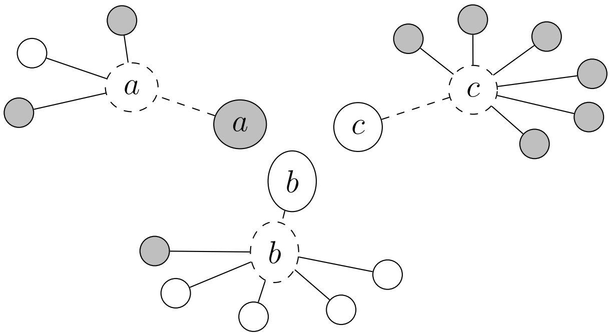


Figure 3.1.: Users' interests in content complicate the offloading strategy. Shaded nodes represent interested users; solid lines link acquaintances; dashed lines and nodes represent nodes' mobility.

users within a reasonable time than *a* can. In general, a cost-effective offloading strategy involves an interplay between users' interests and their social importance.

In addition to deciding *who* shall download the content through cellular links, as in the target-set selection formulation [96], we ask *when*. To appreciate the benefits of including time in the model, we consider a few scenarios.

- Every user downloads his[1] interested content through the cellular link immediately after the content is released. No offloading through the proximate link takes place in this case. This is the baseline *diligent* strategy that mobile data offloading measures against.

- Every user initially waits, in the hope that someone will download the content and forward the content (through the proximity link) to him through one of his acquaintances. However, nobody will receive the content, since nobody has downloaded it. Even if the content is eventually downloaded by some random user, and is forwarded to other interested users, it may have expired. This is the *lazy* strategy that introduces an unacceptably long delay.

- Some well-connected, or socially important, users, whose acquaintances are interested in the content, download the content through the cellular link, and for-

[1] "He" ("his") is to be read "he/she" ("his/her") henceforth.

> ward the content to their acquaintances when they meet through the proximate link. As time passes by, and the risk of the content becoming stale increases, those users who have not received their interested content through either link become impatient in waiting, and eventually download the content through the cellular link if the content has still not been received after a long delay. This *adaptive* strategy is neither too diligent nor too lazy, and provides a trade-off between cellular traffic load and content delivery delay.

A challenge is to design such an adaptive strategy without resorting to central scheduling and coordination through the cellular link, which is costly and less scalable. Although human mobility exhibits patterns [91, 182], contact opportunities are hard to predict precisely. Therefore, effective central scheduling and coordination require prohibitively costly updating.

We address the challenge as follows. Users estimate their relative social importance in the dynamic, opportunistic, proximity-link-based network with a weighted ego-centric betweenness centrality metric (Equation (3.2)); users estimate their (and their acquaintances') aggregated interests (Equation (3.3)) based on their chances of meeting each other (Equation (3.1)); users use a function (Equation (3.4)), which embodies the concept of users' *patience* for the content, to consolidate users' social importance with aggregated interests. This function gives rise to a probabilistic cellular offloading strategy (Equation (3.5)) that assigns a cellular download probability to a user, according to his capability to help offload the topical cellular content. Users then periodically decide whether to download the content through the cellular link by their patience *at that time*.

Thus, our solution is *social*, *content*, and *situation*-aware: Involving topologically important, but otherwise disinterested, users in downloading and forwarding content will help reduce the cellular traffic and improve the offloading efficiency, while satisfying users' content demand.

In the following sections, we formulate the problem (Section 3.2), describe the design of our patience-based cellular offloading strategy (Section 3.3), analyze its properties (Section 3.4), and complement the analysis with trace-driven simulations (Section 3.5). Works that inspire ours are summarized in Section 3.6.

3.2 Model

Consider a group of smartphone users: Each user has a smartphone that can access the Internet through the cellular link, and connect with nearby smartphones through some proximate link. For example, Bluetooth is the current standard for linking proximate devices: Han et al. evaluated and confirmed the feasibility of using Bluetooth to offload cellular data [96]. Other possibilities include Wi-Fi co-location (two users connect to the same Wi-Fi access point) and the upcoming Wi-Fi Direct. The cellular link is persistent but expensive, while the proximity link is opportunistic but free: A smartphone can access the Internet immediately on demand through the cellular link, while two smartphones can connect with each other through the proximate link only when they are nearby. An opportunity for two smartphones to connect through the proximate link is an encounter between them.

The users are interested in some content that is generated and released by some publisher on the Internet. Each piece of content is tagged before its release. We model two aspects of the user-content relationship: users' interests and content's freshness. For a user u:

- The interests of u are represented by a set of tags I_u: u is interested in a piece of content if the content has a tag in I_u.
- For a tag g, u prefers to receive a piece of content with tag g, within a delay of up to f_g. Otherwise, if u has not received the content within that time, the content becomes stale for u. We call f_g the freshness of tag g.

The publisher publishes an aggregate feed, containing the summary, the tags, and the download link for each piece of released content. A user is notified through the feed when new content is released.

The problem is to find a localized strategy that minimizes the number of cellular downloads (which incur costs) while maximizing fresh content deliveries. A localized strategy is one in which each user makes decisions based on information obtained through encounters rather than requiring (global) coordination through the cellular link. Communication through cellular link incurs cost, and keeping track of local changes for global coordination aggravates the problem. In comparison, a localized strategy is cost-effective and adaptive. The decisions to be made include whether to download a piece of content through the cellular link and, if the answer is yes, when.

Before moving on to describe the concrete design, we make our assumptions explicit. Since the proximate link is virtually free, routing on the proximity-link-based network is not a focus of this chapter: To maximize coverage, the content is epidemically forwarded across the opportunistic proximity-link-based network once it is downloaded through the cellular link. A more sophisticated forwarding strategy [124] can be adopted, but is beyond the scope of this chapter. The users are honest and cooperative. In other words, each user will follow the protocols by honestly sharing information and cooperatively reducing the overall cost:

- A user will honestly report their interests to others upon request.
- If downloading, storing, and forwarding a piece of content will reduce the overall cellular cost of the whole network, a user will do it even if he is not interested in the content.

Enforcement and incentive [212, 238, 152], while important, are orthogonal to the current problem and are left for further studies.

3.3 Design

3.3.1 Overview

Intuitively, two groups of users are favored for directly downloading a piece of content with tag g through the cellular link:

- Those who are interested in g and meet with users who are interested in g;
- Those who are socially important, or equivalently, topologically important in the dynamic proximity-link-based network.

The rationale for favoring the first group is obvious: Those users have better chances of directly obtaining or forwarding the content to interested users. However, the scope of this case is restricted to direct acquaintances and is thus oblivious of the topology of the proximity-link-based network, for which the second case tries to remedy. The membership in the two groups may overlap; those who are members of both groups are favored over those who are members of only one group.

Concretely, a user decides his topological importance in the dynamic proximity-link-based network by locally computing his weighted ego-centric betweenness centrality (Section 3.3.3). Along with the aggregated interest of both himself and his acquaintances (Section 3.3.4), the user determines his patience for the content and periodically decides, with a temporal-dependent probability based on his patience, whether to download the content through the cellular link if he has not yet received the content (Section 3.3.5).

In this section, we focus on the design details. Discussions on intuition and rationale are deferred to Section 3.4.

3.3.2 Temporal Tie Strength

Let the set of users that u has met through the proximate link be U_u: U_u is the set of u's acquaintances. For $v \in U_u$, let the chronologically ordered sequence of encounters between u and v be $[a_1, b_1], [a_2, b_2], \ldots, [a_k, b_k]$, and the current time be t; the average interval between consecutive encounters $\hat{s}_u(v)$ is defined as:

$$\hat{s}_u(v) = \begin{cases} \dfrac{(t - b_k) + \sum_{i=1}^{k-1}(a_{i+1} - b_i)}{k} & u \text{ and } v \text{ have met.} \\ +\infty & \text{otherwise.} \end{cases}$$

By definition, $\hat{s}_u(v)$ is symmetric: $\hat{s}_u(v) = \hat{s}_v(u)$; $\hat{s}_u(v) \geq 0$; u can locally compute $\hat{s}_u(v)$ for all $v \in U_u$.

Based on $\hat{s}_u(v)$, the temporal tie strength (tie for short) $s_u(v)$ is defined as:

$$s_u(v) = \begin{cases} \exp(-\alpha_s \hat{s}_u(v)) & s_u(v) \in [0, +\infty), \\ 0 & s_u(v) = +\infty, \end{cases} \tag{3.1}$$

in which $\alpha_s > 0$ is a scaling parameter, adapting to the given scenario, to prevent the tie $s_u(v)$ from dropping too fast with the increase of the average inter-encounter interval $\hat{s}_u(v)$.

Greater $s_u(v)$ corresponds to stronger tie between u and v; $s_u(v) \in [0, 1]$. Like $\hat{s}_u(v)$, $s_u(v)$ is symmetric: $(s_u(v) = s_v(u))$ and u can locally compute $s_u(v)$ for all $v \in U_u$.

3.3.3 Weighted Ego-centric Betweenness Centrality

For $v, w \in U_u$, u can obtain $\hat{s}_u(v)$, $\hat{s}_u(w)$, and $\hat{s}_v(w)$ (or, equally, $\hat{s}_w(v)$) during their encounters. u can construct his neighborhood graph G_u, of which nodes are $\{u\} \cup U_u$ and the edge between $v, w \in U_u \cup \{u\}$ has a weight $\hat{s}_v(w) = \hat{s}_w(v)$ if $\hat{s}_w(v) \neq +\infty$. For $v, w \in U_u$ and $v \neq w$, let $p(v, w)$ be the proposition "(v, u, w) is a shortest path between v and w"; this can be determined by, for example, the Dijkstra's algorithm [113]. From G_u, u can compute a weighted ego-centric betweenness centrality β_u:

$$\beta_u = \begin{cases} \frac{\sum_{v,w \in U_u, v \neq w} [p(v,w)]}{2\binom{|U_u|}{2}} & |U_u| \geq 2, \\ 0 & \text{otherwise.} \end{cases} \tag{3.2}$$

In Equation (3.2) and the following, when p is a proposition, the notation $[p]$ is the propositional indicator function:

$$[p] = \begin{cases} 1 & p \text{ is true,} \\ 0 & p \text{ is false.} \end{cases}$$

From Equation (3.2), $\beta_u \in [0, 1]$.

3.3.4 Interest Aggregation

User u records the interests I_v of user v when they meet. u's aggregated interest $i_u(g)$ on tag g is:

$$i_u(g) = [g \in I_u] + \sum_{v \in U_u} s_u(v)[g \in I_v]. \tag{3.3}$$

$i_u(g) \geq 0$; $i_u(g) < 1$ only if $g \notin I_u$.

3.3.5 Patience and Probabilistic Cellular Downloading Strategy

From the centrality β_u (Equation (3.2)) and aggregated interest $i_u(g)$ on tag g (Equation (3.3)), u determines his patience $p_{u,g}$ for tag g as a function:

$$p_{u,g} : \quad [0, 1] \to [0, 1], \tag{3.4}$$

defined as (for two scaling parameters $\alpha_i > 0$ and $\alpha_\beta > 1$, which correspond to the interest aggregation $i_u(g)$ and the centrality β_u, respectively):

$$p_{u,g}(x) = \begin{cases} \left(1 - e^{-\alpha_i i_u(g)}\right) x^{\alpha_\beta^{(1-2\beta_u)}} & g \in I_u, \\ \left(1 - e^{-\alpha_i i_u(g)}\right) (1-x)^{\alpha_\beta^{(1-2\beta_u)}} & g \notin I_u. \end{cases} \tag{3.5}$$

The patience function defined by Equations (3.4) and (3.5) gives u a strategy to make cellular download decision. u, according to the strategy and based on the situation at that time (Have u received the content? How close to the content expiration?), periodically (at a pre-defined interval for all users) makes a probabilistic cellular download decision as follows. At the moment $t + x \cdot f_g$ ($x \in [0, 1]$) between the time t that u first learns about a piece of content with tag g and the time $t + f_g$ that the content becomes stale for u, u flips a biased coin and, with a probability $p_{u,g}(x)$, downloads the content through the cellular link. As a stipulation, if u is interested in the content himself, but has neither downloaded nor received the content by the time $t + f_g$, u will download the content directly through the cellular link to satisfy his content demand.

3.4 Analysis

3.4.1 Probabilistic Cellular Downloading Strategy Based on Patience

We take the patience function $p_{u,g}(x)$ defined in Equation (3.5) apart:

- The maximal probability that u will download the content through the cellular link *in one round* is $1 - e^{-\alpha_i i_u(g)}$, which is monotonically increasing on $i_u(g)$: Greater aggregated interest $i_u(g)$ corresponds to higher maximal cellular downloading probability.

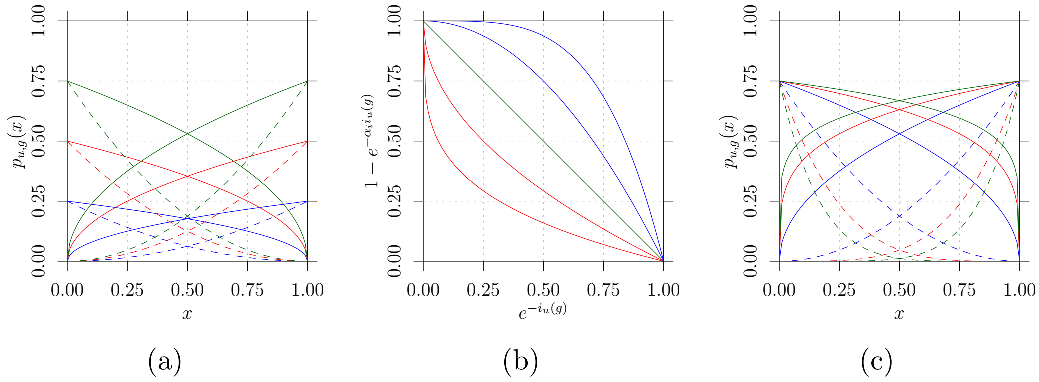


Figure 3.2.: The patience function $p_{u,g}$ and the scaling parameters α_i (interest $i_u(g)$) and α_β (centrality β_u). (a) Given the scaling parameters α_i and α_β, the patience $p_{u,g}$ function is jointly determined by the aggregated interest $i_u(g)$ and the centrality β_u. For $\alpha_i = 1$ and $\alpha_\beta = 2$, the patience functions corresponding to the 12 ($3 \times 2 \times 2$) combinations $i_u(g) = 0.29, 0.69, 1.39$ (corresponding to $1 - e^{-\alpha_i i_u(g)} = 0.25, 0.50, 0.75$; blue, red, green), $\beta_u = 0, 1$ (dashed, solid), and the cases $g \in I_u, g \notin I_u$ (increasing, decreasing) are plotted for comparison. (b) The effect of the (interest) scaling parameter α_i. The maximum of the patience function $(1 - e^{-\alpha_i i_u(g)})$, which corresponds to the maximal probability that u will download the content through the cellular link in one decision, are plotted against the inverse exponential of the aggregated interest ($e^{-i_u(g)}$) with $\alpha_i = 0.25, 0.5, 1, 2, 4$ (greater than 1: blue; less than 1: red; equal to 1: green) for comparison. (c) The effect of the (centrality) scaling parameter α_β. For (interest) scaling parameter $\alpha_i = 1$ and aggregated interest $i_u(g) = 1.39$ (corresponding to the maximal cellular downloading probability $1 - e^{-\alpha_i i_u(g)} = 0.75$), the patience functions $p_{u,g}$ corresponding to the 12 ($2 \times 2 \times 3$) combinations $\beta_u = 0, 1$ (dashed, solid), $\alpha_\beta = 2, 4, 6$ (blue, red, green), and $g \in I_u, g \notin I_u$ (increasing, decreasing) are plotted for comparison.

- The shape (i.e., bends upward or downwards, or mathematically, concave or convex) of the patience function $p_{u,g}$ depends on u's centrality β_u: $\beta_u = \frac{1}{2}$ corresponds to the diagonal; $\beta_u > \frac{1}{2}$ (u is more socially important) corresponds to a concave (bends upward) curve; $\beta_u < \frac{1}{2}$ (u is less socially important) corresponds to a convex (bends downward) curve.

- In all cases, the patience function $p_{u,g}$ is monotonic. The direction of change (i.e., increasing or decreasing) depends on whether u is interested in g himself, i.e., $g \in I_u$ or $g \notin I_u$. If $g \in I_u$, $p_{u,g}$ is monotonically increasing; if $g \notin I_u$, $p_{u,g}$ is monotonically decreasing.

The effect of the parts on the patience function $p_{u,g}$ is illustrated in Figure 3.2a. The effects of the scaling parameter α_i and α_β are shown in Figures 3.2b and 3.2c, respectively.

The probabilistic downloading strategy based on the patience function in Equation (3.5) has a few desirable properties.

Property 4 *If u has higher chances of serving users (possibly including himself) before content expiration, the maximal probability that u will download the content in one round is higher.*

We can validate Property 4 by noticing that the probability $1 - e^{-\alpha_i i_u(g)}$ is a monotonically increasing function depending only on $i_u(g)$ (given the system scaling parameter α_i): We will see in Section 3.4.4 that the intuition behind $i_u(g)$ is exactly to reflect the chances of u being able to serve content in time.

Property 5 *Other things being equal, more socially important users have higher cellular downloading probability.*

Property 5 is evident by comparing each pair of $\beta_u = 0$ and $\beta_u = 1$ curves with the same set of other parameters. Analytically, by Equation (3.5), it is straightforward to verify that a larger β_u leads to a larger $p_{u,g}(x)$ for the same $x \in [0, 1]$.

The intuition behind Property 5 is that a more socially important user has better chances of meeting others, and passing on the downloaded content. Therefore, letting them download with higher probabilities may help offloading the cellular traffic to the proximate link.

Property 6 *If u is* not *interested in a tag g, u's downloading probability will decrease over time; otherwise, u's downloading probability will increase over time.*

Property 6 is evident by noticing that, in Equation (3.5), $p_{u,}$ is monotonically increasing if $g \in I_u$ and monotonically decreasing if $g \notin I_u$.

The intuition behind Property 6 is as follows. If u is not interested in a tag g, u is being purely altruistic in downloading content with g. u can start downloading with a high probability in the hope that he can forward the content to others when they meet later. With the chances of meeting others (and hence forwarding the content to others through the proximate link) dwindling over time, the value of celluar downloading decreases. This is reflected by the monotonically decreasing downloading probability in the second case in Equation (3.5).

Conversely, if u is interested in a tag g, u is helping both himself and others in downloading content with g. u can afford to start downloading with a low probability in the hope that he can receive the content from another user who has the content, and thus, save cellular bandwidth. With the chances of meeting others (and hence receiving the content from others through the proximate link) dwindling over time, u becomes increasingly *impatient* in waiting. This is reflected by the monotonically increasing downloading probability in the first case in Equation (3.5).

3.4.2 Temporal Tie Strength

The average interval between consecutive encounter $\hat{s}_u(v)$ quantifies the frequency of encounters between u and v (thus, the opportunities to offload the cellular traffic to the proximate link), based on their past encounters: If they met frequently in the past, they are more likely to do so in the future. The assumption behind this is that human social contacts are regular and thus predictable, which is confirmed by studies on human mobility [91, 182, 172] and is taken by previous social-assisted routing schemes [107, 124].

$\hat{s}_u(v)$ can be computed efficiently by keeping a running sum of past intervals, a count of encounters, and the timestamp of the last encounter. This is amenable for implementation in a large network where the nodes, which are resource-constrained, have to keep track of a large number of neighbors.

The temporal tie strength $s_u(v)$ between u and v is a monotonically decreasing function on $\hat{s}_u(v)$ that maps into $[0, 1]$: The more frequently u and v meet, the stronger their (social) tie is. The reason for making $s_u(v)$ a number between 0 and 1 is to avoid marginalizing u's own interests in the aggregated interest $i_u(g)$ in Equation (3.3), which will be further discussed later in Section 3.4.4.

3.4.3 Weighted Ego-centric Betweenness Centrality

The weighted ego-centric betweenness centrality β_u defined in Equation (3.2) is inspired and loosely based on the ego-centric betweenness centrality [73]. The difference between the two are the weights on the edges and that, for a pair of u's opportunistic neighbors v and w, we do not divide $[p(v, w)]$ by the number of shortest (weighted) paths between them. The reason is that, given the heuristic nature of the centrality metric, minor relaxation is justified by computation efficiency. The rationale for considering a weighted graph is that, on an intermittently connected graph like the proximity-link-based network, the delay (characterized by the weights on the edges of the graph) matters.

Intuitively, β_u is the ratio (thus, $\beta_u \in [0, 1]$) that, among all pairs of u's opportunistic neighbors, u can pass on content with the shortest delay (the geodesic, or the shortest path). The greater β_u is, the more topologically important, or socially important, that u is on the proximity-link-based network.

3.4.4 Interest Aggregation

u's aggregated interest $i_u(g)$ on a tag g (Equation (3.3)) gives an estimation on the content demand by u and u's acquaintances.

The rationale for u to weigh an acquaintance v's interests by their tie $s_u(v)$ is as follows. u needs to decide whether downloading a piece of content will help meet v's content demand. This is restricted by their chances of meeting each other, as characterized by their tie $s_u(v)$. Even v is interested in a piece of content, if u has little chance of meeting v, there is little point for u to download the content for v.

Again, the rationale for making the tie $s_u(v)$ a number between 0 and 1 in Equation (3.1) is to avoid marginalizing u's own interests in the aggregated interest $i_u(g)$ in Equation (3.3). Downloading a piece of content that u is interested in, himself, will *immediately* satisfy his content demand, while others' content demand will be met by u's celluar downloading only if they meet some time *later*, before the content expires. Therefore, in u's cellular downloading decision, u's own interests are more important than others: Making $s_u(v)$ a number between 0 and 1 does exactly that in Equation (3.3).

By Equation (3.3), if u is interested in a tag g ($g \in I_u$), $i_u(g) \geq 1$: The only possibility that $i_u(g) < 1$ is that $g \notin I_u$. The greater $i_u(g)$ is, the more likely that u downloading a piece of content with tag g will help satisfy users' content demand through the proximate link.

3.5 Experiment

We compare the performance of the proposed patience-based offloading strategy with a recent work by Han et al. on cellular offloading, which is based on the target-set formulation [96]. The comparison is based on simulations driven by two publicly available contact traces: a real-world collected trace, Haggle INFOCOM 2006 [180], and a synthesized trace, NUS contact [189].

3.5.1 Methodology

3.5.1.1 Dataset

The Haggle INFOCOM 2006 contact trace (Haggle, henceforth) contained Bluetooth sightings of 78 attendees and 20 stationary nodes in the conference venue during the

3 days of the 2006 INFOCOM conference. It is widely cited due to its closed-world nature: The attendees met each other often in the conference venue, which produced a trace with repetitive contact patterns in a short time and a confined space. The time-resolution of this dataset is one second.

The NUS contact trace was synthesized from the class schedules and rosters for the Spring 2006 semester in National University of Singapore (NUS). Students attending the same session of a class were considered to have contacts with each other. In our simulation, we chose a group of 1,000 students who shared a class schedule with at least one other student in the group. The time-resolution of this dataset is one hour.

3.5.1.2 Procedure

Han et al. [96] proposed a deterministic, centralized, and heuristic algorithm to choose a set of nodes to serve as the offloading *target set*, i.e., nodes that download the content at the beginning and serve as initial seeds for subsequent proximate propagation). Although the target-set formulation of the cellular offloading problem is elegant, to select the target set, the algorithm (the emphtarget-set strategy henceforth) is centralized and requires the SP to collect individual nodes' contact information (which intrudes users' privacy) through either cellular links (which is costly under the current mobile computing business model) or other non-cellular links (e.g., WiFi, which is either inconvenient or untimely). Moreover, it is unclear what is the best size for the target set. Follow the method used by Han et al. [96], we resolved this by statistically summarizing the simulation results on multiple target sets with different sizes. Since Han et al. did not consider users' interest in their model [96], to fairly compare their target-set strategy with our patience-based one, we set the upper limit on the target set's size to the number of interested users, to eliminate the cases that (unfairly) favors patience-based strategy due to the absence of a parameter in

Table 3.1.: Parameters (from Equations (3.5) and (3.2)) for the three instances (eager, moderate, lazy) of the patience strategy used in the simulation.

		eager	moderate	lazy
Haggle	α_i	0.5	0.1	0.05
	α_β	2		
	α_s	0.01		
NUS	α_i	0.05	0.03	0.01
	α_β	2		
	α_s	0.01		

the target-set model; this allows us to assess the performance of the patience-based offloading strategy more objectively.

In contrast, the patience-based strategy is localized and adaptive. The parameters in Equation (3.5) can be used to tune the balance between maximizing offloading efficiency and minimizing content delivery delay in the patience-based strategy. To study this flexibility, we used three sets of parameters to instantiate the patience-based strategy. The resulting instances differ in their maximal downloading probability ($1 - exp(-\alpha_i i_u(g))$ in Equation (3.5); explained in Section 3.4.1) or, more intuitively, the eagerness in downloading the content through the cellular link early. The three instances are identified as *eager*, *moderate*, and *lazy* and their parameters (from Equations (3.5) and (3.2)) are shown Table 3.1.

Since the focus of our study was on reducing cellular traffic, we adopted a simple strategy in the opportunistic forwarding between proximate users: Once a node u obtained a piece of content (by either downloading through the cellular link or receiving from other nodes through the proximate link), the content would be forwarded through the proximate link to all of u's neighbors when they met u. This is known in literature as epidemic forwarding or flooding.

We simulated the cellular downloading decision processes under these strategies with various numbers of interested users. For each given number of interested users, we generated over 100 interest distributions among the users, and for each interest distribution, the stochastic decision process was repeated 50 times to reduce statistical bias.

3.5.1.3 Metrics

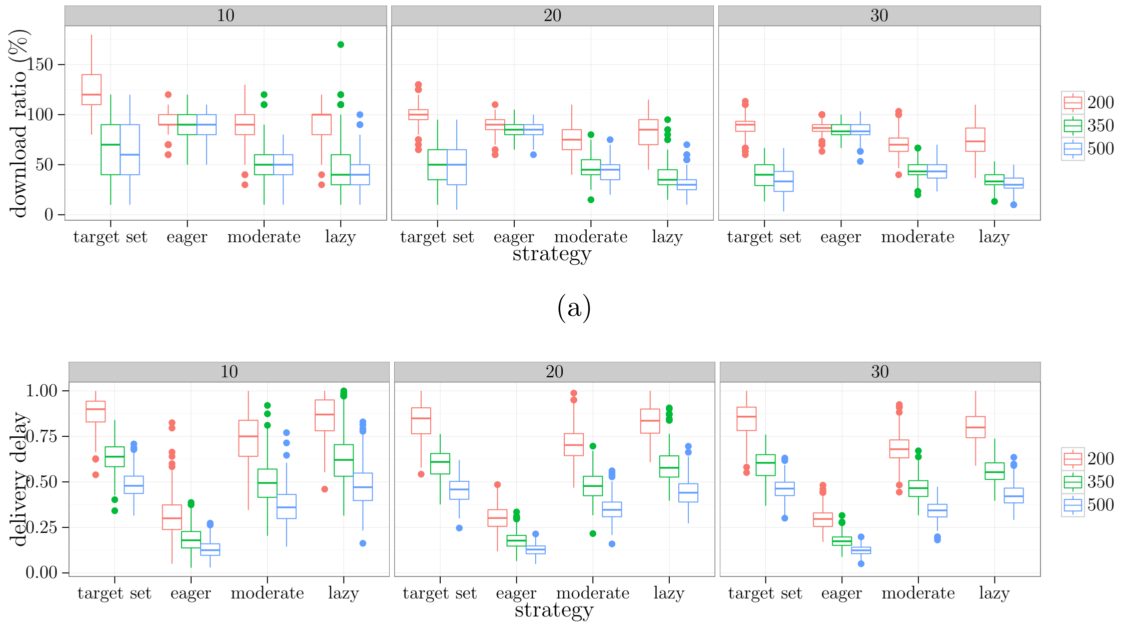
Performance of the strategies are measured by two metrics, *downloading ratio* and *content delivery delay.*

Download ratio. An offloading strategy's efficiency can be measured the number of cellular downloads by the end of the decision process (which is determined by the content's freshness). Quantitatively, if there are n_i users who are interested in the content and, by the end of the offloading process, the content is downloaded through the cellular link d times, the downloading ratio of the offloading strategy is $\frac{d}{n_i} \times 100\%$. An offloading strategy that can satisfy users' content demand with fewer cellular downloads is more efficient.

Content delivery delay. While delay is inevitable for an offloading strategy that does not have every interested user download a piece of content as soon as it becomes available, it is desirable that the delay is minimized. Thus, another aspect of an offloading strategy's efficiency is the content delivery delay that it introduces. Quantitatively, for a piece of content u that is released at the moment 0 and must be delivered by the moment 1[2], let the time of delivery to an interested user $i \in I_u$ be $t_d(i)$, the (average) content delivery delay is $\sum_{i \in I_u} t_d(i)/|I_u|$, which is a value between 0 and 1.

While it would be nice to have an offloading strategy that has low downloading ratio and content delivery delay, these objectives are not always compatible with each other: A trade-off between cellular bandwidth usage and content delivery delay often needs to be made. This is discussed in more detail, in the context of simulation results, in Section 3.5.2.

[2] We can normalize the delay by content's delivery deadline to make the delivery delay to 1. Since all interested users who have not received the content by the delivery deadline download the content directly through cellular link, normalized content delivery delay is never greater than 1.



(a)

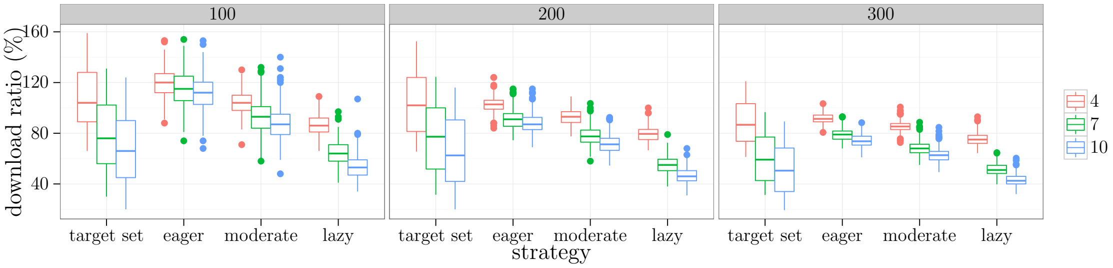
(b)

Figure 3.3.: Download ratio and (normalized) delivery delay of the Haggle dataset. Results with different numbers of interested users (10, 20, and 30 interested users out of the 98 nodes) and content delivery deadline (200, 350, and 500 seconds) are compared. For the patience strategies, a downloading decision is made every 50 seconds.

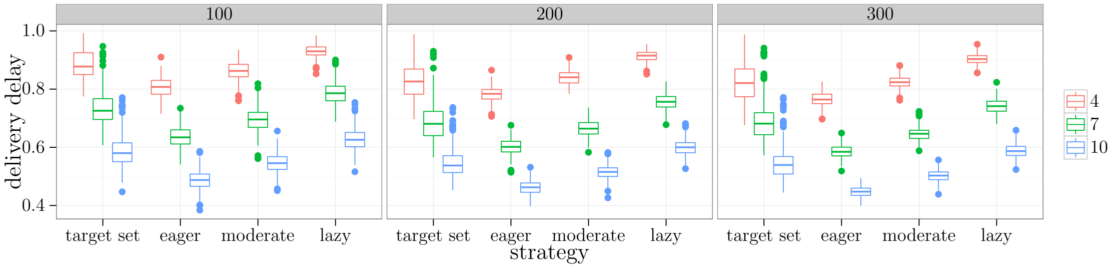
3.5.2 Results

The simulation results of the Haggle dataset are shown in the form of boxplots[3] [138] in Figures 3.3a and 3.3b, respectively. Results with different numbers of interested users (10, 20, and 30 interested users out of the 98 nodes) and content delivery deadline (200, 350, and 500 seconds) are compared. As noted in Section 3.5.1.2, the target-set strategy was enhanced to eliminate the cases in which the size of the target set is greater than the number of interested users; this allows us to access the performance of the strategies more objectively.

[3] Boxplots show the second quartile (i.e., the median) along with the first and third quartiles (i.e., 25% and 75%) in the middle box. The whiskers extend to the extrema within 1.5 times of the inter-quartile range (i.e., the distance between the first and third quartiles). Data beyond the end of the whiskers are outliers and plotted as points [138].



(a)



(b)

Figure 3.4.: Download ratio and (normalized) delivery delay of the NUS dataset. Results with different numbers of interested users (100, 200, and 300 interested users out of the 1,000 nodes) and content delivery deadline (4, 7, and 10 hours) are compared. For the patience strategies, a downloading decision is made every 1 hour.

An ideal offloading strategy has a small download ratio and short delivery delay (Section 3.5.1.3). In reality, these two goals are usually not compatible. This is evident in the three variants of the patience-based strategy: While the eager variant has the shortest delivery delay (Figure 3.3b at the expense of largest download ratio (Figure 3.3a), the lazy variant has the opposite performance trade-off and the moderate variant comes in between. Patience-based strategy, through its parameters (e.g., Table 3.1), provides a control over the trade-off between a small download ratio and short delivery delay.

One type of content that benefits the most from the situation awareness in our patience-based offloading strategy is the content that needs to be delivered quickly. One example is the content that expires after 200 seconds in Figures 3.3a and 3.3b: All variants of the patience-based strategy deliver the content with a significantly lower celluar download ratio and delivery delay than that of the target-set strategy. In this case, an interested user who chooses to wait for content is very likely to either 1) receive the content quickly from other users through the proximate channel or 2) do not receive the content till the content delivery deadline. For the latter case, the patience-based offloading strategy allow those users to realize that they are unlikely to receive the content from others (i.e., become *impatient*) and, hence, download the content directly. In contrast, the same group of users will wait the end of content delivery deadline to download the content under the target-set strategy. This corresponds to the shorter delivery delay of the patience-based strategy in Figure 3.3b.

For a similar reason, the benefit of situation awareness, which is the essence of the patience function (Equation (3.5)), is more pronounced for the type of content with fewer interested users: The sparsity of the interested users will make those interested users to have a higher probability than their (probably uninterested) neighbors to download the content (through the interest aggregation in Equation (3.3)); the interested users who have not received the content will become impatient in waiting and hence download the content sooner than they otherwise would under the target-set strategy.

For the popular content that stays fresh longer, such as the one that is interested by 30 users and has content freshness of 500 seconds, the target-set strategy may have a smaller (i.e., better) download ratio than some variants of the patience-based strategy, as shown for the case of eager variant with 30 interested users in Figure 3.3a. An examination of corresponding cases in Figure 3.3b suggests that this advantage of the target-set strategy is attained at the expense of a significantly longer delivery delay (3 times as long as that of the patience-based strategy). It also suggests that, in these situations (i.e., many interested users and long content freshness), having only a few users to download the content initially and allow the content to propagate through the proximate channel could greatly reduce cellular downloading cost. The patience-based strategy could be optimized for these situations by tuning the parameters to have a small maximal downloading probability (Equation (3.5)), as demonstrated by the lazy variant of patience-based strategy in Figure 3.3a.

Although it is not evident from Figures 3.3a and 3.3b, we note that, unlike the target-set strategy that requires collection of users' contact traces for offline training (to find the target set), the patience-based strategy only requires exchange of information between opportunistically encountered users while achieving comparable, or even better, performance: The patience-based strategy is localized and online. The benefit of the patience-based offloading is that it is more less intrusive to users, has lower maintenance overhead for the service provider, is more scalable, and adapts easily to the preference and connection changes among the users.

Comparable results on the NUS dataset are shown in Figures 3.4a and 3.4b. Despite the increase of scale (from around 100 nodes in Haggle to $1,000$ nodes in NUS) and trace regularity (NUS is synthesized from class schedules and roasters, as described in Section 3.5.1.2), the observation and discussion on performance trade-off and the benefits and limitation of the patience-based strategy drawn from Haggle still hold for NUS.

3.6 Related Work

Mobile data offloading, or mobile cellular traffic loading, is about the trade-off between the persistent but expensive cellular links and the intermittent but cheap (often free) local links. Balasubramanian et al. [21] and Lee et al. [123] conducted empirical studies, and confirmed the feasibility of offloading cellular traffic through intermittent Wi-Fi links in urban vehicular and pedestrian settings, respectively. Han et al. [95] proposed using Bluetooth to offload cellular traffic. The follow-ups [94, 96] formulated mobile data offloading as a target-set selection problem [173], and proved the approximation bound of a greedy approximation algorithm [117]. Ioannidis et al. [108] proved the convexity of the "timely content distribution over mobile social network" problem and studied how the average age of content changes when the number of users increases. Our work complements their contributions by studying the distribution of topical content and modeling users' content preference and time-varying patience for content.

The concept of centrality, which originated in sociology to measure relative importance of social actors, has been applied in studying computer networks. Borgatti [28] surveyed common definitions of the centrality concept (degree, closeness, betweenness, and eigenvector [26]). Hui et al. [107], among others, used centrality as a hint for routing in delay-tolerant networks. Kim and Anderson [118] adapted centralities to temporal-evolving graphs. The significant overhead of gathering information to compute traditional socio-centric centralities prompts researchers to investigate alternative ego-centric centralities, especially ego-centric betweenness centrality [73]. A finding from these investigations is that although the socio-centric and ego-centric versions of the betweenness centrality do not usually match in raw values, they often agree in relative ranking [143]. Brandes, in seeking a faster algorithm to compute the (socio-centric) betweenness centrality, implicitly extends betweenness centrality to weighted graph [30]. Based on the regularity of human mobility pattern [91, 182, 172], we, adapting the ideas of Nanda and Kotz [143] and Brandes [30], define a weighted ego-centric betweenness centrality to help users locally decide their relative temporal topological importance.

Users' content preference was previously considered in the context of content-centric routing [39] and publisher/subscriber architecture [124]. Given the preference variance for the large number of cellular subscribers, it is also relevant for mobile data offloading. Routing through proximity links is not a focus of this work, and we assume flooding. We include content preference in our model, discuss its interplay with social importance and bounded delay tolerance, and provide a method to consolidate them in an adaptive probabilistic cellular offloading strategy.

3.7 Summary and Future Work

In offloading topical cellular content, the virtue of patience is to allow the more capable to have better chances of serving the common good. The patience function (Equation (3.5)) shows one approach to locally synthesizing topological importance and content demand for better offloading efficiency. The simulation results suggest that properly involving topologically important, but disinterested, users in downloading and forwarding content helps in reducing cellular traffic.

These are just the beginnings; plenty of work is left to be done. Enforcement and incentive are two important issues to be further studied once the offloading framework is established. Other practical issues, like packetization, buffer management, and node churning, are omitted in the current work for simplicity, but are unavoidable in real-world implementations.

4 TEMPORAL COVERAGE BASED CONTENT DISTRIBUTION IN HETEROGENEOUS SMART DEVICE NETWORKS

This chapter abstracts prioritized defense deployment (Chapter 2) and mobile cellular offloading (Chapter 3) into a "content distribution problem using opportunistic links" problem, and extends the consideration to a scenario in which *not* all devices have cellular data channel available due to device limitation or cost concerns, but proximate channel can be established between any pair of devices that are close to each other. A piece of content is injected into the network through those devices with cellular channel (the "seeds"), and the objective is to deliver the content to all devices (i.e., full coverage) over opportunistic proximate links with few duplication (i.e., low propagation cost). The essence of this chapter, in light of the other chapters, is to propose 1. a proximate channel opportunistic connectivity quality metric, *temporal coverage quality*, using kernel-density estimation (KDE), 2. and a distributed algorithm that elects a *temporal covering set* (similar to T-dominating set in Chapter 2) using temporal coverage quality that, intuitively, "grows from the seeds." The resulting propagation rule is to restrict forwarding within the (smaller) temporal covering set, instead of the whole network.

This chapter is previously published as a conference paper [155] in IEEE International Conference on Communications (ICC), 2015.

4.1 Introduction

In this chapter, we consider *content distribution* in *heterogeneous* smart device networks. By "heterogeneous smart device network," we mean a collection of smartphones/tablets in which the *cellular data channel* is available on only *some* devices, but *all* devices can transfer data through the *proximate channel* such as Bluetooth/NFC/Wi-Fi Direct. In other words, such networks consist of a few constantly available links to a set of nodes (backed by the cellular channel) and many intermittently available links between (potentially) all the nodes in the network (backed by the proximate channel and defined by the mobility of the nodes). The heterogeneity models the common scenario that the cellular data channel is not available on many tablets (due to the lack of 3G/4G cellular transceivers) or some smartphones (explicitly disabled by their users due to cost or security concerns). On these devices, when the (infrastructural) Wi-Fi channel is not available, the proximate channel is the only means of data communication.

We consider the common scenario in which the mobility of the network nodes, although cannot be predicted precisely, nevertheless have regularity [111, 186, 187]. Examples of such networks are all the smart devices of regular students and faculty/staff members on a university campus or of employees on an enterprise site. In fact, given a densely populated and frequently visited area, the set of smart devices owned by frequent visitors often exhibit such encounter regularity.

Content distribution in smart device networks have multiple applications, two of which are mobile cellular data offloading [95, 96, 154] and prioritized defense deployment in enterprise networks [153]. In these applications, a piece of data (e.g., user-subscribed content in mobile cellular data offloading or vulnerability patches in prioritized defense deployment) is injected into or collected from the network through the cellular channel and is propagated among the nodes in the network through the proximate channel. A common objective is to minimize monetary or energy costs by reducing the number of times the content is downloaded through the cellular channel or duplicated through the proximate channel. Moreover, due to objective (e.g., the high costs or absence of a central coordination mechanism [154]) and subjective (e.g., privacy concerns [153]) constraints, it is desirable that the content distribution process emerges from the collective effect of *localized* forwarding decision made by intermediate nodes without central coordination. These settings are formulated in Section 4.2.1.

The key problem addressed in this work is how to exploit content's delay tolerance for more cost-effective content distribution (i.e., fewer copies over the proximate channel are considered to be more cost-effective) in a heterogenenous smart device network. The key ideas of this work towards addressing this problem are proposing:

- a temporal-spatial structural property (*temporal coverage*) of heterogeneous smart device networks that exploits the temporal regularity of proximate encounters in such networks for effective content distribution.
- algorithms that distributedly (i.e., each device runs the algorithm without central coordination) elect a temporal covering "backbone" from such networks, based on devices' local proximate encounter records.

To the best of our knowledge, novelties and contributions of our work include:

- Unlike existing applications of content distribution in smartphone network [94, 95, 96, 154, 153], in which the network nodes are homogeneous with regard to cellular data communication capability (i.e., all nodes can push/pull data from cellular links at will), we consider the more challenging heterogeneous setting, in which nodes without cellular links can only receive/send data through proximate channels such as Bluetooth/NFC/Wi-Fi Direct.
- We define the concept of temporal coverage based on quantitative metrics of proximate channel's temporal quality using kernel-density estimation (KDE), which preserves certainty about such estimation that is otherwise lost in simpler statistical metrics such as average or expected [153] inter-encounter interval.
- We propose localized algorithms (Section 4.2.3) that distributedly elect temporally covering nodes without central coordination.
- We verify the proposed algorithm's effectiveness for content distribution in heterogeneous smart device networks with simulations using real public Bluetooth encounter traces.

4.2 Design

4.2.1 Problem Formulation

The problem discussed in Section 4.1 can be formulated as follows. Let U be a set of nodes in a heterogeneous smart device network, U_c and $U_{\bar{c}}$ be the sets of nodes with and without cellular data channel, respectively: $U_c \cup U_{\bar{c}} = U$ and $U_c \cap U_{\bar{c}} = \emptyset$. In this network:

- Content are generated outside the network and injected into the network through the cellular channel, i.e., the nodes with cellular data links U_c (i.e., the "seeds" hereafter) are the interface between network and the outside Internet.
- Content can be forwarded between two devices when they move close enough to establish a proximate channel, in which case we say they encounter with each other.
- Every $u \in U$ records its past encounters with other nodes, say, with $v \in U$: $[s_1^{u,v}, e_1^{u,v}], \ldots, [s_{k_{u,v}}^{u,v}, e_{k_{u,v}}^{u,v}]$ $(s_1^{u,v} < e_1^{u,v} < \ldots < s_{k_{u,v}}^{u,v} < e_{k_{u,v}}^{u,v})$, in which u encounters v (and hence u can send data to v through the proximate channel) during the time windows $[s_i^{u,v}, e_i^{u,v}]$ (for $i \in \{1, \ldots, k_{u,v}\}$); conversely, no content can be forwarded between u and v during $(e_i^{u,v}, s_{i+1}^{u,v})$ (for $i \in \{1, \ldots, k_{u,v} - 1\}$) due to the lack of communication channel.

The objective is to minimize the content distribution cost, defined as the number of times the content is forwarded from one device to another through the proximate channel, without central coordination.

The challenge of achieving cost-effective content distribution in heterogeneous smart device networks can be better understood by considering the following schemes. In each case, node u has obtained the content and is deciding whether to forward the content to the nodes it encounters in the future.

Eager multiple forwarding. u forwards the content once to every node it encounters. This is known as flooding or epidemic routing in literature [226]. The overall delivery delay is minimized. However, the delivery cost can be higher than necessary. Nevertheless, it envelops the proximate-channel-based data propagation process from the outside—no data propagation through the proximate channel can deliver the content faster than the eager multiple forwarding.

Eager k forwarding. u forwards the content once to the first k nodes it encounters [186]. The delivery cost is bounded from above by $k|U|$: Each node forwards the content at most k time. Delay at each intermediate nodes is also minimized. Depending on proximate encounter opportunities and the choice of k, eager k forwarding's performance ranges from eager single forwarding [187] to eager multiple forwarding [187]. However, since proximate encounter opportunities are often not uniform among nodes, it is difficult, if not impossible, to find a (global) k that performs optimally.

Random forwarding. Upon encountering another node v, u makes a random decision of whether to forward the content to v. u will forward the data to v at most once to avoid duplication. If the random decision is unbiased (i.e., equal chance of forwarding/not forwarding), the delivery cost is halved comparing with eager multiple forwarding, while random forwarding does not suffer from the delivery failure as in eager single forwarding. The relative delivery cost to eager multiple forwarding can be tuned by adjusting the forwarding decision's odds: Lower forwarding odds correspond to lower delivery costs. However, it is not clear how to optimally tune the forwarding odds without global coordination or the ability to detecting in-network content saturation as required by more sophisticated adaptive random forwarding schemes (e.g., the work by Liu and Wu [129]), which are not generally available.

4.2.2 Temporal Quality Metrics

In light of the schemes discussed at the end of Section 4.2.1, our key idea of improvement towards cost-effective content distribution (Section 4.2.2.3) is to apply these forwarding rules to, instead of the full network, a restricted set of nodes that we call a *temporal covering set* (Section 4.2.2.2). Intuitively, a temporal covering set is a proximate-channel content distribution backbone with *strong* internal connectivity and *full* external coverage of the whole network. Both the internal connectivity and external coverage are defined on quantitative *temporal quality* metrics of proximate channels based on the readily available encounter records.

4.2.2.1 Temporal Quality of Proximate Channels

Based on its past encounters $[s_1^{u,v}, e_1^{u,v}], \ldots, [s_{k_{u,v}}^{u,v}, e_{k_{u,v}}^{u,v}]$ with v, u can estimate the *temporal quality* of its proximate channel with v, in terms of the proximate channel's potential of forwarding the content timely.

A straightforward idea is to use *average inter-encounter interval*, defined as

$$\frac{1}{k_{u,v} - 1} \sum_{i=1}^{k_{u,v}-1} \left(s_{i+1}^{u,v} - e_i^{u,v}\right).$$

A smaller average inter-encounter interval between two regularly encountered nodes indicates that content are more likely can be forwarded from one node to the other timely, and hence their (opportunistic) proximate channel is of a better temporal quality.

However, as will be discussed shortly, average inter-encounter interval fails to capture the *certainty* about proximate channel quality and can lead to counter-intuitive results. Therefore, we propose the following temporal quality metric of proximate channels based on kernel density estimation (KDE).

A KDE[1] $\hat{f}(x)$ of u's inter-encounter intervals to v, with the Epanechnikov kernel $K(x) = \frac{3}{4}(1 - x^2)\mathbf{1}_{|x|\leq 1}$ [71], is:

$$\hat{f}_{u,v}(x) = \frac{1}{k_{u,v} - 1} \sum_{i=1}^{k_{u,v}-1} K(x - (s_{i+1}^{u,v} - e_i^{u,v})), \tag{4.1}$$

in which $\mathbf{1}_{|x|\leq 1}(x)$ is the indicator function on the set $\{x| - 1 \leq x \leq 1\}$ that equals to 1 when $|x| \leq 1$ and equals to 0 otherwise. Then, the *T-coverage (temporal) quality* $d_u^T(v)$ of u's proximate channel to v is defined as:

$$d_u^T(v) = \int_{-\infty}^{T} \hat{f}_{u,v}(x)\mathrm{d}x. \tag{4.2}$$

As a special case, if u has never encountered v, or no inter-encounter interval is less than the parameter T, $d_u^T(v)$ is defined to be 0. By Equation (4.2), $0 \leq d_u^T(v) \leq 1$ and $d_u^T(v) = d_v^T(u)$. Note that $d_u^T(v)$ can be computed locally by u from information readily available to u, i.e., its encounter records with v.

T in Equation (4.2) is a time-domain *quality threshold* parameter that is used to filter out sporadic or long-delay opportunistic links between nodes. Without T as the integral upper bound, the integration (Equation (4.2)) of the kernel $\hat{f}_{u,v}$ (Equation (4.1)) from $-\infty$ to ∞ would always be 1 by the definition of smoothing kernels, and thus cannot be used to compare temporal quality of their proximate channel. In contrast, integration from $-\infty$ up to T in Equation (4.2) endows the temporal quality metric $d_u^T(v)$ the semantics of an estimation of the probability that u will encounter v at least once within a time window of T. Greater $d_u^T(v)$ translates to a better chance that u can deliver content to v *timely* through their opportunistic proximate channel.

Comparing with average inter-encounter interval, KDE-based proximate channel quality estimation (Equations (4.2)) is more nuanced. To see this, consider an example with time unit of seconds, $T = 110$, and 10 groups of inter-encounter interval

[1] We deliberately omit the "smoothing bandwidth" parameter, often denoted by the symbol h, to simplify the (already complex) notation; it is understood that a default smoothing bandwidth (e.g., the R implementation of KDE specifies the algorithm of computing the smoothing bandwidth from inputs) is used here.

Table 4.1.: KDE-based T-coverage temporal quality (with $T = 110$) $d_u^T(v)$ (Equation (4.2)) of u's proximate channel to v can capture the quality differences of the different groups shown in Figure 4.1.

i in 2^i	$d_u^T(v)$	i in 2^i	$d_u^T(v)$
1	0.293	6	0.346
2	0.303	7	0.360
3	0.312	8	0.375
4	0.323	9	0.391
5	0.334	10	0.410

records: Group i ($i \in \{1, 2, \ldots, 10\}$) consists of 2^i pairs of interleaved 100 and 200, i.e., u encounters v with periodic intervals of $100, 200, 100, 200$, etc.. The average inter-encounter interval for all 10 groups has the same value of 150, which is greater than the proximate channel quality threshold $T = 110$. This suggests that the quality of these proximate channels does not meet expectation. However, the fact that "u periodically encounters with v in 100 seconds" suggests otherwise.

In contrast, Figure 4.1 and Table 4.1 show that KDE-based proximate channel quality $d_u^T(v)$ preserves more temporal quality information about the opportunistic proximate channel between u and v (i.e., producing a continuous, rather than binary, degree of satisfying quality expectation with regards to T) and captures the differences in temporal quality of the proximate channel between these groups *in a single number*: Proximate channel temporal quality derived from a group with $2 \times 2^{10} = 2048$ data points (0.410 from Table 4.1) is intuitively better (i.e., more robust) than the estimation that is derived from a group with only $2 \times 2^1 = 4$ data points (0.293 from Table 4.1).

4.2.2.2 Temporally Covering Set

For a pair of nodes $u, v \in U$, if the T-coverage quality $d_u^T(v) > 0$, we define a directed edge from u to v with a weight of $d_u^T(v)$—in this case, we say that u T-*covers*

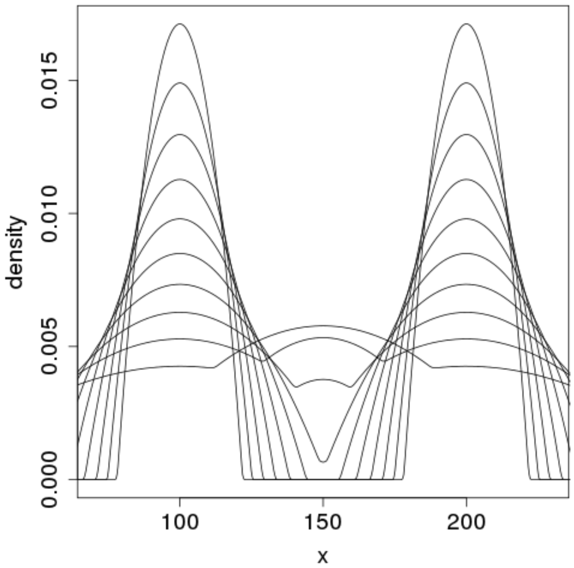


Figure 4.1.: The kernel density estimation (KDE; Equation (4.1)) for 10 groups of inter-encounter interval records with 2^i pairs of interleaved 100 and 200 in group i. The emerging dual peaks with increasing i correspond to increasing certainty about the density distribution of the nodes' inter-encounter intervals. KDE preserves u's certainty about the temporal quality estimation of its proximate channel with v that is otherwise lost in average inter-encounter interval.

v or, equivalently, v is T-covered by u. These edges on U define a directed weighted graph, which we also denote as U when there is no ambiguity in the context. A set of nodes $\mathcal{D}_T \subset U$ is a *temporally covering set with temporal threshold of* T ("T-covering set" for brevity) if:

- (Coverage) For each node $u \in U$, either $u \in \mathcal{D}_T$ or there is a node $v \in \mathcal{D}_T$ such that u is T-covered by v.
- (Connectivity) For each node $u \in \mathcal{D}_T$, either u is a seed (i.e., $u \in U_c$), or there is a seed $v \in U_c$ (i.e., v is equipped with cellular data channel) such that there is a path (i.e., a chain of consecutively T-covered nodes) from v to u.

The nodes $\mathcal{D}_T$ are the T-dominators (or simply "dominators"), and the nodes that are T-covered by other nodes the T-dominatees (or simply "dominatees"), i.e., "dominators T-cover dominatees." By Connectivity, non-seed dominators are also dominatees.

4.2.2.3 Temporal Coverage Based Content Distribution

Coverage and Connectivity, coupled with the interpretation of T-coverage temporal quality as the probability of timely encounters, essentially make a T-covering set a virtual backbone for content distribution in a heterogeneous smart device network. More concretely, if we restrict the "eager multiple forwarding" rule (Section 4.2.1) to the T-covering set (i.e., only T-dominators will forward data):

- Connectivity dictates that each dominator can receive the content through a chain of dominators from the seeds (where the content is injected or collected).
- Coverage dictates that each non-dominator shall be directly reachable from a dominator timely.

Therefore, this temporal coverage based content distribution scheme (i.e., eager multiple forwarding restricted to a T-covering set) allows content to be delivered from seeds to any node with only timely encountered nodes serving as intermediaries. Intuitively, this allows more cost-effective content delivery (since the T-covering set is a subset of the whole network) than eager multiple forwarding, without incurring delay penalty (for delaying content delivery to destinations too much).

In this scheme, delivery cost is positively associated with the size of the T-covering set (number of dominators and the edge density of the covering set). Therefore, the localized dominator election algorithms presented next make efforts to reduce the number of elected dominators.

4.2.3 Algorithm

The core of our solution to cost-effective content distribution in heterogeneous smart device networks is the following *localized* dominator election algorithm, in

Algorithm 1 u's local decision process on whether to change its dominator status when u encounters v after they have exchanged information.

```
1:      ▶ only consider quality opportunistic links
2:  if $d_u^T(v) > 0$ then
3:          ▶ $u$ updates $L_\uparrow(u)$ and $L_\downarrow(u)$
4:      if $v$ is a dominator then
5:          if $v \in U_c$ or $u$ is a non-dominator then
6:              $L_\uparrow(u) \leftarrow L_\uparrow(u) \cup \{v\}$
7:              $L_\downarrow(u) \leftarrow L_\downarrow(u) \setminus \{v\}$
8:          end if
9:          for $w \in L_\downarrow(u)$ do
10:             if $d_v^T(w) > d_u^T(w)$ then   ▶ if $w$ is better $T$-dominated by $v$ than by $u$
11:                 $L_\downarrow(u) \leftarrow L_\downarrow(u) \setminus \{w\}$
12:             end if
13:         end for
14:     else if $v$ is a non-dominator then
15:         $L_\uparrow(u) \leftarrow L_\uparrow(u) \setminus \{v\}$
16:         $x \leftarrow$TRUE   ▶ $x =$TRUE if $u$ is $v$'s best $T$-dominator
17:         for $w \in L_\uparrow(u)$ do
18:             if $d_v^T(w) > d_u^T(v)$ then   ▶ if $v$ is better $T$-dominated by $w$ than by $u$
19:                 $x \leftarrow$FALSE
20:                 go to 23
21:             end if
22:         end for
23:         if $x =$TRUE then   ▶ $u$ is $v$'s best dominator
24:             $L_\downarrow(u) \leftarrow L_\downarrow(u) \cup \{v\}$
25:         end if
26:     end if
27:         ▶ $u$ sets its dominator status based on whether $L_\downarrow(u)$ and $L_\uparrow(u)$ are empty
28:     if $u \notin U_c$ then   ▶ only non-seeds change dominator status
29:         if $L_\downarrow(u) = \emptyset$ then
30:             $u$ turns a non-dominator
31:         else if $L_\uparrow(u) \neq \emptyset$ and $L_\downarrow(u) \neq \emptyset$ then
32:             $u$ turns a dominator
33:         end if
34:     end if
35: end if
```

which the nodes, instead of being coordinated centrally, turn themselve into dominators/non-dominators based on the information they gather from their encounters with other devices. In the algorithm, each node u locally maintains two lists about other nodes: the *upstream list* $L_\uparrow(u)$ and *downstream list* $L_\downarrow(u)$.

Initially: all the seeds U_c (i.e., nodes that are equipped with cellular channel) turn dominators and will remain so throughout the election process; all the non-seeds $U_{\bar{c}}$ (temporally) turn non-dominators and may turn dominiators by localized election later. Both $L_\uparrow(u)$ and $L_\downarrow(u)$ are both initially empty (i.e., $L_\uparrow(u) = L_\downarrow(u) = \emptyset$) for every node u in the network.

When u encounters v, u first updates its T-coverage quality $d_u^T(v)$ (which, as discussed after Equation (4.2), equals to $d_v^T(u)$), and then carries out the following information exchange procedure:

- u sends its seed/dominator status to v.
- u sends $L_{\uparrow}(u)$ and $L_{\downarrow}(u)$ to v.
- u receives $\{d_v^T(w)|w \in L_{\downarrow}(u)\}$ and $\{d_v^T(w)|w \in L_{\uparrow}(u)\}$ from v.

v follows the same procedure by swapping the symbols u and v. The amount of exchanged information is linear to the number of nodes that they have encountered in the past (rather than the size of the network unless the network is dense) and can be, for example, piggy-backed to periodic beacons[2].

After the information exchange, u has all the information needed to independently carry out Algorithm 1, in which u will turn a dominator if its updated upstream/downstream lists are both non-empty, and will turn a non-dominator if its downstream list is empty. v follows the same procedure by, again, swapping the symbols u and v in Algorithm 1.

The essence of Algorithm 1 is that:

- u will include a dominator v as (one of) its upstream $L_{\uparrow}(u)$ (line 6) if u thinks (based its local information after the exchange) that v connects u to one of the seeds, i.e., for Connectivity. If so, u will in turn consider delegate its downstream $w \in L_{\downarrow}(u)$ to v (lines 9–13) if v is a strictly better dominator (defined by the relation $d_v^T(w) > d_u^T(w)$ on line 10).
- u will include a non-dominator v as (one of) its downstream $L_{\downarrow}(u)$ (line 24) if u thinks (again, based on its local information) that none of u's upstream dominates $L_{\uparrow}(u)$ strictly better than u does (the logic on lines 16–25).
- The “strictly better” comparison (lines 10 and 18) prevents two nodes from mutually delegating the dominator responsibility for a third node w to each other and thus leaves w (wrongfully) uncovered.

[2] See, for example, Wu's discussion [214] on the implementation of this information exchange through periodic beacons.

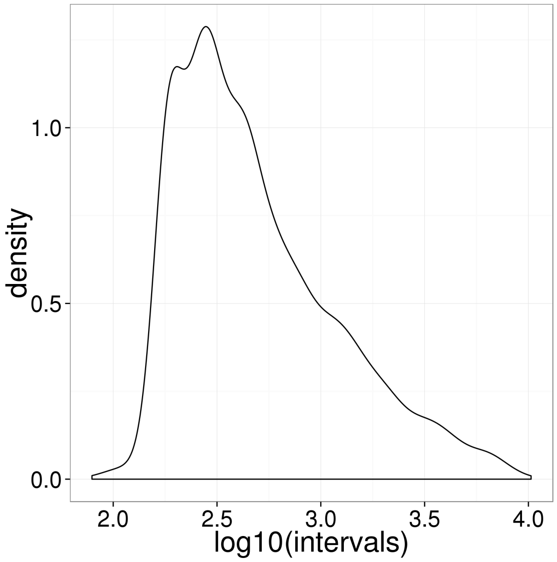


Figure 4.2.: Smoothed density distribution of intervals between consecutive encounters in the sigcomm2009 dataset.

Thus, Connectivity (i.e., content coverage) is maintained and the number of dominators (i.e., delivery cost) is also reduced.

4.3 Experiment

4.3.1 Datasets and Setup

We use the publicly available Bluetooth encounter dataset sigcomm2009 [158, 157], downloaded from the CRAWDAD wireless dataset archive[3]. The raw trace (the "proximity.csv" trace in the dataset) consists of timestamped periodic Bluetooth proximity device discovery records of 76 users during the SIGCOMM 2009 conference.

Based on the meta-data, we filter out sporadic devices (those with ID over 100 in the dataset), and transform the periodic scanning records into encounter events ("sessions"). Specifically, since the devices make a scanning every 120+/-10.24 seconds

[3] http://crawdad.cs.dartmouth.edu/thlab/sigcomm2009/

(randomized) for 10.24 seconds, we combine consecutive scanning records between a pair of devices within that time window into the same session. Moreover, only 48 out of the 76 nodes regularly meet each other up to the trace timestamp of about $12,500$ (out of timestamp of up to about $35,000$, after which the recorded encounter are sporadic and the performance curves shown below go flat). Therefore, we zoom in to that segment of trace to show the details of the results below.

Figure 4.2 shows the (smoothed) density distribution of inter-encounter intervals (delay between two consecutive sessions for a pair of nodes) of the devices. Note that the x axis is in logarithmic scale. In the results shown below, we set the temporal quality threshold T to $1,000$ (corresponding to the x axis value of 3.0 in Figure 4.2) to include enough temporal-spatial links without obliterating the quality value: As briefly discussed in Section 4.2.2.1, choosing a too small or too small threshold T would lead to the quality metric $d_u^T(v)$ to be all 0 (for too small T) or 1 (for too large T) and hence cannot capture the temporal quality of different proximate channels. The results below show that our choice of $1,000$ does a fair job in capturing such quality differences. A general heuristics is left for future work.

We simulate the content distribution processes with the data forwarding rules discussed in Section 4.1: eager multiple forwarding (emulti), eager single forwarding (esingle), and random forwarding with a 50% forwarding chance at each encounter (random 50). As for the proposed algorithm, we consider the T-coverage-based forwarding (Section 4.2.2.3; tdom) and a variant of T-coverage-based forwarding with the dominator has a 50% chance of forwarding at each encounter (tdom50) to be comparable with random50.

4.3.2 Simulations and Results

Figures 4.3 and 4.4 show the average coverage (the number of nodes that receive the content) and delivery cost (the number of times the content get sent from one node to another) normalized with emulti (i.e., by arithmetic division of the raw numbers)

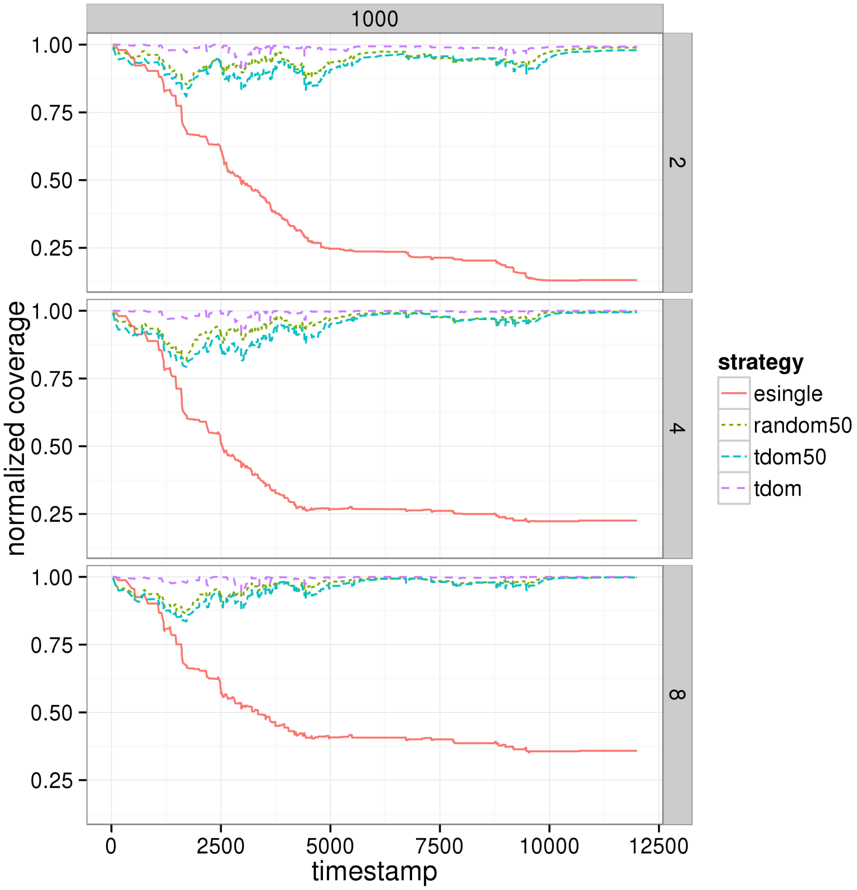


Figure 4.3.: Average content distribution coverage normalized by the eager multiple forwarding scheme with different numbers of seeds over 100 random runs. The row headings show the number of seeds and the column heading shows the temporal quality threshold T for the temporal coverage based schemes. Scheme notation: esingle (eager single forwarding), random50 (50% random forwarding), tdom50 (50% random T-coverage-based forwarding), tdom (T-coverage-based forwarding).

Table 4.2.: Average content delivery delay comparing to the eager multiple forwarding scheme with the same settings and notation as in Figure 4.3.

	esingle	random50	tdom50	tdom
2	5577	271	397	81
4	5530	199	306	29
8	4725	173	241	25

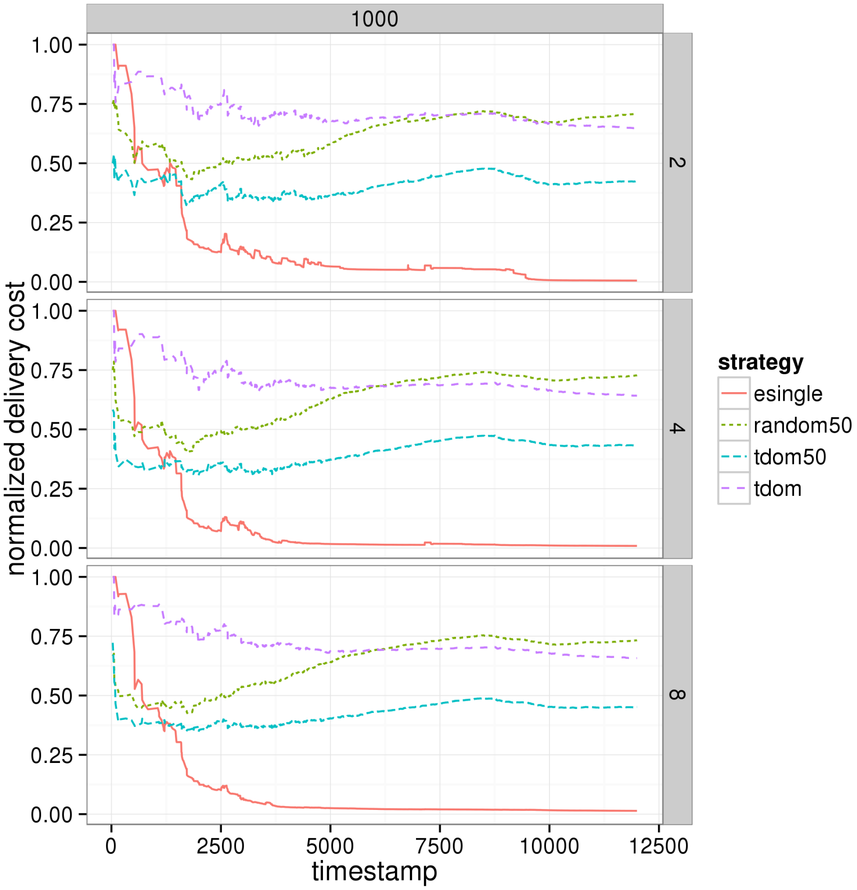


Figure 4.4.: Average content delivery cost normalized by the eager multiple forwarding scheme with the same settings and notation as in Figure 4.3.

throughout the content distribution processes for different numbers of seeds over 100 random runs; Table 4.2 shows the average content delivery delay comparing to emulti with the same settings. In computing the delays, nodes that have not received the content by the end of the content distribution process (there are many such instances for esingle) are considered to receive the content at the last timestamp; otherwise, the indefinite delay could not be used for computing the average delay.

Since emulti envelops the proximate-channel-based content distribution process from the outside (as discussed in Section 4.2.1), normalizing the results with emulti in Figures 4.3 and 4.4 and in Table 4.2 clearly show how each scheme exploits content's delay tolerance to improve content delivery costs. The results indicate that, by restricting the eager multi forwarding rule to the locally elected temporal covering set, tdom reduces content delivery cost by 25% (Figure 4.4) with minimum delays (Table 4.2) and little sacrifice in coverage (Figure 4.3) comparing with alternatives such as random50 and esingle. Moreover, if modest delays and coverage loss are allowed, tdom50 can be applied to reduce the delivery cost of random50 by another 25%. In summary, these results show that the temporal covering set is a cost-effective (virtual) content distribution backbone for heterogeneous smart device networks.

4.4 Related Work

The work is motivated by extending applications of content distribution in homogeneous smart device networks (in which *all* nodes have cellular data capability that can be activated on demand) such as mobile cellular [95, 96, 154] and enterprise network defense prioritization [153] to heterogenenous networks (in which only *some* nodes have cellular data capability). One challenge of the heterogeneous setting is the requirement of Connectivity to seeds. In particular, the concept of temporal coverage is inspired by the work on enterprise network defense prioritization [154], which extends previous works [213, 220, 177] on (spatially) connected dominating set (CDS) based routing in ad hoc network (MANET) to the temporal dimension, by exploiting the regularity [111, 186, 187] exhibited by many proximity-channel-based smartphone networks, as a prominent application of delay-tolerant networks (DTNs) [75, 107] that have received significant research in the past decade.

4.5 Summary and Future Work

We propose temporal coverage based content distribution to effectively exploits content's delay tolerance for reducing content distribution costs in heterogeneous smart device networks. KDE is used to process readily available encounter records to capture the temporal quality of the proximate channel that eludes simpler measurements such as the average inter-encounter interval. Using real Bluetooth encounter traces, we demonstrate that temporal coverage based content distribution significantly cuts content delivery cost with minimal delay and no sacrifice in coverage. Future work includes implementing the proposed methodology and deploying it in real heterogeneous smart device network for further insights.

5 BEHAVIORAL MALWARE DETECTION IN DELAY TOLERANT NETWORKS

Taking an angle different from Chapters 2, 3, and 4, which focus on facilitating distribution of useful content (e.g., vulnerability patches in Chapter 2 and user-subscribed content in Chapter 3), this chapter considers the opposite problem of preventing the distribution of unwanted content over opportunistic proximate links. Using a probabilistic behavioral model for proximity malware (malware that propagates through proximate links), this chapter addresses several challenges of sequential evidence collection that are unique to the decentralized and opportunistic nature of the proximate channel, such as balancing between infection risk and service loss, and handling potentially false indirect evidence in evidence sharing.

This chapter is previously published as a journal article [156] in the January 2014 issue of IEEE Transactions on Parallel and Distributed Systems (TPDS), as an extension to the conference paper, “Behavioral Detection and Containment of Proximity Malware in Delay Tolerant Networks” [151], that is published in IEEE International Conference on Mobile Ad-hoc and Sensor Systems (MASS), 2011.

5.1 Introduction

The popularity of mobile consumer electronics, like laptop computers, PDAs, and more recently and prominently, smartphones, revives the delay-tolerant-network (DTN) model as an alternative to the traditional infrastructure model. The widespread adoption of these devices, coupled with strong economic incentives, induces a class of malware that specifically targets DTNs. We call this class of malware *proximity malware.*

An early example of proximity malware is the Symbian-based *Cabir* worm, which propagated as a Symbian Software Installation Script (.sis) package through the Bluetooth link between two spatially proximate devices [196]. A later example is the iOS-based *Ikee* worm, which exploited the default SSH password on jailbroken [80] iPhones to propagate through IP-based Wi-Fi connections [197]. Previous research [6] quantify the threat of proximity malware attack and demonstrate the possibility of launching such an attack, which is confirmed by recent reports on hijacking hotel Wi-Fi hotspots for drive-by malware attacks [122]. With the adoption of new short-range communication technologies such as NFC [146] and Wi-Fi Direct [210] that facilitate spontaneous bulk data transfer between spatially proximate mobile devices, the threat of proximity malware is becoming more realistic and relevant than ever.

Proximity malware based on the DTN model brings unique security challenges that are not present in the infrastructure model. In the infrastructure model, the cellular carrier centrally monitors networks for abnormalities; moreover, the resource scarcity of individual nodes limits the rate of malware propagation. For example, the installation package in *Cabir* and the SSH session in *Ikee*, which were used for malware propagation, cannot be detected by the cellular carrier. However, such central monitoring and resource limits are absent in the DTN model. Proximity malware exploits the opportunistic contacts and distributed nature of DTNs for propagation.

A prerequisite to defending against proximity malware is to detect it. In this chpater, we consider a general behavioral characterization of proximity malware. Behavioral characterization, in terms of system call and program flow, has been previously proposed as an effective alternative to pattern matching for malware detection [119, 23]. In our model, malware-infected nodes' behaviors are observed by others during their multiple opportunistic encounters: Individual observations may be *imperfect*, but abnormal behaviors of infected nodes are *identifiable* in the long-run. For example, a single suspicious Bluetooth connection or SSH session request *during one encounter* does not confirm a *Cabir* or *Ikee* infection, but repetitive suspicious requests *spanning multiple encounters* is a strong indication for malware infection. The imperfection of a single, local observation was previously in the context of distributed IDS against slowly propagating worms [58].

Instead of assuming a sophisticated malware containment capability, such as patching or self-healing [239, 125], we consider a simple “cut-off” strategy: If a node i suspects another node j of being infected with the malware, i simply ceases to connect with j in the future to avoid being infected by j. Our focus is on how individual nodes shall make such cut-off decisions against potentially malware-infected nodes, based on direct and indirect observations.

A comparable example from everyday experience is fire emergency. An early indication, like dark smoke, prompts two choices. One is to report fire emergency immediately; the other is to collect further evidence to make a better informed decision later. The first choice bears the cost of a false alarm, while the second choice risks missing the early window to contain the fire.

In the context of DTNs, we face a similar dilemma when trying to detect proximity malware: Hyper-sensitivity leads to false positives, while hypo-sensitivity leads to false negatives. In this chapter, we present a simple, yet effective solution, *look-ahead*, which naturally reflects individual nodes’ intrinsic risk inclinations against malware infection, to balance between these two extremes. Essentially, we extend the Naive Bayesian model, which has been applied in filtering email spams [15, 93, 224], detecting botnets [206], and designing IDSs [58, 5], and address two DTN-specific, malware-related, problems.

1. *Insufficient evidence vs. evidence collection risk.* In DTNs, evidence (such as Bluetooth connection or SSH session requests) is collected only when nodes come into contact. But contacting malware-infected nodes carries the risk of being infected. Thus, nodes must make decisions (such as whether to cut off other nodes and, if yes, when) *online* based on potentially insufficient evidence.

2. *Filtering false evidence sequentially and distributedly.* Sharing evidence among opportunistic acquaintances helps alleviating the aforementioned insufficient evidence problem; however, false evidence shared by malicious nodes (the *liars*) may negate the

benefits of sharing. In DTNs, nodes must decide whether to accept received evidence sequentially and distributedly.

Our contributions are summarized below.

1. We present a general behavioral characterization of proximity malware, which captures the functional but imperfect nature in detecting proximity malware (Section 5.2).

2. Under the behavioral malware characterization, and with a simple cut-off malware containment strategy, we formulate the malware detection process as a distributed decision problem. We analyze the risk associated with the decision, and design a simple, yet effective, strategy, *look-ahead*, which naturally reflects individual nodes' intrinsic risk inclinations against malware infection. Look-ahead extends the Naive Bayesian model, and addresses the DTN-specific, malware-related, "insufficient evidence vs. evidence collection risk" problem (Section 5.3.1).

3. We consider the benefits of sharing assessments among nodes, and address challenges derived from the DTN model: liars (i.e., bad-mouthing and false-praising malicious nodes) and defectors (i.e., good nodes that have turned rogue due to malware infections). We present two alternative techniques, *dogmatic filtering* and *adaptive look-ahead*, that naturally extend look-ahead to consolidate evidence provided by others, while containing the negative effect of false evidence. A nice property of the proposed evidence consolidation methods is that the results will not worsen even if liars are the majority in the neighborhood (Section 5.3.2). Real contact traces are used to verify the effectiveness of the methods (Section 5.4).

5.2 Model

Consider a DTN consisting of n nodes. The neighbors of a node are the nodes it has (opportunistic) contact opportunities with.

Proximity malware is a malicious program that disrupts the host node's normal function and has a chance of duplicating itself to other nodes during (opportunistic) contact opportunities between nodes in the DTN. When a duplication occurs, the other node is infected with the malware.

In our model, we assume that each node is capable of assessing the other party for suspicious actions after each encounter, resulting in a *binary* assessment. For example, a node can assess a Bluetooth connection or a SSH session for potential *Cabir* or *Ikee* infection. The watchdog components in previous works on malicious behavior detection in MANETs [135] and distributed reputation systems [140, 35] are other examples. A node is either evil or good, based on if it is or is not infected by the malware. The suspicious-action assessment is assumed to be an *imperfect* but *functional* indicator of malware infections: It may occasionally assess an evil node's actions as "non-suspicious" or a good node's actions as "suspicious", but most suspicious actions are correctly attributed to evil nodes. A previous work on distributed IDS presents an example for such imperfect but functional binary classifier on nodes' behaviors [58].

The functional assumption characterizes a malware-infected node by the assessments of its neighbors. If node i has N (pair-wise) encounters with its neighbors and s_N of them are assessed as suspicious by the neighbors, its *suspiciousness* S_i is defined as:

$$S_i = \lim_{N \to \infty} \frac{s_N}{N}. \tag{5.1}$$

By Equation (5.1), $S_i \in [0, 1]$. A number $L_e \in (0, 1)$ is chosen as the *line between good and evil.* L_e depends on the quality of a particular suspicious-action assessment and, if the assessment is a functional discriminant feature of the malware and the probabilistic distribution of the suspiciousness of both good and evil nodes are known, L_e can be chosen as the (Bayesian) decision boundary, which minimizes classification errors [66]. Node i is good if $S_i \leq L_e$, or evil if $S_i > L_e$: We draw a fine line between good and evil, and judge a node by its deeds.

Instead of assuming a sophisticated malware coping mechanism, such as patching or self-healing, we consider a simple and widely applicable malware containment strategy: Based on past assessments, a node i decides whether to refuse future connections ("cut off") with a neighbor j.

5.3 Design

In the following discussion, we investigate the decision process of a node i, which has k neighbors $\{n_1, n_2, \ldots, n_k\}$, against a neighbor j; with no loss of generality, let j be n_1.

5.3.1 Household Watch

Consider the case in which i bases the cut-off decision against j *only* on i's own assessments on j. Since only direct assessments are involved, we call this model *household watch* (the naming will become more evident by the beginning of Section 5.3.2).

Let $\mathcal{A} = (a_1, a_2, \ldots, a_A)$ be the assessment sequence (a_i is either 0 for "non-suspicious" or 1 for "suspicious") in chronological order, i.e., a_1 is the oldest assessment, and a_A is the newest one.

Bayes' theorem tells us:

$$P(S_j|\mathcal{A}) \propto P(\mathcal{A}|S_j) \times P(S_j). \tag{5.2}$$

$P(S_j)$ encodes our prior belief on j's suspiciousness S_j; $P(\mathcal{A}|S_j)$ is the likelihood of observing the assessment sequence $\mathcal{A}$ given S_j; $P(S_j|\mathcal{A})$ is the posterior probability, representing the plausibility of j having a suspiciousness of S_j given the observed assessment sequence $\mathcal{A}$. Since the evidence $P(\mathcal{A})$ does not involve S_j and serves as a normalization factor in the computation, we omit it and write the quantitative relationship in the less cluttered proportional form[1].

We have the following observations:

- By the *principle of maximal entropy* [109] (which states that, subject to known constraints, or *testable information*, the probability assignment that best represents our state of knowledge is the one which maximizes the *entropy*, as defined by Shannon [181]), before obtaining any assessment, a node i, which *holds no*

[1] When we use proportional form in this chapter, we have implicitly done the same thing.

presumption on another node j's suspiciousness, should assign a *uniform* distribution to the prior $P(S_j)$, which is:

$$P(S_j) = 1, \tag{5.3}$$

since, by definition, $S_j \in [0, 1]$. Any other assignment of $P(S_j)$ reflects prejudice that i holds against j, which is *not* warranted by our assumption on the background knowledge B.

- The independence between pairs of assessments implies the *equivalence* of *batch* and *sequential* computation for $P(S_j|\mathcal{A})$. If we apply the assessment sequentially by using the posterior of the previous round as the prior of this round, we have:

$$\begin{aligned} P(S_j|\mathcal{A}) &= P(S_j|a_1, \dots, a_A) \\ &\propto P(a_D|S_j, a_1, \dots, a_{D-1}) \\ &\quad \times P(S_j|a_1, \dots, a_{A-1}) \\ &= P(a_D|S_j) \times P(Sj|a_1, \dots, a_{A-1}) \\ &\dots \\ &\propto P(S_j) \prod_{k=1}^{D} P(a_k|S_j). \end{aligned} \tag{5.4}$$

By the definition of suspiciousness S_j and the independence among assessments, we have:

$$P(a_k|S_j) = \begin{cases} S_j & \text{for} \quad a_k = 1 \\ 1 - S_j & \text{for} \quad a_k = 0 \end{cases} . \tag{5.5}$$

By Equations 5.3, 5.4, and 5.5, we obtain Equation 5.6:

$$P(S_j|\mathcal{A}) \propto S_j^{s_{\mathcal{A}}} (1 - S_j)^{|\mathcal{A}| - s_{\mathcal{A}}} \tag{5.6}$$

in which $s_{\mathcal{A}}$ is the number of suspicious assessments in $\mathcal{A}$ (i.e., the assessments equal to 1), and $|\mathcal{A}|$ is the number of assessments collected so far.

By Equation 5.6, we can calculate the $S_j \in [0,1]$ which maximizes $P(S_j|\mathcal{A})$. Let $a = s_{\mathcal{A}}$ and $b = A - s_{\mathcal{A}}$. If $a = 0$ and $b \neq 0$, $S_j = 0$ is the maximizer; conversely, if $a \neq 0$ and $b = 0$, $S_j = 1$ is the maximizer. If both a and b are both non-zero, let $\mathcal{C}$ be the normalization constant in Equation 5.6 (which is a constant for S_j), we have:

$$\begin{aligned}
\frac{\mathrm{d}P(S_j|A)}{\mathrm{d}S_j} &= \frac{\mathrm{d}}{\mathrm{d}S_j}\left(\mathcal{C}S_j^a\sum_{k=0}^{b}\binom{b}{k}(-S_j)^k\right) \\
&= \mathcal{C}aS_j^{a-1}\sum_{k=0}^{b}\binom{b}{k}(-S_j)^k \\
&\quad - \mathcal{C}bS_j^a\sum_{k=0}^{b-1}\binom{b-1}{k}(-S_j)^k \\
&= \mathcal{C}S_j^{a-1}(1-S_j)^{b-1}\left(a(1-S_j)-bS_j\right).
\end{aligned}$$

The unique $S \in (0,1)$ which makes $\frac{\mathrm{d}}{\mathrm{d}S_j}P(S_j|A) = 0$ is the S_j which satisfies $a(1 - S_j) - bS_j = 0$, i.e., $S_j = \frac{a}{a+b}$. Moreover, it maximizes $P(S_j|\mathcal{A})$, even when either a or b (but not both) is zero. Therefore, we have:

$$\operatorname*{arg\,max}_{S_j\in[0,1],\mathcal{A}\neq\emptyset} P(S_j|\mathcal{A}) = \frac{s_{\mathcal{A}}}{|\mathcal{A}|}\,, \tag{5.7}$$

Figure 5.1 shows the normalized posterior distributions $P(S_j|\mathcal{A})$ for assessment samples with different sizes, given by Equation 5.6. In each case, the ratio between suspicious and non-suspicious assessments is the same, i.e., 1:3; by Equation 5.7, $S_j = \frac{1}{1+3} = 0.25$ is the maximizer of $P(S_j|\mathcal{A})$, which is clearly shown in Figure 5.1. The distribution becomes sharper with a larger sample, which accords to the intuition of the increasing certainty on the suspiciousness S_j.

The uncertainty over j's suspiciousness S_j (and, hence, the risk of losing a good neighbor) holds i back from cutting j off immediately, based on insufficient evidence. In the following discussion, we consider two alternative approaches, *distribution* and

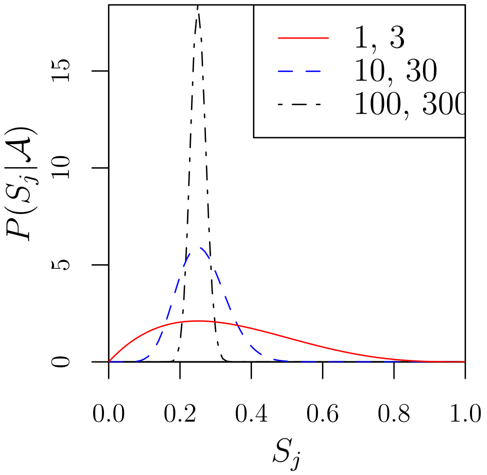


Figure 5.1.: The normalized posterior distribution $P(S_j|\mathcal{A})$ for assessment samples with different sizes. The two numbers for each line in the legend show the number of suspicious and non-suspicious assessments, respectively. In each case, the ratio between suspicious and non-suspicious assessments is 1 : 3. All distributions have a maximal value at $S_j = \frac{1}{1+3} = 0.25$. However, the distribution becomes shaper with a larger sample, which corresponds to a sense of increasing certainty regarding the suspiciousness S_j.

maximizer, to handle the insufficient-evidence problem, based on Equations (5.6) and (5.7), respectively.

In the *distribution* approach, i considers the whole posterior suspiciousness distribution (Equation (5.6)) in making the cut-off decision against j. From i's perspective, after observing an assessment sequence $\mathcal{A}$, the probability $P_g(\mathcal{A})$ that j is good is:

$$P_g(\mathcal{A}) = \int_0^{L_e} P(S_j|\mathcal{A})\mathrm{d}S_j; \tag{5.8}$$

the probability $P_e(\mathcal{A})$ that j is evil is:

$$P_e(\mathcal{A}) = 1 - P_g(\mathcal{A}) = \int_{L_e}^1 P(S_j|\mathcal{A})\mathrm{d}S_j. \tag{5.9}$$

Let $\mathcal{C} = (\int_0^1 S_j^{s_{\mathcal{A}}}(1-S_j)^{|\mathcal{A}|-s_{\mathcal{A}}}\mathrm{d}S_j)^{-1}$ be the (probability) normalization factor in Equation 5.6; we have:

$$P_g(\mathcal{A}) = \mathcal{C}\int_0^{L_e} S_j^{s_{\mathcal{A}}}(1-S_j)^{|\mathcal{A}|-s_{\mathcal{A}}}\mathrm{d}S_j \tag{5.10}$$

and

$$P_e(\mathcal{A}) = \mathcal{C}\int_{L_e}^{1} S_j^{s_{\mathcal{A}}}(1-S_j)^{|\mathcal{A}|-s_{\mathcal{A}}}\mathrm{d}S_j. \tag{5.11}$$

One property of $P_g(\mathcal{A})$ and $P_e(\mathcal{A})$ that is of use later is their monotonicity on $s_{\mathcal{A}}$. By Equations 5.8 and 5.9, we have $P_g(\mathcal{A}) = 1 - P_e(\mathcal{A})$. Thus, we only need to prove the monotonicity of any one of them; the other follows naturally. Here, we prove that $P_g(\mathcal{A})$ is a monotonically decreasing function on $s_{\mathcal{A}}$.

Let $a = s_{\mathcal{A}}$ and $b = A - s_{\mathcal{A}}$; we only need to prove:

$$\begin{aligned}&(\int_0^1 S_j^a(1-S_j)^{b+1}\mathrm{d}S_j)^{-1}\int_0^{L_e} S_j^a(1-S_j)^{b+1}\mathrm{d}S_j\\ \geq\ &(\int_0^1 S_j^{a+1}(1-S_j)^{b}\mathrm{d}S_j)^{-1}\int_0^{L_e} S_j^{a+1}(1-S_j)^{b}\mathrm{d}S_j,\end{aligned}$$

or, equivalently:

$$\begin{aligned}&\int_0^1 S_j^{a+1}(1-S_j)^{b}\mathrm{d}S_j\int_0^{L_e} S_j^a(1-S_j)^{b+1}\mathrm{d}S_j\\ \geq\ &\int_0^1 S_j^a(1-S_j)^{b+1}\mathrm{d}S_j\int_0^{L_e} S_j^{a+1}(1-S_j)^{b}\mathrm{d}S_j.\end{aligned}$$

Subtract $\int_0^{L_e} S_j^{a+1}(1-S_j)^{b}\mathrm{d}S_j \int_0^{L_e} S_j^a(1-S_j)^{b+1}\mathrm{d}S_j$ from both sides, we get:

$$\int_{L_e}^1 S_j^{a+1}(1-S_j)^{b}\mathrm{d}S_j\int_0^{L_e} S_j^a(1-S_j)^{b+1}\mathrm{d}S_j$$

for the left side and:

$$\int_0^{L_e} S_j^{a+1}(1-S_j)^{b}\mathrm{d}S_j\int_{L_e}^1 S_j^a(1-S_j)^{b+1}\mathrm{d}S_j$$

for the right side. Finally, we have:

$$
\begin{aligned}
\text{left} = \int_{L_e}^{1} S_j^{a+1}(1-S_j)^b \mathrm{d}S_j \int_{0}^{L_e} S_j^{a}(1-S_j)^{b+1} \mathrm{d}S_j \\
\geq \int_{L_e}^{1} L_e S_j^{a}(1-S_j)^b \mathrm{d}S_j \int_{0}^{L_e} (1-L_e) S_j^{a}(1-S_j)^{b} \mathrm{d}S_j \\
= \int_{0}^{L_e} L_e S_j^{a}(1-S_j)^b \mathrm{d}S_j \int_{L_e}^{1} (1-L_e) S_j^{a}(1-S_j)^{b} \mathrm{d}S_j \\
\geq \int_{0}^{L_e} S_j^{a+1}(1-S_j)^b \mathrm{d}S_j \int_{L_e}^{1} S_j^{a}(1-S_j)^{b+1} \mathrm{d}S_j = \text{right}.
\end{aligned}
$$

Thus, we have proven that "$P_g(\mathcal{A})$ is a monotonically decreasing function on $s_{\mathcal{A}}$" and "$P_e(\mathcal{A})$ is a monotonically increasing function on $s_{\mathcal{A}}$".

When $P_g(\mathcal{A}) \geq P_e(\mathcal{A})$, the evidence collected so far (i.e., $\mathcal{A}$) is favorable to j. However, when $P_g(\mathcal{A}) < P_e(\mathcal{A})$, the evidence is unfavorable to j and suggests that j might be an evil node. i needs to *decide whether to cut j off*.

The structure of the behavioral malware characterization model (specifically, a single threshold L_e is used to distinguish the nature of a node) gives rise to a subtlety concerning i's prejudice against j in the distribution approach. By Equation 5.7, if i makes no presumption on j's suspiciousness and no assessment has been made yet (i.e., $\mathcal{A} = \emptyset$), $P(S_j|\mathcal{A}) = 1$. If $L_E \neq 0.5$, by Equations (5.8) and (5.9), either $P_g(\mathcal{A}) < P_e(\mathcal{A})$ (if $L_E < 0.5$) or $P_g(\mathcal{A}) > P_e(\mathcal{A})$ (if $L_E > 0.5$). In other words, while i makes no presumption on j's suspiciousness, i may nevertheless be prejudiced against j by the distribution approach's decision rule.

This leads to a discussion on whether such prejudices are warranted. The choice of L_e depends on the assessment mechanism itself and, as mentioned previously, if the probabilistic distributions of suspiciousness of both good and evil nodes are known, can be determined by minimizing Bayesian decision errors. If $L_e > 0.5$, the assessment mechanism is biased towards false positive (good nodes' actions being assessed as suspicious); if $L_e < 0.5$, the assessment mechanism is biased towards false negative (evil nodes' actions being assessed as non-suspicious). However, before any assessment is made, i has no clue about the true nature of j. A bias in the assessment mechanism should not affect the i's neutrality on j's nature before the first assessment is made. Thus, we stipulate that the comparison between $P_g(\mathcal{A})$ and $P_e(\mathcal{A})$ should be made only when $\mathcal{A} \neq \emptyset$.

Alternatively, in the *maximizer* approach, i uses the suspiciousness distribution's maximizer (Equation (5.7)) when making the cut-off decision against j. The justification for the maximizer approach is that the suspicious distribution's maximizer is the *single most probable* estimation of j's suspicisousness given the evidence. The maximizer approach precludes the prejudice problem, because the maximizer is undefined when $\mathcal{A} = \emptyset$. Similar to the distribution approach, i compares evidence that is both favorable and unfavorable to j. Evidence $\mathcal{A}$ is favorable to j if $s_{\mathcal{A}}/|\mathcal{A}| \leq L_e$ and is unfavorable to j if $s_{\mathcal{A}}/|\mathcal{A}| > L_e$. The maximizer approach significantly reduces the computation cost, in comparison with the distribution approach, while partially discarding information contained in the suspiciousness distribution derivable from the evidence collected so far.

Whichever approach is taken, the cut-off decision problem has an *asymmetric* structure in the sense that cutting j off will immediately terminate the decision process (i.e., i will cease connecting with j; no further evidence will be collected), while the opposite decision will not. Thus, we only need to consider the decision problem when i considers cutting j off due to unfavorable evidence against j.

The cut-off decision is made based on the risk estimation of such a decision. The key insight is that i shall estimate the cut-off decision's risk by *looking ahead.*

More specifically, given the current assessment sequence $\mathcal{A} = (a_1, \ldots, a_A)$, the next assessment a_{A+1} (which has not been taken yet) might be either 0 (non-suspicious) or 1 (suspicious). Let $\mathcal{A}' = (\mathcal{A}, a_{A+1})$.

If $a_{A+1} = 1$, by the monotonicity of $P_g(\mathcal{A})$ and $P_e(\mathcal{A})$ on $s_{\mathcal{A}}$ (proved in Section **??**), either $P_g(\mathcal{A}') < P_g(\mathcal{A}) < P_e(\mathcal{A}) < P_e(\mathcal{A}')$ (the distribution approach) or $s_{\mathcal{A}'}/|\mathcal{A}'| = (1 + s_{\mathcal{A}})/(1 + |\mathcal{A}|) > s_{\mathcal{A}}/|\mathcal{A}| > L_e$ (the maximizer approach): The evidence against j becomes more unfavorable.

However, if $a_{A+1} = 0$, the evidence might become either favorable or unfavorable to j. If the evidence is still unfavorable toward j, we say that i's decision of cutting j off is *one-step-ahead robust*. If the cut-off decision is one-step-ahead robust, i is certain that exposing itself to the potential danger of infection by collecting *one further assessment* on j will not change the outlook that j is evil.

Similarly, i can look *multiple* steps ahead. In fact, the number of steps i is willing to look ahead is a *parameter* of the decision process rather than a *result* of it. This parameter shows i's willingness to be exposed to a higher infection risk in exchange for a higher certainty about the nature of j and a lower risk of cutting off a good neighbor; in other words, it reflects i's *intrinsic* risk inclination against malware infection.

Definition 2 (Look-ahead λ) *The* look-ahead λ *is the number of steps i is willing to look ahead before making a cut-off decision.*

We can make a similar decision-robustness definition for look-ahead λ.

Definition 3 (λ-robustness) *At a particular point in i's cut-off decision process against j (with assessment sequence $\mathcal{A} = (a_1, \ldots, a_A)$), i's decision of cutting j off is said to be* λ-step-ahead robust, *or simply* λ-robust, *if 1) the current evidence $\mathcal{A}$ is unfavorable toward j; 2) even if the next λ assessments $(a_{A+1}, \ldots, a_{A+\lambda})$ all turn out to be non-suspicious (i.e., 0), the evidence against j is still unfavorable.*

Given the look-ahead λ, the proposed malware containment strategy is *to cut j off if the cut-off decision is λ-robust, and not to cut j off otherwise.*

The look-ahead λ reflects individual nodes' intrinsic risk inclinations against the malware as follows.

λ must be large enough so that the decision process will not terminate prematurely. For example, after the first suspicious-action assessment against J, depending on L_e, the evidence might become unfavorable toward j, and i will consider whether to cut j off. If λ happens to be too small, depending on L_e, the cut-off decision may be

λ-robust at this very point (i.e., after the first assessment), and i will cut j off by the decision rule. Thus, λ should be properly chosen to ensure the decision process will bootstrap.

However, the look-ahead λ is related to the potential risk of being infected if the look-ahead has been carried out. Suppose that i's infection risk (against j) is $R(n)$ where n is the number of encounters between i and j; since direct contact is the only propagation channel of the proximity malware, $R(n)$ and n are positively correlated: more encounters mean a higher risk of being infected. One reasonable instantiation of $R(n)$ is $R(n) = 1 - (1 - p)^n$, where p is the (fixed) infection probability in a single encounter.

Suppose that i's cost of cutting j off (and hence losing j's service) is $C_i(j)$. To be comparable with the instantiation $R(n) = 1-(1-p)^n$, let $0 < C_i(j) < 1$. $C_i(j)$ reflects the value of j's service to i. One possible instantiation of $C_i(j)$ is j's social significance as perceived by i. For example, i can collect past communication/forwarding records or even initiate (opportunistic) local social community detection and use techniques such as ego-betweenness [57] to estimate j's social significance to i. The social cost $C_i(j)$ can be estimated once and kept fixed or can otherwise be updated regularly throughout the decision process.

If the evidence is unfavorable toward j, the look-ahead λ can be chosen by $\lambda = \max\{n|R(n) \leq C_i(j)\} = \max\{n|1 - (1 - p)^n \leq C_i(j)\}$: i is willing to give j chance (by looking λ steps ahead and hence not cutting j off immediately) as long as the infection risk (positively correlated with λ) is less than the cost of losing j's service (if j is a good neighbor). Depending on the relation between the infection risk $R(n)$ and the social cost $C_i(j)$, λ can be either static or dynamic across multiple encounters. To put it another way, a large λ is chosen as long as the (potential) benefit of maintaining connection with j justifies the (infection) risk.

5.3.2 Neighborhood Watch

Besides using i's own assessments, i may incorporate other neighbors' assessments in the cut-off decision against j. This extension to the evidence collection process is inspired by the real-life neighborhood (crime) watch program, which encourages residents to report suspicious criminal activities in their neighborhood. Similarly, i shares assessments on j with its neighbors, and receives their assessments on j in return.

In the neighborhood-watch model, the malicious nodes that are able to transmit malware (we will see next that there may be malicious nodes whose objective is other than transmitting malware) are assumed to be *consistent over space and time*. These are common assumptions in distributed trust management systems (summarized in Section 5.5), which incorporate neighboring nodes' opinions in estimating a local trust value.

By being consistent over space, we mean that evil nodes' suspicious actions are observable to all their neighbors, rather than only a few. If this is not the case, the evidence provided by neighbors, even if truthful, will contradict local evidence and, hence, cause confusions: Nodes shall discard received evidence and fall back to the household watch model.

By being consistent over time, we mean that evil nodes can not play strategies to fool the assessment mechanism. This is equivalent to the functional assumption in characterizing the nature of nodes by suspiciousness (Equation 5.1). The case in which the evil nodes can circumvent the suspiciousness characterization (such as by first accumulating good assessments, and then launch an attack through a short burst of concentrated suspicious actions) calls for game-theoretic analysis and design, and is beyond the scope of this chapter. Instead, we propose a behavioral characterization of proximity malware; further game-theoretic analysis and design could base on this foundation.

5.3.2.1 Challenges

Two cases complicate the neighborhood watch model: *liars* and *defectors*.

Liars are those evil nodes who confuse other nodes by sharing false assessments. A false assessment is either a *false praise* or a *false accusation*. False praises understate evil nodes' suspiciousness, while false accusations exaggerate good nodes' suspiciousness. Furthermore, a liar can fake assessments on nodes that it has never met with. To hide their true nature, liars may do no evil other than lying, and, therefore, have low suspiciousness.

Defectors are those nodes that change their nature due to malware infections. They start out as good nodes and faithfully share assessments with their neighbors; however, due to malware infections, they become evil. Their behaviors after the infection are under the control of the malware.

These complications call for *evidence consolidation*. Two extremal, but naive, evidence-consolidation strategies are 1) to trust no one and 2) to trust everyone. The former degenerates to the household-watch model with the twist of the defectors (defectors change their nature and hence their behavioral pattern); the latter leads to confusions among good nodes.

5.3.2.2 Evidence

For a pair of neighboring nodes i and j, let $\mathcal{N}_i$ and $\mathcal{N}_j$ be the neighbors of i and j, respectively. At each encounter, i shares with j its assessments on the neighbor set $\mathcal{N}_i - \{j\}$, and j shares with i its assessments on the neighbor set $\mathcal{N}_j - \{i\}$.

Since the cut-off decision only needs to be made against a neighbor, i only considers the assessments of its own neighbors $\mathcal{N}_i \cap (\mathcal{N}_j - \{i\})$ from the evidence provided by j. Without superimposed trust relationships among the nodes in the model, i and j only share *their own* assessments, instead of forwarding the ones provided by their neighbors.

5.3.2.3 Evidence Aging

The presence of defectors breaks the assumption when we characterize a node's nature by suspiciousness in Equation 5.1. A defector starts as a good node but turns evil due to malware infections; the assessments collected before the defector's change of nature, even truthful, are misleading.

To alleviate the problem of outdated assessments, old assessments are discarded in a process called *evidence aging*. Each assessment is associated with a timestamp. Only assessments with timestamps less than a specific *aging window* T_E from now are included in the cut-off decision.

To see that the aging window T_E alleviates the defector problem, consider a node that is infected at time T. Without evidence aging, all evidence before T mounts to testify that the node is good; if the amount of this prior evidence is large, it may take a long time for its neighbors to find out about the change in its nature. In comparison, with evidence aging, at time $T + T_E$, all prior evidence expires and only those assessments after the infection are considered, which collectively testify against the node.

However, in practice, the choice of the aging window T_E depends on the context. While a small T_E may speed up the detection of defectors by reducing the impact of stale information, T_E must be large enough to accommodate enough assessments to make a sound cut-off decision. If T_E is too small, a node will not have enough assessments to make a λ-robust cut-off decision.

5.3.2.4 Evidence Consolidation

We propose two alternative methods, *dogmatic filtering* and *adaptive look-ahead*, for consolidating evidence provided by other nodes, while containing the negative impact of liars. For exposition, we consider a scenario in which node i uses the assessments within the evidence aging window $[T - T_E, T]$ provided by i's neighbors (other than one of the neighbors, say, j) in making the cut-off decision against j.

The following observation inspires our solution: Given enough assessments, i is more likely to correctly estimate j's suspiciousness than otherwise. Consider a simple numerical illustration. If j has in total 4 suspicious actions and 12 non-suspicious actions assessed by its neighbors, its (true) suspiciousness is $\frac{4}{4+12} = 0.25$. If i has made 4 out of the $4 + 12 = 16$ assessments, by the space-consistency assumption, i is equally likely to obtain *any* sub-sequence of the 16 assessment sequence. The total possibilities of i making x $(0 \leq x \leq 4)$ suspicious assessments and $4-x$ non-suspicious assessments are $\binom{4}{x}\binom{12}{4-x}$; a straightforward calculation shows that the number is maximized when $x = 1$. In other words, i is more likely to estimate j to be $\frac{1}{1+3} = 0.25$, which agrees with the true suspiciousness, compared to otherwise.

In general, suppose j has been assessed n times by its neighbors, and s of them are suspicious. Its suspicisousness, by definition, is $\frac{s}{n}$. If n' $(0 < n' \leq n)$ of the assessments are from i and s' $(s - (n - n') \leq s' \leq \min(s, n'))$ of them are suspicious (thus, from i's perspective, j's suspiciousness is $\frac{x'}{n'}$), s' is more likely to be either $\lfloor \frac{s}{n} n' \rfloor$ or $\lceil \frac{s}{n} n' \rceil$ (i.e., i's estimation of j's suspiciousness agrees with the true suspiciousness) than otherwise, since, as in the previous numerical example, $\binom{s}{s'}\binom{n-s}{n'-s'}$ is maximized when $\frac{s'}{s} \approx \frac{n'-s'}{n-s}$ for a given n'.

The implications are: 1. Given enough assessments, honest nodes are likely to obtain a close estimation of a node's suspiciousness (suppose they have not cut the node off yet), even if they only use their own assessments. 2. The liars have to share a significant amount of false evidence to sway the public's opinion on a node's suspiciousness. 3. The most susceptible victims of liars are the nodes that have little evidence.

Dogmatic filtering *Dogmatic filtering* is based on the observation that one's own assessments are truthful and, therefore, can be used to bootstrap the evidence consolidation process. A node shall only accept evidence that will not sway its current opinion too much. We call this observation the *dogmatic principle.*

Our interpretation of the dogmatic principle depends on the following generalization of Definition 3.

Definition 4 (λ-robust judgment) *Let $\mathcal{A}$ be the suspicious-action assessments that i has on j. We say that i's* judgment on j's nature is λ-robust (or $(-\lambda)$-robust) based on $\mathcal{A}$, *if 1) the evidence $\mathcal{A}$ is favorable (or unfavorable) toward j, 2) the evidence remains so* even if *the next λ assessments are all suspicious (or non-suspicious), and 3) the evidence becomes unfavorable (or favorable) toward j if the next $\lambda + 1$ assessments are all suspicious (or non-suspicious).*

As a special case, if a judgment is not even 1-robust (or (-1)-robust), we say that the judgment is 0-robust or not robust at all.

λ-robust judgment reflects i's certainty of its judgment on j's nature (based on the evidence collected so far). The λ-robust cut-off decision against j (Definition 3) is equivalent to the $(-\lambda)$-robust judgment on the (evil) nature of j. The sign of λ in Definition 4 represents j's nature: A negative number represents evilness, and a positive number represents goodness.

i's cut-off decision against j works as follows with dogmatic filtering. 1. i will not consider cutting j off until i has at least one assessment on j. 2. After its first encounter with j and with its own assessments $\mathcal{A}$ with the evidence aging window

$[T-T_E, T]$, i considers whether or not to take another neighbor k's alleged assessments on j within the same window $\mathcal{B}$ when i and k meet. 3. Suppose that the judgment on j's nature is $\lambda_{\mathcal{A}}$-robust and $\lambda_{(\mathcal{A}+\mathcal{B})}$-robust, based on $\mathcal{A}$ and $(\mathcal{A}+\mathcal{B})$, respectively. i will take $\mathcal{B}$ only if $\lambda_{\mathcal{A}} \neq 0$ and $\frac{|\lambda_{\mathcal{A}}-\lambda_{(\mathcal{A}+\mathcal{B})}|}{|\lambda_{\mathcal{A}}|} \leq \delta$; $\delta > 0$ and is called the *dogmatism*. 4. i makes a λ-robust cut-off decision against j, based on either $\mathcal{A}$ or $(\mathcal{A}+\mathcal{B})$, depending on whether $\mathcal{B}$ has passed the dogmatism test.

With dogmatic filtering, i is very conservative when its certainty about j's nature is still low (i.e., $\lambda_{\mathcal{A}}$ is small). At this early stage, i will accept the evidence provided by j only if the evidence would not significantly change its certainty on j's nature. In particular, if $\lambda \leq 1$, i will *never* accept a piece of evidence that would change its judgment on j's nature because $|\lambda_{\mathcal{A}} - \lambda_{(\mathcal{A}+\mathcal{B})}| > |\lambda_{\mathcal{A}}|$ if $\mathcal{A}$ and $(\mathcal{A}+\mathcal{B})$ are of different signs.

Dogmatic filtering significantly contains the impact of liars on i while still allowing a change of certainty (on j's nature) comparable to its own. The aforementioned observation that the liars have to fabricate a significant amount of false evidence to confuse honest nodes means that the evidence $\mathcal{B}$ provided by a liar k must have a high $\lambda_{\mathcal{B}}$ (albeit of the wrong sign) to be effective in confusing i. The liar's strategy will not work because i will refuse to take $\mathcal{B}$ when $|\lambda_{\mathcal{A}}|$ is small with dogmatic filtering, while $\lambda_{\mathcal{A}}$ and $\lambda_{\mathcal{B}}$ should be of different signs when $\lambda_{\mathcal{A}}$ is large (because by then, i should have a close estimation of j's true suspiciousness, and hence, $\lambda_{\mathcal{A}}$ is of the correct sign). The evidence filtering works even when the liars are the majority among i's neighbors.

Adaptive look-ahead *Adaptive lookahead* takes a different approach towards evidence consolidation. Instead of deciding whether to use the evidence provided by others *directly* in the cut-off decision, adaptive lookhead *indirectly* uses the evidence by adapting the *steps to look ahead* to the diversity of opinion.

Adaptive look-ahead works as follows. 1. Suppose that at a particular moment, the distribution maximizer derived from the assessments (within the evidence aging window) on j (Equation (5.7)) made by i is s_0; similarly, the distribution maximizer derived from the assessments (within the evidence aging window) on j that i *received* from its neighbors is $s_1, s_2, \ldots, s_n$. 2. i computes the following *ego-centric variance* σ_i as a metric on the diversity of opinions (from its own assessments):

$$\sigma_i = \frac{\sqrt{\sum_{i=1}^{n}(s_i - s_0)^2}}{n}. \tag{5.12}$$

3. Let the *maximal* ego-centric variance up to (and including) now be σ_i^* (thus, we have $\sigma_i \leq \sigma_i^*$). i makes its cut-off decision against j *if the decision is* $f(\sigma_i, \sigma_i^*, \lambda)$*-robust*, where $f(\cdot, \cdot, \cdot)$ is a three-parameter integer function ranging from 0 to λ, which we call the *adaptive-lookahead function*. A particular instantiation is the *linear* adaptive lookahead function.

$$f(\sigma_i, \sigma_i^*, \lambda) = \lceil \lambda \frac{\sigma_i}{\sigma_i^*} \rceil. \tag{5.13}$$

The idea of adaptive look-ahead is to adapt the risk inclination, embodied in the λ-robust cut-off decision in Definition 3, to the diversity of public opinions, embodied in the ego-centric variance in Equation (5.12). The dogmatism principle underlies the use of the ego-centric variance: The agreement of the public's opinions with that of i is an indication that i is approaching the true suspiciousness; thus, to expedite the detection of evil nodes (and hence reduce the risk of infection from further contact), i reduces the steps to look ahead in making the cut-off decision.

Because the value of the adaptive-lookahead function is no greater than 1, the worst that liars can do is to degenerate i's cut-off decision to a λ-robust one. Also, since i has a chance of estimating a close-to-true suspiciousness than otherwise, liars' false opinions are likely to be different from that of i, and good nodes' opinions are likely to agree with that of i. Thus, i will be more proactive if good nodes make up the majority of its neighborhood and less so if the liars are the majority.

Table 5.1.: Dataset statistics.

	nodes	entries	time span	avg. interval
Haggle	41	112, 295	15 days	12 secs
MIT reality	96	114, 046	490 days	371 secs

5.4 Experiment

5.4.1 Datasets

We verify our design with two real mobile network traces: Haggle [179] and MIT reality [67].

The raw datasets are rich in information, some of which is irrelevant to our study, e.g., call logs and cell tower IDs in MIT reality. Therefore, we remove the irrelevant fields and retain the node IDs and time-stamps for each pair-wise node encounter. Since the Haggle dataset has only 22, 459 entries spanning over 3 days, we repeat it another 4 times to make it into a dataset with 112, 295 entries spanning over 15 days, and thus make it comparable to the MIT reality dataset in quantity. Some statistics of the processed datasets are summarized in Table 5.1.

5.4.2 Setup

Without loss of generality, we choose $L_e = 0.5$ to be the line between good and evil. For each dataset, we randomly pick 10% of the nodes to be the evil nodes and assign them with suspiciousness greater than 0.5; the rest of the nodes are good nodes and are assigned suspiciousness less than 0.5.

For a particular pairwise encounter, a uniform random number is generated for each node; a node receives a “suspicious” assessment (by the other node) if the random number is greater than its suspiciousness and receives a “non-suspicious” assess-

Table 5.2.: Neighbor nature and cut-off decision combination.

	…gets cut off.	…stays connected.
An evil neighbor…	True positive.	False negative.
A good neighbor…	False positive.	True negative.

ment otherwise. Thus, each assessment is binary, while the frequency of "suspicious" assessments for a particular node reflects its suspiciousness in the long term.

5.4.3 Performance Metric

The performance comparison is based on two metrics: *detection rate* and *false positive rate*. The categories of the "neighbor's nature" and "cut-off decision" combinations are shown in Table 5.2. For each combination, we sum up all the decisions made by *good* nodes (evil nodes' cut-off decisions are irrelevant) and obtain four counts: TP (true positives), FN (false negatives), TN (true negatives), and FP (false positives). The detection rate DR is defined as:

$$DR = \frac{TP}{TP + FN} \times 100\%,$$

and the false positive rate FPR is defined as:

$$FPR = \frac{FP}{FP + TN} \times 100\%.$$

A high detection rate and a low false positive rate are desirable. When a balance must be stricken between the two, one might be emphasized over the other, depending on the context.

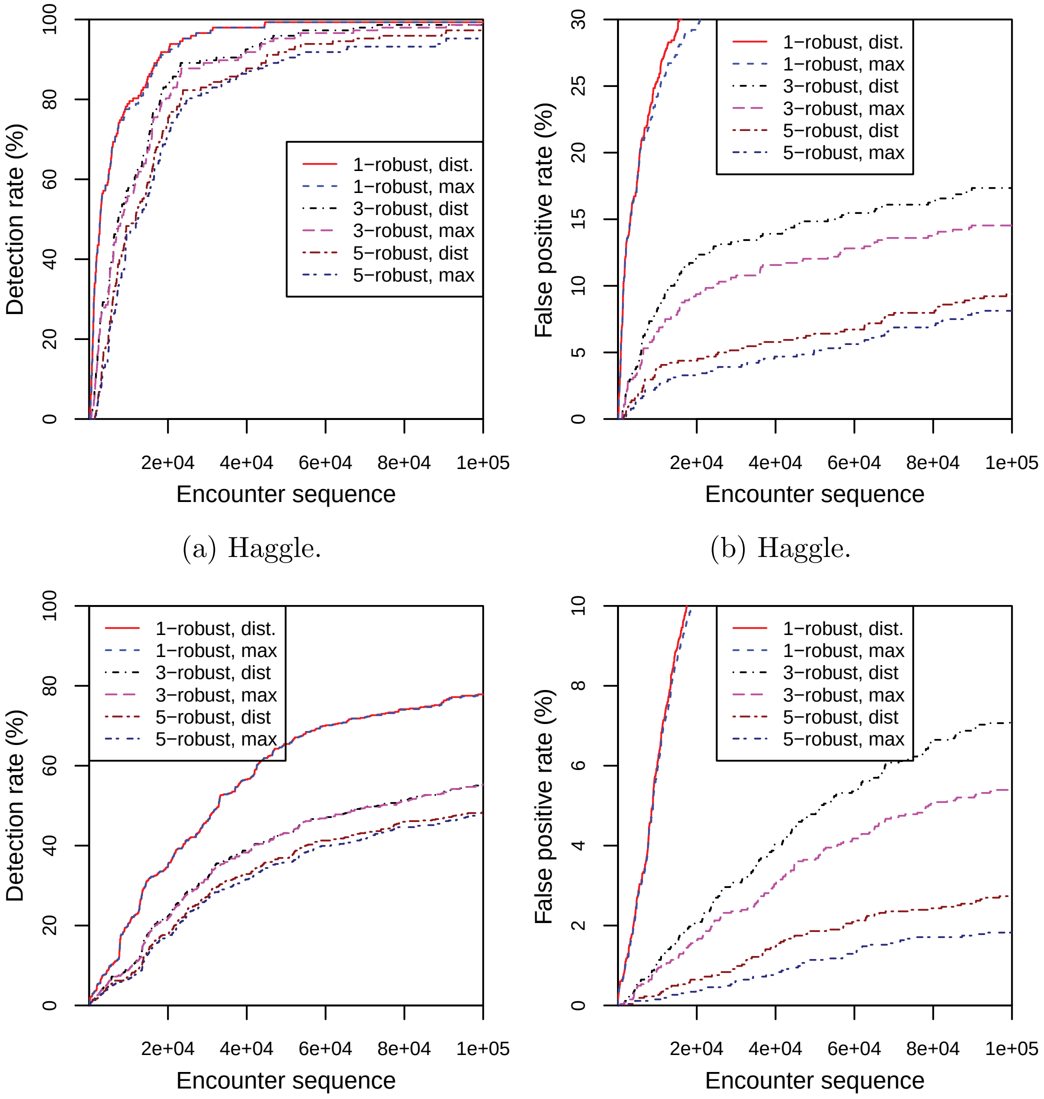
5.4.4 Results

5.4.4.1 Look-ahead: Distribution vs. Maximizer

We compare the two alternative approaches, distribution and maximizer, to the look-ahead strategy (Section 5.3.1). The results are shown in Figure 5.2.

The look-ahead parameter λ reflects a node's intrinsic (infection) risk inclination. In both Haggle (Figures 5.2a and 5.2b) and MIT reality (Figures 5.2c and 5.2d), the λ-robust cut-off strategy with a larger λ corresponds to a higher detection rate (in the early stage for Haggle and throughout for MIT reality) and a significantly lower false positive rate (for both datasets). In Haggle, the eventual detection rates for all three look-ahead parameters are close to 100%. The difference in the eventual detection rate between Haggle and MIT reality is attributed to the different contact patterns in these datasets: The contact pattern in Haggle is more homogeneous than that in MIT reality, in the sense that the variation of the interval between encounters is significantly higher and a few nodes contribute most of the assessments in MIT reality. Thus, the detection rate is more sensitive to the change of λ in MIT reality than in Haggle.

In both datasets, the detection rate and false positive rate are comparable for the distribution and maximizer approach, with the distribution approach having a slightly higher detection rate and false positive rate. The small difference in performance, coupled with the significant reduction in computation overhead (integration for the distribution approach versus arithmetic operations for the maximizer approach), make



(a) Haggle.

(b) Haggle.

(c) MIT reality.

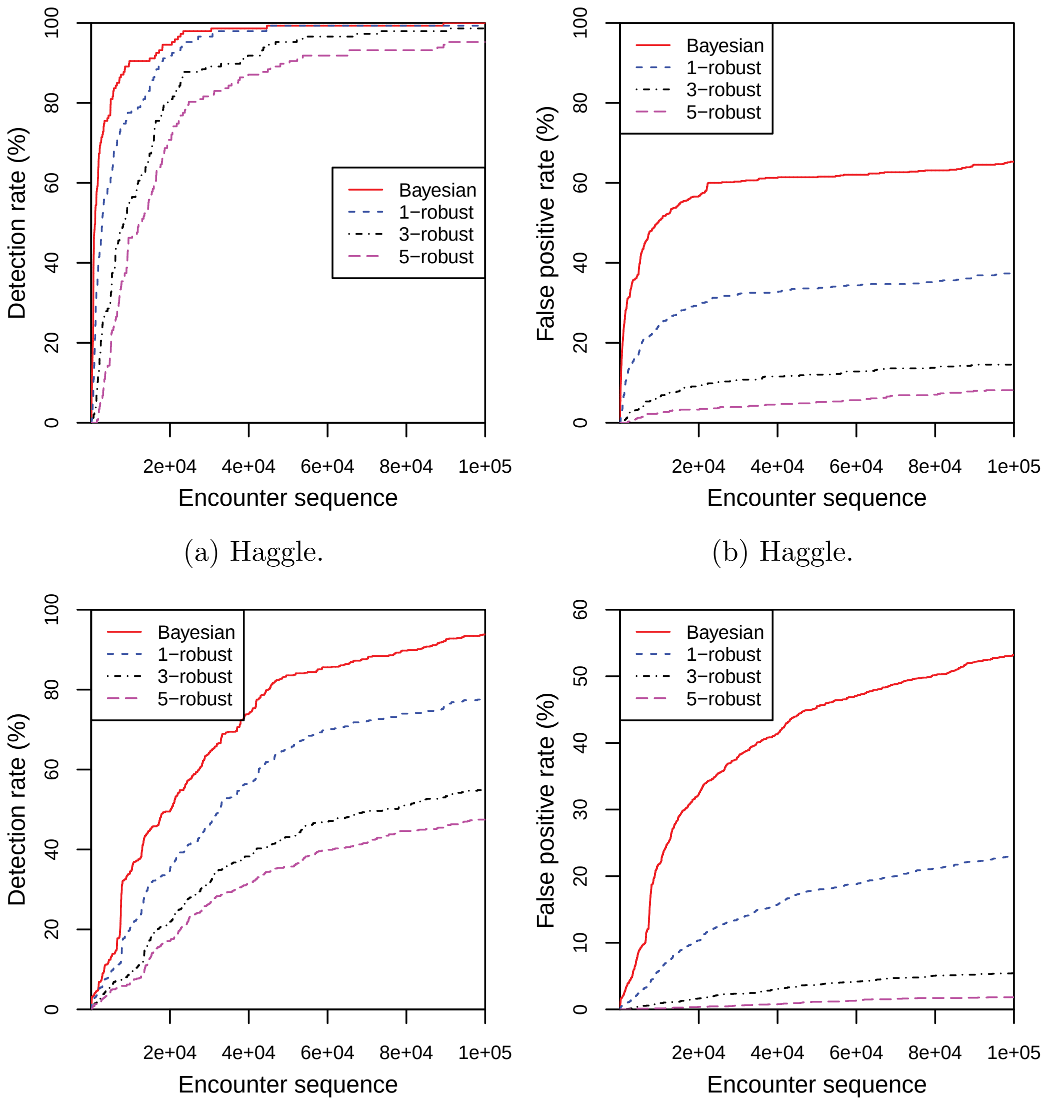
(d) MIT reality.

Figure 5.2.: Performance comparison between the λ-robust cut-off strategy with the distribution (dist) and maximizer (max) evidence weighing approaches; $\lambda = 1, 3,$ and 5.

the maximizer approach with a moderate λ as the preferred look-ahead strategy. In the following sections, we show results for the maximizer approach with $\lambda = 3$.

5.4.4.2 Look-ahead

We compare Bayesian-based strategies with, and without, the look-ahead extension (i.e., λ-robust cut-off decision) under the household-watch model (i.e., no evidence



(a) Haggle.

(b) Haggle.

(c) MIT reality.

(d) MIT reality.

Figure 5.3.: Performance comparison between the vanilla Bayesian (degenerated 0-robust) cut-off strategy and the 3-robust look-ahead cut-off strategy.

exchange). The vanilla Bayesian strategy does not look ahead and proceeds with cutting-off once the evidence becomes unfavorable to the neighbor. It can be seen as a degenerated λ-robust cut-off strategy with $\lambda = 0$. The results are shown in Figure 5.3.

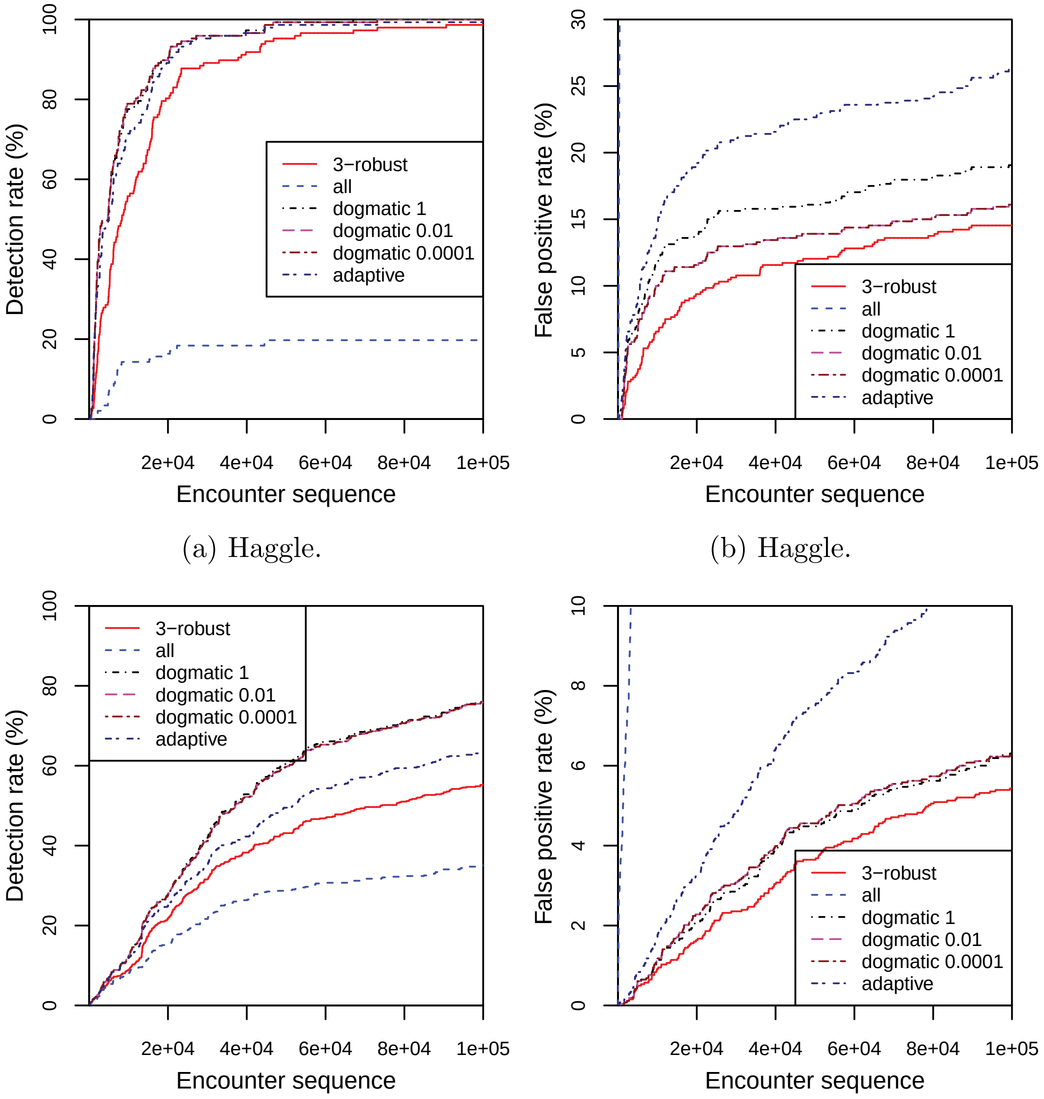
In Figure 5.3, the vanilla Bayesian strategy has the highest detection and false positive rate. Both rates drop with an increasing look-ahead parameter. However, the false detection rate drops much faster than the detection rate. Indeed, for Haggle, the 1-robust and the vanilla Bayesian strategies have almost the same detection rate after $30,000$ encounters, but there is a 30% difference in the false positive rate. The difference in detection rate is more pronounced for MIT reality, but the reduction in false positive rate far outweighs that of detection rate. For the risk-taking nodes, sacrificing a little detection rate for a large reduction in false positive rate is desirable: the look-ahead parameter λ provides an effective mechanism to tune for a desirable balance.

The results confirm the intuition that leads to the look-ahead extension to the vanilla Bayesian strategy: Being conservative in making cut-off decisions (by looking ahead) pays off by retaining utility without sacrificing much security.

5.4.4.3 Evidence Consolidation

We also evaluate the benefits of sharing assessments among nodes, and the effect of the proposed evidence consolidation strategies in minimizing the negative impact of liars on the shared evidence's quality. We compare the dogmatic filtering (with dogmatism of 0.0001, 0.01, and 1, respectively) and adaptive look-ahead evidence consolidation methods with two other (naive) evidence consolidation methods: 1) taking *no* indirect evidence, i.e., look-ahead with no evidence consolidation, and 2) taking *all* indirect evidence without filtering.

In our study, 10% of the evil nodes play the dual roles of evil-doers and liars. There are many possible liar strategies. Based on our observations in Section 5.3.2.4, we adopt an *exaggerated false praise/accusation* liar strategy. More specifically, a liar (falsely) accuses good nodes of suspicious actions and (falsely) praises other evil nodes for non-suspicious actions. Besides, to exert a significant influence on the public opinion, they exaggerate the false praises/accusations by 10 times (since they are only



(a) Haggle.

(b) Haggle.

(c) MIT reality.

(d) MIT reality.

Figure 5.4.: Performance impact of various evidence consolidation methods on the look-ahead cut-off strategy. *all*: naive strategy without filtering (Section 5.3.2); *dogmatic* δ: dogmatic filtering with dogmatism δ (Section 5.3.2.4); *adaptive*: adaptive lookahead (Section 5.3.2.4).

10% of the whole population). The results on the performance of various evidence consolidation strategies under this setting are shown in Figure 5.4.

Figure 5.4 clearly shows the negative impact of liars on malware detection if evidence is not filtered: Under the influence of liars, the naive “all” strategy has a low detection rate and a high false positive rate. This calls for a non-trivial evidence consolidation strategy to deal with the liars.

Both dogmatic filtering and adaptive look-ahead show significant increases in detection rate and modest increases in false positive rate over the baseline 3-robust lookahead strategy with no evidence filtering. Together with Figure 5.3, the results indicate that the 3-robust look-ahead, with either dogmatic filtering or adaptive lookahead, is comparable in detection rate and, even in the presence of liars, shows a significantly lower false positive rate in comparison with both the Bayesian and 1-robust strategies.

In Figure 5.4, the eventual detection rates converge to almost 100% for Haggle but diverge for MIT reality. The convergence in detection rate is expected for a homogeneous dataset like Haggle, in which most nodes are well-connected and are able to collect enough evidence to eventually make a sound cut-off decision. In this case, evidence consolidation helps to expedite the decision-making process without driving the false positive rate up too much. A closer look at MIT reality indicates that this dataset is highly heterogeneous: A few well-connected nodes contribute most of the assessments, and leave the other less well-connected nodes with insufficient evidence to make a λ-robust judgment alone. In this case, evidence consolidation helps the latter nodes in collecting enough evidence to make a λ-robust decision.

Two of the dogmatic filtering strategies (with a dogmatism of 0.01 and 0.0001) show almost the same performance, with the other dogmatic filtering strategy (with a dogmatism of 1) show a slight difference in comparison with other strategies. In both datasets, the adaptive look-ahead strategy shows an inferior performance in comparison to the three variations of the dogmatic filtering strategy. However, it automatically (i.e., with no parameter to tune) achieves superior detection rate over both Bayesian and 3-robust strategies in the presence of liars.

5.5 Related Work

Proximity malware and mitigation schemes. Su et al. collected Bluetooth traces and demonstrated that malware could effectively propagate via Bluetooth with simulations [191]. Yan et al. developed a Bluetooth malware model [217]. Bose and Shin showed that Bluetooth can enhance malware propagation rate over SMS/MMS [29]. Cheng et al. analyzed malware propagation through proximity channels in social networks [45]. Akritidis et al. quantified the threat of proximity malware in wide-area wireless networks [6]. Li et al. discussed optimal malware signature distribution in heterogeneous, resource-constrained mobile networks [126]. In traditional, non-DTN, networks, Kolbitsch et al. [119] and Bayer et al. [23] proposed to detect malware with learned behavioral model, in terms of system call and program flow. We extend the Naive Bayesian model, which has been applied in filtering email spams [15, 93, 224] , detecting botnets [206], and designing IDSs [58, 5], and address DTN-specific, malware-related, problems. In the context of detecting slowly propagating Internet worm, Dash et al. presented a distributed IDS architecture of local/global detector that resembles the neighborhood-watch model, with the assumption of attested/honest evidence, i.e., without liars [58].

Mobile network models and traces. In mobile networks, one cost-effective way to route packets is via the short-range channels of intermittently connected smartphones [199, 37, 72]. While early work in mobile networks used a variety of simplistic random i.i.d. models, such as random waypoint, recent findings [103] show that these models may not be realistic. Moreover, many recent studies [57], based on real mobile traces, revealed that a node's mobility shows certain social network properties. Two real mobile network traces were used in our study.

Reputation and trust in networking systems. In the neighborhood watch model, suspiciousness, defined in Equation (5.1), can be seen as nodes' reputation; to cut a node off is to decide that the node is not trustworthy. Thus, our work can be viewed from the perspective of reputation/trust systems. Three schools of thoughts emerge from previous studies. The first one uses a central authority, which by convention is called the trusted third party. In the second school, one global trust value is drawn and published for each node, based on other nodes' opinions of it; eigenTrust [115] is an example. The last school of thoughts includes the trust management systems that allow each node to have its own view of other nodes [34, 188]. Our work differs from previous trust management work in addressing two DTN-specific, malware-related, trust management problems: 1) insufficient evidence vs. evidence collection risk and 2) sequential and distributed online evidence filtering.

5.6 Summary and Future Work

Behavioral characterization of malware is an effective alternative to pattern matching in detecting malware, especially when dealing with polymorphic or obfuscated malware. Naive Bayesian model has been successfully applied in non-DTN settings, such as filtering email spams and detecting botnets. We propose a general behavioral characterization of DTN-based proximity malware. We present *look-ahead*, along with *dogmatic filtering* and *adaptive look-ahead*, to address two unique challenging in extending Bayesian filtering to DTNs: "insufficient evidence vs. evidence collection risk" and "filtering false evidence sequentially and distributedly". In prospect, extension of the behavioral characterization of proximity malware to account for strategic malware detection evasion with game theory is a challenging yet interesting future work.

6 WEB OF APKS (WOA): A DECLARATIVE APPROACH FOR STATIC ANDROID PACKAGE (APK) BINARY ANALYSIS

Chapter 5 *abstracts* the mechanism of detecting proximity mobile malware in a probabilistic behavioral model. This chapter supplements Chapter 5 with *concrete* investigation on a popular and real mobile computing platform, Android. This chapter presents the design, implementation, and evaluation of a declarative approach for static Android app binary analysis, which, among several other applications, supports malware analysis on the Android platform. In light of existing literature (surveyed in Section 6.2), the highlights of the proposed approach are: 1. the expressiveness and efficiency of its query capability using a graph representation that captures key elements of Android apps' semantics. 2. its robust handling of implicit control flows and obfuscations to account for adversarial app analysis scenarios such as app plagiarism detection and malware analysis.

6.1 Introduction

6.1.1 Problem Definition and Motivation

Initially released on 5 November 2007 [211] as the first product of the Open Handset Alliance (an industry consortium of, as of January 2015, 84 "technology and mobile companies" [7]), the Android mobile platform [12] has seen wide adoption in the consumer market. Ephemeral market statistics [84] of "estimated worldwide Android-based mobile device shipments reaching 1.4 billion units in 2015" aside, a recent measurement study [205] on the official Android app market (i.e., Google Play Store) that crawls over 1.1 million apps furnishes enough evidence on the scale of the current Android mobile platform. This tremendous growth is reflected by a surge of academic research on the application [183, 184, 25, 116, 152] and, particularly, security [204, 69, 236, 149, 228, 43] of the Android platform. The research body is too numerous to be enumerated here; some recent representative works that are closely related to the objective of this chapter are surveyed in Section 6.2.

The research objective of this chapter is to propose, implement, and evaluate declarative graph analysis as a scalable and robust approach for identifying and explaining similarity between Android application packages (known by its acronym "APK" in Android platform development documentation). Before moving on to present the design (Sections 6.4 and 6.5) and evaluation (Section 6.6) of this work, we first address the following questions:

- What do we mean by "similarity between APKs"?
- Why do we need to identify and explain such similarity?
- Why emphasizing the attributes "declarative, scalable, and robust?"

What do we mean by "similarity between APKs"? Similar APKs perform similar functionality and/or share similar program structure due to, for example, identical authorship, common libraries, or software theft/plagiarism. Apps instantiated from similar APKs produce similar effects on the *environment* that include other apps and the underlying Android system. A closely related concept from software engineering research is *software birthmarks* [142, 132, 20, 235, 215, 209, 110], defined as a unique characteristic of a program that can be used "as a software theft detection technique" [142] or "to determine the program's identity" [110]—both focusing on establishing the identity or origin of programs for detecting software theft/plagiarism. As we will discuss next, detecting similar APKs have wider practical applications than software theft/plagiarism.

Why do we need to identify and explain such similarity? Identifying similar APKs is a common theme underlying several current research problems, which can be categorized into adversary scenario and non-adversary scenario.

Two prominent applications in adversary scenario are app plagiarism detection [62, 233, 55, 97, 234, 56, 229, 44, 225] and malware detection [85, 232, 164, 128, 219, 78, 16]. App plagiarism is a specialization of software theft/plagiarism (briefly discussed above) in the domain of Android apps, with the idiosyncrasies of the Android platform both simplifying (e.g., identifying platform API) and complicating (e.g., multiple-entry and reactive nature of apps) the task (this will be elaborated in Sections 6.4 and 6.5). A major line of recent research [85, 164, 219, 78, 16] that uses machine learning techniques for Android malware detection is based on the assumption that malware has structural/behavioral similarity that can be captured by proper training on a representative dataset [16]. The observation that real-world malware samples often share common malicious payloads or code [237, 16] (due to, for example, black market trading [77]) support the approach of detecting malware through similarity.

Applications in non-adversary scenario include app categorization [60, 216] ("this app is a flashlight tool") and app recommendation [54, 223, 208, 127] ("you might also like these apps that provide similar functions"). A recent measurement study [205] reveals inadequacies in current app market such as coarse granularity of categories, mutual exclusion of categories in which apps may reasonably belong to multiple categories, and app authors may exaggerate or misrepresent their apps' functionality and quality for undeserved benefits. Identifying app similarity facilitate the categorization and vetting of apps on the market.

Explaining "how a group of apps are related", or result explainability/interpretability, complements the identification of these apps and often provides insights that are needed to vet the identification results (we will discuss concrete examples later). Machine-learning-based techniques are often susceptible to the choice *between* accuracy and interpretability. One treatise on this topic [31] puts this trade-off plainly in its title "Machine learning: Between accuracy and interpretability." The most accurate models in practice are rarely the most explainable ones [32]. Result interpretability emerges as a main objective in recent Android malware detection research [85, 16].

Why emphasizing the attributes “declarative, scalable, and robust?” The size and growth of the Android app market as revealed by recent measurement study of the official Google Play Store [205] (and not to consider the numerous alternative markets such as Amazon Appstore [8]) leads to what researchers refer to as “a billion opcode problem” [44, 97]: The number of Android apps, and the resources/opcodes that determine their semantics, has grown to a scale that the performance gap between linear and quadratic complexity in processing and cross-examining them (such as the task we are undertaking) make the difference between practical and impractical. Pair-wise comparison of apps that are used in early app plagiarism detection works [62, 233, 55] do not scale well, and hence later works focus on improving this aspect by, for example, localizing the search of similar apps (such as the use of Vintage Point Tree [222] in detecting piggybacked apps [234]) or applying clustering techniques to apps encoded in metric spaces [97].

Real-world app analysis rarely has access to app source code, and is often performed in an uncooperative or even hostile scenario, in which the apps under analysis employ techniques to prevent such analysis. One example is the real Android malicious app analyzed in Section 6.3. A robust analysis is one that assumes an uncooperative/hostile scenario by default, and can withstand *certain* anti-analysis attempts at an *affordable cost*: As the adages “there is not free lunch” and “there is no panacea” go, an app analysis that claims robustness must be clear on “what scenarios it applies to” and “at what cost.” We will elaborate on this point later concerning the presented work.

The benefits of the declarative approach to computational problem solving [130] are manifested in, for example, the standardization of SQL [59] for database queries and the application of Prolog [50, 88] to artificial intelligence research. A declarative approach to Android app analysis has the similar benefit of *separation of concerns*: An analyst, who has the domain knowledge of which app features are pertaining to the task at hand, only needs to declare *what* features to find, rather than specify *how* to find the features. As surveyed in details in Section 6.2, most existing works on Android app analysis towards malware/plagiarism detection focus on specific aspects of the "how," and the support for "what"-style exploration receives little investigation. An exception is the recent work by Feng et al. [78], in which the proposed system supports using Datalog [4], a subset of Prolog, to find privacy-leaking apps. A major objective/contribution of the present work is to advance the research on this front by proposing a methodology and devising a software tool to support declarative APK analysis.

The rest of the chapter presents an approach that addresses these aspects in the task of identifying/explaining Android app similarity in specific, and of Android app analysis in general.

6.1.2 A Preview of Declarative APK Analysis

This section presents two simple examples to illustrate the declarative graph analysis approach towards APK analysis, which is a central theme of this work. Figure 6.1 visualizes the result from issuing the following Neo4j [144] Cypher [145] query[1]:

```
MATCH (a:Apk) WHERE a.package="com.agewap.om"
MATCH (sk:SigningKey)-->a-->(d:Dex)-->(cp:Component)-->(cb:Callback)--(m:Method)
MATCH a-->(cgr:CallGraphNode)-[:INVOKE*..5]->(cgn:CallGraphNode)
return *
```

against a prototype million-node-scale Web of APKs (WoA) for Android malware samples[2] with the package name `com.agewap.om` (details will be explained in later sections; we focus on the result here).

The result of this query indicates that:

[1] This is the first instance of a number of concrete queries presented in this work for result reproducibility. Since the Neo4j graph database [144] and its Cypher graph query language [145] are used in the current implementation of this work, concrete queries are presented in Cypher. Section 6.4.1.2 presents a short introduction to the query syntax.

[2] The prototype WoA contains over 5,000 Android malware samples from the Drebin [16] and Android Malware Genome Project (AMGP) [236] projects. More evaluation using this prototype is presented in Section 6.6.

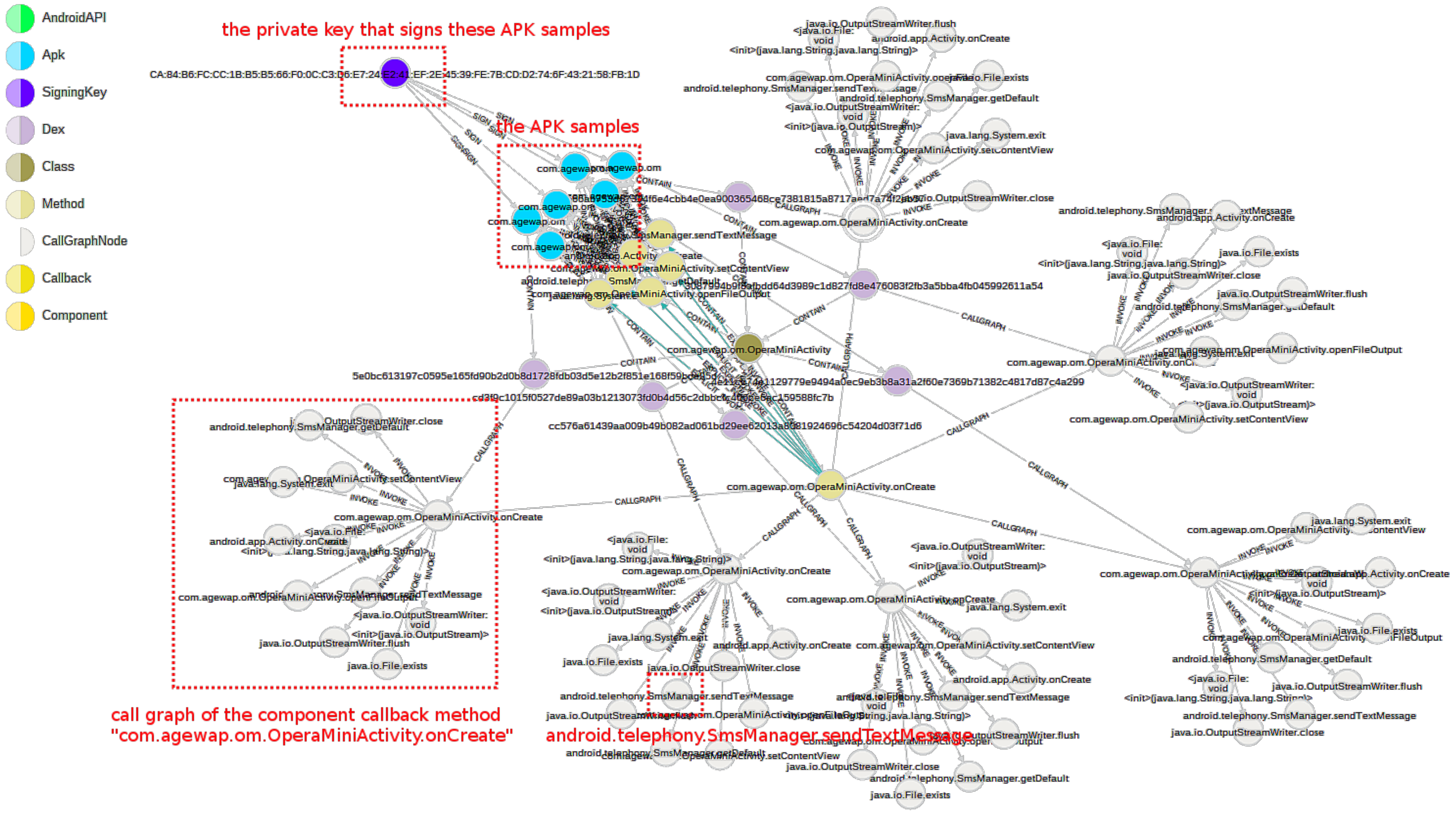

AndroidAPI
Apk
SigningKey
Dex
Class
Method
CallGraphNode
Callback
Component
the private key that signs these APK samples
CA:84:B6:FC:CC:1B:B5:B5:66:F0:0C:C3:D6:E7:24:E2:41:EF:2E:45:39:FE:7B:CD:D2:74:6F:43:21:58:FB:1D
the APK samples
call graph of the component callback method "com.agewap.om.OperaMiniActivity.onCreate"
android.telephony.SmsManager.sendTextMessage
com.agewap.om.OperaMiniActivity.onCreate
com.agewap.om.OperaMiniActivity
SIGN
CONTAIN
CALLGRAPH
INVOKE

Figure 6.1.: The relationship of the `com.agewap.om` APK samples in the prototype Web of APKs.

- Same authorship: There are 6 such samples, all signed by the same private key, the SHA-256 checksum of which begins with `CA:84:B6:FC:CC:1B`.

- Not simple repackaging: The 6 samples contain different executable codes (Dalvik EXecutables, or DEX, (the executable file format of Android's Dalvik virtual machine)) with different SHA-256 checksums.

- Structurally similar: All calls similar Java methods, including the Android API method `android.telephony.SmsManager.sendTextMessage`.

- Shallow call graphs: All external API invocations are done in the entry Activity callback method `com.agewap.om.OperaMiniActivity.onCreate`.

- No implicit control flows: All method calls are *explicit* invokes (i.e., the `*-invoke` Dalvik VM instruction) rather than *implicit* ones (e.g., Java reflections [137]). A real example of extensive implicit invokes is discussed in Section 6.3.

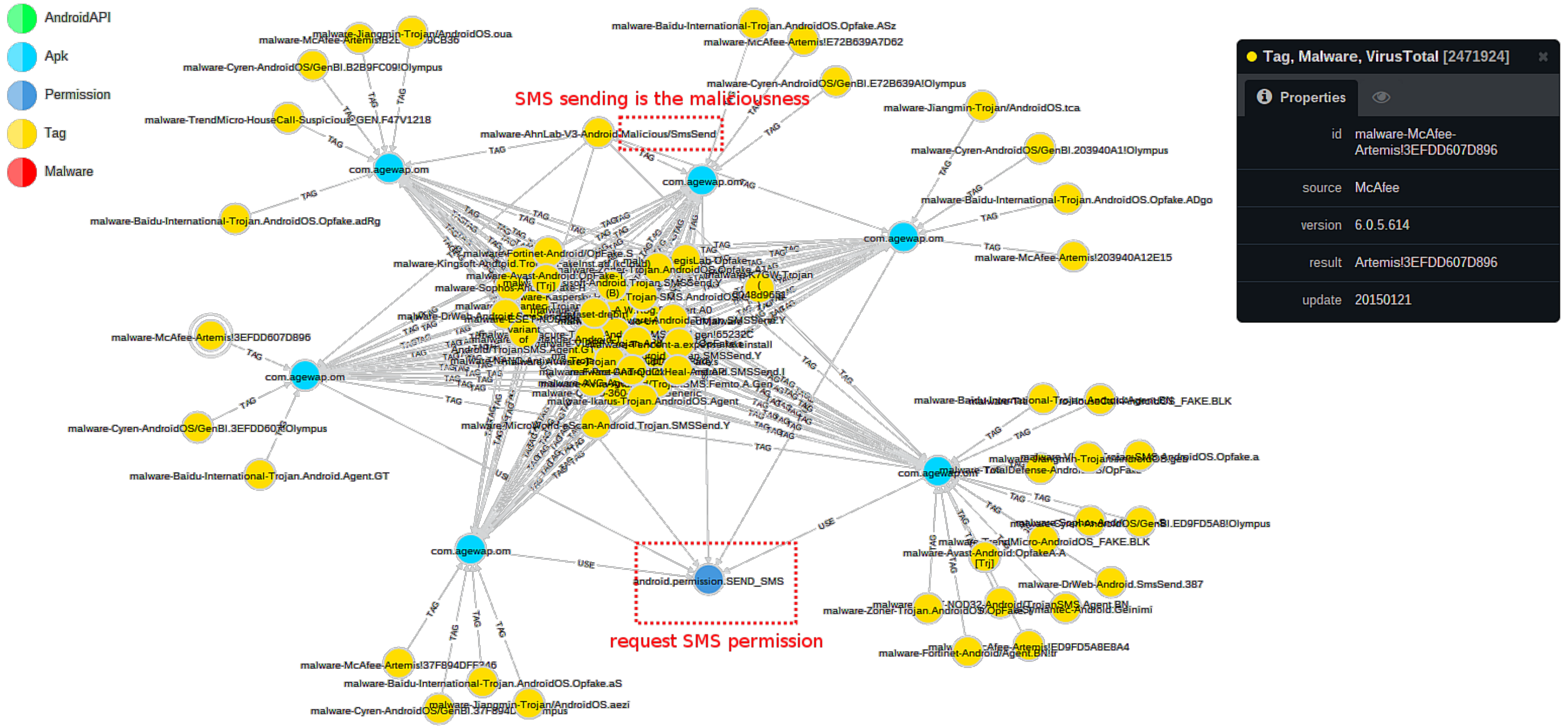
With the following query:

```
MATCH (a:Apk) WHERE a.package="com.agewap.om"
MATCH (p:Permission)<--a<--(t:Tag)
return *
```

we observe in Figure 6.2 that all 6 samples request the `android.permission.SEND_SMS` permission (i.e., to invoke SMS-sending API) and are flagged by multiple anti-virus (AV) software vendors [207]. Some of the labels indicate that its maliciousness is sending premium SMS without user's consent, for example, the AhnLab[3]'s label is `Malicious/SmsSend`.

These are simple examples of the insights about a group of APKs (and their relationship) that can be answered by the work presented in this section. More intricate examples and further evaluation are presented in later sections. What is

[3] `http://ahnlab.com`


AndroidAPI
Apk
Permission
Tag
Malware
SMS sending is the maliciousness
request SMS permission
com.agewap.om
android.permission.SEND_SMS
USE
TAG
malware-AhnLab-V3-Android.Malicious/SmsSend
malware-Baidu-International-Trojan.AndroidOS.Opfake.ASz
malware-McAfee-Artemis!E72B639A7D62
malware-Cyren-AndroidOS/GenBl.E72B639A!Olympus
malware-Jiangmin-Trojan/AndroidOS.tca
malware-Cyren-AndroidOS/GenBl.203940A1!Olympus
malware-Baidu-International-Trojan.AndroidOS.Opfake.ADgo
malware-McAfee-Artemis!203940A12E15
malware-Jiangmin-Trojan/AndroidOS.oua
malware-Cyren-AndroidOS/GenBl.B2B9FC09!Olympus
malware-TrendMicro-HouseCall-Suspicious_GEN.F47V1218
malware-Baidu-International-Trojan.AndroidOS.Opfake.adRg
malware-McAfee-Artemis!3EFDD607D896
malware-Cyren-AndroidOS/GenBl.3EFDD607!Olympus
malware-Baidu-International-Trojan.Android.Agent.GT
malware-Kingsoft-Android.Troj...
malware-Fortinet-Android/OpFake.S
malware-Avast-Android:OpFake-...
malware-Sophos-AndrOpFake...
malware-DrWeb-Android.SmsSend...
malware-ESET-NOD32...
malware-Ikarus-Trojan.AndroidOS.Agent
malware-MicroWorld-eScan-Android.Trojan.SMSSend.Y
malware-TrendMicro-AndroidOS_FAKE.BLK
malware-Avast-Android:OpfakeA-A [Trj]
malware-DrWeb-Android.SmsSend.387
malware-Zoner-Trojan.AndroidOS.OpFake...
malware-Fortinet-Android/Agent.BN!tr
malware-McAfee-Artemis!ED9FD5A8E8A4
malware-Cyren-AndroidOS/GenBl.ED9FD5A8!Olympus
malware-McAfee-Artemis!37F894DFF246
malware-Baidu-International-Trojan.AndroidOS.Opfake.aS
malware-Jiangmin-Trojan/AndroidOS.aezi
malware-Cyren-AndroidOS/GenBl.37F894D...!Olympus

● Tag, Malware, VirusTotal [2471924]

Properties

id	malware-McAfee-Artemis!3EFDD607D896
source	McAfee
version	6.0.5.614
result	Artemis!3EFDD607D896
update	20150121

Figure 6.2.: The anti-virus software scanning result labels and the permissions requested by the `com.agewap.om` APK samples in the prototype Web of APKs.

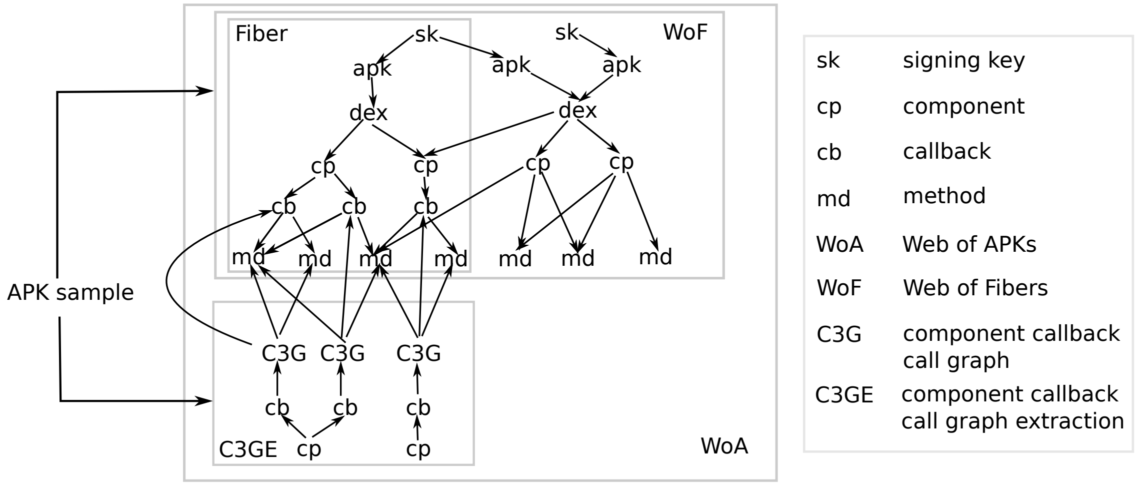


Figure 6.3.: Web of APKs (WoA), which represents both “a declarative graph analysis approach towards APK anlaysis” and “a system that implements this approach,” consists of two interacting components: Web of Fibers (WoF) and Component Callback Call Graph Extraction (C3GE). Details of WoF and C3GE are presented in Sections 6.4 and 6.5, respectively.

to be noted here is the declarative approach of addressing these questions, and the simplicity of the queries that generate the results.

6.1.3 Objective

Figure 6.3 illustrates the essence of this work. Web of APKs (WoA), which represents both “a declarative graph analysis approach towards APK anlaysis” and “a system that implements this approach,” consists of two interacting components: Web of Fibers (WoF) and Component Callback Call Graph Extraction (C3GE). The function of and relation between these components are:

- A Fiber is a hierarchical property graph model of an individual APK sample that consists of the following layers of *structural nodes*: signing key, APK, DEX (Dalvik EXecutable; the executable format of APK), component classes (components), component callback methods (callbacks), transitive invocatees of the callbacks, call graph nodes, and method invocation instances (optional).

- Auxiliary information, such as Android permissions defined or used by the APK, Android intent filters that designate events to trigger component callbacks, and arbitrary tags (such as data source, manual malware family labels, and anti-virus software scanning results tags crawled from the VirusTotal service [207]), are attached to Fiber for use in query refinement.

- The inter-procedural structural information extracted from an APK sample by C3GE supplies both "the syntactic information for the lowest (transitive invocatee) layer of a Fiber" and "the structural signature that complements the syntactic information of the upper layers of a Fiber."

- Fibers of the whole APK pool are "weaved" into a single Web of Fibers (WoF), which can be declaratively queried against to answer APK analysis questions.

Essentially, by syntactic matching and structural signatures computed with the help of C3GE, the Fibers of APKs are weaved into a single Web of Fibers, which can be queried against to support the proposed declarative graph analysis approach towards APK analysis. The "fiber weaving into a web" metaphor gives each part its name.

Given this, the objective of this chapter is to demonstrate that:

> The declarative approach as proposed and implemented in Web of APKs (WoA) is an effective approach for identifying and explaining similarity between APKs.

To support the objective, we first review existing literature (Section 6.2) in the following lines of research on the Android platform: app repackaging/plagiarism detection (Section 6.2.1), malware detection (Section 6.2.2), dynamic app analysis (Section 6.2.3), and APK obfuscations (Section 6.2.4). Analysis of a real Android malware sample is presented in Section 6.3, which, besides elucidating some real malware detection evasion techniques and demonstrating the need for more robust APK analysis tools (a need that this work attempts to meet), also serves as a concrete running example in the following sections. We then present the model, design, and algorithms of the two pillars of this work—Web of Fibers (WoF) and Component Callback Call Graph Extraction (C3GE)—in Sections 6.4 and 6.5, respectively. Evaluation using a million-node-scale Web of APKs modeling over 5,000 Android malware samples is presented in Section 6.6.

6.1.4 Contribution

The chapter makes the following contributions.

- Propose and implement a declarative approach (Web of APKs/WoA) to Android APK analysis, which formulates the "similar APK identification and explanation" task as a graph analysis problem.
- Propose and implement a set of static-analysis techniques for:
  - Detecting and resolving (external-dependency-free) implicit control flows on Android that include Java reflections and Java/Android asynchronous threads/tasks.
  - Extracting Component Callback Call Graphs (C3Gs) with explicit considerations for adversary scenarios such as malware analysis.
- The improved inter-procedural control flow extraction capability of these techniques combined together over the popular *Androguard* APK analysis tool [62, 63] used in multiple previous works [85, 164, 105, 163, 219] is demonstrated on real Android malware samples. Obfuscation robustness is analyzed and evaluated against the obfuscators in ADAM APK obfuscation framework [231].
- Build a million-node-scale Web of APKs modeling over 5,000 Android malware samples and evaluate the proposed methodology over it.

In addition, the underlying implementation expounded in this chapter is publicly released [4] in both source and executable formats for result reproducibility [150] and

[4] `https://github.com/pw4ever/web-of-apks`

future extensions. Thus, the chapter and the implementation are a symbiotic pair: The implementation provides results and supports reproducibility of the chapter; the chapter documents the motivation, algorithms, design trade-offs, and internals of the implementation.

6.2 Related Work

In this section, we review a few lines of research on Android app/APK analysis that closely relate to the work presented in this chapter: app repackaging/plagiarism detection (Section 6.2.1), malware detection (Section 6.2.2), dynamic app analysis (Section 6.2.3), and APK obfuscations (Section 6.2.4).

6.2.1 Android App Repackaging/Plagiarism Detection

Desnos [62] defines a Dalvik bytecode similarity metric based on Normalized Compression Distance (NCD) [48] and applies it to find lists of identical, similar, and new/deleted methods between a pair of Android apps. NCD is a symmetric similarity metric between a pair of strings that is based on the idea that compressing the concatenation of two similar strings will produce a shorter output than if the two strings are not similar. The essence of this work are: 1. For each method of an APK sample, generate a string that encode a (linearized) control flow graph, external API (Android/Java) invocations, and Java exceptions. 2. Use NCD to compute similarity between a pair of such strings. 3. Similar strings correspond to similar methods.

Zhou et al. adapt Context Triggered Piece-wise Hashing (CTPH) [120] in their DroidMOSS [233] system to measure pair-wise similarity of apps for detecting app repackaging. The essence of this work are: 1. Split each long sequence of executable code into short sequences when the hash value of a sliding window is prime; this procedure is named “resetting” in the paper. 2. Local changes, such as insertion of an invocation instruction in a repackaged app, is localized by the sliding windows and resetting criteria. 3. Hashes of the short sequences are concatenated, and the edit distance [36] of two such concatenated hashes are their corresponding code's similarity. 4. Only the opcode, rather than the easily replaceable operands, of each code piece is used in computing the above hashes.

Crussell et al. use Program Dependence Graphs (PDGs) [79] in their DNADroid [55] system to detect similar apps with different authorship, or app "clones" as called in the paper. The essence of this work are: 1. Convert Android app code from (Android's) DEX format into (Java's) JAR format using the tool `dex2jar` [194]. 2. Remove common libraries from the code by Java package name and code checksum. 3. Use the existing Java analysis framework WALA [41] to extract the data dependency part of PDGs. 4. The justification of not including the control flow part of PDGs is "so that our detection is more robust against statement reordering, insertion and deletion" [55]. 5. Apply heuristics (called "filters" in the paper) to remove pairs of apps that are unlikely to be clones from the pair-wise comparison; one example of such heuristics is to remove PDGs that have less than 10 nodes. 6. Use the VF2 subgraph isomorphism algorithm [52] to identify isomorphic subgraphs in the PDGs, and use the node number ratio of isomorphic subgraphs in the PDGs as app similarity score.

Hanna et al. use k-gram feature of code sequence and feature hashing in their Justapp[97] system to detect method-level similarity between Android apps. The essence of this work are: 1. Extract basic blocks of each Java method from the APK. 2. Hash every k opcodes, along with all constant operands (but not variable operands), and set a corresponding bit in a feature bit-vector of the method. 3. The justification of dropping variable operands but retaining constant operands is that variable operands are easily changeable but constant operands may contain control flow information as used by Java's reflection mechanism [46]. 4. Use agglomerative hierarchical clustering [61] on the aforementioned feature bit-vector to group similar apps together.

In their PiggyApp [234] system, Zhou et al. identify one type of app repackaging (what they call "piggybacked" apps), characterized by the relative independence of the added code (the "rider") with the original apps (the "carrier"), and propose to identify pigbacked apps by isolating and comparing the primary module (corresponding to the carrier) for similarity. The essence of this work are: 1. Build a weighted graph of Java packages that represents Java package's data/control dependence (including class inheritance, package homogeny, method calls, and member field references). 2. Use agglomerative clustering algorithm to group class packages into different modules. 3. Use information declared in `AndroidManifest.xml` to determine the primary modules, and use identifier (string) similarity to break tie. 4. Generate a bit vector (the feature vector) of the primary modules that representing the presence/absence of the following features: Requested permission, Android API calls, intent types, use of native code/external classes, and authorship information. 5. Embed the bit vector into a metric space, and use Jaccard distance of primary module feature vectors as similarity score of apps. 6. Use Vantage Point Tree (VPT) [222] as the metric space representation to reduce complexity from $O(n^2)$ to $O(n \log n)$.

In AnDarwin [56], Crussell et al. base on works of Gabel et al. [82] and Jiang et al. [112] to extract "semantic vectors" (histogram of code type frequencies) and identifying similar code using Locality Sensitive Hashing (LSH) [9]. The essence of this work are: 1. Split data dependency graph by connected components (the "semantic blocks") and use the histograms of code type frequencies (the "semantic vector") as the feature of each semantic block. 2. Semantic blocks with fewer than 10 nodes are discarded to reduce program size, "because they represent trivial and uncharacteristic code." 3. LSH are applied to each group of similarly sized semantic vectors (another heuristics to reduce problem size) to identify similar semantic vectors. 4. Common libraries are identified by their frequencies in all semantic vectors, and are removed from the data feeding to the next stage. 5. MinHash [33] are used to compute an efficient approximation to Jaccard-distance-based distance between pair-wise apps.

In FSquaDRA [229], Zhauniarovich et al. use non-code resources contained in APKs for app repackaging detection to eschew the computational complexity associated with code analysis. This is also used by Viennot et al. in their large-scale study [205] of the Google play store for detecting repackaged apps.

Unlike several of the methods mentioned above that eschew control flow graph (CFG) in favor of data dependence graph (DDG), in their work [44], Chen et al. embed Java-method-level CFG of apps (the “3D-CFG”) into a 3-dimensional metric space, and use a weighted centroid (center of mass) of the 3D-CFG to represent the underlying method. Similarity of app methods is computed from the distance between their 3D-CFG centroids. Scalability is derived from the observed “centroid effect”, i.e., proximity of centroids is positively correlated with method similarity, and is achieved by localizing the search space to similar methods with centroids that are proximate in the embedding metric space.

In ViewDroid [225] by Zhang et al., a “featured view graph” is constructed from each APK sample, and app repackaging/plagiarism detection is performed pair-wisely on the corresponding featured view graphs with subgraph isomorphism. A featured view graph consists of user interface views (corresponding to Java classes that are derived from `android.app.Activity` in Android) as nodes and the transition relationship between the views as edges; Android APIs and the event callback method that triggers view transition are used to label the nodes and edges, respectively. Like in Crussell et al.'s work [55], the VF2 [52] subgraph isomorphism detection algorithm is used for this purpose, perhaps due to the availability of its implementation [114].

6.2.2 Android Malware Detection

Zhou et al. propose and implement a set of heuristics, including risky permission request and dynamic Java class or native library loading, for detecting suspicious behaviors in DroidRanger [237]. The suspicious apps are further manually triaged to determine whether they are malicious. The discovered malware samples are collected and released as the Android Malware Genome Project (AGMP) [236], which is highly cited and also used in evaluating the present work, as presented in Section 6.6.

In their work [85], Gascon et al. use Androguard to extract call graphs from APK, label each extracted method with a 15-dimensional binary vector that represent

Dalvik bytecode categories that are used in that method, and use a procedure inspired by neighborhood hash graph kernel (NHGK) [100] to encode 1-hop neighborhood of each method on the call graph. Histograms of neighborhood hashes are used as the kernel function for a linear support vector machine (SVM) [192] classifier to classify malware.

In DroidMiner [219], Yang et al. propose a two-tier graphical model to capture behavioral modality of Android apps. The upper layer, Component Dependency Graph (CDG), represent interactions among the four types of Android components (Activities, Services, Broadcast Receivers, and Content Observers). The lower layer, Component Behavior Graph (CBG), "represents control-flow logic of those permission-related Android and Java API functions, and actions performed on particular resources of each component." [219] Malware modalities are defined as ordered sequence of sensitive functions from known malware samples' behavior graphs, and the detection of malware is formulated as an association rule mining [101] problem from modalities to malicious behaviors. Although the implementation is not (yet) released, it is mentioned specifically that "DroidMiner extracts it callee methods by analyzing the invoke-kind instruction (e.g., invoke-virtual nad invoke-direct) used in the method." [219]

In Apposcopy [78], Feng et al. combine static taint analysis with a program representation called inter-component call graph (ICCG) to derive control/data flow signatures of privacy-leaking malware. A Datalog [4]-based specification language is proposed for querying whether app samples match signatures of known malware families.

In Drebin [16], Arp et al. extract multiple textual features, including hardware components, requested permission, app components, intents, API calls, used permission, network addresses, from app's manifest and executable code, and build a high-dimensional metric space consisting of binary dimensions (indicating whether a textual feature appears in the app sample). Linear SVM is trained and applied as a malware classifier, and the dimensions that contribute the most to the classification are extracted as explanations for the results. A contribution made in this work is the release of a 5,000+ malware dataset, which subsumes the aforementioned AGMP dataset (sans the malware family labels). This dataset is also used in building the prototype WoA that is evaluated in Section 6.6.

Some other works on Android malware detection includes the following. Zheng et al. propose a 3-level signatures for identifying Android apps: The method-level signature is a hash of a method's Android API call sequence; the class-level signature is sorted concatenation of method-level signatures; the app-level signature is concatenation of class-level signatures of the classes that are used in the APK. Protsenko and Müller propose to use software complexity metrics from software engineering research, such as McCabe's Cyclomatic Complexity [136] and the Chidamber and Kemerer Metrics Suite [47], for Android malware detection [164].

Also related to the present work is a line of works that uses app's requested permission [68, 22, 76, 18, 178, 149, 86, 87] or "requested permission plus de-compiled source code" [40] for risk assessment. These works and the present work complement each other by providing better information extraction and processing capabilities.

6.2.3 Dynamic Android App analysis

TaintDroid [70] is a highly cited work that brings privacy leak in Android to the attention of security research community and technology media circles. The abundance of personal and private information about their users on Android devices, coupled with the well-specified and standard app execution environment on Android, make the *sources* (e.g., the IMEI phone identifier) and *sinks* (e.g., out-going Internet connection) of *sensitive* information straightforward to identify and, therefore, the privacy leak problem well defined: Any sensitive information that is obtained from some source and eventually flow into some sink constitutes a *potential* privacy leak.

The word "potential" in the aforementioned definition of privacy leak is underlined by AppIntent [221], which differentiates between "user-intended data transmission" and "unintended data transmission." AppIntent presents a symbolic execution technique called *event-space constraint guided symbolic execution* to derive the UI actions that trigger sensitive data transmission; these UI actions are reported as context information for the analyst to decide whether the data transmission is intended by user or not—only the latter case is designated as a privacy leak.

Another contribution of TaintDroid is that it first implements and demonstrates the use of a venerable system analysis technique, *taint analysis* (the TaintDroid paper [70] has a succinct but extensive survey on previous applications of taint analysis), in Android. TaindDroid's *dynamic* taint analysis implementation is adopted by later works such as AppsPlayground [168].

Static taint analysis is an alternative to the approach taken by TaintDroid. The advantage of the static approach is (its potential of being) more comprehensive in coverage (due to the reactive nature of Android framework, the coverage of the static approach highly depends on the sophistication of the implementation) and that it will not be detected and countered by malicious apps. FlowDroid [17, 81] is a recent work that addresses the main concern of *static* taint analysis—fidelity of the static tainting model. More specifically, FlowDroid presents a static tainting model with "context, flow, field, and object sensitivity" that handles callbacks invoked by the Android framework and reduces false alarms. Another contribution of FlowDroid is to present an open taint-analysis benchmark called DroidBench.

Another line of works, started by DroidScope [218] and followed up by, for example, CopperDroid [170], focuses on understanding malicious app's behaviors by reconstructing app semantics through virtual machine introspection (VMI) techniques. The advantage of these "out-of-the-box" malware analysis frameworks is their transparency—they are invisible to even priviledged malware in the system under observation. DroidScope demonstrates the capability of tracing system calls, dalvik virtual machine instruction traces, and Android API invocations, while CopperDroid focuses on tracing system calls and inter-process communications (IPCs) through the Android-specific Binder mechanism. However, their advantage in transparency through being “out-of-the-box” also restricts their applicability to environments which can be effectively virtualized and instrumented, in other words, in Android’s own emulators. Porting these works to different emulators (for example, the Genymotion emulator[5]) or even a different version of Android is a non-trivial effort—this is demonstrated by the incomplete port of DroidScope to Android 4.3 (Jellybean) from their original 2.3 (Gingerbread) version as of June 2014[6]

The event-driven nature of Android apps breaks the linearity of traditional batch-processing programs, and requires specific testing of its GUI-driven behaviors for thoroughness. Early works on Android UI testing [104] adapts the widely applied fuzzing testing technique (which essentially tests the app against numerous randomly generated and, sometimes, ill-formed input events). AppInspector [90] mentions the use of symbolic execution for more thoroughly exercising GUIs.

Later works, such as SmartDroid [230], Dynodroid [133], and AppsPlayground [168], focus on improving the efficiency and effectiveness of UI testing by “being less random and more guided.” SmartDroid focuses on identifying UI-based trigger conditions of sensitive behaviors in Android apps, with the intention of combining with privacy leak detector such as TaintDroid to reveal such leaks. The main idea is to use statically extracted “activity activation” and “function call graphs” to trim the exponentially growing condition space. Dynodroid instruments the Android framework to observe UI event responses of apps, and therefore forms an “observe-select-execute” loop at the granularity of single input events. AppsPlayground incorporates TaintDroid for privacy leak detection and presents a heuristics to de-duplicate UI element identifications.

Some other ideas that have been explored include the followings. Crowdroid [38] presents the idea of crowsourcing Android app analysis to detect apps that “have the same name/version but behave differently” as an indicator of trojan-horse infection. Whyper [148] applies Natural Language Processing (NLP) analysis of human-oriented app metadata (e.g., description on Play Store) to decide whether a requested permis-

[5] `http://www.genymotion.com/`

[6] `https://code.google.com/p/decaf-platform/wiki/DroidScope`

sion is warranted. AsDroid [106] detect apparent discrepancies of the human-oriented metadata (e.g., the text of a Button widget) and the triggered function of UI elements to identify app's stealthy behavior.

6.2.4 APK Obfuscations

APK obfuscations are techniques that transform APK binaries between different forms while preserving the app semantics. The official Android development environment integrates into the app development process a tool named `ProGuard` that "shrinks, optimizes, and obfuscates your code by removing unused code and renaming classes, fields, and methods with semantically obscure names." [14] As discussed below, works on APK obfuscations often adapt ideas from earlier research on Java program obfuscations [227, 102, 176, 51, 131] due to Android's root in Java: The officially supported development process begins with Java source code, and first goes through Java compilation before eventually compiled to the Android Dalvik virtual machine bytecodes. App plagiarists and malware authors obfuscate their apps to evade detection. Benign app authors obfuscate their apps to increase the difficulty of plagiarization. Even without such intentions, the generated app binaries could by changing/upgrading app build toolchain or enabling aggressive optimization during the build process.

In their work on stress-testing Android anti-virus systems [231], Zheng et al. discusses two broad categories of obfuscation techniques: repackaging and bytecode-level obfuscations. More specifically, repackaging techniques include: 1. Realigning uncompressed data within an APK file on n-byte boundaries so that all portions can be directly memory mapped. Android's built-in `zipalign` tool uses $n = 4$ by default, and Zheng et al. uses $n = 8$. 2. Resigning APK with uncertified private keys with `keytool` and `jarsigner` tools. 3. Unpacking and rebuilding the APK, which, besides the difference in signing key as above, may generate a different APK due to the difference in build environment/toolchain. Bytecode-level obfuscations include: 1. Inserting defunct methods to change the method table in the Dalvik EXecutable (DEX) binary contained in APK. 2. Renaming methods to evade method-name matching. 3. Inserting goto instructions to change the control flow within a method. 4. Applying simple string obfuscation such as alphabet rotation to evade direct string matching.

In DroidChameleon [169, 167], Rastogi et al. present a more extensive sets of obfuscation techniques that include repackaging and the following two classes of obfuscation, named "transformation attacks detectable by static analysis (DSA)" and "transformation attacks non-detectable by static analysis (NSA)" by the author. 1. DSA: package/identifier renaming, call indirections, code reordering, junk code insertion, function outlining/inlining, payload obfuscations 2. NSA: Java reflections, bytecode encryption using dynamic class loaders.

In PANDORA [163], besides some common obfuscation techniques as mentioned above, Protsenko and Muller propose the following techniques: 1. Replacing plain strings/numbers by arithmetic/string operations that produce equivalent values; examples: Vigenere encryption, replacing $4 + 1$ with $6 - 1$, array index shift, move Java method locals to composite types (maps/sets/lists) in the containing Java class. 2. OOD obfuscations: extract method, encapsulate field (using getter/setter), move fields/methods from one class to another, merge methods.

Although it is noted in literature that few techniques beyond what is provided by ProGuard are widely applied in real apps [14], and some of the techniques surveyed above are difficult to apply automatically without violating Android's stringent executable format verification [105], they nevertheless show what dedicated adversary could do to evade detection.

6.3 Analysis of a Real Android Malware Sample

This section presents the detailed analysis of a real Android malicious app. Besides revealing some detection evasion techniques employed by real Android malware, the analysis motivates the design and implementation of this work, in particular, Component Callback Call Graph Extraction (C3GE) presented in Section 6.5, by showing the inadequacy of a recursive call graph construction algorithm (Algorithm 2) adopted by, for example, the popular [85, 164, 105, 163, 219] Android analysis tool Androguard [63]. Moreover, this malware sample is used in later sections as a running example for illustrating various elements of this work.

6.3.1 About the Malware Sample

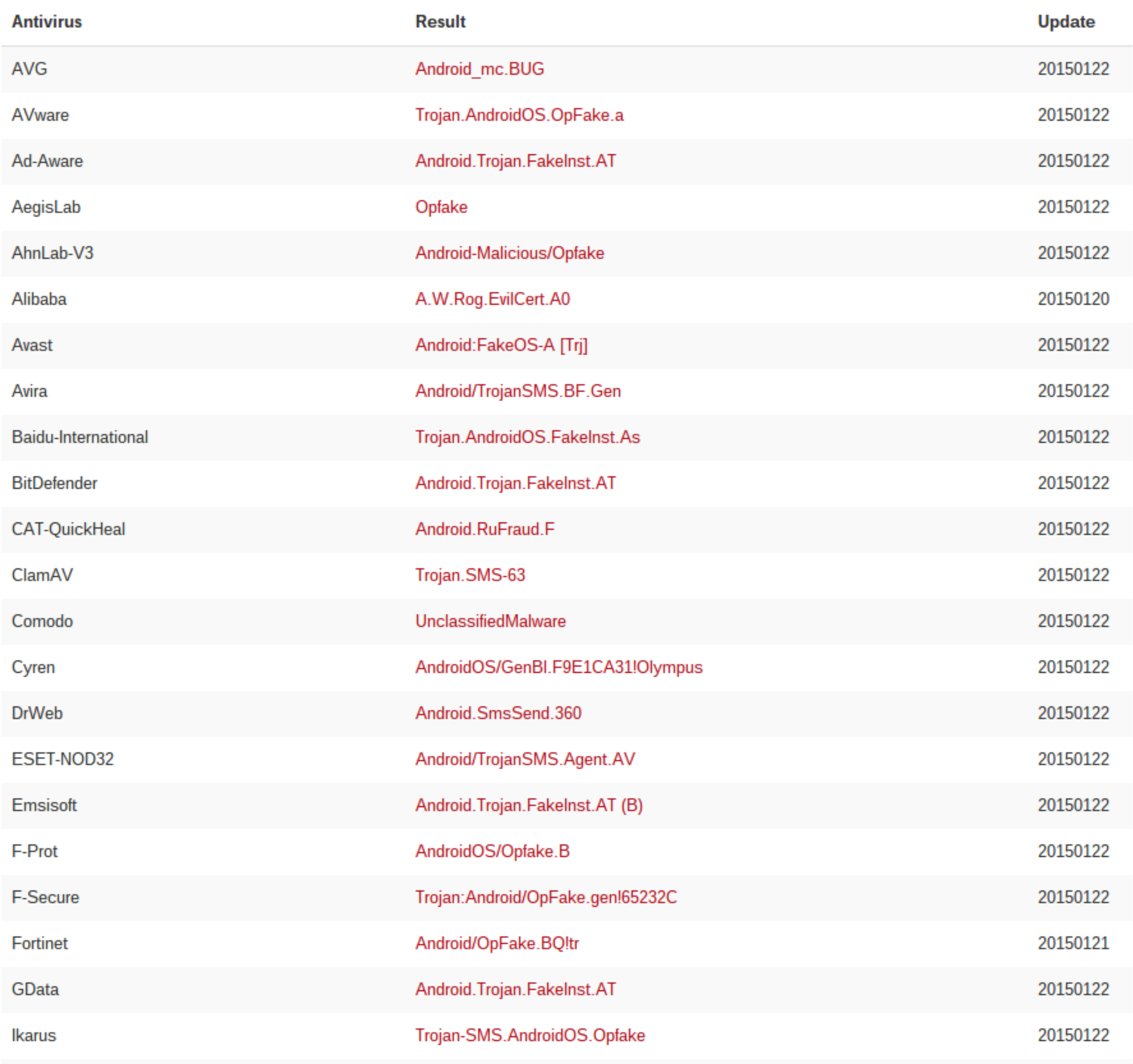
The malware is contained in the Drebin Android malware dataset [16, 185, 3] with the following identification information:

- SHA-256 checksum: a00f2b489dac150e513526ab285141d41a127133cd3be2115046e22e189ff2a3.
- Android package name: Jk7H.PwcD.
- Version code: 1.
- Size: 40,551 bytes.
- VirusTotal [207] result page: `http://goo.gl/6PqUYM`, with a snapshot obtained on January 2015 shown in Figure 6.4.

Reasons for using this malware sample for demonstration include: 1. It is structurally simple enough for an in-depth analysis that can be repeated by the reader. 2. It packs in its compact size some representative detection evasion techniques, including class name obfuscation, yet the class fields/methods retain their human-readable names. 3. It belongs to a multiply duplicated malware family that is captured in the Drebin dataset.

6.3.2 Obtaining and Decompiling the Sample

Since this malware sample is detected by multiple anti-virus software (Figure 6.4), there is limited risk in making it available for result reproducibility. Therefore, efforts



Antivirus	Result	Update
AVG	Android_mc.BUG	20150122
AVware	Trojan.AndroidOS.OpFake.a	20150122
Ad-Aware	Android.Trojan.FakeInst.AT	20150122
AegisLab	Opfake	20150122
AhnLab-V3	Android-Malicious/Opfake	20150122
Alibaba	A.W.Rog.EvilCert.A0	20150120
Avast	Android:FakeOS-A [Trj]	20150122
Avira	Android/TrojanSMS.BF.Gen	20150122
Baidu-International	Trojan.AndroidOS.FakeInst.As	20150122
BitDefender	Android.Trojan.FakeInst.AT	20150122
CAT-QuickHeal	Android.RuFraud.F	20150122
ClamAV	Trojan.SMS-63	20150122
Comodo	UnclassifiedMalware	20150122
Cyren	AndroidOS/GenBl.F9E1CA31!Olympus	20150122
DrWeb	Android.SmsSend.360	20150122
ESET-NOD32	Android/TrojanSMS.Agent.AV	20150122
Emsisoft	Android.Trojan.FakeInst.AT (B)	20150122
F-Prot	AndroidOS/Opfake.B	20150122
F-Secure	Trojan:Android/OpFake.gen!65232C	20150122
Fortinet	Android/OpFake.BQ!tr	20150121
GData	Android.Trojan.FakeInst.AT	20150122
Ikarus	Trojan-SMS.AndroidOS.Opfake	20150122

Figure 6.4.: Part of the VirusTotal scanning result of the APK sample a00f2b489dac150e513526ab285141d41a127133cd3be2115046e22e189ff2a3 obtained on 22 January 2015.

are made to simplify and document the procedure of reproducing the results shown in this and later sections. The reader is encouraged to follow these procedures to increase confidence in the results.

In an Internet-connected Linux machine with `bash` [161], `wget` [162], and `java` [53] (the Java virtual machine command-line starter) installed, execute the following shell commands to download the malware APK sample to user's home directory.

```
( URL1="https://raw.githubusercontent.com/pw4ever/web-of-apks/gh-pages/01sample/malware/";
  URL2="Jk7H.PwcD-1-a00f2b489dac150e513526ab285141d41a127133cd3be2115046e22e189ff2a3.apk";
  wget -nc -nd -O "$HOME/a00f2b.apk" \
    "${URL1}/${URL2}"; )
```

Next, set up utilities that will be used later, which include WoA's command-line interface `woa` and a script for decompiling APK binaries using the Soot [200, 121] Java optimization framework.

```
(
TARGET="$HOME/bin/";
mkdir -p ${TARGET};
wget -nc -nd -P ${TARGET} \
  https://raw.githubusercontent.com/pw4ever/web-of-apks/gh-pages/bin/soot-apk
wget -nc -nd -P ${TARGET} \
  https://raw.githubusercontent.com/pw4ever/web-of-apks/gh-pages/bin/woa
chmod +x ${TARGET}/{soot-apk,woa}
)
```

Lastly, decompile the DEX in the APK sample into Soot's Jimple [200] intermediate representation (IR) format, which is a typed 3-address IR format as shown shortly in, for example, Figure 6.5.

```
"$HOME/bin/soot-apk" a00f2b.apk
```

After this, a directory named `a00f2b` will be created in the user's home directory that contains the files mentioned below.

6.3.3 Analysis of an Invocation Path to Malicious Functionality

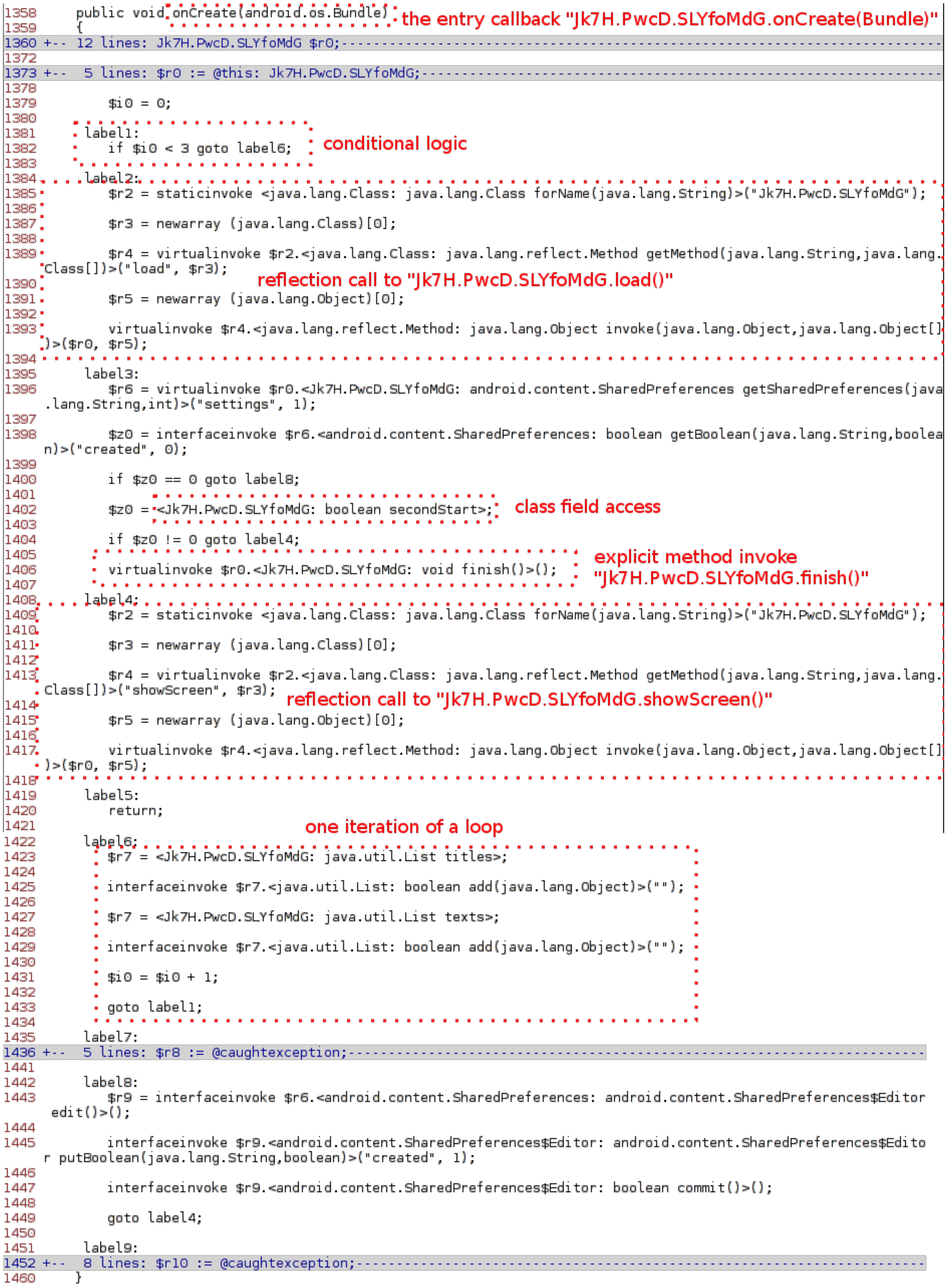
Anti-virus software scanning results (e.g., `DrWeb`'s label of this app sample contains the word `SmsSend`, as shown in Figure 6.4) suggest that the maliciousness of this app sample lies in unauthorized SMS message sending. In this section, *an* invocation path to the malicious SMS sending function is revealed to illustrate the detection evaision techniques employed by this real malware sample and, in Section 6.3.4, motivates the needs for APK analysis tools that are robust against these detection evasion techniques. Also in Section 6.3.4, other invocation paths besides the one presented here will be revealed by a simple query in the proposed system.

6.3.3.1 Step 1: `Jk7H.PwcD.SLYfoMdG.onCreate`

Figure 6.5 shows the first non-initializer Java method that is executed upon entry into the app: the method `Jk7H.PwcD.SLYfoMdG.onCreate`, which is the creation-event callback of the Activity class `Jk7H.PwcD.SLYfoMdG`. Points of interest include: 1. There is a 3-fold iteration between label 6 and label 7, with the conditional check on line 1,382. These iterations add 3 empty strings to class fields `titles` and `texts`. 2. Two methods, `load` and `showScreen`, of class `Jk7H.PwcD.SLYfoMdG` are implicitly invoked through Java reflections. 3. Method `finish` of class `Jk7H.PwcD.SLYfoMdG` is explicitly invoked if the boolean class field `secondStart` is false. We now follow the invocation path to `showScreen` next.

6.3.3.2 Step 2: `Jk7H.PwcD.SLYfoMdG.showScreen`

`showScreen`, the logic of which is shown in Figure 6.6, is invoked from `onCreate` implicitly through Java reflection. Points of interest include: 1. Field `state` of class `Jk7H.PwcD.SLYfoMdG` is accessed through Java reflection, and is compared against the number 0 in the branching conditional on line 1,876. 2. If `state` equals to 0 (line 1,874 and 1,876), method `setMain` of class `Jk7H.PwcD.SLYfoMdG` is invoked implicitly through Java reflection. 3. Otherwise, if `state` equals to 1 (line 1,890 and 1,892), method `setLicense` of class `Jk7H.PwcD.SLYfoMdG` is invoked implicitly through Java reflection. 4. Otherwise, if `state` equals to 2 (line 1,915 and 1,917), method `setEnd` of class `Jk7H.PwcD.SLYfoMdG` is invoked implicitly through Java reflection. It should be noted the extensive use of Java reflection for class field and method access here. We now follow the invocation path to `setMain` next.



```
    public void onCreate(android.os.Bundle)
    {
+--  12 lines: Jk7H.PwcD.SLYfoMdG $r0;-----------------------------------------------------------
+--   5 lines: $r0 := @this: Jk7H.PwcD.SLYfoMdG;---------------------------------------------------

        $i0 = 0;

     label1:
        if $i0 < 3 goto label6;

     label2:
        $r2 = staticinvoke <java.lang.Class: java.lang.Class forName(java.lang.String)>("Jk7H.PwcD.SLYfoMdG");

        $r3 = newarray (java.lang.Class)[0];

        $r4 = virtualinvoke $r2.<java.lang.Class: java.lang.reflect.Method getMethod(java.lang.String,java.lang.Class[])>("load", $r3);

        $r5 = newarray (java.lang.Object)[0];

        virtualinvoke $r4.<java.lang.reflect.Method: java.lang.Object invoke(java.lang.Object,java.lang.Object[])>($r0, $r5);

     label3:
        $r6 = virtualinvoke $r0.<Jk7H.PwcD.SLYfoMdG: android.content.SharedPreferences getSharedPreferences(java.lang.String,int)>("settings", 1);

        $z0 = interfaceinvoke $r6.<android.content.SharedPreferences: boolean getBoolean(java.lang.String,boolean)>("created", 0);

        if $z0 == 0 goto label8;

        $z0 = <Jk7H.PwcD.SLYfoMdG: boolean secondStart>;

        if $z0 != 0 goto label4;

        virtualinvoke $r0.<Jk7H.PwcD.SLYfoMdG: void finish()>();

     label4:
        $r2 = staticinvoke <java.lang.Class: java.lang.Class forName(java.lang.String)>("Jk7H.PwcD.SLYfoMdG");

        $r3 = newarray (java.lang.Class)[0];

        $r4 = virtualinvoke $r2.<java.lang.Class: java.lang.reflect.Method getMethod(java.lang.String,java.lang.Class[])>("showScreen", $r3);

        $r5 = newarray (java.lang.Object)[0];

        virtualinvoke $r4.<java.lang.reflect.Method: java.lang.Object invoke(java.lang.Object,java.lang.Object[])>($r0, $r5);

     label5:
        return;

     label6:
        $r7 = <Jk7H.PwcD.SLYfoMdG: java.util.List titles>;

        interfaceinvoke $r7.<java.util.List: boolean add(java.lang.Object)>("");

        $r7 = <Jk7H.PwcD.SLYfoMdG: java.util.List texts>;

        interfaceinvoke $r7.<java.util.List: boolean add(java.lang.Object)>("");

        $i0 = $i0 + 1;

        goto label1;

     label7:
+--   5 lines: $r8 := @caughtexception;----------------------------------------------------------

     label8:
        $r9 = interfaceinvoke $r6.<android.content.SharedPreferences: android.content.SharedPreferences$Editor edit()>();

        interfaceinvoke $r9.<android.content.SharedPreferences$Editor: android.content.SharedPreferences$Editor putBoolean(java.lang.String,boolean)>("created", 1);

        interfaceinvoke $r9.<android.content.SharedPreferences$Editor: boolean commit()>();

        goto label4;

     label9:
+--   8 lines: $r10 := @caughtexception;---------------------------------------------------------
    }
```

the entry callback "Jk7H.PwcD.SLYfoMdG.onCreate(Bundle)"

conditional logic

reflection call to "Jk7H.PwcD.SLYfoMdG.load()"

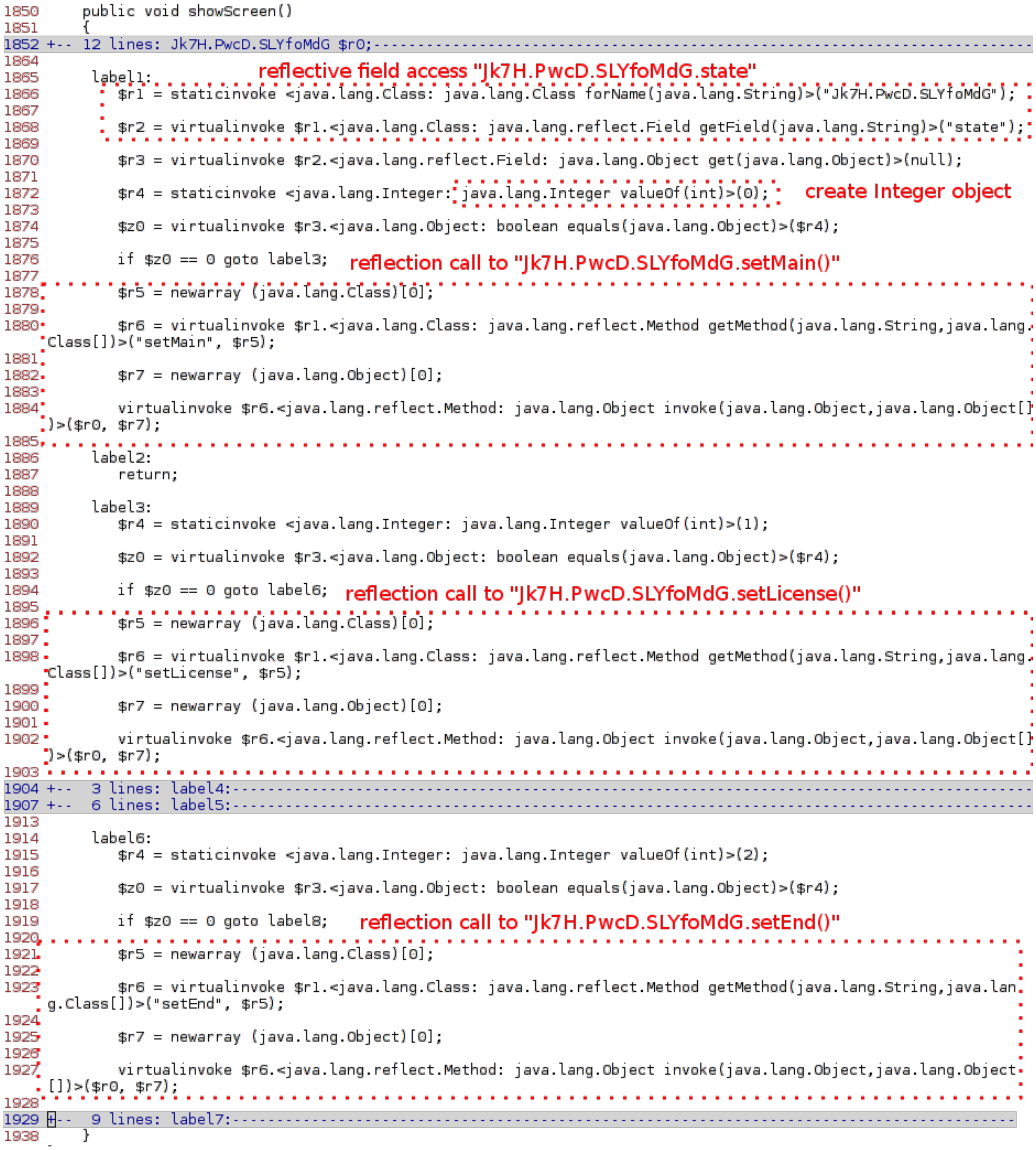
class field access

explicit method invoke "Jk7H.PwcD.SLYfoMdG.finish()"

reflection call to "Jk7H.PwcD.SLYfoMdG.showScreen()"

one iteration of a loop

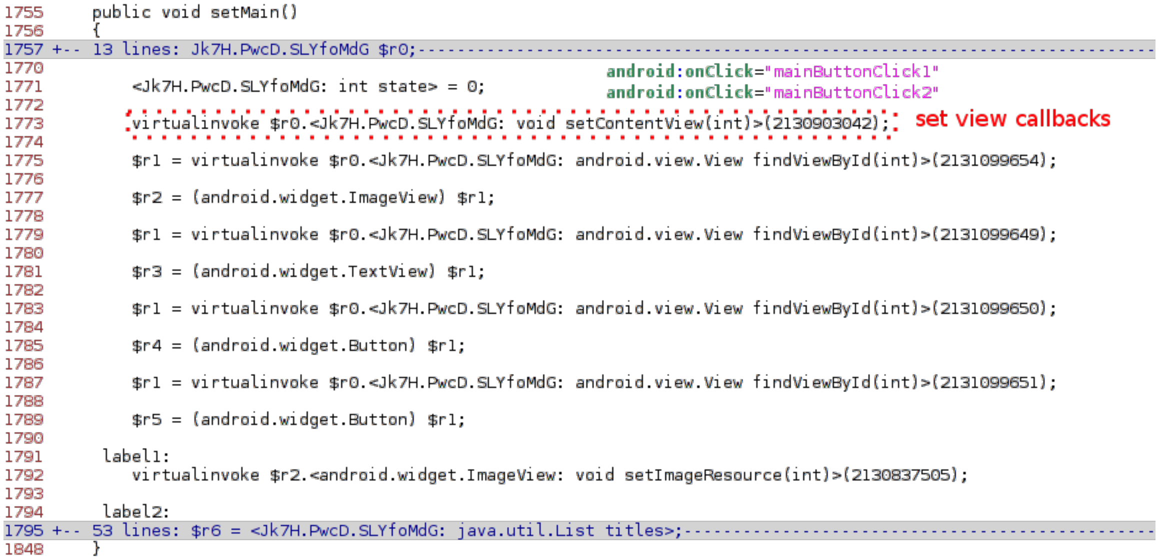
Figure 6.5.: The app component callback method Jk7H.PwcD.SLYfoMdG.onCreate, which is the first entry into the malware sample Jk7H.PwcD. File: Jk7H.PwcD.SLYfoMdG.jimple.



```
    public void showScreen()
    {
+-- 12 lines: Jk7H.PwcD.SLYfoMdG $r0;--------------------------------------------------------------------------

     label1:
        $r1 = staticinvoke <java.lang.Class: java.lang.Class forName(java.lang.String)>("Jk7H.PwcD.SLYfoMdG");

        $r2 = virtualinvoke $r1.<java.lang.Class: java.lang.reflect.Field getField(java.lang.String)>("state");

        $r3 = virtualinvoke $r2.<java.lang.reflect.Field: java.lang.Object get(java.lang.Object)>(null);

        $r4 = staticinvoke <java.lang.Integer: java.lang.Integer valueOf(int)>(0);

        $z0 = virtualinvoke $r3.<java.lang.Object: boolean equals(java.lang.Object)>($r4);

        if $z0 == 0 goto label3;

        $r5 = newarray (java.lang.Class)[0];

        $r6 = virtualinvoke $r1.<java.lang.Class: java.lang.reflect.Method getMethod(java.lang.String,java.lang.
Class[])>("setMain", $r5);

        $r7 = newarray (java.lang.Object)[0];

        virtualinvoke $r6.<java.lang.reflect.Method: java.lang.Object invoke(java.lang.Object,java.lang.Object[]
)>($r0, $r7);

     label2:
        return;

     label3:
        $r4 = staticinvoke <java.lang.Integer: java.lang.Integer valueOf(int)>(1);

        $z0 = virtualinvoke $r3.<java.lang.Object: boolean equals(java.lang.Object)>($r4);

        if $z0 == 0 goto label6;

        $r5 = newarray (java.lang.Class)[0];

        $r6 = virtualinvoke $r1.<java.lang.Class: java.lang.reflect.Method getMethod(java.lang.String,java.lang.
Class[])>("setLicense", $r5);

        $r7 = newarray (java.lang.Object)[0];

        virtualinvoke $r6.<java.lang.reflect.Method: java.lang.Object invoke(java.lang.Object,java.lang.Object[]
)>($r0, $r7);

+--  3 lines: label4:-------------------------------------------------------------------------------------------
+--  6 lines: label5:-------------------------------------------------------------------------------------------

     label6:
        $r4 = staticinvoke <java.lang.Integer: java.lang.Integer valueOf(int)>(2);

        $z0 = virtualinvoke $r3.<java.lang.Object: boolean equals(java.lang.Object)>($r4);

        if $z0 == 0 goto label8;

        $r5 = newarray (java.lang.Class)[0];

        $r6 = virtualinvoke $r1.<java.lang.Class: java.lang.reflect.Method getMethod(java.lang.String,java.lan
g.Class[])>("setEnd", $r5);

        $r7 = newarray (java.lang.Object)[0];

        virtualinvoke $r6.<java.lang.reflect.Method: java.lang.Object invoke(java.lang.Object,java.lang.Object
[])>($r0, $r7);

+--  9 lines: label7:------------------------------------------------------------------------------------------
    }
```

reflective field access "Jk7H.PwcD.SLYfoMdG.state"
create Integer object
reflection call to "Jk7H.PwcD.SLYfoMdG.setMain()"
reflection call to "Jk7H.PwcD.SLYfoMdG.setLicense()"
reflection call to "Jk7H.PwcD.SLYfoMdG.setEnd()"

Figure 6.6.: The method `Jk7H.PwcD.SLYfoMdG.showScreen`, implicitly invoked by the method `Jk7H.PwcD.SLYfoMdG.onCreate` through Java reflection. File: `Jk7H.PwcD.SLYfoMdG.jimple`.



```
1755     public void setMain()
1756     {
1757 +-- 13 lines: Jk7H.PwcD.SLYfoMdG $r0;-------------------------------------------------------------------
1770
1771         <Jk7H.PwcD.SLYfoMdG: int state> = 0;
1772
1773         virtualinvoke $r0.<Jk7H.PwcD.SLYfoMdG: void setContentView(int)>(2130903042);
1774
1775         $r1 = virtualinvoke $r0.<Jk7H.PwcD.SLYfoMdG: android.view.View findViewById(int)>(2131099654);
1776
1777         $r2 = (android.widget.ImageView) $r1;
1778
1779         $r1 = virtualinvoke $r0.<Jk7H.PwcD.SLYfoMdG: android.view.View findViewById(int)>(2131099649);
1780
1781         $r3 = (android.widget.TextView) $r1;
1782
1783         $r1 = virtualinvoke $r0.<Jk7H.PwcD.SLYfoMdG: android.view.View findViewById(int)>(2131099650);
1784
1785         $r4 = (android.widget.Button) $r1;
1786
1787         $r1 = virtualinvoke $r0.<Jk7H.PwcD.SLYfoMdG: android.view.View findViewById(int)>(2131099651);
1788
1789         $r5 = (android.widget.Button) $r1;
1790
1791      label1:
1792         virtualinvoke $r2.<android.widget.ImageView: void setImageResource(int)>(2130837505);
1793
1794      label2:
1795 +-- 53 lines: $r6 = <Jk7H.PwcD.SLYfoMdG: java.util.List titles>;-----------------------------------------------
1848     }
```

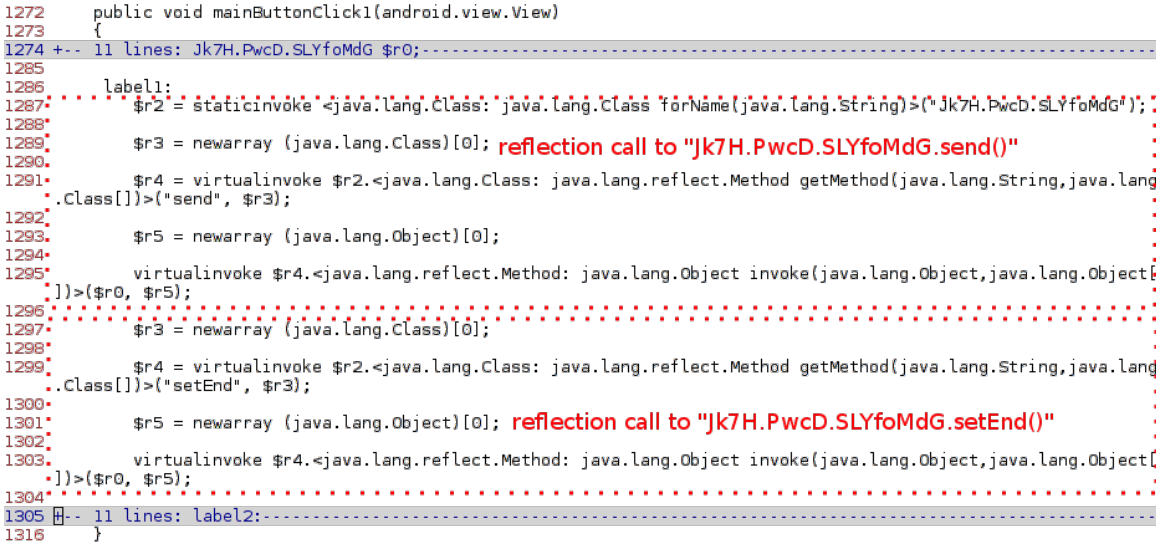
android:onClick="mainButtonClick1"
android:onClick="mainButtonClick2"
set view callbacks

Figure 6.7.: The method `Jk7H.PwcD.SLYfoMdG.setMain`, implicitly invoked by the method `Jk7H.PwcD.SLYfoMdG.showScreen` through Java reflection. File: `Jk7H.PwcD.SLYfoMdG.jimple`.

6.3.3.3 Step 3: `Jk7H.PwcD.SLYfoMdG.setMain`

`setMain`, the logic of which is shown in Figure 6.7, is invoked from `showScreen` implicitly through Java reflection. `setMain` uses the Android API `setContentView` of class `android.app.Activity` (inherited by the app class `Jk7H.PwcD.SLYfoMdG`) to set up an interactive user interface (UI) on the device's current screen, i.e., the current Activity in Android's parlance.

Analysis of the resource files contained in the APK reveals that the UI set up on line 1,773 contains two Button widgets (two "Views" in Android's parlance) that can be clicked by the user. Two methods, `mainButtonClick1` and `mainButtonClick2`, are invoked in response to user's button clicking action. Therefore, although the widget callback method `mainButtonClick1` is not *directly* invoked by `setMain`, there



```
    public void mainButtonClick1(android.view.View)
    {
+-- 11 lines: Jk7H.PwcD.SLYfoMdG $r0;---------------------------------------------------------------------

     label1:
        $r2 = staticinvoke <java.lang.Class: java.lang.Class forName(java.lang.String)>("Jk7H.PwcD.SLYfoMdG");

        $r3 = newarray (java.lang.Class)[0];

        $r4 = virtualinvoke $r2.<java.lang.Class: java.lang.reflect.Method getMethod(java.lang.String,java.lang
.Class[])>("send", $r3);

        $r5 = newarray (java.lang.Object)[0];

        virtualinvoke $r4.<java.lang.reflect.Method: java.lang.Object invoke(java.lang.Object,java.lang.Object[
])>($r0, $r5);

        $r3 = newarray (java.lang.Class)[0];

        $r4 = virtualinvoke $r2.<java.lang.Class: java.lang.reflect.Method getMethod(java.lang.String,java.lang
.Class[])>("setEnd", $r3);

        $r5 = newarray (java.lang.Object)[0];

        virtualinvoke $r4.<java.lang.reflect.Method: java.lang.Object invoke(java.lang.Object,java.lang.Object[
])>($r0, $r5);

+-- 11 lines: label2:----------------------------------------------------------------------------------
    }
```

reflection call to "Jk7H.PwcD.SLYfoMdG.send()"

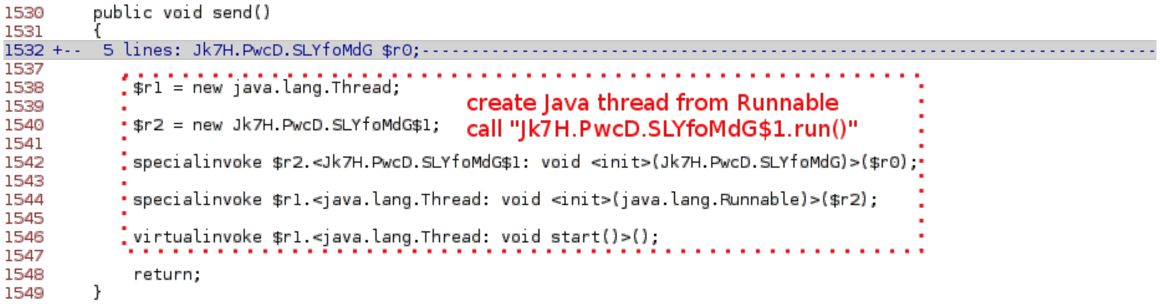
reflection call to "Jk7H.PwcD.SLYfoMdG.setEnd()"

Figure 6.8.: The method `Jk7H.PwcD.SLYfoMdG.mainButtonClick1`, implicitly invoked by the method `Jk7H.PwcD.SLYfoMdG.setMain` through view callback. File: `Jk7H.PwcD.SLYfoMdG.jimple`.

is a logically *implicit* invocation path from the latter to the former. We now follow this logic invocation path to `mainButtonClick1` next.

6.3.3.4 Step 4: `Jk7H.PwcD.SLYfoMdG.mainButtonClick1`

`mainButtonClick1`, the logic of which is shown in Figure 6.8, is invoked from `setMain` implicitly through button widget callback. `mainButtonClick1` invokes two methods, `send` and `setEnd`, implicitly through Java reflection. Note that there are two methods named `send` in class `Jk7H.PwcD.SLYfoMdG`, overloaded by their different arities (i.e., the number of method arguments). It is the 0-arity `send` that is invoked here. The 2-arity `send` is invoked later in the invocation path we analyze here (Section 6.3.3.7). We now follow the invocation path to the 0-arity `send` next.



```
1530     public void send()
1531     {
1532 +--   5 lines: Jk7H.PwcD.SLYfoMdG $r0;----------------------------------------------------------------------
1537
1538         $r1 = new java.lang.Thread;
1539
1540         $r2 = new Jk7H.PwcD.SLYfoMdG$1;
1541
1542         specialinvoke $r2.<Jk7H.PwcD.SLYfoMdG$1: void <init>(Jk7H.PwcD.SLYfoMdG)>($r0);
1543
1544         specialinvoke $r1.<java.lang.Thread: void <init>(java.lang.Runnable)>($r2);
1545
1546         virtualinvoke $r1.<java.lang.Thread: void start()>();
1547
1548         return;
1549     }
```

create Java thread from Runnable
call "Jk7H.PwcD.SLYfoMdG$1.run()"

Figure 6.9.: The method `Jk7H.PwcD.SLYfoMdG.send` (0-arity version), implicitly invoked by the method `Jk7H.PwcD.SLYfoMdG.mainButtonClick1` through Java reflection. File: `Jk7H.PwcD.SLYfoMdG.jimple`.

6.3.3.5 Step 5: (Zero-arity) `Jk7H.PwcD.SLYfoMdG.send`

The 0-arity method `send` of class `Jk7H.PwcD.SLYfoMdG`, the logic of which is shown in Figure 6.9, is invoked from `mainButtonClick1` implicitly through Java reflection. Although this method is a single basic block without any conditional logic, it uses a new implicit control flow that has not been discussed so far: asynchronous Java thread.

Specifically, class `Jk7H.PwcD.SLYfoMdG$1`, which implements the `java.lang.Runnable` interface, is fed as the initialization argument to an instance of the `java.lang.Thread` class on line 1, 544. Then, the `Thread` instance is started on line 1,546.

After this, although no internal methods (i.e., methods implemented in the APK) is directly invoked by the 0-arity `send`, method `run` of class `Jk7H.PwcD.SLYfoMdG$1` will be executed as a result of the asynchronous thread mechanism of the Dalvik virtual machine, which mirrors the identical mechanism on the Java platform. We now follow this logical invocation path to `Jk7H.PwcD.SLYfoMdG$1.run` next.

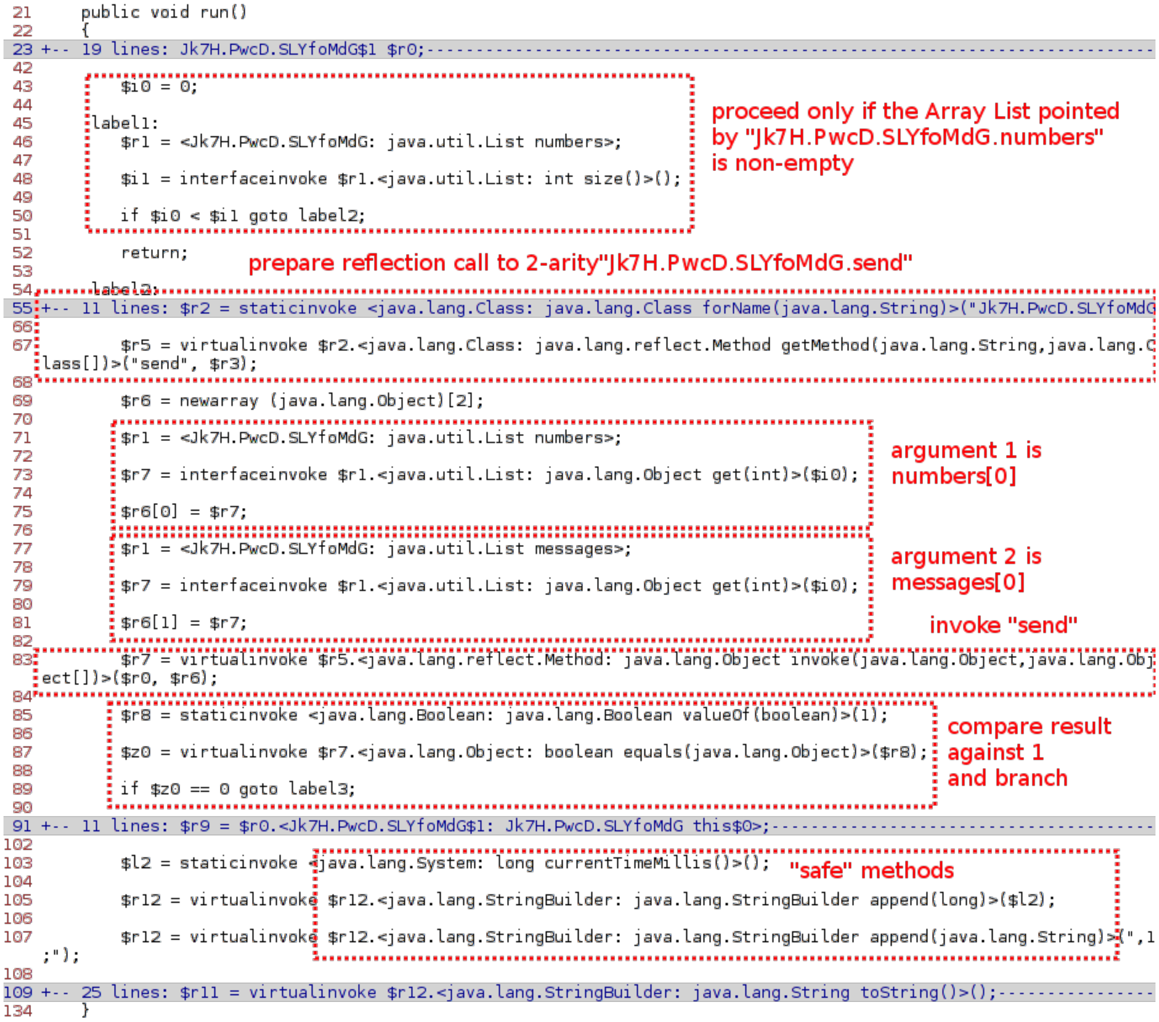
6.3.3.6 Step 6: `Jk7H.PwcD.SLYfoMdG$1.run`

Method `run` of class `Jk7H.PwcD.SLYfoMdG$1`, (a segment of) the logic of which is shown in 6.10, is invoked when the `java.lang.Thread` instance started in the 0-arity method `send` of class `Jk7H.PwcD.SLYfoMdG` is scheduled for execution by the hosting virtual machine. The logic of interest here is that, if the `java.util.ArrayList` pointed to by the field `numbers` of class `Jk7H.PwcD.SLYfoMdG` is non-empty (lines 43–50; the implication of this conditional on our design will be examined later in Section 6.5), the *2-arity* method `send` of class `Jk7H.PwcD.SLYfoMdG` is implicitly invoked through Java reflection. We now follow the invocation path to the 2-arity `send` next.

6.3.3.7 Step 7: (Two-arity) `Jk7H.PwcD.SLYfoMdG.send`

The 2-arity method `send` of class `Jk7H.PwcD.SLYfoMdG`, the logic of which is shown in 6.11, culminates the trace of following the invocation path by implicitly invoking Android API method that sends SMS text messages, i.e., method `sendTextMessage` of class `android.telephony.SmsManager`.

It is noted that the title (line 162) and content (line 166) of the text message are determined by the method arguments fed into `send`. Refer back to Figure 6.10, the two arguments are determined by the first elements of the Array Lists referred-to by the fields `numbers` and `messages` of class `Jk7H.PwcD.SLYfoMdG`, respectively.



```
    public void run()
    {
+-- 19 lines: Jk7H.PwcD.SLYfoMdG$1 $r0;-------------------------------------------------------------------------

        $i0 = 0;

     label1:
        $r1 = <Jk7H.PwcD.SLYfoMdG: java.util.List numbers>;

        $i1 = interfaceinvoke $r1.<java.util.List: int size()>();

        if $i0 < $i1 goto label2;

        return;

     label2:
+-- 11 lines: $r2 = staticinvoke <java.lang.Class: java.lang.Class forName(java.lang.String)>("Jk7H.PwcD.SLYfoMdG

        $r5 = virtualinvoke $r2.<java.lang.Class: java.lang.reflect.Method getMethod(java.lang.String,java.lang.C
lass[])>("send", $r3);

        $r6 = newarray (java.lang.Object)[2];

        $r1 = <Jk7H.PwcD.SLYfoMdG: java.util.List numbers>;

        $r7 = interfaceinvoke $r1.<java.util.List: java.lang.Object get(int)>($i0);

        $r6[0] = $r7;

        $r1 = <Jk7H.PwcD.SLYfoMdG: java.util.List messages>;

        $r7 = interfaceinvoke $r1.<java.util.List: java.lang.Object get(int)>($i0);

        $r6[1] = $r7;

        $r7 = virtualinvoke $r5.<java.lang.reflect.Method: java.lang.Object invoke(java.lang.Object,java.lang.Obj
ect[])>($r0, $r6);

        $r8 = staticinvoke <java.lang.Boolean: java.lang.Boolean valueOf(boolean)>(1);

        $z0 = virtualinvoke $r7.<java.lang.Object: boolean equals(java.lang.Object)>($r8);

        if $z0 == 0 goto label3;

+-- 11 lines: $r9 = $r0.<Jk7H.PwcD.SLYfoMdG$1: Jk7H.PwcD.SLYfoMdG this$0>;--------------------------------------

        $l2 = staticinvoke <java.lang.System: long currentTimeMillis()>();

        $r12 = virtualinvoke $r12.<java.lang.StringBuilder: java.lang.StringBuilder append(long)>($l2);

        $r12 = virtualinvoke $r12.<java.lang.StringBuilder: java.lang.StringBuilder append(java.lang.String)>(",1
;");

+-- 25 lines: $r11 = virtualinvoke $r12.<java.lang.StringBuilder: java.lang.String toString()>();---------------
    }
```

proceed only if the Array List pointed by "Jk7H.PwcD.SLYfoMdG.numbers" is non-empty

prepare reflection call to 2-arity"Jk7H.PwcD.SLYfoMdG.send"

argument 1 is numbers[0]

argument 2 is messages[0]

invoke "send"

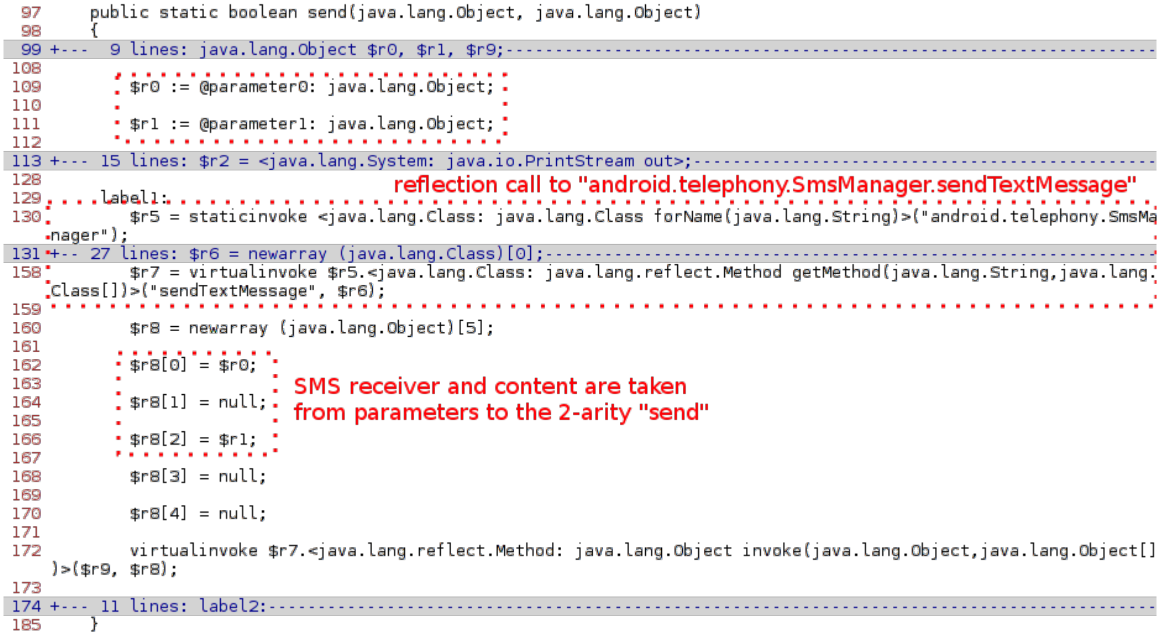
compare result against 1 and branch

"safe" methods

Figure 6.10.: The method `Jk7H.PwcD.SLYfoMdG$1.run`, implicitly invoked by the method `Jk7H.PwcD.SLYfoMdG.send` (0-arity version) through Java thread. File: `Jk7H.PwcD.SLYfoMdG$1.jimple`.

6.3.3.8 Summary

We have just traced the following 7-hop invocation path from the entry callback method `onCreate` to the malicious invocation of `sendTextMessage` above (omit the class names): `onCreate` ⇒ `showScreen` ⇒ `setMain` ⇒ `mainButtonClick1` ⇒ (0-arity) `send` ⇒ `run` ⇒ (2-arity) `send` ⇒ `sendTextMessage`. A remarkable fact about this invocation path is that none of the hops is triggered by the usual direct in-



```
    public static boolean send(java.lang.Object, java.lang.Object)
    {
+---  9 lines: java.lang.Object $r0, $r1, $r9;------------------------------------------------------------

        $r0 := @parameter0: java.lang.Object;

        $r1 := @parameter1: java.lang.Object;

+--- 15 lines: $r2 = <java.lang.System: java.io.PrintStream out>;------------------------------------------

     label1:
        $r5 = staticinvoke <java.lang.Class: java.lang.Class forName(java.lang.String)>("android.telephony.SmsMa
nager");
+-- 27 lines: $r6 = newarray (java.lang.Class)[0];-------------------------------------------------------
        $r7 = virtualinvoke $r5.<java.lang.Class: java.lang.reflect.Method getMethod(java.lang.String,java.lang.
Class[])>("sendTextMessage", $r6);

        $r8 = newarray (java.lang.Object)[5];

        $r8[0] = $r0;

        $r8[1] = null;

        $r8[2] = $r1;

        $r8[3] = null;

        $r8[4] = null;

        virtualinvoke $r7.<java.lang.reflect.Method: java.lang.Object invoke(java.lang.Object,java.lang.Object[]
)>($r9, $r8);

+--- 11 lines: label2:-----------------------------------------------------------------------------------
    }
```

reflection call to "android.telephony.SmsManager.sendTextMessage"

SMS receiver and content are taken from parameters to the 2-arity "send"

Figure 6.11.: The method `Jk7H.PwcD.SLYfoMdG.send` (2-arity version), implicitly invoked by the method `Jk7H.PwcD.SLYfoMdG$1.run` through Java reflection. SMS text messages are sent through reflection call to `sendTextMessage` Android API. File: `Jk7H.PwcD.SLYfoMdG.jimple`.

vocation mechanism (i.e., the *-invoke virtual machine instructions), but by implicit control flow mechanisms such as Java reflection, thread, or Android event callbacks.

For a 40KBytes app, the invocation path to `sendTextMessage` taken by `Jk7H.PwcD` is untypically complicated and surreptitious. For example, those Java reflection invokes can be replaced by direct invoke virtual machine instructions (`*-invoke`) without changing user observable behaviors of the app for both simplicity and performance (Java reflection incurs non-trivial overhead in comparison to direct invokes [147]).

However, `Jk7H.PwcD` demonstrates, within its compact size, what a dedicated adversary might do to evade detection. This example motivates the explicit consideration of adversary scenario for Android APK analysis.

6.3.4 The Need for More Robust Call Graph Extraction Algorithm

6.3.4.1 Call Graph Extraction in Androguard

Andorguard [62, 63] is a publicly released and maintained suite of APK analysis tools implemented in the Python programming language. Androguard is used in multiple previous works [85, 164, 105, 163]. We apply the `androgexf` tool from Androguard's latest 1.9 release[7] and obtain the call graph shown in Figure 6.12.

More precisely, Figure 6.12 shows the connected subgraph of `Jk7H.PwcD`'s call graph.Note that `androgexf` only shows internal method invocations, i.e., invocations to methods that are contained in the APK; external invocations to Android API are not included in the extracted call graph. Figure 6.12 shows that only two internal method invocations are detected: `finish` and `getSharedPreferences`. The other nodes shown in Figure 6.12 are internal methods that directly invoke either of these methods, but are not otherwise invoked from `onCreate`. In other words, `androgexf` only shows that these methods are co-invocators with `onCreate` to either `finish` and `getSharedPreferences`, but do not have an invocator-invocatee relationship between them.

Given the lengthy analysis in Section 6.3.3, the explanation for the result shown in Figure 6.12 is that `androgexf` is not designed to handle implicit control flows such as Java reflection and thread. Moreover, the results indicate that `androgexf`'s call graph extraction logic is (Algorithm 2):

[7] To be precise, we use the `python2-androguard-hg` package maintained in Arch Linux's AUR repository that is last updated on 5 June 2014, available at `http://goo.gl/GcmoXC`.

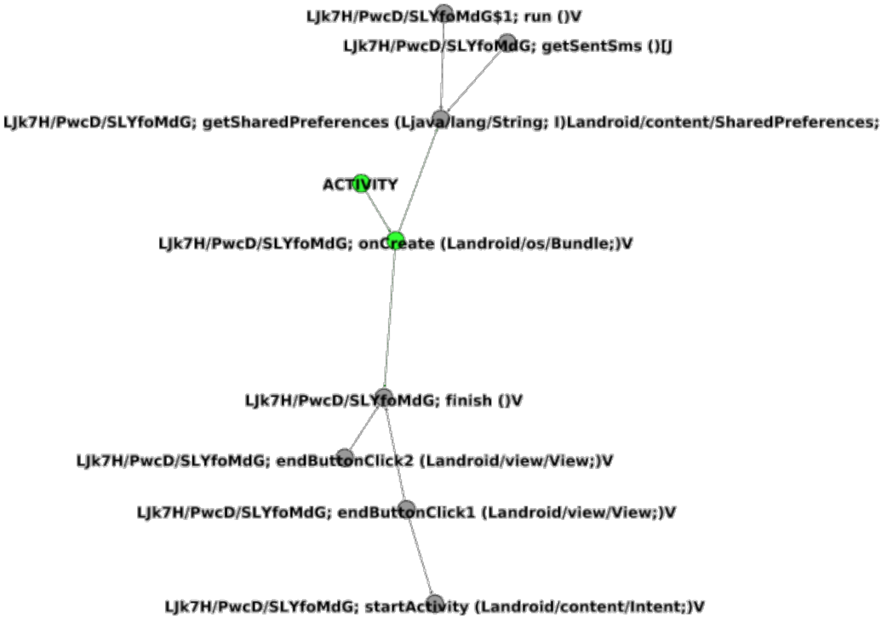

LJk7H/PwcD/SLYfoMdG$1; run ()V
LJk7H/PwcD/SLYfoMdG; getSentSms ()[J
LJk7H/PwcD/SLYfoMdG; getSharedPreferences (Ljava/lang/String; I)Landroid/content/SharedPreferences;
ACTIVITY
LJk7H/PwcD/SLYfoMdG; onCreate (Landroid/os/Bundle;)V
LJk7H/PwcD/SLYfoMdG; finish ()V
LJk7H/PwcD/SLYfoMdG; endButtonClick2 (Landroid/view/View;)V
LJk7H/PwcD/SLYfoMdG; endButtonClick1 (Landroid/view/View;)V
LJk7H/PwcD/SLYfoMdG; startActivity (Landroid/content/Intent;)V

Figure 6.12.: The subgraph of the call graph generated by Androguard that is rooted at the entry callback method `onCreate`. Note that only two internal methods, `finish` and `getSharedPreferences`, that are directly invoked as shown in Figure 6.5 are detected. The other methods shown here also invoke (one of) these two methods, but are not invoked from `onCreate`.

> Scan the current method body, identify `*-invoke` virtual machine instructions, resolve the target method, recursively follow the target method if it is an internal method that is implemented in the current APK.

Although this is a simple and elegant (in the recursive formulation) call graph extraction algorithm, given the existence of app such as the one analyzed in Section 6.3.3, there is a clear need for more robust call graph extraction algorithms. This is a task undertaken in the present work.

It must be pointed out that Androguard provides much more functionality than call graph extraction; in particular, it provides a Python programming interface to access and manipulate internal structures of an APK, which is used in other works for various extensions. Moreover, Androguard is publicly available, can be easily set up, and is well maintained—all the traits make Androguard immensely valuable for

Algorithm 2 Androguard's call graph extraction algorithm.

```
Input: $B_m$: the sequence of VM instructions that comprises method $m$'s body
Output: invocation paths from method $m$
function EXTRACT-INTERNAL-INVOKES($B_m$)
    $I \leftarrow \emptyset$    ▶ $I$ holds invocation paths from $m$
    for $i \in \mathcal{B}_m$ do    ▶ loop over the body of $m$
        if $i$ is of type *-invoke then
            $m' \leftarrow$ target method of $i$
            if $m'$ is an internal method then
                $B_{m'} \leftarrow$ method body of $m'$
                $I \leftarrow I \cup \{m, \text{EXTRACT-INTERNAL-INVOKES}(B_{m'})\}$
            else    ▶ $m'$ is an external method
                $I \leftarrow I \cup \{[m, m']\}$
            end if
        end if
    end for
    return $I$
end function
```

research, as evidenced by numerous works that is based on Androguard. The purpose of this section is not to belittle Androguard[8], but to point out the limitations of what it provides, and to motivate the work presented in next sections.

6.3.4.2 What this Work Provides

In contrast, as shown in Figure 6.13, the following query on the prototype WoA:

```
MATCH
(a:Apk {sha256:"a00f2b489dac150e513526ab285141d41a127133cd3be2115046e22e189ff2a3"})
  <-[:INVOKED_BY]- (m:Method)
WHERE m.name=~".*android\\.telephony.*"
WITH a, m
MATCH (a)-->(d:Dex)-->(c:Class)-->(cb:Callback)
RETURN *
```

[8] Androguard sets a good example of publishing *and maintaining* their *implementation* that is often (unfortunately) not followed. Androguard is a source of inspiration for making the present work reproducible for the benefits of the whole research community.

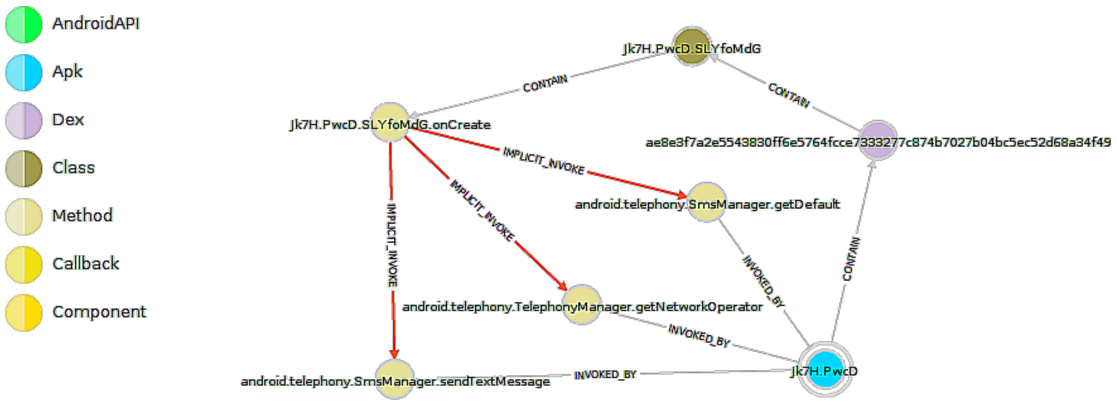

AndroidAPI
Apk
Dex
Class
Method
Callback
Component
Jk7H.PwcD.SLYfoMdG
CONTAIN
CONTAIN
Jk7H.PwcD.SLYfoMdG.onCreate
ae8e3f7a2e5543830ff6e5764fcce7333277c874b7027b04bc5ec52d68a34f49
IMPLICIT_INVOKE
android.telephony.SmsManager.getDefault
IMPLICIT_INVOKE
IMPLICIT_INVOKE
INVOKED_BY
CONTAIN
android.telephony.TelephonyManager.getNetworkOperator
INVOKED_BY
android.telephony.SmsManager.sendTextMessage
INVOKED_BY
Jk7H.PwcD

Figure 6.13.: With the first query shown in Section 6.3.4.2, WoA reveals that 3 methods, `getDefault`, `getNetworkOperator`, and `sendTextMessage`, from Java packages `android.telephony.*` are *implicitly* invoked from the entry method `onCreate` of class `Jk7H.PwcD.SLYfoMdG`.

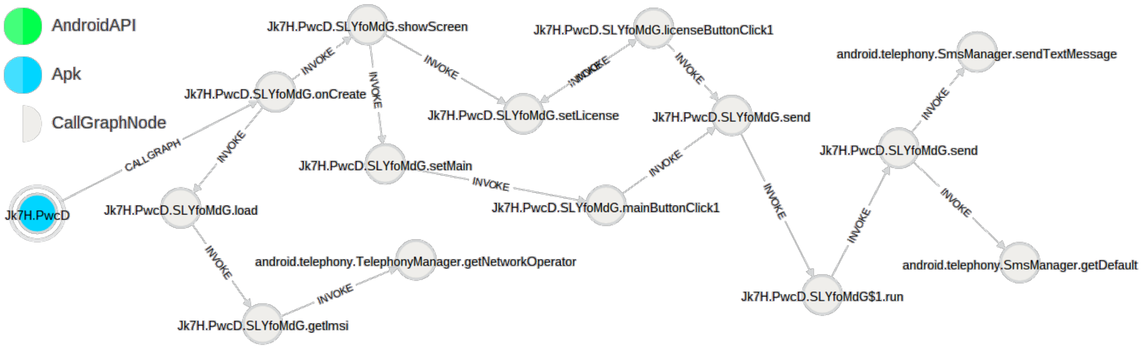

AndroidAPI
Apk
CallGraphNode
Jk7H.PwcD.SLYfoMdG.showScreen
Jk7H.PwcD.SLYfoMdG.licenseButtonClick1
android.telephony.SmsManager.sendTextMessage
INVOKE
INVOKE
INVOKE
INVOKE
Jk7H.PwcD.SLYfoMdG.onCreate
INVOKE
Jk7H.PwcD.SLYfoMdG.setLicense
Jk7H.PwcD.SLYfoMdG.send
INVOKE
CALLGRAPH
INVOKE
Jk7H.PwcD.SLYfoMdG.send
Jk7H.PwcD.SLYfoMdG.setMain
INVOKE
INVOKE
Jk7H.PwcD
Jk7H.PwcD.SLYfoMdG.load
INVOKE
Jk7H.PwcD.SLYfoMdG.mainButtonClick1
INVOKE
INVOKE
INVOKE
android.telephony.TelephonyManager.getNetworkOperator
android.telephony.SmsManager.getDefault
INVOKE
Jk7H.PwcD.SLYfoMdG$1.run
Jk7H.PwcD.SLYfoMdG.getImsi

Figure 6.14.: With the second query shown in Section 6.3.4.2, WoA reveals a different invocation path to `sendTextMessage` from the one analyzed in Section 6.3.3: `onCreate` ⇒ `showScreen` ⇒ `setLicense` ⇒ `licenseButtonClick1` ⇒ (0-arity) `send` ⇒ `run` ⇒ (2-arity) `send` ⇒ `sendTextMessage`.

shows that 3 methods, `getDefault`, `getNetworkOperator`, and `sendTextMessage`, from Java packages `android.telephony.*` are *implicitly* invoked from the method `onCreate` of class `Jk7H.PwcD.SLYfoMdG`. In our current implementation using the Neo4j graph database [144], the query that generates Figure 6.12 takes 0.1 seconds to complete in a freshly booted instance (so that cache is not used to bias the result). As shown in Figure 6.14, the following query on the prototype WoA:

```
MATCH
(a:Apk {sha256:"a00f2b489dac150e513526ab285141d41a127133cd3be2115046e22e189ff2a3"})
  -[:CALLGRAPH]-> (cg:CallGraphNode)
MATCH p=(cg)-[:INVOKE*..8]->(m)
WHERE m.name=~".*android\\.telephony.*"
RETURN *
```

reveals a different invocation path to `sendTextMessage` from the one analyzed in Section 6.3.3: `onCreate` ⇒ `showScreen` ⇒ `setLicense` ⇒ `licenseButtonClick1` ⇒ (0-arity) `send` ⇒ `run` ⇒ (2-arity) `send` ⇒ `sendTextMessage`. The two paths diverge at `showScreen` and re-converges at 0-arity `send`. The query that generates Figure 6.14 takes 1.647 seconds to complete in a freshly booted instance.

6.4 Web of Fibers

6.4.1 Introduction

6.4.1.1 Relation to Existing Literature

Figure 6.15 shows a (non-comprehensive) categorization of the techniques proposed in existing literature surveyed in Section 6.2. Within this framework, the task of measuring similarity of APKs for plagiarism/malware detection consists of two sub-tasks: 1. encoding individual APK samples and 2. identifying plagiarism/malware by computing similarity on the encoded APK samples. For example, Desnos encodes each Java method of an APK into a string, and uses the Normalized Compression Distance (NCD) of the strings to measure similarity of the corresponding methods [62].

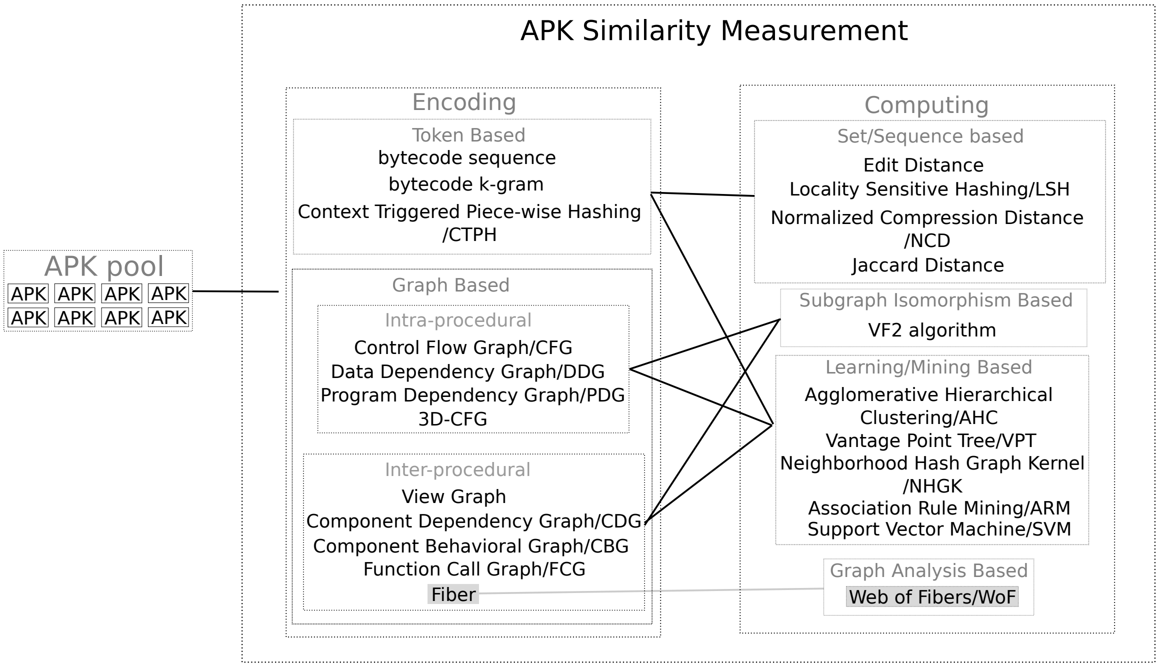

APK Similarity Measurement
APK pool
APK APK APK APK
APK APK APK APK
Encoding
Token Based
bytecode sequence
bytecode k-gram
Context Triggered Piece-wise Hashing /CTPH
Graph Based
Intra-procedural
Control Flow Graph/CFG
Data Dependency Graph/DDG
Program Dependency Graph/PDG
3D-CFG
Inter-procedural
View Graph
Component Dependency Graph/CDG
Component Behavioral Graph/CBG
Function Call Graph/FCG
Fiber
Computing
Set/Sequence based
Edit Distance
Locality Sensitive Hashing/LSH
Normalized Compression Distance /NCD
Jaccard Distance
Subgraph Isomorphism Based
VF2 algorithm
Learning/Mining Based
Agglomerative Hierarchical Clustering/AHC
Vantage Point Tree/VPT
Neighborhood Hash Graph Kernel /NHGK
Association Rule Mining/ARM
Support Vector Machine/SVM
Graph Analysis Based
Web of Fibers/WoF

Figure 6.15.: A (non-comprehensive) categorization of techniques, as proposed in related work reviewed in Section 6.2, for measuring similarity between samples from a pool of APKs. Wihtin this framework, Web of APKs (WoA) encodes individual APK samples into the inter-procedural graph-based representation Fibers, which are then combined into a single Web of Fibers (WoF) by syntactic matching of attributes. APK analysis problems, including similarity measurement, are formulated as graph analysis queries on WoF.

The spectrum of APK encodings can be clustered into the following categories based on their granularity and scope.

Token-based encodings encode short Dalvik bytecode sequences or all bytecodes contained in individual Java methods of an APK into non-graph-structured tokens. They are fine in granularity (i.e., at individual byte code level) and local in scope (i.e., bytecode segments within individual methods). Examples of such encodings include stringified bytecode sequences [62], bytecode k-grams [97], and context triggered piece-wise hashing (CTPH) [233].

Intra-procedural-graph-based encodings encode control and data flow information within individual Java methods of an APK into graphs. Comparing with token-based encodings, they are coarser in granularity (i.e., bytecodes are abstracted away) but less local in scope (i.e., individual methods). Examples of such encodings include data dependency graphs (DDG) [56], program dependence graphs (PDGs) [55], variants of control flow graphs (CFGs), e.g., 3D-CFG [44]

Inter-procedural graph-based encodings encode global structural information of an APK between individual Java methods or, at even coarser granularity, Android components. Examples of such encodings include view graphs [225], component dependency graphs (CDG) and component behavioral graphs (CBG) [219], function call graph (FCG) [85], and, as will be discussed shortly, Fibers.

Techniques of computing similarity of APKs using the above encodings include the following categories.

Set/sequence-based techniques use distance metrics for set or sequence for similarity measurement. Examples of using these techniques include edit distance of CTPH [233], locality sensitive hashing (LSH) of semantic blocks [56], normalized compression distance of stringified bytecode sequences [62], and Jaccard distance of feature vectors [234].

Subgraph-isomorphism-based techniques measure APK similarity by identifying isomorphic subgraphs of graph-based APK encodings and using, for example, Jaccard distance to measure how significant the isomorphic subgraphs are. Examples of using these techniques include applying isomorphic subgraph identification algorithms to PDGs [55] and view graphs [225]. Interestingly, both works use the VF2 algorithm [52], perhaps due to the public availability of its implementation [114]. This makes a case in releasing supporting implementation for result reproducibility and the benefit of future research—a practice that is followed in this work.

Machine-learning/pattern-mining-based techniques encompass a significant portion of the research body that applies advances in machine learning and pattern mining research for identifying similar APKs. Since machine learning tasks, such as classification and clustering, are often frame in a continuous metric space (e.g., Euclidean spaces) rather than a discrete combinatorial space (e.g., sequences and permutations), combinatorial structures extracted from APKs (e.g., bytecode sequences, *k*-grams, feature hashes) are often first transformed into continue features before proceeding. Example of using these techniques include applying agglomerative hierarchical clustering (AHC) [61] to *k*-gram-based feature bit-vectors [97] and to group APK class packages into different modules [234], using vantage point tree (VPT) [222] to reduce metric space search complexity [234], using neighborhood hash graph kernel (NHGK) [100] to encode 1-hop neighborhoods on method call graphs, formulating malware detection as an association rule mining [101] problem from extracted features to observed malicious behaviors [219], using support vector machines (SVM) [192] to malicious app detection [85, 16], and using probabilistic generative models [203] on Android permissions requested by APKs for ranking their perceived risks [149].

Parallel to the aforementioned approaches, Web of APKs, as presented in this work, adopts a declarative graph analysis approach to the problem of identifying and explaining semantic similarity of APKs. As briefly discussed in Section 6.1.3, this approach does not work by pairing individual APK encodings for comparison or training classifiers on a carefully selected representative data sets, but, instead, by:

- Combining Fibers of individual APK samples into a single Web of Fibers (WoF) through both syntactic matching of essential attributes and structural similarity.
- Leveraging the efficiency of index-free adjacency [175] access and query capabilities of modern graph database systems to support declarative graph analysis of the relationship between APK samples.

6.4.1.2 About the Queries

As briefly mentioned in Section 6.1.2, the Neo4j graph database [144] and its companion Cypher graph query language [145] are leveraged in the current implementation of WoA to manage, query, and visualize WoF. Therefore, concrete queries that scatter across this chapter for result reproducibility are presented in Cypher [145].

Fortunately, Cypher is designed to facilitate easy comprehension by using pattern matching and self-explanatory English prose and iconography. The following examples, accompanied by references to the official documentation [145], suffice for understanding query instances presented in this section.

- An example of graph node pattern is `(a:Apk {sha256:"1234"})`, in which a node of label `Apk` (a node can have more than one labels) and property `sha256` (of value `1234`) is bound to the name `a` that can be referred to later in the pattern. Note: node patterns are enclosed by pairs of parentheses `()`, and properties are enclosed by pairs of braces `{}`.

- An example of graph edge ("relationship" in Cypher's terminology) pattern is `(a)-[e:INVOKE]->(b)`, in which a directed edge from node `a` to node `b` of type `INVOKE` (an edge can have at most one type) is bound to the name `e`. Note: edge patterns are enclosed by pairs of brackets `[]`, with edge directionality represented by `-[]->` (left to right), `<-[]-` (right to left), or `-[]-` (either direction).

- An example of graph path (an ordered interleaving sequence of nodes and edges) pattern is `(cgr:CallGraphNode)-[:INVOKE*..5]->(cgn:CallGraphNode)`, which matches an interleaving sequence of label-`CallGraphNode` nodes and type-`-INVOKE` edges with up to (and including) 5 edges on the path.

6.4.2 Fibers: The Property Graph Model of Individual APKs

6.4.2.1 Overview

It has been mentioned in Section 6.1.3 and illustrated in Figure 6.3 that:

> A Fiber is a hierarchical property graph model of an individual APK sample that consists of the following layers of *structural nodes*: signing key,

Table 6.1.: List of edge types in Fibers, which include edges that connect both structural nodes (e.g., SigningKey, Apk, and Dex) and informational nodes (e.g., Tag, Permission, and IntentFilterAction). Rows marked with * are only available in the Full Mode. Refer to Table 6.2 for the list of node types/attributes in Fibers.

From Node Type	Relationship Type ⇒	To Node Type
SigningKey	SIGN	Apk
Tag	TAG	Apk
Apk	USE	Permission
Apk	DEFINE	Permission
Apk	CONTAIN	Dex
Apk	CALLGRAPH	CallGraphNode
Dex	CONTAIN	Class
Dex	CONTAIN	Component/Class
Package	CONTAIN	Class
Class	DESCEND	AndroidAPI/Class
Class	CONTAIN	Method
Class	CONTAIN	Method/Callback
Method/Callback	EXPLICIT_INVOKE	Method
Method/Callback	IMPLICIT_INVOKE	Method
Method/Callback	CALLGRAPH	CallGraphNode
Apk	CALLBACK_SIGNATURE	CallbackSignature
Callback	CALLBACK_SIGNATURE	CallbackSignature
CallGraphNode	INVOKE	CallGraphNode
Method	INVOKED_BY	Apk
* Method/Callback	EXPLICIT_INVOKE	MethodInstance
* Method/Callback	IMPLICIT_INVOKE	MethodInstance
* MethodInstance	INSTANCE_OF	Method
IntentFilterAction	TRIGGER	Component/Class
IntentFilterCategory	TRIGGER	Component/Class

APK, DEX (Dalvik EXecutable; the executable format of APK), component classes (components), component callback methods (callbacks), transitive invocatees of the callbacks, call graph nodes, and method invocation instances (optional). Besides structural nodes, there are also several types of *informational nodes*: arbitrary tags (e.g., data source, anti-virus software scanning label), requested/defined permissions, and Android Intent Filter actions/categories.

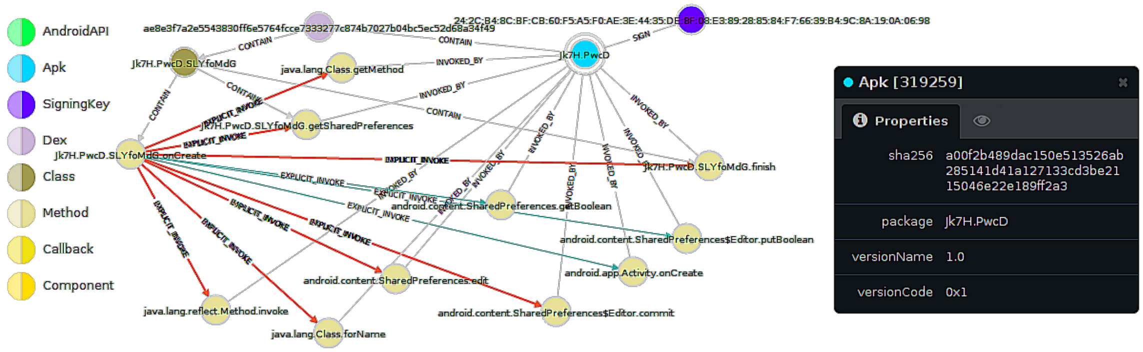
Property graph, as used in the context of graph database [175] and also here, is a graph data model in which properties (i.e., key-value pairs) can be attached to both graph nodes and edges, and patterns or conditionals can be specified using such properties. Fibers are hierarchical in that the nodes in a Fiber are of different types (i.e., have different labels) that belong to different logical layers, and edges exist only between specific types of nodes in different layers.

Table 6.2.: List of node types/attributes in Fibers, which include both structural nodes (e.g., SigningKey, Apk, and Dex) and informational nodes (e.g., Tag, Permission, and IntentFilterAction). Rows marked with * are only available in Full Mode. Refer to Table 6.1 for the list of edge types in Fibers.

Node Type	Attributes
SigningKey	sha256
Apk	sha256, package, versionCode, versionName
Dex	sha256
Permission	name
Package	name
Class	name
Method	name
* MethodInstance	name, args
Callback	name
CallbackSignature	name, apk, signature
CallGraphNode	name, apk, signature
IntentFilterAction	name
IntentFilterCategory	name

More concretely, Tables 6.1 and 6.2 enumerate the edge types and node types/attributes, respectively. For example, the first row in Table 6.1 reads "edge from node of type `SigningKey` to node of type `Apk` is of type `SIGN`;" the first row in Table 6.2 reads "node of type `SigningKey` has a sole attribute named `sha256`." Next, in Section 6.4.2.2, concrete examples of these node/edge types/attributes are presented using the real Android malware example analyzed in Section 6.3. Thoughts behind the model design are presented in Section 6.4.3.

One note on Tables 6.1 and 6.2 is that there are two modes of operation in WoF: the Compact Mode and the Full Mode, with the Compact Mode being the default mode of operation. The difference between the two is that the Full Mode has an additional type of node `MethodInstance`, which records not only invoked method's name (as in the type-`Method` node) but also the actual arguments used in invocation instances. The thoughts behind the design of having both modes and choosing the Compact Mode as the default is presented in Section 6.4.3.


AndroidAPI
Apk
SigningKey
Dex
Class
Method
Callback
Component
ae8e3f7a2e5543830ff6e5764fcce7333277c874b7027b04bc5ec52d68a34f49
24:2C:B4:8C:BF:CB:60:F5:A5:F0:AE:3E:44:35:DE:8F:08:E3:89:28:85:84:F7:66:39:B4:9C:8A:19:0A:06:98
SIGN
CONTAIN
INVOKED_BY
EXPLICIT_INVOKE
Jk7H.PwcD
Jk7H.PwcD.SLYfoMdG
java.lang.Class.getMethod
Jk7H.PwcD.SLYfoMdG.getSharedPreferences
Jk7H.PwcD.SLYfoMdG.onCreate
Jk7H.PwcD.SLYfoMdG.finish
android.content.SharedPreferences.getBoolean
android.content.SharedPreferences$Editor.putBoolean
android.app.Activity.onCreate
android.content.SharedPreferences.edit
android.content.SharedPreferences$Editor.commit
java.lang.reflect.Method.invoke
java.lang.Class.forName
Properties

Apk [319259]

sha256 a00f2b489dac150e513526ab285141d41a127133cd3be2115046e22e189ff2a3

package Jk7H.PwcD

versionName 1.0

versionCode 0x1

Figure 6.16.: The explicit-invocation part of Fiber skeleton of `Jk7H.PwcD` (Section 6.3).

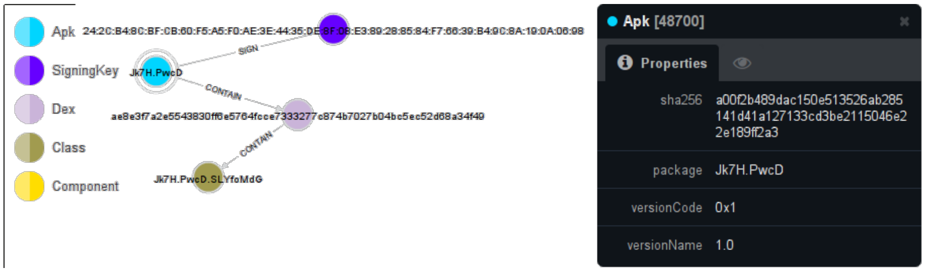
6.4.2.2 Anatomy of the Model

This section complements the previous one by taking apart the Fiber constructed from the real Android malware sample analyzed in Section 6.3.

Figure 6.16 visualizes the result of the following query:

```
MATCH (apk:Apk{sha256:"a00f2b489dac150e513526ab285141d41a127133cd3be2115046e22e189ff2a3"})
OPTIONAL MATCH (signingKey:SigningKey) -[:SIGN]-> (apk) -[:CONTAIN]-> (dex:Dex)
  -[:CONTAIN]-> (class:Class) -[:CONTAIN]-> (callback:Callback)
OPTIONAL MATCH (explicitInvoke) <-[:EXPLICIT_INVOKE]- (callback)
RETURN *
```

which extracts the explicitly-invoked part of the example app's Fiber skeleton in the Compact Mode: SigningKey, Apk, Dex, Component, Callback, and (explicitly invoked) Method. Both internal method invocations in the call graph produced by Androguard (Figure 6.12), `finish` and `getSharedPreferences` of class `Jk7H.PwcD.SLYfoMdG`, are included in this Fiber. In addition, external method invocations, such as to Android API method `edit` (of class `android.content.SharedPreferences`) or Java API method `getMethod` (of class `java.lang.Class`), are also included in the `Method` layer of the Fiber. These external API invocations reveal the externally observed behavior of the app that is not captured in an internal-method-only call graph.


Apk
SigningKey
Dex
Class
Component
24:20:B4:80:BF:0B:60:F5:A5:F0:AE:3E:44:35:DE:8F:08:E3:89:28:85:84:F7:66:39:B4:90:8A:19:0A:06:98
SIGN
Jk7H.PwcD
CONTAIN
ae8e3f7a2e5543830ff6e5764fcce7333277c874b7027b04bc5ec52d68a34f49
CONTAIN
Jk7H.PwcD.SLYfoMdG
Apk [48700]
Properties

sha256	a00f2b489dac150e513526ab285141d41a127133cd3be2115046e22e189ff2a3
package	Jk7H.PwcD
versionCode	0x1
versionName	1.0

Figure 6.17.: Upper layers of `Jk7H.PwcD`'s Fiber: SigningKey, Apk, Dex, and Component.

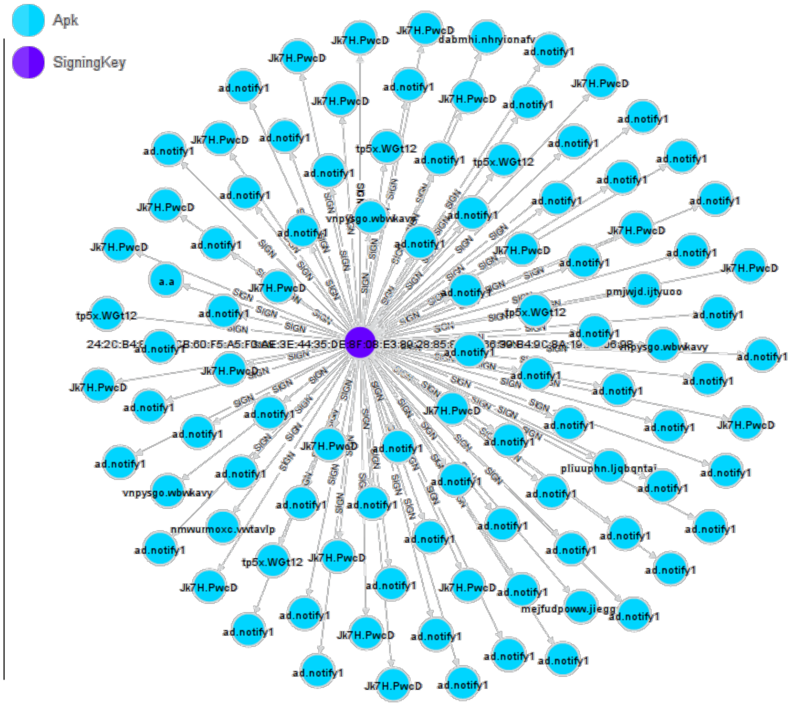


Figure 6.18.: 100 of the APK samples, signed by the author of `Jk7H.PwcD` (Section 6.3), that have more than 40 anti-virus vendors flagging them as malware.

SigningKey. Figure 6.18 visualizes the result generated by the following query:

```
MATCH (apk:Apk{sha256:"a00f2b489dac150e513526ab285141d41a127133cd3be2115046e22e189ff2a3"})
OPTIONAL MATCH (signingKey:SigningKey) -[:SIGN]-> (apk)
WITH signingKey MATCH (signingKey) -[:SIGN]-> (n:Apk) <-- (m:Malware)
WITH signingKey, n, count(m) as cm WHERE cm>40
RETURN signingKey, n LIMIT 100
```

which shows 100 of the APK samples, signed by the author of `Jk7H.PwcD`, that have more than 40 anti-virus vendors flagging them as malware. The result indicates that the signing key (begins with `24:2C:B4`) belong to a prolific malware author.

Apk. Figure 6.19 visualizes the result generated by the following query:

```
MATCH (apk:Apk{sha256:"a00f2b489dac150e513526ab285141d41a127133cd3be2115046e22e189ff2a3"})
MATCH (p:Permission) -- (apk) -- (n:Tag)
RETURN *
```

which shows anti-virus vendor labels (obtained from the VirusTotal [207] service) and permission used/defined by `Jk7H.PwcD`. Note that `Jk7H.PwcD` requests 4 Android permissions:

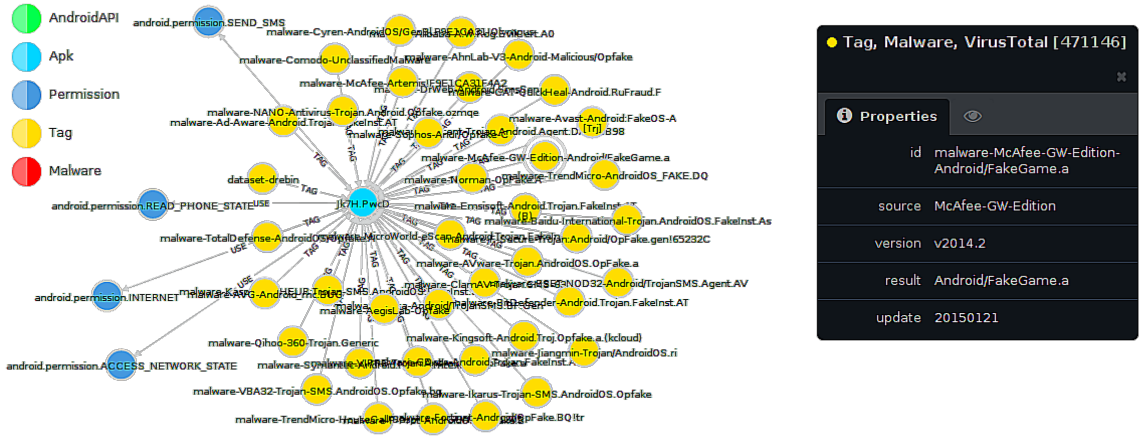
- `android.permission.SEND_SMS`,
- `android.permission.READ_PHONE_STATE`,
- `android.permission.INTERNET`,
- and `android.permission.ACCESS_NETWORK_STATE`,

of which the use of `android.permission.SEND_SMS` for the malicious function is analyzed in detail in Section 6.3.3.

Dex and Component. Figure 6.20 visualizes the result generated by the following query:

```
MATCH (apk:Apk{sha256:"a00f2b489dac150e513526ab285141d41a127133cd3be2115046e22e189ff2a3"})
MATCH (apk) -- (dex:Dex)
WITH dex
MATCH (dex) -- (n)
RETURN *
```

which shows all nodes in the prototype WoF that connect to the Dex node of `Jk7H.PwcD` (SHA-256 checksum begins with `a00f2b`), which include a single Component node `Jk7H.PwcD.SLYfoMdG` of type Activity. It can be observed that this particular Dex is multiply repackaged in a number of APKs, all of which share the same package name


AndroidAPI
Apk
Permission
Tag
Malware
android.permission.SEND_SMS
android.permission.READ_PHONE_STATE
android.permission.INTERNET
android.permission.ACCESS_NETWORK_STATE
Jk7H.PwcD
dataset-drebin
malware-Comodo-UnclassifiedMalware
malware-AhnLab-V3-Android-Malicious/Opfake
malware-McAfee-Artemis!F9E1CA31F4A2
malware-NANO-Antivirus-Trojan.Android.Opfake.ozmqe
malware-Ad-Aware-Android.Trojan.FakeInst.AT
malware-Avast-Android:FakeOS-A
malware-McAfee-GW-Edition-Android/FakeGame.a
malware-TrendMicro-AndroidOS_FAKE.DQ
malware-Norman-OpFake.A
malware-Emsisoft-Android.Trojan.FakeInst.AT (B)
malware-Baidu-International-Trojan.AndroidOS.FakeInst.As
malware-TotalDefense-AndroidOS/Opfake.A
malware-AVware-Trojan.AndroidOS.OpFake.a
malware-ESET-NOD32-Android/TrojanSMS.Agent.AV
malware-BitDefender-Android.Trojan.FakeInst.AT
malware-AegisLab-Opfake
malware-Qihoo-360-Trojan.Generic
malware-Kingsoft-Android.Troj.Opfake.a.(kcloud)
malware-Jiangmin-Trojan/AndroidOS.ri
malware-VBA32-Trojan-SMS.AndroidOS.Opfake.bq
malware-Ikarus-Trojan-SMS.AndroidOS.Opfake
malware-TrendMicro-HouseCall
TAG
USE

Tag, Malware, VirusTotal [471146]

Properties

id	malware-McAfee-GW-Edition-Android/FakeGame.a
source	McAfee-GW-Edition
version	v2014.2
result	Android/FakeGame.a
update	20150121

Figure 6.19.: Anti-virus vendor labels (obtained from the VirusTotal [207] service) and permission used/defined by `Jk7H.PwcD`

`Jk7H.PwcD`. The differences in these APKs can be accounted for by, for example, resigning using a different key, changing of meta information (e.g., requested permission), adding/removing contained resources, or even simply rebuilding or realigning the APK binary [231].

Figure 6.21 visualizes the result generated by the following query:

```
MATCH (apk:Apk{sha256:"a00f2b489dac150e513526ab285141d41a127133cd3be2115046e22e189ff2a3"})
MATCH (apk) --> (:Dex) --> (component:Component)
WITH component
MATCH (component) --> (callback:Callback) -[:EXPLICIT_INVOKE]-> (method:Method)
RETURN *
```

which shows the lower layers of Fiber nodes in `Jk7H.PwcD`: component callback methods and (explicitly invoked) transitively invoked methods.

The type-`Method` nodes invoked by the type-`Callback` node `onCreate` of class `Jk7H.PwcD.SLYfoMdG` are *transitive invocatees* that include both directly and indirectly invoked methods: If methods m_1 invoke m_2 and m_2 invokes m_3, the transitive invocatees of m_1 include both m_2 and m_3. The reason that the explicitly invoked transitive invocatees shown in Figure 6.21 (i.e., the `finish` and `getSharedPreferences` nodes) coincide with that of the call graph extracted by Androguard (Figure 6.12)

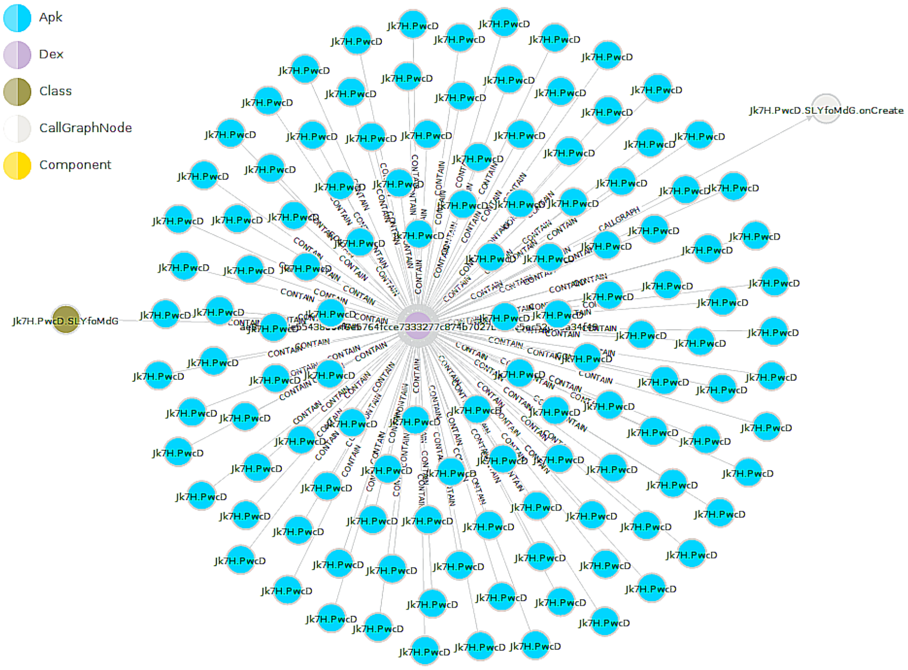


Figure 6.20.: All nodes in the prototype WoF that connect to the Dex node of `Jk7H.PwcD` (Section 6.3), which include a single Component node `Jk7H.PwcD.SLYfoMdG` of type Activity.

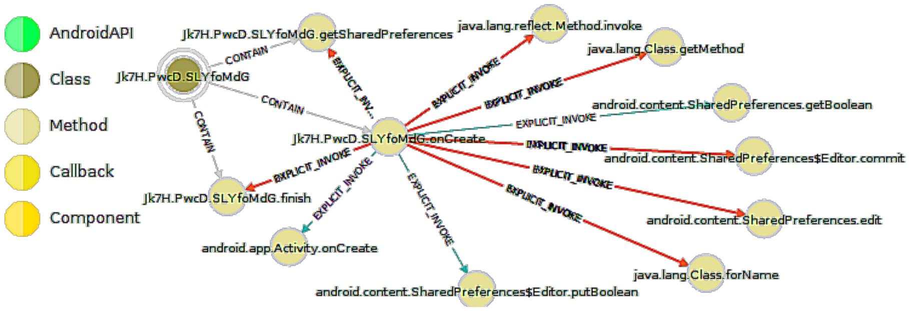

AndroidAPI
Class
Method
Callback
Component
Jk7H.PwcD.SLYfoMdG.getSharedPreferences
java.lang.reflect.Method.invoke
java.lang.Class.getMethod
Jk7H.PwcD.SLYfoMdG
CONTAIN
EXPLICIT_INVOKE
android.content.SharedPreferences.getBoolean
Jk7H.PwcD.SLYfoMdG.onCreate
android.content.SharedPreferences$Editor.commit
Jk7H.PwcD.SLYfoMdG.finish
android.content.SharedPreferences.edit
android.app.Activity.onCreate
java.lang.Class.forName
android.content.SharedPreferences$Editor.putBoolean

Figure 6.21.: The lower layers of Fiber nodes in `Jk7H.PwcD`: component callback methods and (explicitly invoked) transitively invoked methods.

is that, as analyzed in Section 6.3.3, `Jk7H.PwcD` extensively employs implicit control flow invocations, which is the subject of the next section.

Figure 6.22 visualizes the result generated by the following query:

```
MATCH (apk:Apk{sha256:"a00f2b489dac150e513526ab285141d41a127133cd3be2115046e22e189ff2a3"})
OPTIONAL MATCH (signingKey:SigningKey) -[:SIGN]-> (apk) -[:CONTAIN]-> (dex:Dex)
  -[:CONTAIN]-> (class:Class) -[:CONTAIN]-> (callback:Callback)
```

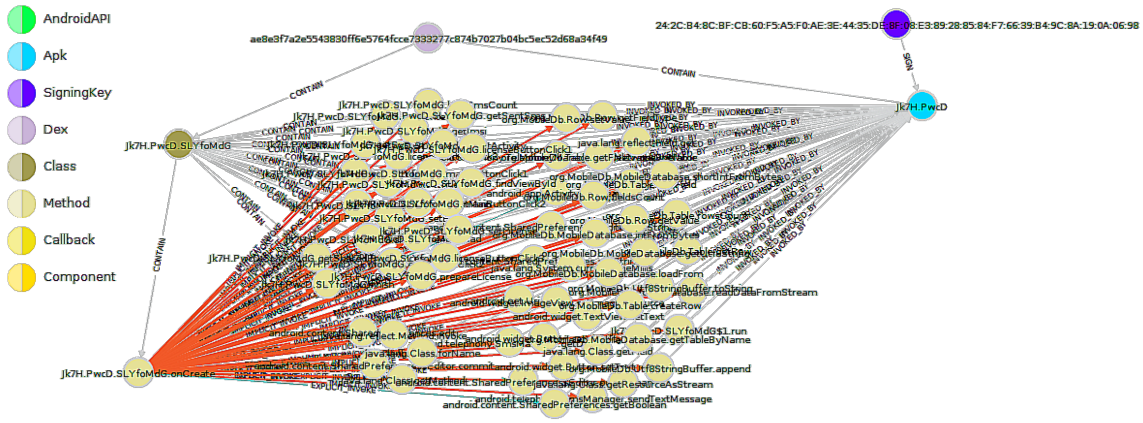

AndroidAPI
Apk
SigningKey
Dex
Class
Method
Callback
Component
ae8e3f7a2e5543830ff6e5764fcce7333277c874b7027b04bc5ec52d68a34f49
24:2C:B4:8C:BF:CB:60:F5:A5:F0:AE:3E:44:35:DE:8F:08:E3:89:28:85:84:F7:66:39:B4:9C:8A:19:0A:06:98
SIGN
CONTAIN
Jk7H.PwcD
Jk7H.PwcD.SLYfoMdG
Jk7H.PwcD.SLYfoMdG.onCreate
INVOKED_BY
EXPLICIT_INVOKE

Figure 6.22.: The Fiber skeleton of `Jk7H.PwcD` *with* implicitly invoked transitive invocatees included.

```
OPTIONAL MATCH (explicitInvoke) <-[:EXPLICIT_INVOKE]- (callback)
  -[:IMPLICIT_INVOKE]-> (implicitInvoke)
RETURN *
```

which shows the Fiber skeleton of `Jk7H.PwcD` *with* implicitly invoked transitive invocatees included. Contrast Figure 6.22 with the earlier Figures 6.16 and 6.21, it can be seen that far more methods are implicitly invoked in `Jk7H.PwcD` than are explicitly invoked, which is not surprising given the analysis of `Jk7H.PwcD` in Section 6.3.3. Although many of these methods are indirectly invoked, they are all directly connected to the Callback method `onCreate` in `Jk7H.PwcD`'s Fiber per the edge model of Fibers (Table 6.1).

The thoughts behind the design choice of compressing invocation *chains* into the *flat* transitive invocatee set is discussed shortly in Section 6.4.3. Essentially, this allows: 1. efficient query of all externally observable behaviors from a callback as defined by the API method invocations and 2. tolerating non-essential changes in APK internal structures due to, for example, obfuscation.

Transitive invocatees, as shown in Figure 6.22, are insensitive to invocation paths of APKs. While this provides some degree of robustness against invocation-path obfuscation, it fails to meet the demand when there is a need for investigating such invocation paths as in, for example, the query that produce Figure 6.14.

Therefore, to make up for transitive invocatees' invocation-path insensitivity, the Fiber model also include nodes/edges that represent the invocation paths: the type-`CallGraphNode` nodes and type-`INVOKE` edges between such nodes. The connected subgraph of such nodes/edges, starting from a given component callback method m, is called m's component callback call graph (C3G).

Figure 6.23 visualizes the result generated by the following query:

```
MATCH
(a:Apk {sha256:"a00f2b489dac150e513526ab285141d41a127133cd3be2115046e22e189ff2a3"})
  -[:CALLGRAPH]->(cg:CallGraphNode)
MATCH (cg)-[:INVOKE*..4]->(m:CallGraphNode)
RETURN *
```

which shows the C3G from the component callback method `onCreate` within 4-hop from the root node that represents `onCreate` (the number of hops is chosen for visual clarity, rather than due to any limitation on the query). Comparing with the call graph (Figure 6.12) extracted by Algorithm 2, the need for more robust call graph extraction algorithms—which is fully examined in Section 6.5—is plain to see. For this chapter, it suffices to know that C3G is an integrated part of WoF.

6.4.3 A Review of WoF's Design

This section explains the thoughts behind the design of WoF in a series of questions and answers.

6.4.3.1 How the element types/attributes are selected in WoF?

The selection of node types/attributes in WoF is the result of balancing two principles that can be summarized as "less is more" and "simple but not simpler."

Less is more. As previously discussed in Section 6.4.2.1, unlike existing works that encode individual APKs separately, the encoded APKs in WoF (i.e., Fibers) are

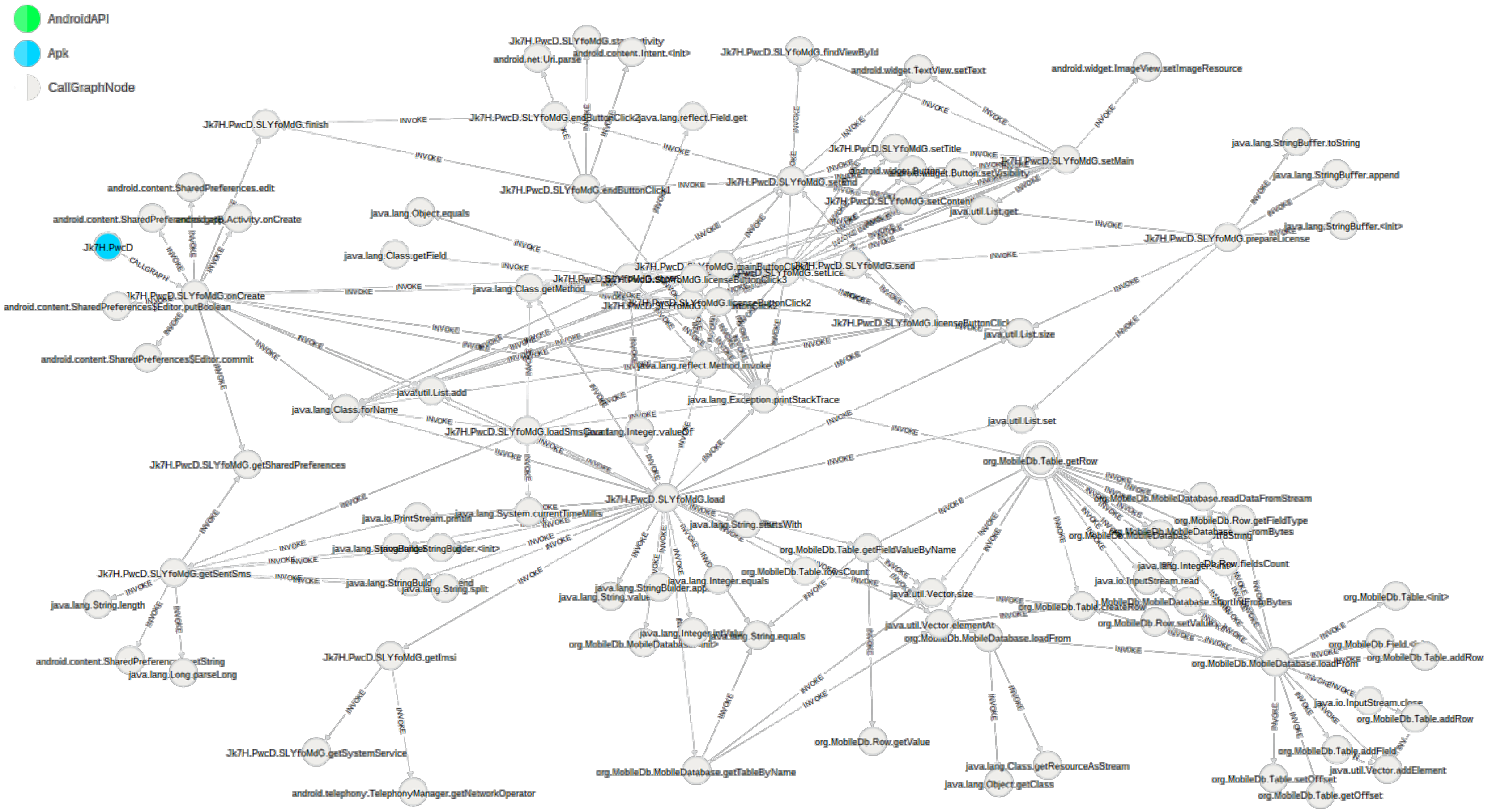

AndroidAPI
Apk
CallGraphNode
Jk7H.PwcD
CALLGRAPH
INVOKE
Jk7H.PwcD.SLYfoMdG.onCreate
android.content.SharedPreferences$Editor.putBoolean
android.content.SharedPreferences$Editor.commit
android.content.SharedPreferences.edit
android.app.Activity.onCreate
Jk7H.PwcD.SLYfoMdG.finish
Jk7H.PwcD.SLYfoMdG.getSharedPreferences
Jk7H.PwcD.SLYfoMdG.getSentSms
java.lang.String.length
android.content.SharedPreferences.getString
java.lang.Long.parseLong
Jk7H.PwcD.SLYfoMdG.getImsi
Jk7H.PwcD.SLYfoMdG.getSystemService
android.telephony.TelephonyManager.getNetworkOperator
android.net.Uri.parse
android.content.Intent.<init>
Jk7H.PwcD.SLYfoMdG.endButtonClick2
java.lang.reflect.Field.get
Jk7H.PwcD.SLYfoMdG.endButtonClick1
Jk7H.PwcD.SLYfoMdG.findViewById
android.widget.TextView.setText
android.widget.ImageView.setImageResource
Jk7H.PwcD.SLYfoMdG.setTitle
Jk7H.PwcD.SLYfoMdG.setMain
android.widget.Button.setVisibility
Jk7H.PwcD.SLYfoMdG.setContent
java.util.List.get
Jk7H.PwcD.SLYfoMdG.send
Jk7H.PwcD.SLYfoMdG.setLicense
Jk7H.PwcD.SLYfoMdG.mainButtonClick
Jk7H.PwcD.SLYfoMdG.licenseButtonClick3
Jk7H.PwcD.SLYfoMdG.licenseButtonClick2
Jk7H.PwcD.SLYfoMdG.licenseButtonClick
java.util.List.size
java.lang.Object.equals
java.lang.Class.getField
java.lang.Class.getMethod
java.lang.reflect.Method.invoke
java.lang.Exception.printStackTrace
java.util.List.add
java.lang.Class.forName
Jk7H.PwcD.SLYfoMdG.loadSmsCount
java.lang.Integer.valueOf
java.util.List.set
java.lang.StringBuffer.toString
java.lang.StringBuffer.append
java.lang.StringBuffer.<init>
Jk7H.PwcD.SLYfoMdG.prepareLicense
Jk7H.PwcD.SLYfoMdG.load
java.io.PrintStream.println
java.lang.System.currentTimeMillis
java.lang.StringBuilder.<init>
java.lang.StringBuilder.append
java.lang.String.split
java.lang.String.startsWith
java.lang.StringBuilder.append
java.lang.String.valueOf
java.lang.Integer.equals
java.lang.Integer.intValue
java.lang.String.equals
org.MobileDb.MobileDatabase.<init>
org.MobileDb.Table.getFieldValueByName
org.MobileDb.Table.rowsCount
java.util.Vector.size
java.util.Vector.elementAt
org.MobileDb.MobileDatabase.loadFrom
org.MobileDb.Row.getValue
org.MobileDb.MobileDatabase.getTableByName
java.lang.Class.getResourceAsStream
java.lang.Object.getClass
org.MobileDb.Table.getRow
org.MobileDb.MobileDatabase.readDataFromStream
org.MobileDb.Row.getFieldType
org.MobileDb.MobileDatabase.toString
org.MobileDb.Row.fieldsCount
java.lang.Integer.<init>
java.io.InputStream.read
org.MobileDb.Table.createRow
org.MobileDb.MobileDatabase.shortIntFromBytes
org.MobileDb.Row.setValue
org.MobileDb.MobileDatabase.loadFrom
org.MobileDb.Table.<init>
org.MobileDb.Field.<init>
org.MobileDb.Table.addRow
java.io.InputStream.close
org.MobileDb.Table.addRow
org.MobileDb.Table.addField
java.util.Vector.addElement
org.MobileDb.Table.setOffset
org.MobileDb.Table.getOffset

Figure 6.23.: The component callback call graph (C3G) from the sole component callback method `onCreate` of `Jk7H.PwcD` within 4-hop from the root node that represents `onCreate`.

merged into a *single* WoF. Therefore, the number of elements incorporated into Fiber will directly impact the scale of WoF, in which a smaller Fiber translates to more efficient access and query for WoF. Moreover, if non-essential elements (i.e., elements of APKs that does not characterize the APK, are shared by many APKs, or can be easily changed/obfuscated) are included and overwhelm the essential elements, similarity between APKs can be obscured like needles in haystack. For both reasons, it is preferable to reduce of the number/variety of elements that are included in the model.

Simple but not simpler. To be useful, WoF should be able to not only *detect similar APKs*, but also *differentiate dissimilar APKs*. For this, despite "less is more," WoF should include enough elements to capture the various ways that APKs can differ from each other. For example, if only `package` is recorded for type-`Apk` nodes in Table 6.2, two different Apks with the same package name could not be distinguished. Contrarily, if only `sha256` is recorded, we would not see that the Dex shown in Figure 6.20 is included by multiple APK with the *same* package name `Jk7H.PwcD`.

Considering these two principles, the types/attributes of nodes/edges of WoF (Tables 6.1 and 6.2) are selected to capture *essential* syntactic similarities between APKs without unnecessarily inflating the model.

6.4.3.2 Why have both transitive invocatees and call graphs in Fiber?

Indeed, each transitive invocatee has at least one corresponding call graph node (there can be more than one on different invocation paths). The same information about an invocated method will redundantly appear in both the transitive invocatee and call graph parts of a Fiber.

However, they serve different purposes. In Section 6.4.4, we will discuss two different kinds of similarity that is captured in Fiber: syntactic similarity and structural similarity. Essentially, at the method invocation level of the Fiber model, transitive invocatee provides syntactic similarity while call graph provides structural similarity. Methods extracted from *different* APKs that have the same identifier (i.e., “package name” and “method name”) map to the *same* transitive invocatee node, whereas such methods map to *different* call graph nodes—This is why a `CallGraphNode` node has an `apk` attribute that uniquely associate it with an `Apk` node (Table 6.2).

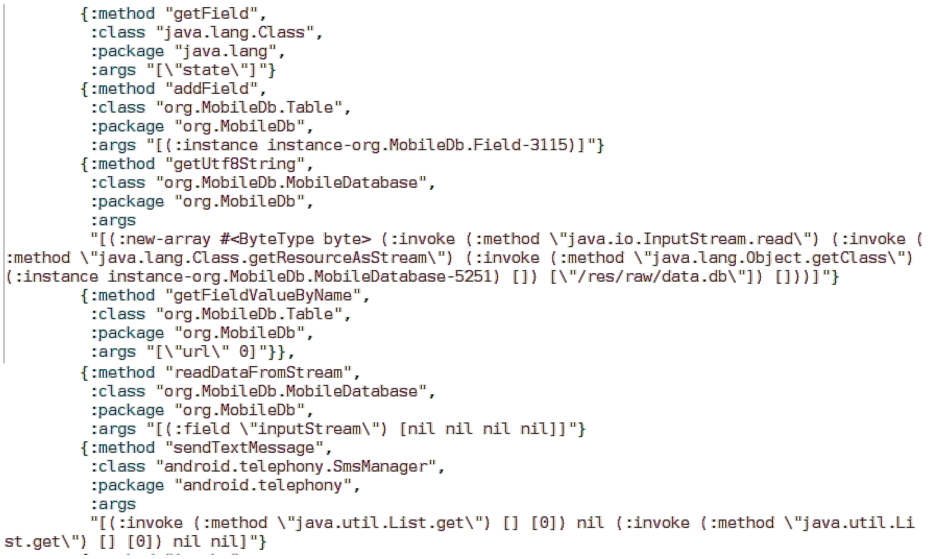
The intention of having both transitive invocatee and call graph parts in the same Fiber is to support efficient queries of different types. Figures 6.13 and 6.14 are two concrete examples: Transitive invocatee supports efficient *set* queries (“does `Jk7H.PwcD` invoke `sendTextMessage`, and from which method callback?”), whereas call graph supports efficient *path* queries (“what are the invocation paths from `onCreate` to `sendTextMessage`?”). Moreover, unlike an approach that implements transitive invocatees in a (for example) hash set data structure, the explicit-graph-connection approach adopted by WoF can efficiently find all APKs that use certain methods without doing individual set membership test—this is one example of the efficiency provided by WoF’s index-free-graph-traversal queries.

6.4.3.3 Why WoF does not include bytecode-level feature, e.g., bytecode *k*-gram?

The reasons are three-fold.

Firstly, a single method can contain hundreds of bytecodes. Moreover, as the obfuscation techniques surveyed in Section 6.2.4 suggest, bytecodes can be easily added/removed/replaced without disturbing overall app semantics. Therefore, the “less is more” principle (discussed above) suggests the exclusion of bytecode features in WoF.

Secondly, although WoF does not directly encode bytecodes, WoF implicitly includes bytecode-level features by encoding the invoke-type bytecode in the transitive invocatees and the C3G—the construction of C3G, as will be discussed in Section 6.5,



```
        {:method "getField",
         :class "java.lang.Class",
         :package "java.lang",
         :args "[\"state\"]"}
        {:method "addField",
         :class "org.MobileDb.Table",
         :package "org.MobileDb",
         :args "[(:instance instance-org.MobileDb.Field-3115)]"}
        {:method "getUtf8String",
         :class "org.MobileDb.MobileDatabase",
         :package "org.MobileDb",
         :args
         "[(:new-array #<ByteType byte> (:invoke (:method \"java.io.InputStream.read\") (:invoke (
:method \"java.lang.Class.getResourceAsStream\") (:invoke (:method \"java.lang.Object.getClass\")
(:instance instance-org.MobileDb.MobileDatabase-5251) []) [\"/res/raw/data.db\"]) []))]"}
        {:method "getFieldValueByName",
         :class "org.MobileDb.Table",
         :package "org.MobileDb",
         :args "[\"url\" 0]"}},
        {:method "readDataFromStream",
         :class "org.MobileDb.MobileDatabase",
         :package "org.MobileDb",
         :args "[(:field \"inputStream\") [nil nil nil nil]]"}
        {:method "sendTextMessage",
         :class "android.telephony.SmsManager",
         :package "android.telephony",
         :args
         "[(:invoke (:method \"java.util.List.get\") [] [0]) nil (:invoke (:method \"java.util.Li
st.get\") [] [0]) nil nil]"}
```

Figure 6.24.: Snippets of method invocation information extracted from `Jk7H.PwcD` during the construction of its Fiber.

involve bytecode analysis that are much more semantic-preserving than, for example, k-gram.

Lastly, WoF is designed to *complement*, rather than, *replace* bytecode-level APK similarity detection techniques such as k-gram. Moreover, if such needs arise, WoF can be easily extended to encode bytecode by adding additional node/edge types for the purpose, without having to change other parts of the design such as the queries.

6.4.3.4 Why the Full Mode is optional and not the default?

While the Full Mode capturing more information (i.e., actual method invocation arguments, as shown in Figure 6.24) than the Compact Mode with no sacrifice in query capabilities, our experience in building prototype WoA from real APK samples suggests that Full Mode is best reserved for deep analysis on a small pool of APKs. For example, an early WoA built from the 1,200+ APK samples in the AGMP dataset contains 70,848 type-`Method` nodes and 609,128 type-`MethodInstance` nodes. The ratio of 10-to-1 between type-`MethodInstance` and type-`Mehotd`, or even worse for some APK analysis modes in Section 6.5, makes the use of Full Mode unwieldy for analysis on a large pool of APKs. Therefore, the Compact Mode is the default mode of operation and will be the only mode examined henceforth.

In the current implementation, method invocation arguments are extracted and stored for query as a byproduct of the C3GE process (Section 6.5). Therefore, a user only needs to decide whether to incorporate such information into WoF for visualization, rather than whether having access to such information.

6.4.4 Syntactic and Structural Similarity

This section discusses two kinds of APK sample similarity that are captured in WoF: syntactic and structural similarity. Section 6.6 presents an extensive evaluation of WoA's APK analysis support capability with both kinds of similarity using a million-node-scale graph built from real malware samples.

6.4.4.1 Syntactic Similarity

Syntactic similarity between APK samples is captured in WoF by the matching of their Fiber elements' attributes. Combining with the "less is more" principle (Section 6.4.3.1), syntactic similarity explains the selection of attributes in Fiber (Table 6.2): They are used to identify nodes from different APKs by their syntactic similarity. For example, `SigningKey`, `Apk`, and `Dex` are uniquely identified by their SHA-256 checksum: two APKs samples that have the `SigningKey` with the same `sha256` attribute value share the same `SigningKey` node in the WoF. Therefore, APKs signed by the same author can be identified in WoF by their common connection to the same `SigningKey` node; similarly, repackaged APKs can be identified by their common connection to the same `Dex` node.

The previous examples in Section 6.4.2 that illustrate different parts of WoF are some examples of using syntactic similarity in APK analysis: 1. The query that generates Figure 6.18 returns malicious APKs that share the same authorship. 2. The query that generates Figure 6.20 returns repackaged APKs that share the same executable code.

These simple queries can be extended to answer other questions that may be of interest to the analyst. For example, a trivial extension to the query of Figure 6.18 is:

```
MATCH (apk:Apk{sha256:"a00f2b489dac150e513526ab285141d41a127133cd3be2115046e22e189ff2a3"})
MATCH (signingKey:SigningKey) -[:SIGN]-> (apk)
WITH signingKey MATCH (signingKey) -[:SIGN]-> (n:Apk) <-- (m:Malware)
WITH n, count(m) as cm WHERE cm>40
WITH collect(DISTINCT n.package) as coln
UNWIND coln as pkgname
MATCH (a:Apk) WHERE a.package=pkgname
WITH pkgname, count(a) as ca
RETURN pkgname, ca ORDER BY ca DESC
```

which counts the different package names of the multiply flagged (by over 40 AV vendors) malware signed by `Jk7H.PwcD`'s author:

```
Jk7H.PwcD,116
ad.notify1,94
tp5x.WGt12,18
vnpysgo.wbwkavy,3
```

Further evaluation is presented in Section 6.6.

6.4.4.2 Structural Similarity

While syntactic similarity supports discovery of related APKs by matching their attributes at different levels, it can be obscured by active obfuscation in adversary scenarios such as plagiarism/malware detection. Take our running example (Section 6.3) for instance, given that the method names (e.g., `send`) are plain English words, it is almost certain that: 1. `Jk7H.PwcD` is an obfuscated package name for the app and 2. `Jk7H.PwcD.SLYfoMdG` is the obfuscated class name for the main class. A dedicated adversary may go further to consistently change all internal method names (albeit external API methods cannot be obfuscated by renaming alone).

To extend utility of WoA's declarative graph analysis to such cases, WoF complements syntactic similarity by structural similarity. As shown in Table 6.2, a node type `CallbackSignature` with attributes `name`, `apk`, and `signature` is introduced into the Fiber model, in which a structural signature (the content of `signature`) is associated with each component callback method (identified by `name`, e.g., `Jk7H.PwcD.SLYfoMdG.onCreate`) of a given APK binary (identified by the checksum of the APK in `apk`). The intention is that `signature` can be used in query to identify component callbacks that behave similarly but have different names (perhaps due to obfuscation).

Given such intention, existing APK method encoding techniques (surveyed in Section 6.2), such as the string encoding proposed by Desnos [62], CTPH-based or bytecode-k-gram-based feature hashes [97], LSH-based semantic vectors [56], and NHGK-based encoding [85], can conceivably be used for `CallbackSignature`'s `signature`. In WoF, we devise an *inter-procedural-control-flow-based* signature, Component Callback Call Graph Degree Frequency Distribution (C3GDFD), and use it for the `signature` attribute of `CallbackSignature` nodes based on the following considerations.

- Existing research on APK obfuscation (Section 6.2.4) shows that inter-procedural control flow is more difficult to be obfuscated than both bytecode sequences and intra-procedural control flow.

- Token-based or intra-procedural-control-flow-based encoding techniques (Figure 6.15) only characterize a single method, e.g., the entry callback `onCreate`.

- By leveraging WoA's robust C3GE (Section 6.5), C3GDFD-based signature characterizes the whole call graph rooted at the callback.

Given the *directed* call graph G_m (an example is shown in Figure 6.23) of a component callback method m extracted by C3GE (Section 6.5), Component Callback Call Graph Degree Frequency Distribution (C3GDFD) based signature of m is a tuple of the following 5 numbers computed from G_m:

- the *node count* of G_m,
- the *mean* of G_m's degree frequency distribution (DFD, defined next),
- the *standard deviation* of G_m's DFD,
- the *skewness* [134] of G_m's DFD, and
- the *kurtosis* [134] of G_m's DFD.

Formally, suppose the node set of G_m is $N_m = \{n_1, n_2, \ldots, n_{\mathbf{card}}(N_m)\}$ ($\mathbf{card}(N_m)$ is the cardinality function of N_m, i.e., the number of elements in N_m), with the *out-degree* of node n_i ($i = 1, 2, \ldots \mathbf{card}(N_m)$) (i.e., the number of *egress* edges from node n_i) is d_i, the (out-)degree frequency distribution (DFD) of G_m is a function f that maps from a non-negative integer j to the number of nodes in G_m that has j as the out-degree:

$$\begin{aligned} f \colon \mathbb{N} &\to \mathbb{N} \\ j &\mapsto \mathbf{card}(\{k | k \in N_m, d_k = j\}). \end{aligned} \tag{6.1}$$

Based on Equation (6.1), the last 4 numbers of the G_m's C3GDFD-based signature are defined as follows[9]:

- the mean μ_m:

$$\mu_m = \frac{1}{\mathbf{card}(N_m)} \sum_{i=0}^{+\infty} f(i). \tag{6.2}$$

[9] There are variations of skewness and kurtosis's definitions; we use the definition from statistics library (incanter; `http://incanter.org/`) used in our implementation here. Reference: `http://goo.gl/5SBu9L`

- the standard deviation σ_m:

$$\sigma_m = \sqrt{\frac{1}{\mathbf{card}(N_m)} \sum_{i=0}^{+\infty} (f(i) - \mu_m)^2}. \tag{6.3}$$

- the skewness γ_m:

$$\gamma_m = \frac{1}{\sigma_m^3} \sum_{i=0}^{+\infty} (f(i) - \mu_m)^3 . \tag{6.4}$$

- the kurtosis β_m:

$$\beta_m = \frac{1}{\sigma_m^4} \sum_{i=0}^{+\infty} (f(i) - \mu_m)^4 - 3. \tag{6.5}$$

Taking Equations (6.2), (6.3), (6.4), and (6.5) together, the C3GDFD-based signature s_m of callback m is:

$$s_m = (\mathbf{card}(N_m), \mu_m, \sigma_m, \gamma_m, \beta_m). \tag{6.6}$$

with any undefined constituents[10] replaced by 0 (the number zero).

The thoughts behind the design (Equation (6.6)) are:

- Statistical shape attributes (node count, mean, standard deviation, skewness, and kurtosis) of the C3G can be used in queries to find callback methods that have structurally similar call graphs, even if the method names have been obfuscated beyond recognition by syntactic similarity alone.

- Since a unique signature is associated with *each* callback (which can be many in a large WoA), signature should be efficient to compute and store, which Equation (6.6) satisfies: 5 numbers to store per signature, with each number has at worst $O(n)$ time-complexity for an n-node call graph to compute.

For example, the following query returns `Jk7H.PwcD`'s C3GDFD-based signatures:

```
MATCH (:Apk{sha256:"a00f2b489dac150e513526ab285141d41a127133cd3be2115046e22e189ff2a3"})
```

[10] Skewness γ_m and kurtoisis β_m can be undefined if, for example, standard deviation $\sigma_m = 0$.

```
  --> (sig:CallbackSignature)
RETURN sig.name, sig.signature
```

which is:

```
Jk7H.PwcD.SLYfoMdG.onCreate,"[102,1.804,3.825,2.526,6.103]"
```

To find component callbacks that have similar signatures, we can issue the following WoA query:

```
MATCH (:Apk{sha256:"a00f2b489dac150e513526ab285141d41a127133cd3be2115046e22e189ff2a3"})
--> (sig:CallbackSignature)
WITH sig
MATCH (siga:CallbackSignature)
WHERE
  abs(sig.signature[0]-siga.signature[0])<5 AND
  abs(sig.signature[0]*sig.signature[1]-siga.signature[0]*siga.signature[1])<5 AND
  abs(sig.signature[2]-siga.signature[2])<0.1 AND
  abs(sig.signature[3]-siga.signature[3])<0.1 AND
  abs(sig.signature[4]-siga.signature[4])<0.1
MATCH siga <--(a:Apk)
RETURN substring(a.sha256, 0, 6) as sa, siga.name, siga.signature
LIMIT 10
```

which find 10 component callbacks that have similarly-shaped C3G (e.g., has a maximal edge/node count difference of 5). We get the result:

```
a7e5a9,Jk7H.PwcD.SLYfoMdG.onCreate,"[102,1.804,3.825,2.526,6.103]"
631c4e,Jk7H.PwcD.SLYfoMdG.onCreate,"[102,1.804,3.825,2.526,6.103]"
726699,Jk7H.PwcD.SLYfoMdG.onCreate,"[102,1.804,3.825,2.526,6.103]"
47a579,Jk7H.PwcD.SLYfoMdG.onCreate,"[102,1.804,3.825,2.526,6.103]"
49a5cc,Jk7H.PwcD.SLYfoMdG.onCreate,"[102,1.804,3.825,2.526,6.103]"
43dbdd,Jk7H.PwcD.SLYfoMdG.onCreate,"[102,1.804,3.825,2.526,6.103]"
cf9190,Jk7H.PwcD.SLYfoMdG.onCreate,"[102,1.804,3.825,2.526,6.103]"
20a0d0,Jk7H.PwcD.SLYfoMdG.onCreate,"[102,1.804,3.825,2.526,6.103]"
ad5e23,Jk7H.PwcD.SLYfoMdG.onCreate,"[102,1.804,3.825,2.526,6.103]"
07f3c7,Jk7H.PwcD.SLYfoMdG.onCreate,"[102,1.804,3.825,2.526,6.103]"
```

which shows that the component callback `Jk7H.PwcD.SLYfoMdG.onCreate` from the repakcaged `Jk7H.PwcD` APKs are detected by the structural similarity of their C3Gs.

6.5 Component Callback Call Graph Extraction (C3GE)

6.5.1 Introduction

This section presents the design and core algorithms of Component Callback Call Graph Extraction (C3GE), which is the other major component of WoA besides the WoF presented in the previous section (Section 6.4). The need for C3GE has been mentioned multiple times in previous sections, for example:

- In Section 6.3.4.2, the insufficiency (Figure 6.12) of a straightforward call graph extraction algorithm (Algorithm 2) for APK analysis is demonstrated using a real Android malware sample (Section 6.3.3). Figures 6.13 and 6.14, which are used as contrast to Figure 6.12, are based on the call graphs extracted by C3GE.

- In Section 6.4.4.2, the C3GDFD-based signature that is used for capturing structural similarity in WoF is based on call graphs extracted by C3GE.

The design of C3GE is motivated by the following observations:

- Invocations to externally defined Java methods in APKs are useful in APK analysis. For example, invocation to the externally define Android API method `android.telephony.SmsManager.sendTextMessage` is the defining behavior of our running example `Jk7H.PwcD` (Section 6.3).

- *Value*, in addition to type, of arguments to method invocations are useful in APK analysis. In addition to providing context to the invocation for analysis (e.g., the target phone number and the content of SMS message are the first and third arguments of `sendTextMessage`, respectively), argument value is critical in resolving implicit control flows such as Java reflection.

- Value exists in APK not only as constants (e.g., integers and `java.lang.String` constants), but can also be constructed through primitive operations (e.g., addition/subtraction) or Java methods (e.g., `java.lang.String.length`).

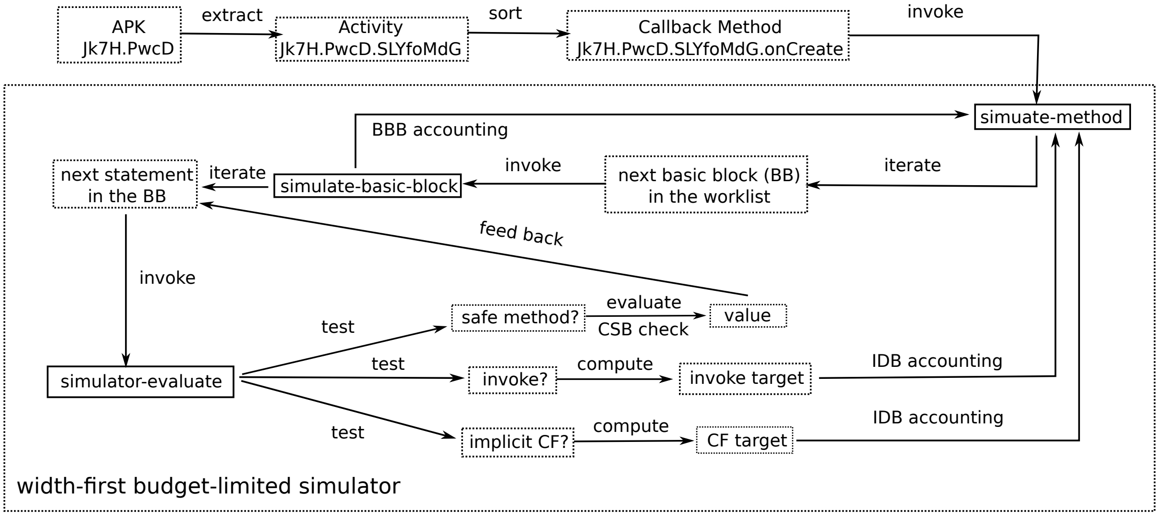


Figure 6.25.: Components of C3GE and their interactions.

- Adversary scenarios such as malware analysis require explicit consideration of anti-analysis techniques that may be employed by malware to evade detection.

These are not mere theoretic considerations: In Section 6.6, we evaluate C3GE against the ADAM APK obfuscation framework [231], in which addressing these points, particularly dynamic value construction, is vital for robustness against obfuscation.

To the best of our knowledge, the C3GE presented here is the first work that addresses all of these points without relying on instrumented virtualization environments such as in DroidScope [218] and AppsPlayground [168], an approach which presents a different set of challenges in analysis coverage and efficiency.

6.5.2 Major Components and Their Interactions

Figure 6.25 illustrates C3GE's components and their interactions. Initially, C3GE takes an APK binary as input and, based on the information contained in its metadata (`AndroidManifest.xml`) and executable code (`Dex`), identifies the Java classes corresponding to app components such as Activity, Service, ContentProvider, and BroadcastReceiver [11]. For example, class `Jk7H.PwcD.SLYfoMdG` is the sole Activity component in our running example `Jk7H.PwcD`.

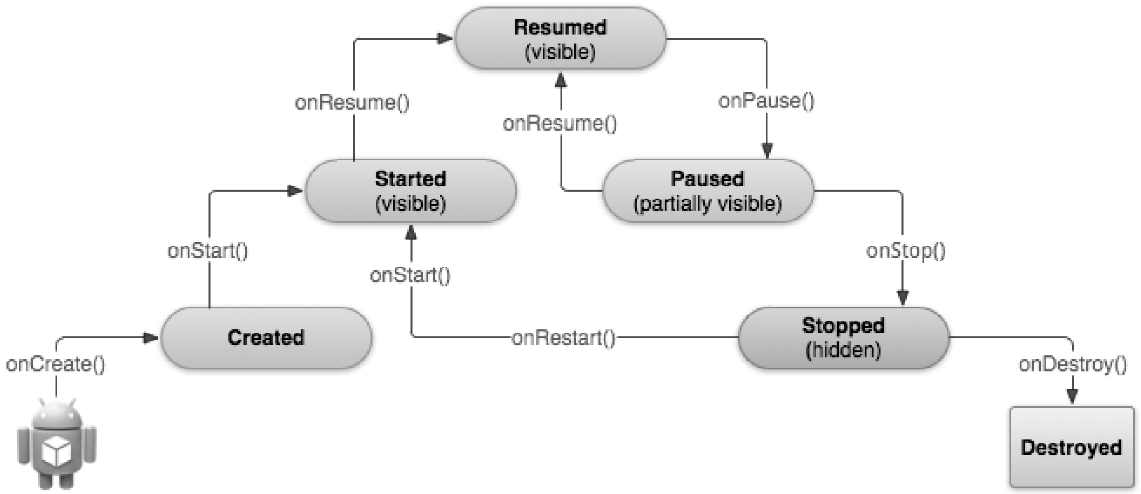


Figure 6.26.: Android Activity lifecycle. Source: `https://developer.android.com/images/training/basics/basic-lifecycle.png`

Then, component callback methods (e.g., `onCreate` and `onStart` of Activity) of these component classes are isolated and sorted according to the Android component lifecycle model [10]. As shown in Figure 6.26, the component lifecycle model imposes an order on the callback methods, so that, for example, `onCreate` is executed before `onStart`. In `Jk7H.PwcD`, the Activity class `Jk7H.PwcD.SLYfoMdG` only has a single callback `onCreate`.

After the callback methods are identified and sorted, each method is submitted to a *width-first budget-limited simulator* to extract its Component Callback Call Graph (C3G). The simulator is the *core* of C3G, which consists of the following interacting procedures:

- `simulate-method`: method simulation,
- `simulate-basic-block`: basic block simulation, and
- `simulate-evaluate`: expression evaluation.

The major data/control flows between these procedures are illustrated in Figure 6.25, summarized next, and is fully examined later in Sections 6.5.3 and 6.5.4.

6.5.2.1 `simulate-method`

`simulate-method` *iteratively* dequeues the next item from the *worklist* and feeds the item into `simulate-basic-block`. After `simulate-basic-block` finishes, the new worklist items returned from it are enqueued into the worklist. Each processed work *deducts* 1 unit from the current `simulate-method` invocation's basic-block budget (BBB). `simulate-method` terminates when either the worklist becomes empty or BBB is depleted, i.e., becomes 0.

Some points to note:

- Each work corresponds to a Soot [200, 121] statement in the current method, from which `simulate-basic-block` will begin to process.
- Initially, the worklist consists of a single work that corresponds to the first statement in the current method.
- The dequeue/enqueue discipline using worklist makes the simulation process *width-first*: In the control flow graph of the current method, all children branches of branching point (e.g., the `then` and `else` branches of a `if` statement) are processed before further descendants of any of the branches.
- The "width-first" processing prevents unbalanced exploration of branches, which might be exploited by an adversary to waste BBB on uninteresting branches. More on this in Section 6.5.4.3.

6.5.2.2 `simulate-basic-block`

Starting from a statement supplied by `simulate-method`, `simulate-basic-block` extracts the subsequent basic block (i.e., a consecutive sequence of statements with no intermediate branching and a single branching at the end), processes each statement in that basic block, and evaluate all operations/method-invocations contained in those statements with `simulator-evaluate`. The processing consists of the following stages: 1. iterate through the intermediate non-branching statements, 2. examine the last branching statement, evaluate it if necessary, and return "next statements" to `simulate-method`.

Some points to note:

- `simulate-basic-block`'s behavior depends on the simulation strategy (conservative branching, aggressive branching (the default), and linear scan) specified at program startup.
  - Under the "aggressive branching" and "conservative branching" strategies, the behavior is to "process the next basic block" as described above.
  - Under the "linear scan" strategy, the behavior is to sequentially process all statements in the method that follow the beginning statement, rather than only the next basic block.
- As will be further discussed in Section 6.5.4.4 after a concrete examination of the algorithms in Section 6.5.3, the simulation strategies provide different trade-offs between simulation efficiency, accuracy, and coverage.

6.5.2.3 `simulator-evaluate`

While `simulate-method` and `simulate-basic-block` are mostly about scheduling statements for analysis, the actual work of analysis is done in `simulator-evaluate`.

As shown in Figure 6.25, `simulate-evaluate` behaves differently based on the expression under evaluation.

- If the expression under evaluation is a Java method invocation corresponding to the `*-invoke` Dalvik bytecode, `simulate-evaluate` evaluates the method by passing it to `simulate-method` if Invocation Depth Budget (IDB) is not exceeded.
  - Simulation of the new method follows the same process as previously described for `simuate-method`.
  - IDB sets an upper bound on the depth that `simulate-method` will be nestly invoked, and prevents the simulation process from getting trapped in an infinite invocation loop.
- If the expression under evaluation is a *safe method*, `simulate-method` executes the safe method on the host JVM if the allocated space does not exceed the Collection Size Budget (CSB).
  - A safe method is a Java method that is shared by Android and JVM, and is either informational or has no other side effects beside creating and manipulating data storage.
  - Examples of safe methods are methods of class `java.lang.String` and `java.util.ArrayList`.
  - The safe methods in our implementation is listed in Figure 6.34.
  - CSB prevents simulator's resources (e.g, JVM heap space) from being exhausted by malicious storage allocation (e.g., a 1 GBytes array) planted by the adversary in the app's bytecode that is never executed in regular app usage.
- If the expression under evaluation is an implicit control flow (e.g., Java reflection), `simulate-method` resolves the control flow target and invokes the target

with `simulate-method` as if it is a regular method invocation (the first case above).

Several regular cases of evaluation that are not shown in Figure 6.25 include:

- assignment to Java class fields or local variables,
- primitive arithmetic operations such as additions and multiplications, and
- Java array allocations (subject to CSB check).

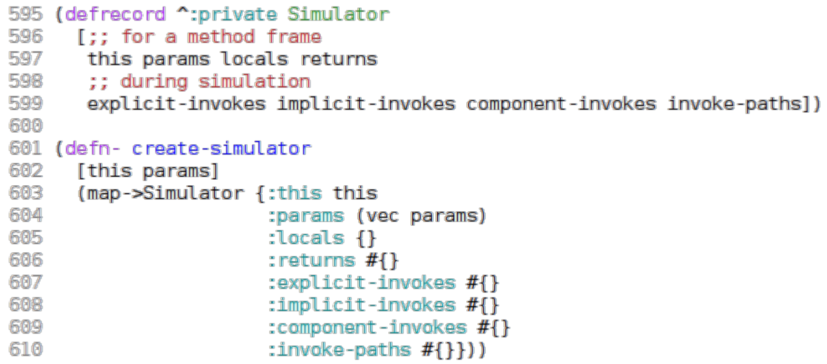
6.5.3 Techniques and Algorithms

After an overview of CG3E's major components in the previous section, we examine its key techniques and algorithms in this section. For result reproducibility, discussion is accompanied by real code snippets (with unrelated code suppressed but source code line numbers retained) from the implementation. Besides the definiteness of real code (unlike psuedocode, real code has well-defined *executable* semantics and must work to be meaningful), the use of a set of techniques, homoiconicity [202], higher-order functions [171], and functional persistent data structures [65], in our implementation enables succinct and concrete expression of algorithms that are otherwise difficult to express. The code snippets shown below is written in the Clojure programming language [99] (a modern Lisp dialect hosted on the Java Virtual Machine (JVM) platform) and corresponds to commit `f39b41` on the GitHub source code repository[11].

6.5.3.1 Simulator Definition and Initialization

Figure 6.27 shows the record structure and the initialization procedure of the C3GE simulator. A `Simulator` instance has two kinds of data records:

[11] `https://github.com/pw4ever/web-of-apks/tree/f39b41`



```
(defrecord ^:private Simulator
  [;; for a method frame
   this params locals returns
   ;; during simulation
   explicit-invokes implicit-invokes component-invokes invoke-paths])

(defn- create-simulator
  [this params]
  (map->Simulator {:this this
                   :params (vec params)
                   :locals {}
                   :returns #{}
                   :explicit-invokes #{}
                   :implicit-invokes #{}
                   :component-invokes #{}
                   :invoke-paths #{}}))
```

Figure 6.27.: Record structure and initialization of the C3GE simulator. File: `src/-woa/apk/dex/soot/simulator.clj`.

- Method frame records, which include the `this` pointer, method argument values, local variables, and return values.

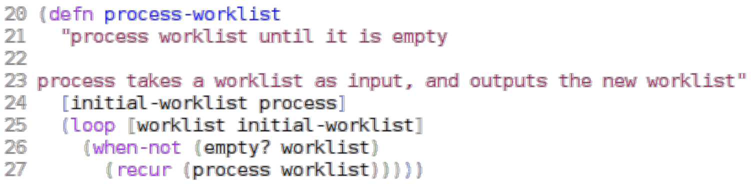
- Invocations of various types encountered during simulation, which include explicit invokes (i.e., through `*-invoke` bytecode), implicit invokes (e.g., Java reflection or thread), and invocation paths that are used to reconstruct call graphs.

`create-simulator` initializes a `Simulator` instance from the supplied `this` reference and method argument values, and initialize other data records to be either empty maps (`{}`) or empty sets (`#{}`).

As will be discussed later with Figures 6.29 and 6.30, each invocation of `simulate--basic-block` by `simulate-method` will be executed in an *independent* simulator instance, implemented with immutable persistent data structure [19], that allows basic block simulations of different branches to be parallelized.

6.5.3.2 Worklist Processing: `process-worklist`

Figure 6.28 shows the worklist processing procedure `process-worklist`, which processes items in the worklist until the worklist becomes empty. The use of Clojure's



```
20 (defn process-worklist
21   "process worklist until it is empty
22
23 process takes a worklist as input, and outputs the new worklist"
24   [initial-worklist process]
25   (loop [worklist initial-worklist]
26     (when-not (empty? worklist)
27       (recur (process worklist)))))
```

Figure 6.28.: The worklist processing procedure `process-worklist`, which *iteratively*, instead of recursively, processes items in the worklist until the worklist becomes empty. File: `src/woa/util.clj`.

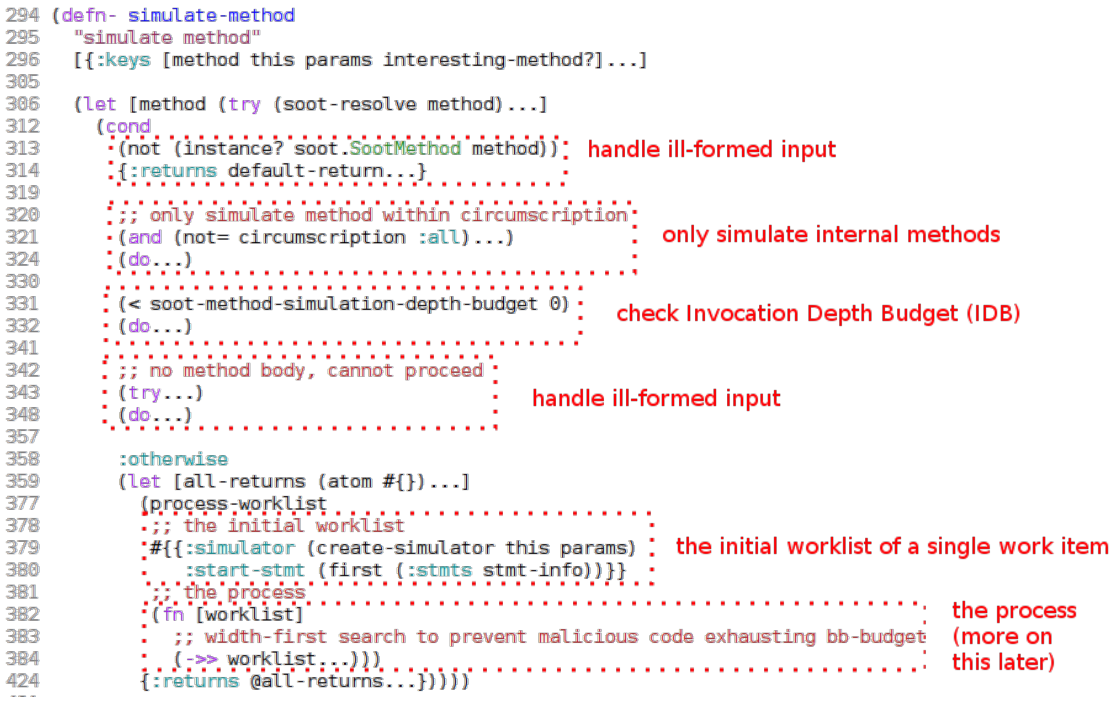
`loop/recur` control structure allows `process-worklist` to be written in *recursive* form but executes *iteratively*, which avoids stack overflow during worklist processing due to deep recursions. `loop/recur` can be viewed as manual, single-function, tail-call optimization (TCO) [49].

More specifically, `process-worklist` takes an initial worklist `initial-worklist` as an argument to bootstrap the iterative work processing process (line 25 in Figure 6.28); it terminates when all items in the worklist has been processed and the worklist becomes empty (line 26 in Figure 6.28). The higher-order function, supplied as the argument `process` to `process-worklist`, takes the worklist of the current iteration and returns the new worklist for the next iteration. This abstraction of `process` is used in multiple places in the implementation, with the prime example being the next one in `simulate-method`.

6.5.3.3 Simulating Method: `simulate-method`

As previously summarized in Section 6.5.2.1, `simulate-method` iteratively schedule worklist items for `simulate-basic-block` until the worklist becomes empty. Figure 6.29 shows the overall logic of `simulate-method` and Figure 6.30 focuses on the worklist processing.

Specifically, before initiating worklist processing (by `process-worklist` on line 377), `simulate-method` does some state integrity check: 1. Filtering out ill-formed



```
294 (defn- simulate-method
295   "simulate method"
296   [{:keys [method this params interesting-method?]...]
305
306   (let [method (try (soot-resolve method)...]
312     (cond
313       (not (instance? soot.SootMethod method))
314       {:returns default-return...}
319
320       ;; only simulate method within circumscription
321       (and (not= circumscription :all)...)
324       (do...)
330
331       (< soot-method-simulation-depth-budget 0)
332       (do...)
341
342       ;; no method body, cannot proceed
343       (try...)
348       (do...)
357
358       :otherwise
359       (let [all-returns (atom #{})...]
377         (process-worklist
378          ;; the initial worklist
379          #{{:simulator (create-simulator this params)
380             :start-stmt (first (:stmts stmt-info))}}
381          ;; the process
382          (fn [worklist]
383            ;; width-first search to prevent malicious code exhausting bb-budget
384            (->> worklist...)))
424         {:returns @all-returns...}))))
```

handle ill-formed input

only simulate internal methods

check Invocation Depth Budget (IDB)

handle ill-formed input

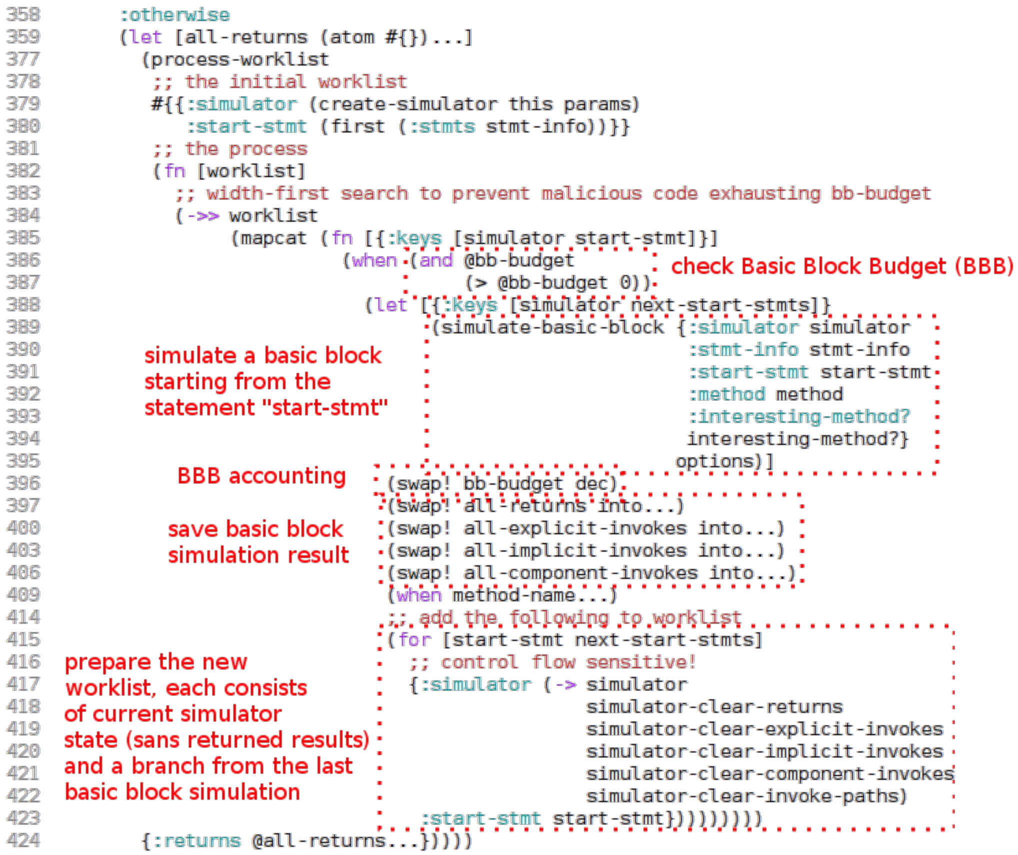
the initial worklist of a single work item

the process
(more on
this later)

Figure 6.29.: Overall method simulation logic of `simulate-method`, with unrelated details suppressed by suffixing triple dots "..." (suppression is also applied in later figures). File: `src/woa/apk/dex/soot/simulator.clj`.

input (lines 313–318 and 342–356); 2. Do not proceed if the simulator runs out of Invocation Depth Budget (IDB; lines 331–340); 3. Check if the method under simulation is an internal method, i.e., methods that are implemented in the APK rather than being part of external Android API (lines 321–329). Afterwards, a single worklist item, consisting of the freshly created simulator instance (line 379) and the first statement of the current method (line 380), is used as the initial worklist to initiate the worklist processing logic (lines 382–423).

As shown in Figure 6.30, for each worklist item, if there is still Basic Block Budget (BBB) left (lines 386–387), `simulate-method` invokes `simulate-basic-block` with this item (lines 389–395) and, after `simulate-basic-block` returns, takes one unit of BBB off for the worklist item (i.e., does BBB accounting on line 396) and saves the results (lines 397–408).



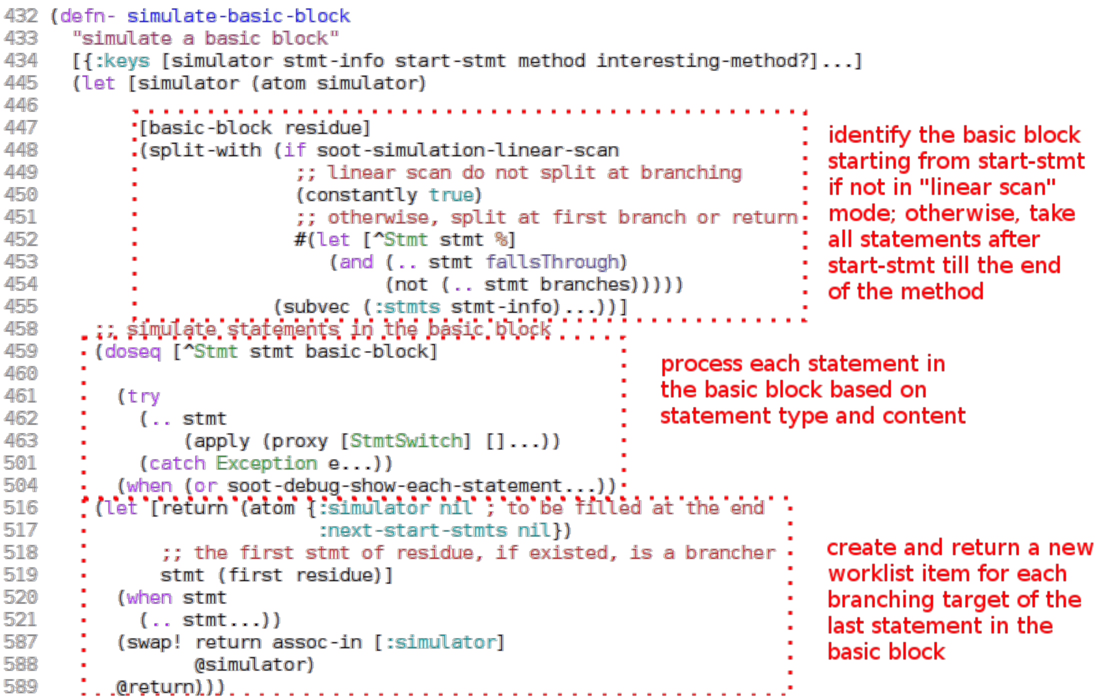
```
358       :otherwise
359       (let [all-returns (atom #{})...]
377         (process-worklist
378          ;; the initial worklist
379          #{{:simulator (create-simulator this params)
380              :start-stmt (first (:stmts stmt-info))}}
381          ;; the process
382          (fn [worklist]
383            ;; width-first search to prevent malicious code exhausting bb-budget
384            (->> worklist
385                 (mapcat (fn [{:keys [simulator start-stmt]}]
386                            (when (and @bb-budget
387                                        (> @bb-budget 0))
388                              (let [{:keys [simulator next-start-stmts]}
389                                     (simulate-basic-block {:simulator simulator
390                                                            :stmt-info stmt-info
391                                                            :start-stmt start-stmt
392                                                            :method method
393                                                            :interesting-method?
394                                                            interesting-method?}
395                                                           options)]
396                                (swap! bb-budget dec)
397                                (swap! all-returns into...)
400                                (swap! all-explicit-invokes into...)
403                                (swap! all-implicit-invokes into...)
406                                (swap! all-component-invokes into...)
409                                (when method-name...)
414                                ;; add the following to worklist
415                                (for [start-stmt next-start-stmts]
416                                  ;; control flow sensitive!
417                                  {:simulator (-> simulator
418                                                  simulator-clear-returns
419                                                  simulator-clear-explicit-invokes
420                                                  simulator-clear-implicit-invokes
421                                                  simulator-clear-component-invokes
422                                                  simulator-clear-invoke-paths)
423                                   :start-stmt start-stmt})))))))))
424         {:returns @all-returns...}))))
```

check Basic Block Budget (BBB)
simulate a basic block starting from the statement "start-stmt"
BBB accounting
save basic block simulation result
prepare the new worklist, each consists of current simulator state (sans returned results) and a branch from the last basic block simulation

Figure 6.30.: Worklist processing logic in `simulate-method`. File: `src/woa/apk/-dex/soot/simulator.clj`.

As will be discussed shortly in Section 6.5.3.4, `simulate-basic-block` can return zero (for `return` statement), one (for `goto` statement), or two (for `if` statement) “next starting statements” (`next-start-stmts` on line 415). `simulate-method` creates one new worklist item per each next-starting-statement (`start-stmt` on line 415), which consists of the simulator state after `simulate-basic-block` returns (lines 417–422) and `start-stmt` (line 423).

A notable point in `simulate-method` is the use of immutable persistent data structure [19] (as provided by Clojure’s built-in data structure implementation) to represent simulator state (refer to `create-simulator` in Figure 6.27 and updates on lines 417–422 in Figure 6.30), which allows efficient (through structural sharing) and



```
432 (defn- simulate-basic-block
433   "simulate a basic block"
434   [{:keys [simulator stmt-info start-stmt method interesting-method?]...]
445   (let [simulator (atom simulator)
446
447         [basic-block residue]
448         (split-with (if soot-simulation-linear-scan
449                       ;; linear scan do not split at branching
450                       (constantly true)
451                       ;; otherwise, split at first branch or return
452                       #(let [^Stmt stmt %]
453                          (and (.. stmt fallsThrough)
454                               (not (.. stmt branches)))))
455                     (subvec (:stmts stmt-info)...))]
458     ;; simulate statements in the basic block
459     (doseq [^Stmt stmt basic-block]
460
461       (try
462         (.. stmt
463             (apply (proxy [StmtSwitch] []...))
501         (catch Exception e...))
504       (when (or soot-debug-show-each-statement...)))
516     (let [return (atom {:simulator nil ; to be filled at the end
517                         :next-start-stmts nil})
518           ;; the first stmt of residue, if existed, is a brancher
519           stmt (first residue)]
520       (when stmt
521         (.. stmt...))
587       (swap! return assoc-in [:simulator]
588              @simulator)
589       @return)))
```

identify the basic block starting from start-stmt if not in "linear scan" mode; otherwise, take all statements after start-stmt till the end of the method

process each statement in the basic block based on statement type and content

create and return a new worklist item for each branching target of the last statement in the basic block

Figure 6.31.: Overall basic block simulation logic of `simulate-basic-block`. File: `src/woa/apk/dex/soot/simulator.clj`.

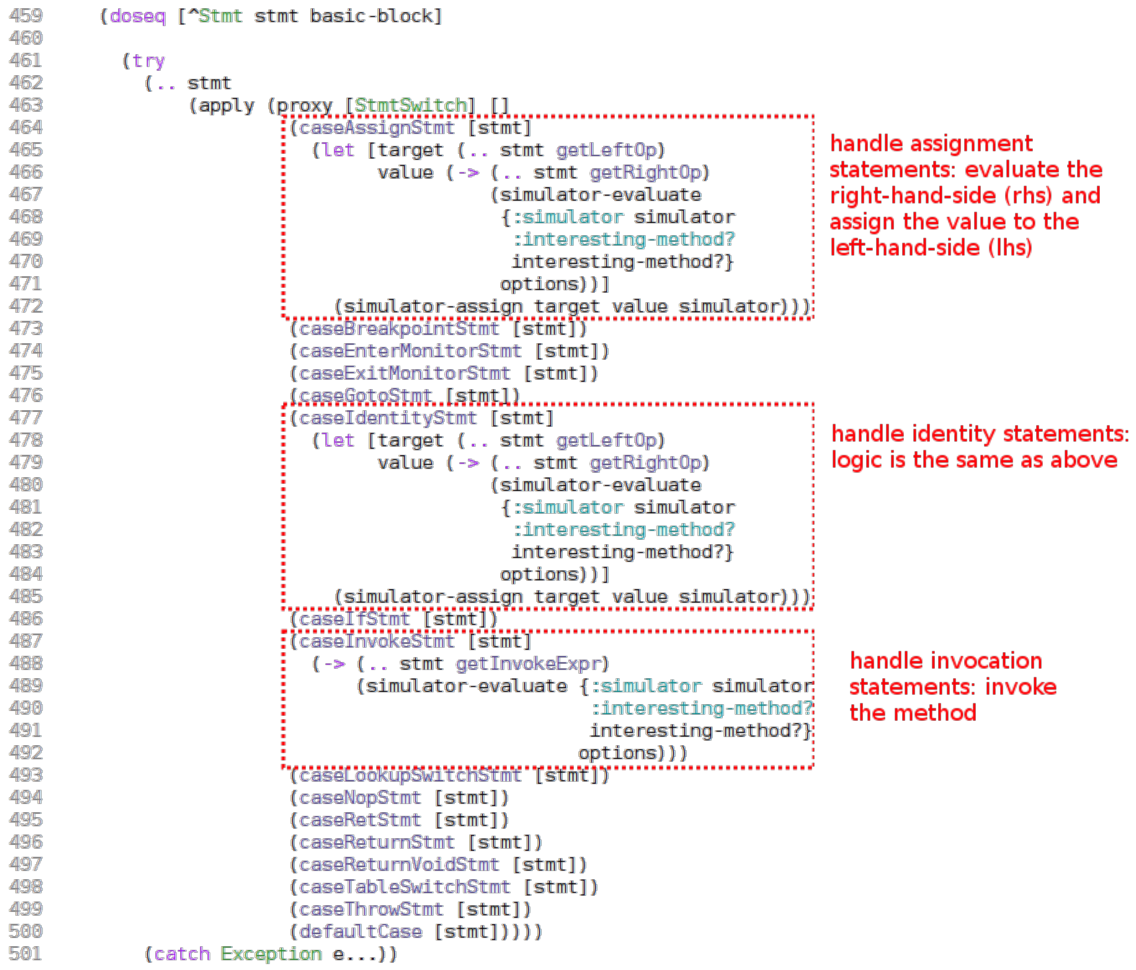
succinct (without explicit state copying) representation of worklist items that diverge from the current simulator state in future iterations.

6.5.3.4 Simulating Basic Block: `simulate-basic-block`

Each worklist item passed to `simulate-basic-block` consists of a simulator state and a starting statement, i.e., `simulator` and `start-stmt` on line 434 of Figure 6.31. Figure 6.31 shows the overall structure of `simulate-basic-block`'s algorithm. `simulate-basic-block`'s logic consists of the following steps.

1. Identify the basic block starting from `start-stmt` (lines 447–455);

2. Process each statement in the basic block based on its statement type and content (lines 461–503; details in Figure 6.32);



```
459     (doseq [^Stmt stmt basic-block]
460
461       (try
462         (.. stmt
463             (apply (proxy [StmtSwitch] []
464                      (caseAssignStmt [stmt]
465                        (let [target (.. stmt getLeftOp)
466                              value (-> (.. stmt getRightOp)
467                                        (simulator-evaluate
468                                         {:simulator simulator
469                                          :interesting-method?
470                                          interesting-method?}
471                                         options))]
472                          (simulator-assign target value simulator)))
473                      (caseBreakpointStmt [stmt])
474                      (caseEnterMonitorStmt [stmt])
475                      (caseExitMonitorStmt [stmt])
476                      (caseGotoStmt [stmt])
477                      (caseIdentityStmt [stmt]
478                        (let [target (.. stmt getLeftOp)
479                              value (-> (.. stmt getRightOp)
480                                        (simulator-evaluate
481                                         {:simulator simulator
482                                          :interesting-method?
483                                          interesting-method?}
484                                         options))]
485                          (simulator-assign target value simulator)))
486                      (caseIfStmt [stmt])
487                      (caseInvokeStmt [stmt]
488                        (-> (.. stmt getInvokeExpr)
489                            (simulator-evaluate {:simulator simulator
490                                                 :interesting-method?
491                                                 interesting-method?}
492                                                options)))
493                      (caseLookupSwitchStmt [stmt])
494                      (caseNopStmt [stmt])
495                      (caseRetStmt [stmt])
496                      (caseReturnStmt [stmt])
497                      (caseReturnVoidStmt [stmt])
498                      (caseTableSwitchStmt [stmt])
499                      (caseThrowStmt [stmt])
500                      (defaultCase [stmt]))))
501         (catch Exception e...))
```

handle assignment statements: evaluate the right-hand-side (rhs) and assign the value to the left-hand-side (lhs)

handle identity statements: logic is the same as above

handle invocation statements: invoke the method

Figure 6.32.: Process basic block statements before the last branching statement in `simulate-basic-block`. File: `src/woa/apk/dex/soot/simulator.clj`.

3. Create and return a new worklist item for each branching target of the last statement in the basic block (lines 516–589; details in Figure 6.33).

An exception to this logic is that, in Step 1, if the "linear scan" simulation strategy is selected in processing the APK (i.e., `soot-simulation-linear-scan` is true on line 448), *all* statements starting from `start-stmt` to the end of the current method, instead of the next basic block, are processed in Step 2. Therefore, under the "linear scan" simulation strategy, statements are processed once by their appearance order in the method without intermediate branching.

Figure 6.32 shows the algorithm of processing basic block statements before the last branching in simulate-basic-block. Three cases are considered:

- Assignment statement (lines 464–472): A common statement type that assigns the value of some expression to a left-hand-side (lhs) expression (e.g., a method frame local or Java class field). An example is line 1387 in Figure 6.5, which assigns a Java array of type java.lang.Class to the method frame local r3.

- Identity statement (lines 477–485): Special assignment statement for method arguments. An example is line 1373 in Figure 6.5, which assigns this instance to the method frame local r0.

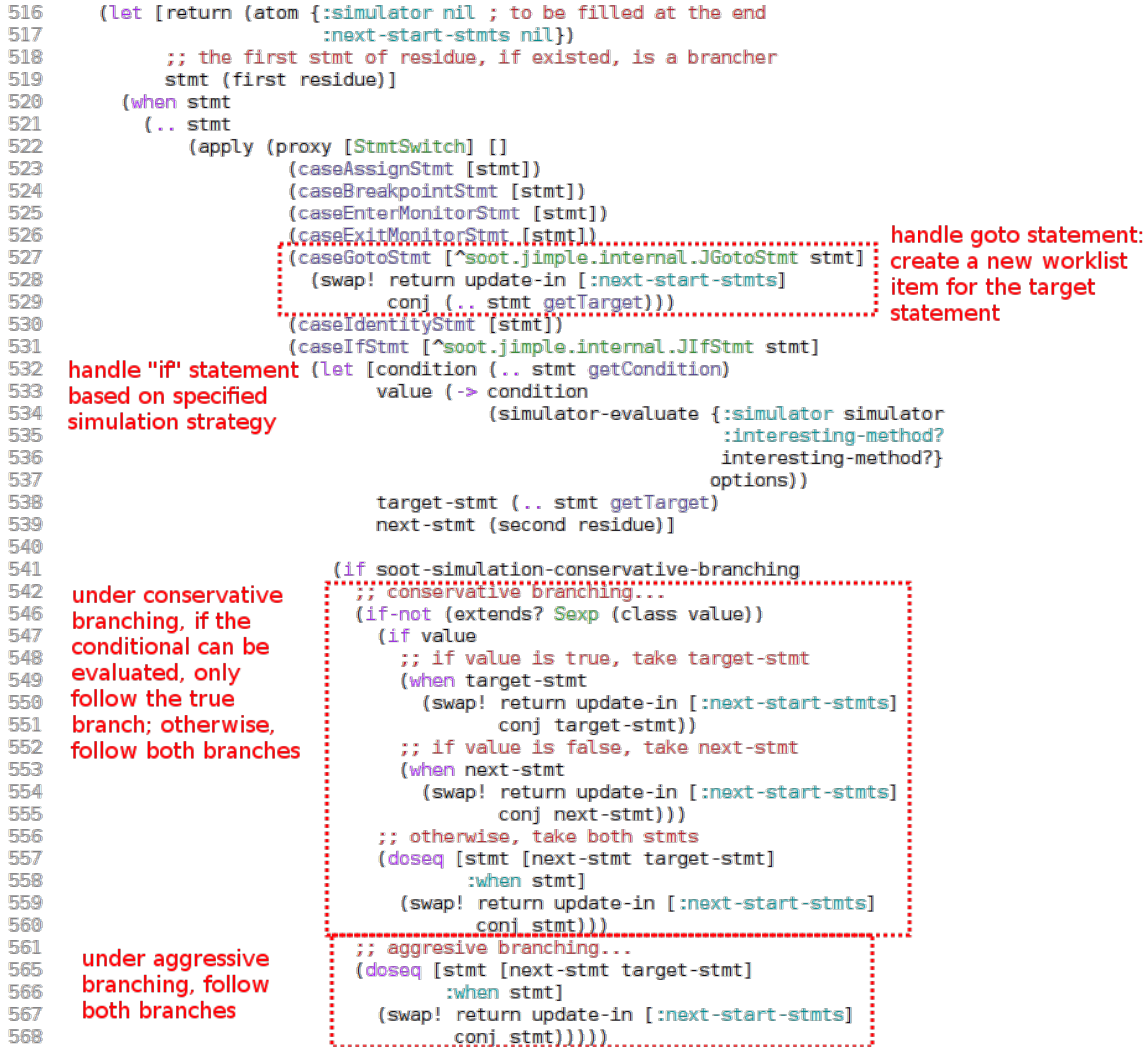
- Invoke statement (lines 487–492): A standalone method invocation statement. An example is line 1393 in Figure 6.5, which invokes the Java method invoke of class java.lang.reflect.Method.

The handling logic of these cases are as expected: Expressions are evaluated in the context of the current simulator state, and the simulator state is accordingly updated if there is any assignment. Note that majority of the work is centralized in simulator-evaluate, which will be examined shortly in Section 6.5.3.5.

Mirroring the above is the algorithm of processing the last branching statement in the current basic block in simulate-basic-block, as shown in Figure 6.33. Besides the return statement case (which returns a value to simulate-method; not shown in Figure 6.33), there are two cases, both conditional statements, to handle:

- goto statement (lines 527–529): Unconditional branching to a target statement. An example is line 1433 in Figure 6.5, which unconditionally jumps to the statement on label 1.

- if statement (lines 531–568): Conditional branching to one of two target statements. An example is line 1382 in Figure 6.5, which jumps to the statement on label 6 if method frame local i0 $<$ 3, or falls through to the next statement (on label 2) otherwise, i.e., i0 $\geq$ 3.



```
516     (let [return (atom {:simulator nil ; to be filled at the end
517                         :next-start-stmts nil})
518           ;; the first stmt of residue, if existed, is a brancher
519           stmt (first residue)]
520       (when stmt
521         (.. stmt
522             (apply (proxy [StmtSwitch] []
523                      (caseAssignStmt [stmt])
524                      (caseBreakpointStmt [stmt])
525                      (caseEnterMonitorStmt [stmt])
526                      (caseExitMonitorStmt [stmt])
527                      (caseGotoStmt [^soot.jimple.internal.JGotoStmt stmt]
528                        (swap! return update-in [:next-start-stmts]
529                               conj (.. stmt getTarget)))
530                      (caseIdentityStmt [stmt])
531                      (caseIfStmt [^soot.jimple.internal.JIfStmt stmt]
532                        (let [condition (.. stmt getCondition)
533                              value (-> condition
534                                        (simulator-evaluate {:simulator simulator
535                                                             :interesting-method?
536                                                             interesting-method?}
537                                                            options))
538                              target-stmt (.. stmt getTarget)
539                              next-stmt (second residue)]
540
541                          (if soot-simulation-conservative-branching
542                            ;; conservative branching...
546                            (if-not (extends? Sexp (class value))
547                              (if value
548                                ;; if value is true, take target-stmt
549                                (when target-stmt
550                                  (swap! return update-in [:next-start-stmts]
551                                         conj target-stmt))
552                                ;; if value is false, take next-stmt
553                                (when next-stmt
554                                  (swap! return update-in [:next-start-stmts]
555                                         conj next-stmt)))
556                              ;; otherwise, take both stmts
557                              (doseq [stmt [next-stmt target-stmt]
558                                      :when stmt]
559                                (swap! return update-in [:next-start-stmts]
560                                       conj stmt)))
561                            ;; aggresive branching...
565                            (doseq [stmt [next-stmt target-stmt]
566                                    :when stmt]
567                              (swap! return update-in [:next-start-stmts]
568                                     conj stmt)))))
```

handle goto statement: create a new worklist item for the target statement

handle "if" statement based on specified simulation strategy

under conservative branching, if the conditional can be evaluated, only follow the true branch; otherwise, follow both branches

under aggressive branching, follow both branches

Figure 6.33.: Process the last branching statement of a basic block in `simulate--basic-block`. File: `src/woa/apk/dex/soot/simulator.clj`.

Note that both `goto` and `if` statements are *intra-procedural* control flow transfer, which means that the target statements must be in the same method as the conditional statement. The only mechanism for inter-procedural control transfer is through `invoke` statement/expression.

The effect of both `goto` and `if` statements is to create and return new worklist items for the next iteration of `simulate-method`. While the `goto` statement simply returns the unconditional branching target (lines 528–529), the behavior of the `if` statement depends on two things: 1. simulation branching strategy: conservative branching or aggressive branching (the default); 2. whether the conditional expression can be evaluated in the context of the simulator state. More precisely:

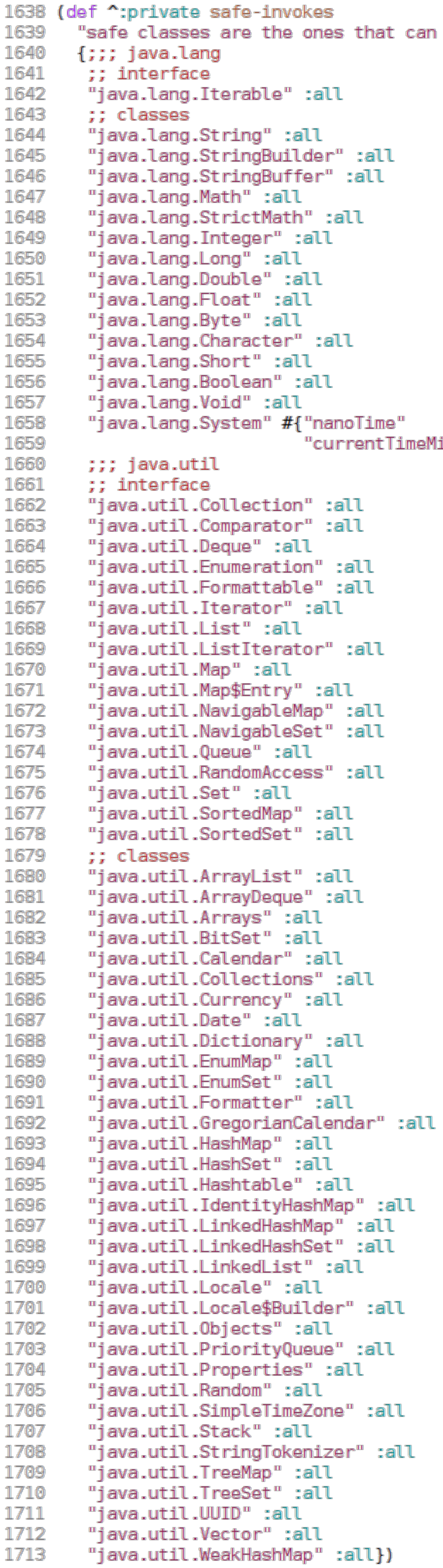
- Under conservative branching (lines 542–560), if the conditional expression can be evaluated in the context of the simulator statement (lines 547–555), only the true branch is followed; otherwise (if the conditional expression can *not* be evaluated; lines 557–560), both branches are followed.
- Under aggressive branching (lines 561–568), both branches are followed.

These different simulation branching strategies provide different choices that balance between analysis coverage, efficiency, and accuracy. While conservative branching is more accurate in the sense that dead branches (branches of conditionals that evaluate to false) will not be followed, it may miss branches that may evaluate to true at run time. In Section 6.6, we will discuss such examples using real app samples.

6.5.3.5 Evaluating Expressions: `simulator-evaluate`

The essential work of the C3GE simulator is to evaluate expressions that appear in statements. Such logic is centralized in `simulator-evaluate`, which is examined in this section. Specifically, we examine the following keys techniques behind `simulator-evaluate`'s logic, summarized previously in Section 6.5.2.3: *safe method evaluation* and *implicit control flow resolution.*

In Section 6.5.2, we point out: 1. The value of expression values in providing content for analysis and in resolving implicit control flows. 2. Values can be dynamically constructed, in addition to be hard-coded in executable code (Dex) as constants.



```
(def ^:private safe-invokes
  "safe classes are the ones that can be simulated in Clojure"
  {;;; java.lang
   ;; interface
   "java.lang.Iterable" :all
   ;; classes
   "java.lang.String" :all
   "java.lang.StringBuilder" :all
   "java.lang.StringBuffer" :all
   "java.lang.Math" :all
   "java.lang.StrictMath" :all
   "java.lang.Integer" :all
   "java.lang.Long" :all
   "java.lang.Double" :all
   "java.lang.Float" :all
   "java.lang.Byte" :all
   "java.lang.Character" :all
   "java.lang.Short" :all
   "java.lang.Boolean" :all
   "java.lang.Void" :all
   "java.lang.System" #{"nanoTime"
                        "currentTimeMillis"}
   ;;; java.util
   ;; interface
   "java.util.Collection" :all
   "java.util.Comparator" :all
   "java.util.Deque" :all
   "java.util.Enumeration" :all
   "java.util.Formattable" :all
   "java.util.Iterator" :all
   "java.util.List" :all
   "java.util.ListIterator" :all
   "java.util.Map" :all
   "java.util.Map$Entry" :all
   "java.util.NavigableMap" :all
   "java.util.NavigableSet" :all
   "java.util.Queue" :all
   "java.util.RandomAccess" :all
   "java.util.Set" :all
   "java.util.SortedMap" :all
   "java.util.SortedSet" :all
   ;; classes
   "java.util.ArrayList" :all
   "java.util.ArrayDeque" :all
   "java.util.Arrays" :all
   "java.util.BitSet" :all
   "java.util.Calendar" :all
   "java.util.Collections" :all
   "java.util.Currency" :all
   "java.util.Date" :all
   "java.util.Dictionary" :all
   "java.util.EnumMap" :all
   "java.util.EnumSet" :all
   "java.util.Formatter" :all
   "java.util.GregorianCalendar" :all
   "java.util.HashMap" :all
   "java.util.HashSet" :all
   "java.util.Hashtable" :all
   "java.util.IdentityHashMap" :all
   "java.util.LinkedHashMap" :all
   "java.util.LinkedHashSet" :all
   "java.util.LinkedList" :all
   "java.util.Locale" :all
   "java.util.Locale$Builder" :all
   "java.util.Objects" :all
   "java.util.PriorityQueue" :all
   "java.util.Properties" :all
   "java.util.Random" :all
   "java.util.SimpleTimeZone" :all
   "java.util.Stack" :all
   "java.util.StringTokenizer" :all
   "java.util.TreeMap" :all
   "java.util.TreeSet" :all
   "java.util.UUID" :all
   "java.util.Vector" :all
   "java.util.WeakHashMap" :all})
```

Figure 6.34.: The list of safe methods in our implementation. Safe methods of each class are listed after the class name, with `:all` representing that “all methods of the class are considered safe.” File: `src/woa/apk/dex/soot/simulator.clj`.

Safe methods are a key part of our solution to the value problem.

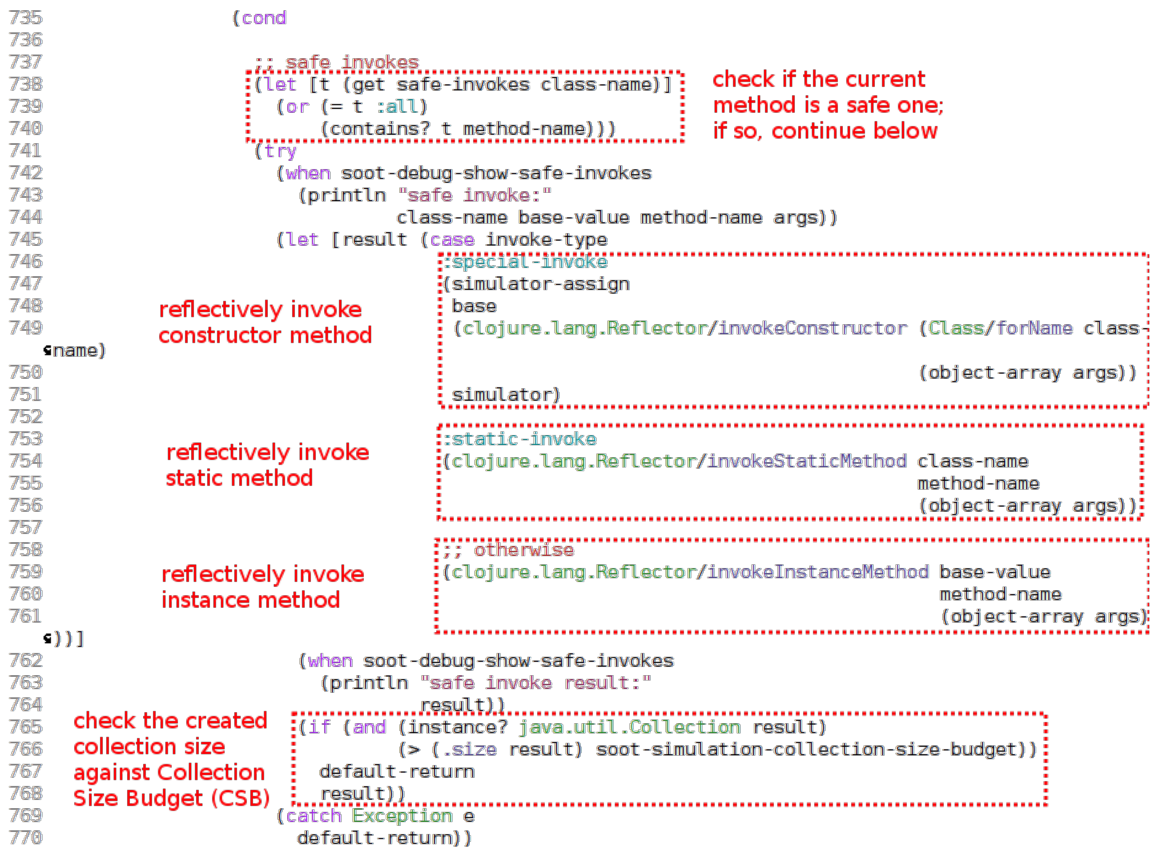
A safe method is a Java method that is shared by Android and JVM, and is either informational or has no other side effects beside creating and manipulating data storage. Examples of safe methods are the Java methods of classes `java.lang.String` and `java.util.ArrayList`. Safe methods are extensively used in our running example `Jk7H.PwcD`, for example, on line 1425 and 1429 in Figure 6.5.

The list of safe methods in our implementation is shown in Figure 6.34. The defining characteristics of safe methods are:

- They are available both in Android API and in Java API;
- They are either informational (e.g., `java.lang.System.nanoTime`) or only create/manipulate data storage.

The key idea is that effects of invoking safe methods in APK (i.e., obtaining information or creating data structures to store values) can be simulated on the JVM that hosts the analysis program by using Java reflection. The advantage of this technique is achieving high simulation fidelity without the burden of re-implementing/maintaining these methods. Note that this technique exploits Android's root in Java and can be used only if the analysis platform is implemented on JVM. For instance, since the Androguard APK analysis toolkit [63] (Section 6.3.4.1) is implemented in Python, such techniques cannot be used there.

Figure 6.35 shows the logic of safe method simulation in `simulator-evaludate`. When the current method is a safe method (lines 738–740), the corresponding method on the hosting JVM is reflectively invoked for constructor methods (lines 746–751), static methods (lines 753–756), and instance methods (lines 758–761). Moreover, to prevent an adversary exploiting safe method simulation for evading detection by maliciously allocating a large collection (for example, a 1 GBytes `java.util.ArrayList`) and hence exhausting resources on the analysis platform, the size of the created collection is checked against the pre-specified Collection Size Budget (CSB) on lines 765–768.



```
(cond

  ;; safe invokes
  (let [t (get safe-invokes class-name)]
    (or (= t :all)
        (contains? t method-name)))
  (try
    (when soot-debug-show-safe-invokes
      (println "safe invoke:"
               class-name base-value method-name args))
    (let [result (case invoke-type
                   :special-invoke
                   (simulator-assign
                    base
                    (clojure.lang.Reflector/invokeConstructor (Class/forName class-name)
                                                              (object-array args))
                    simulator)

                   :static-invoke
                   (clojure.lang.Reflector/invokeStaticMethod class-name
                                                              method-name
                                                              (object-array args))

                   ;; otherwise
                   (clojure.lang.Reflector/invokeInstanceMethod base-value
                                                                method-name
                                                                (object-array args)))]
      (when soot-debug-show-safe-invokes
        (println "safe invoke result:"
                 result))
      (if (and (instance? java.util.Collection result)
               (> (.size result) soot-simulation-collection-size-budget))
        default-return
        result))
    (catch Exception e
      default-return))
```

check if the current method is a safe one; if so, continue below

reflectively invoke constructor method

reflectively invoke static method

reflectively invoke instance method

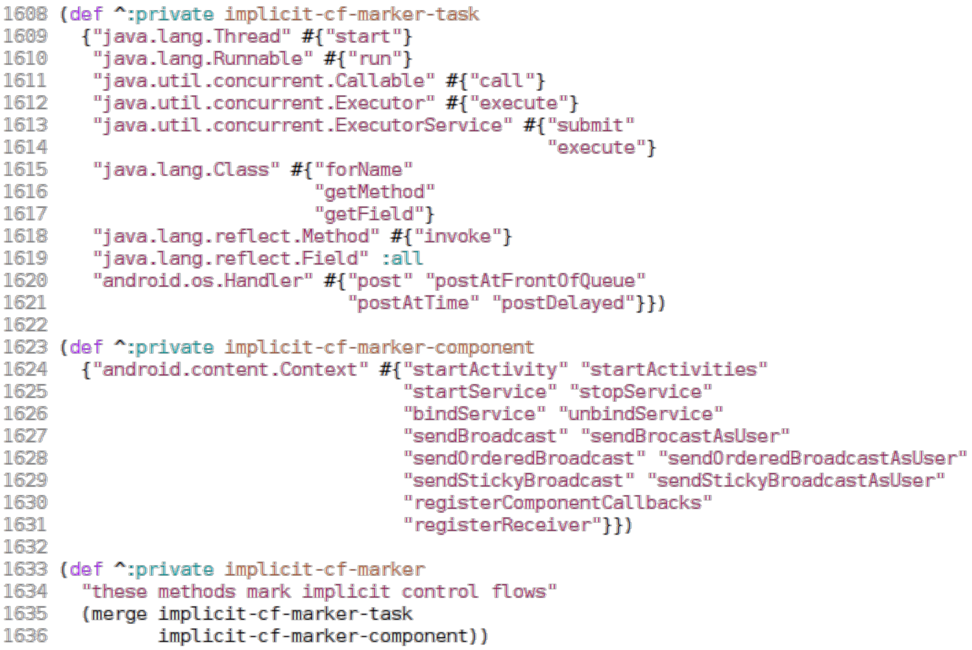
check the created collection size against Collection Size Budget (CSB)

Figure 6.35.: Safe method simulation logic in `simulator-evaludate`. File: `src/woa/apk/dex/soot/simulator.clj`.

In Section 6.6, the value of safe method simulation is demonstrated using real app examples.

Implicit control flow resolution accounts for a significant portion of `simulator-evaluate`'s logic, because there are multiple different cases to handle. Analysis of our running example in Section 6.3.3 illustrates 2 mechanisms that are analyzed in this section: Java reflection and Java/Android multi-thread/asynchronous execution. Figure 6.36 shows most of the implicit control flow mechanisms that are handled in the current implementation of C3GE, which include:

- Java reflection mechanisms: the methods in classes `java.lang.Class`, `java.lang.reflect.Method`, and `java.lang.reflect.Field`.



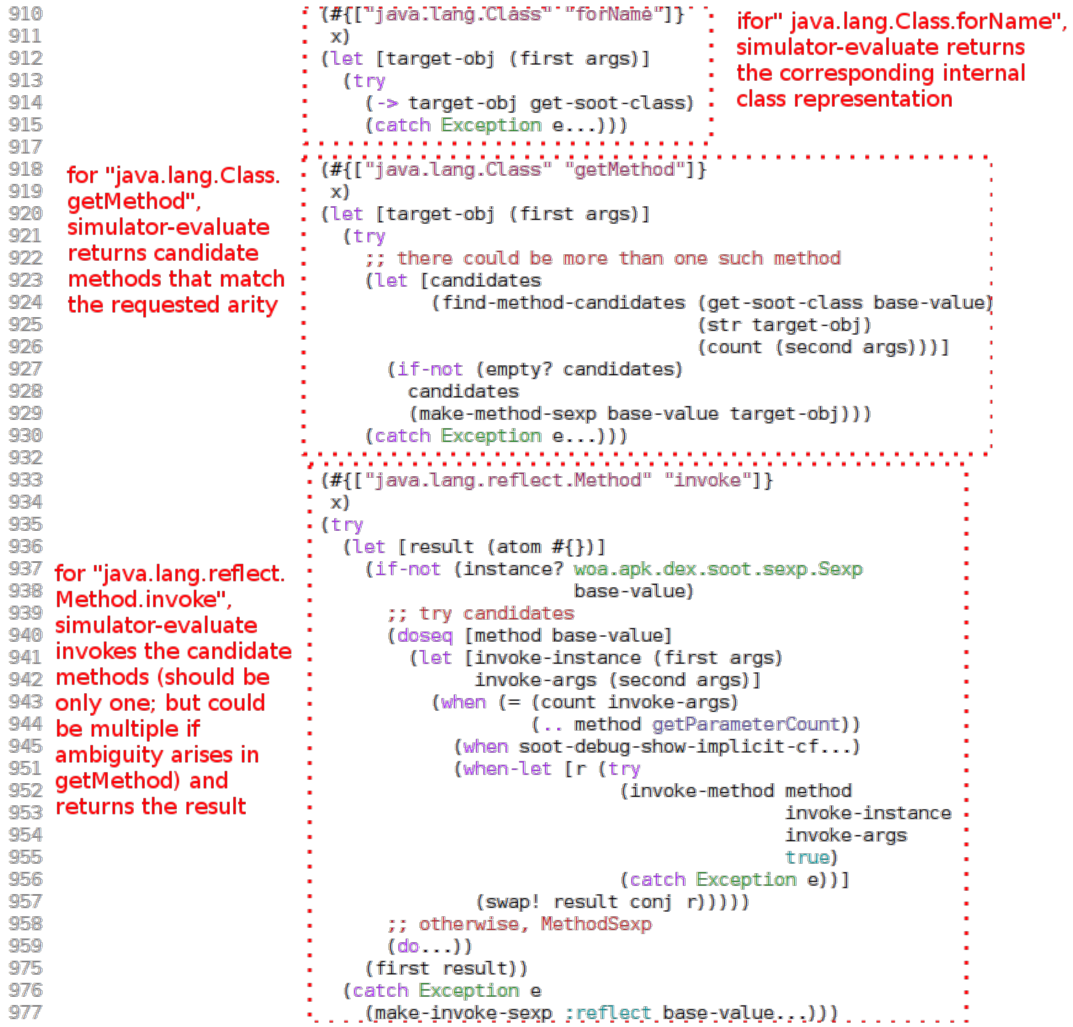
```
(def ^:private implicit-cf-marker-task
  {"java.lang.Thread" #{"start"}
   "java.lang.Runnable" #{"run"}
   "java.util.concurrent.Callable" #{"call"}
   "java.util.concurrent.Executor" #{"execute"}
   "java.util.concurrent.ExecutorService" #{"submit"
                                            "execute"}
   "java.lang.Class" #{"forName"
                       "getMethod"
                       "getField"}
   "java.lang.reflect.Method" #{"invoke"}
   "java.lang.reflect.Field" :all
   "android.os.Handler" #{"post" "postAtFrontOfQueue"
                          "postAtTime" "postDelayed"}})

(def ^:private implicit-cf-marker-component
  {"android.content.Context" #{"startActivity" "startActivities"
                               "startService" "stopService"
                               "bindService" "unbindService"
                               "sendBroadcast" "sendBrocastAsUser"
                               "sendOrderedBroadcast" "sendOrderedBroadcastAsUser"
                               "sendStickyBroadcast" "sendStickyBroadcastAsUser"
                               "registerComponentCallbacks"
                               "registerReceiver"}})

(def ^:private implicit-cf-marker
  "these methods mark implicit control flows"
  (merge implicit-cf-marker-task
         implicit-cf-marker-component))
```

Figure 6.36.: Implicit control flow mechanisms handled in current implementation of C3GE. File `src/woa/apk/dex/soot/simulator.clj`.

- Java/Android multi-thread/asynchronous execution mechanisms: the methods in classes/packages `java.lang.Thread`, `java.lang.Runnable`, `java.util.concurrent.*`, and `android.os.Handler`.

A third category, Android app component activation mechanism (e.g., `android.content.Context.startActivity`), currently does not receive special handling in C3GE, because the WoF model does not require such inter-component information. But if the need to handle inter-component activation arises in the future, the techniques presented below can be extended to include this case.

Figure 6.37 shows the Java reflection handling logic in `simulator-evaluate`. The essence of the logic is the following chain of class/method resolution: 1. When the invoked method is `java.lang.Class.forName` (lines 910–911), `simulator-evaluate` returns a reference to the Java class that has the corresponding name (lines 912–916). 2. When the invoked method is `java.lang.Class.getMethod` (lines 918–919), based on the class found by `java.lang.Class.forName`, `simulator-evaluate` returns a list



```
910 (#{["java.lang.Class" "forName"]}
911  x)
912 (let [target-obj (first args)]
913   (try
914     (-> target-obj get-soot-class)
915     (catch Exception e...)))
917
918 (#{["java.lang.Class" "getMethod"]}
919  x)
920 (let [target-obj (first args)]
921   (try
922     ;; there could be more than one such method
923     (let [candidates
924           (find-method-candidates (get-soot-class base-value)
925                                   (str target-obj)
926                                   (count (second args)))]
927       (if-not (empty? candidates)
928         candidates
929         (make-method-sexp base-value target-obj)))
930     (catch Exception e...)))
932
933 (#{["java.lang.reflect.Method" "invoke"]}
934  x)
935 (try
936   (let [result (atom #{})]
937     (if-not (instance? woa.apk.dex.soot.sexp.Sexp
938                        base-value)
939       ;; try candidates
940       (doseq [method base-value]
941         (let [invoke-instance (first args)
942               invoke-args (second args)]
943           (when (= (count invoke-args)
944                    (.. method getParameterCount))
945             (when soot-debug-show-implicit-cf...)
946             (when-let [r (try
952                            (invoke-method method
953                                           invoke-instance
954                                           invoke-args
955                                           true)
956                            (catch Exception e))]
957               (swap! result conj r)))))
958       ;; otherwise, MethodSexp
959       (do...))
975     (first result))
976   (catch Exception e
977     (make-invoke-sexp :reflect base-value...)))
```

ifor" java.lang.Class.forName", simulator-evaluate returns the corresponding internal class representation
for "java.lang.Class. getMethod", simulator-evaluate returns candidate methods that match the requested arity
for "java.lang.reflect. Method.invoke", simulator-evaluate invokes the candidate methods (should be only one; but could be multiple if ambiguity arises in getMethod) and returns the result

Figure 6.37.: Java reflection handling logic in `simulator-evaluate`. File: `src/woa/-apk/dex/soot/simulator.clj`.

of candidate methods that match the method name (lines 920–931) and requested arity (the number of arguments). Ideally, there should be one such candidate; if ambiguity arises, more than one candidates can be returned as a conservative approximation. 3. When the invoked method is `java.lang.reflect.Method.invoke` (lines 933–934), based on the method candidates found by `java.lang.Class.getMethod`, `simulator-evaluate` invokes the corresponding method candidates and returns the results (lines 935–977).

Following this logic, the resolution of reflection call to `Jk7H.PwcD.SLYfoMdG.load` in `Jk7H.PwcD.SLYfoMdG.onCreate` (Figure 6.5) can be reenacted as follows.

- On line 1385, `java.lang.Class.forName` is invoked with the String `Jk7H.PwcD.SLYfoMdG` as argument. A reference to the corresponding class is returned by `simulator-evaluate` and stored in simulator local `$r2` by `simulate-basic-block`.

- On line 1389, `java.lang.Class.getMethod` is invoked on `$r2` that requests a 0-arity with the name "load." The unique candidate method `Jk7H.PwcD.SLYfoMdG.load` is thus returned by `simulator-evaluate` and stored in simulator local `$r4` by `simulate-basic-block`.

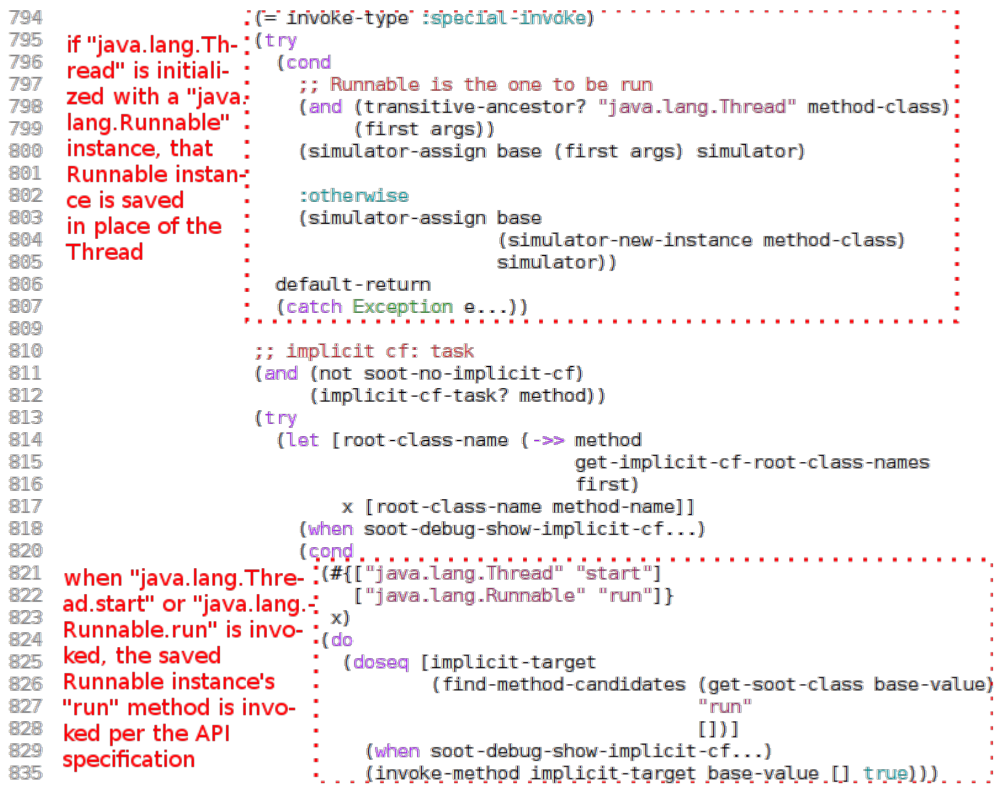
- On line 1393, `java.lang.reflect.Method.invoke` is invoked on the `this` isntance with an empty array. The unique candidate method `Jk7H.PwcD.SLYfoMdG.load` is thus invoked by `simulator-evaluate` and the invocation result is returned to `simulate-basic-block`.

Similarly, Figure 6.38 shows the Java thread handling logic in `simulator-evaluate`. Following the logic, the implicit control flow from the 0-arity `Jk7H.PwcD.SLYfoMdG.send` to `Jk7H.PwcD.SLYfoMdG$1.run` (Figure 6.9) can be reenacted as follows.

- On line 1544, the `java.lang.Thread` instance in `$r1` is initialized with an instance of `Jk7H.PwcD.SLYfoMdG$1`. `simulator-evaluate` stores a reference to the `Jk7H.PwcD.SLYfoMdG$1` instance in `$r1`.

- On line 1546, the `start` method is invoked on the `java.lang.Thread` instance in `$r1`. `simulator-evaluate` invokes the `run` method on the referred `Jk7H.PwcD.SLYfoMdG$1` instance, and hence resolves the implcit contorl flow.

Besides implicit control flow mechanisms discussed above, another complication for Android app analysis is the handling of UI widget's callback methods. In Sec-

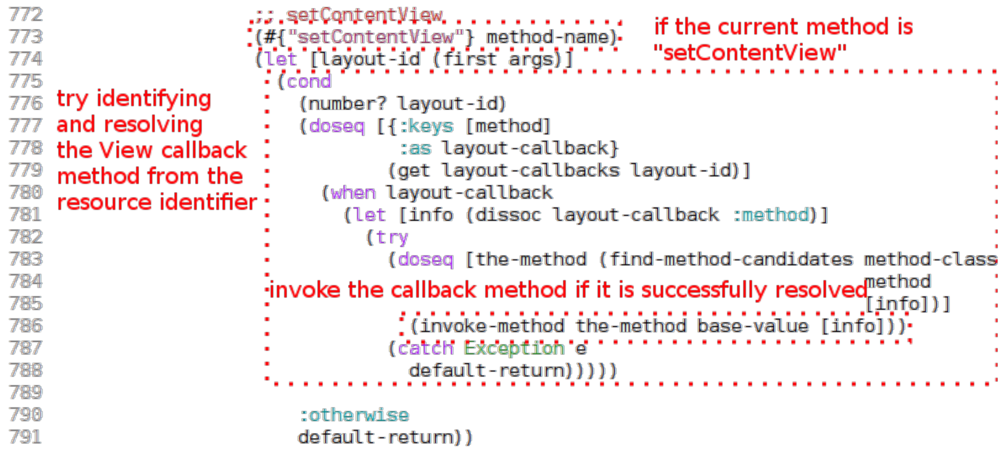


```
(= invoke-type :special-invoke)
(try
  (cond
    ;; Runnable is the one to be run
    (and (transitive-ancestor? "java.lang.Thread" method-class)
         (first args))
    (simulator-assign base (first args) simulator)

    :otherwise
    (simulator-assign base
                      (simulator-new-instance method-class)
                      simulator))
  default-return
  (catch Exception e...))

;; implicit cf: task
(and (not soot-no-implicit-cf)
     (implicit-cf-task? method))
(try
  (let [root-class-name (->> method
                             get-implicit-cf-root-class-names
                             first)
        x [root-class-name method-name]]
    (when soot-debug-show-implicit-cf...)
    (cond
     (#{["java.lang.Thread" "start"]
        ["java.lang.Runnable" "run"]}
      x)
     (do
       (doseq [implicit-target
               (find-method-candidates (get-soot-class base-value)
                                       "run"
                                       [])]
         (when soot-debug-show-implicit-cf...)
         (invoke-method implicit-target base-value [] true)))
```

if "java.lang.Thread" is initialized with a "java.lang.Runnable" instance, that Runnable instance is saved in place of the Thread

when "java.lang.Thread.start" or "java.lang.Runnable.run" is invoked, the saved Runnable instance's "run" method is invoked per the API specification

Figure 6.38.: Java thread handling logic in `simulator-evaluate`. File: `src/woa/-apk/dex/soot/simulator.clj`.



```
;; setContentView
(#{"setContentView"} method-name)
(let [layout-id (first args)]
  (cond
    (number? layout-id)
    (doseq [{:keys [method]
             :as layout-callback}
            (get layout-callbacks layout-id)]
      (when layout-callback
        (let [info (dissoc layout-callback :method)]
          (try
            (doseq [the-method (find-method-candidates method-class
                                                       method
                                                       [info])]
              (invoke-method the-method base-value [info]))
            (catch Exception e
              default-return)))))

    :otherwise
    default-return))
```

if the current method is "setContentView"

try identifying and resolving the View callback method from the resource identifier

invoke the callback method if it is successfully resolved

Figure 6.39.: UI widget's callback method handling logic in `simulator-evaluate`. File: `src/woa/apk/dex/soot/simulator.clj`.

tion 6.3.3.3 and Figure 6.7, we discuss how `Jk7H.PwcD` set up two button-widget callback methods that will be invoked only when the user clicks on the buttons. Since

Android app are UI-driven, the need for proper handling of such callback methods is common, rather than just a corner case.

Figure 6.39 shows how C3GE handle widget callback methods. When the method `setContentView` of class `android.app.Activity` (or classes that descend from it, which is the more common case, as in `Jk7H.PwcD.SLYfoMdG`) is invoked (line 773), the sole integer argument is resolved as a Android layout resource identifier (extracted from XML resource files in the `res` directory of the APK binary), and the corresponding callback methods are resolved. If the resolution is successful (line 763), the resolved method is invoked (line 766).

Through this procedure, widget callback methods are connected to the root method (the component callback method) in C3G, as previously shown for `Jk7H.PwcD.SLYfoMdG` in Figure 6.14: the button `onClick` callback method `mainButtonClick1` is connected to the root method `onCreate` through the method `setMain` that invokes `setContentView`.

6.5.4 A Review of C3GE's Design

With the detailed examination in the previous section (Section 6.5.3), we are in a position to reexamine the design of C3GE illustrated in Figure 6.25. In particular, we will answer the following questions in this section.

- Why call the core of C3GE a simulator?
- Why is the simulator "budget-limited" and what are the budgets for?
- Why is the simulator "width-first" and what problem does it address?
- What are the alternative simulation strategies and their trade-offs?
- What are safe methods? Why are they not used before in, e.g., Androguard?
- What adversary scenarios are considered and how are they handled?

6.5.4.1 Why call the core of C3GE a simulator?

As pointed out in Section 6.5.2, the core of C3GE consists of the procedures `simulate-method`, `simulate-basic-block`, `simulator-evaluate`. These procedures work by *simulating* the statements extracted from bytecode, as if the statements are actually executed. Thus, "simulator" is an apt term for the process.

However, it must be pointed out that the purpose of C3GE is *not* to provide a full Android emulation environment, but to simulate as much of app bytecode semantics as needed to extract the C3G. The considerations are:

- Android emulation is a solved problem due to the existence of, for example, Android's QEMU-based emulator [13] and Genymotion [89]. The challenge of using them to extract C3G is that of efficiency and coverage, which our C3GE aims to address.
- The approach taken by our C3GE does not require setting up virtualized environment and making the app runnable in it. Instead, any app can be analyzed because the analysis is based on bytecode. Essentially, the arguments for "static over dynamic analysis" [17, 81] apply here.
- By simulating safe method invocations and evaluating values, C3GE addresses much of the challenges (such as identifying implicit control flows) for call graph extraction without resorting to full emulation using virtualized environments.

6.5.4.2 Why is the simulator "budget-limited" and what are the budgets for?

Simulation in C3GE is bound by 3 types of budgets: basic block budget (BBB), (method) invocation depth budget (IDB), and constant size budget (CSB). An operation is canceled if there is no budget for it:

- For BBB, cancellation means `simulate-method` stops processing worklist items and returns to its caller.
- For IDB, cancellation means `simulator-evaluate` stops evaluating method invocation at deeper invocation depth and returns to its caller.
- For CSB, cancellation means `simulator-evaluate` stop allocating the Java array/collection object (from either safe method or primitive operation evaluation) in the heap space of the JVM that `simulator-evaluate` is running on.

These budgets are CG3E's precautions against adversary scenarios.

- BBB prevents the C3GE simulator from getting trapped in an infinite intra-procedural loop (e.g., `goto` loop) that is never executed in regular app usage, but will trap an analysis process that is not designed to handle it.
- IDB is the inter-procedural counterpart of BBB, which prevents the C3GE simulator from getting trapped in an infinite invocation loop.
- CSB prevents simulator's resources (e.g. JVM heap space) from being exhausted by malicious storage allocation (e.g, a 1 GBytes array) planted by the adversary in the app's bytecode that is never executed in regular app usage.

It should be mentioned that an alternative to IDB for preventing infinite loop is to detect loops in invocation paths and stop as soon as such a loop is detected (invocation loop detection/ILD). C3GE chooses IDB over ILD for the following reasons:

- IDB allows simulating recursive invocation while ILD does not.
- IDB is more efficient in computation/storage than ILD: While IDB only needs to store/compare a single number (the current invocation depth), ILD needs the full invocation path to decide if there is a loop.

6.5.4.3 Why is the simulator "width-first" and what problem does it address?

The simulator is width-first because, by the worklist processing logic of `simulate-method` (Figures 6.28 and 6.29), *sibling* worklist items (i.e., branching from the same `if` statement) are processed before descendant ones.

The reason to make the simulator width-first is to prevent an adversary from exploiting BBB to evade detection. If the simulator is depth-first, an adversary can deliberately set up a junk loop in a basic block branch, so that the BBB for the current invocation to `simulate-method` will be wasted on this loop without exploring the other branch. A width-first simulator handles this adversary scenario by exploring the other branch *before* the next iteration of the junk loop.

6.5.4.4 What are the alternative simulation strategies and their trade-offs?

As discussed above in Sections 6.5.2 and 6.5.3, the alternative simulation strategies that can be specified at program startup are: conservative branching, aggressive branching, and linear scan, with aggressive branching being the default strategy.

- Under the "aggressive branching" and "conservative branching" strategies, the behavior is to "process the next basic block" as described above. The different between the two is that, if the conditional can be evaluated to a boolean value (true or false) in an `if` statement, the conservative strategy will only follow the true branch, while the aggressive strategy will follow both branches regardless of the result of conditional evaluation.

- Under the "linear scan" strategy, the behavior is to sequentially process *all* statements in the method that follow the beginning statement, rather than only the next basic block.

These simulation strategies provide different trade-offs between simulation efficient, accuracy, and coverage.

- Conservative branching emphasizes accuracy, in which branches whose conditional evaluates to false at analysis time are not followed. However, conservative branching may miss some branches that could be reached at runtime. An example is the `if` statement on line 50 in Figure 6.10: That conditional always evaluates to false at analysis time because the `numbers` field is empty; but an asynchronous thread may add elements to `numbers` at *run time* to make the conditional evaluates to true.

- Linear scan emphasizes coverage and efficiency, but is the least accurate strategy among the three. It has good coverage and is efficient because every statement in a method is processed *once* and *once only*: No missing statement, and no repeatedly processing of the same statement. It is least accurate because it does *not* respect intra-procedural control flow.

- Aggressive branching provides a middle ground between conservative branching and linear scan. While it respects intra-procedural control flow by following `goto` and `if` branching targets (unlike linear scan), it covers both branches in a width-first manner (unlike conservative branching) so that no branches are left unexplored.

Based on these considerations, aggressive branching is selected as the default simulation strategy in C3GE.

In Section 6.6, we will investigate this further with real Android app samples.

6.5.4.5 What are safe methods? Why are they not used in, e.g., Androguard?

A safe method is a Java method that is shared by Android and JVM, and is either informational or has no other side effects beside creating and manipulating data storage. The list of safe methods in our implementation is shown in Figure 6.34. They extend the applicability of value-based static analysis (such as context analysis and Java reflection resolution) from Java primitives (e.g., primitive `int` and `double`) to a much broader set of commonly used objects (e.g., `java.lang.String` or `java.util.ArrayList`). Thus, for example, C3GE can resolve reflection calls in which the method/class names are dynamically constructed from `java.lang.StringBuilder`.

The "safe method" technique exploits the fact that the Android platform and JVM share the same semantics for the safe methods, and their behaviors in Android app can be mirrored to the analysis platform on JVM. Therefore, analysis platforms that are not hosted on JVM (such as Androguard, being a Python-based tool) cannot apply the technique, and hence provides limited support for dynamic-value-based analysis.

6.5.4.6 What adversary scenarios are considered and how are they dealt with?

Based on the discussion above, it can be seen that the following adversary scenarios are handled in C3GE.

- Hiding app logic through Java reflection or Android/Java asynchronous execution mechanisms.
  - This is dealt with in `simulator-evaluate` (Section 6.5.3.5).
- Set up infinite loops to trap analysis.
  - The intra-procedural case of infinite loops is dealt with by limiting the number of basic blocks that are processed in one invocation of `simulate-method`, i.e., the Basic Block Budget (BBB).
  - The inter-procedural case of infinite loops is dealt with by limiting the nesting depth of `simulate-method`, i.e., the Invocation Depth Budget (IDB)
- Allocate large arrays/collections to overflow analysis program's evaluation logic.

  - This is dealt with by limiting the size of collections that the analysis program will allocate, i.e., the Collection Size Budget (CSB).

6.6 Evaluation

This section presents analysis and evaluation of WoA's utility in analyzing APKs using a million-node-scale prototype WoA generated from public APK datasets.

6.6.1 The Prototype WoA

6.6.1.1 Data Source

We build a prototype WoA using the APK samples from the following public data sets: Android Malware Genome Project (AGMP) [236] (1.6 GBytes) and Drebin [16] (6.9 GBytes). The two datasets are overlapped and, after merging the datasets together and filtering out ill-formed APKs, 5,485 APK samples, including the running example analyzed in Section 6.3 and its variants, are included in the prototype WoA. Anti-virus (AV) software scanning results of 55 AV solution vendors of these APK sample (updated as of 22 January 2015) are obtained from VirusTotal [207] and added as `Tag/VirusTotal` nodes to the WoA.

6.6.1.2 Generation Parameter and Procedure

The prototype WoA is generated with the following parameters (these parameters were discussed in Section 6.5): 1. Aggressive branching strategy. 2. Basic Block Budget (BBB): 20. 3. Invocation Depth Budget (IDB): 8. 4. Collection Size Budget (CSB): 10,000.

Suppose the APK binaries are stored in the `dataset` directory of the filesystem, the results can be reproduced by the following Bash [161] command (see Section 6.3.2 for instructions of downloading and setting up the `woa` program):

```
find dataset -type f -name ’*.apk’ | woa --prep-tags ’’ | \
  JVM_HEAP=5g \
  woa -sv --dump-model /dev/null -j5 \
    --soot-basic-block-simulation-budget 20 \
    --soot-method-simulation-depth-budget 8 \
    --soot-simulation-collection-size-budget 10000
```

After the completion of this command, the extracted information of these APKs will be stored in `.model-dump` files, with the SHA-256 checksum of the APK binaries being the file name before the `.model-dump` suffix. Then, a WoA (backing by a Neo4j graph database) can be generated by the following Bash commands:

```
JVM_HEAP=5g woa -vL -l <(ls *.model-dump) -D prototype

( TARGET="$HOME/bin/";
  mkdir -p ${TARGET};
  wget -nc -nd -P ${TARGET} \
    https://raw.githubusercontent.com/pw4ever/web-of-apks/gh-pages/bin/neo4j-batch-import \
  && \
  chmod +x ${TARGET}/neo4j-batch-import )

"$HOME/bin/neo4j-batch-import" graph.db prototype.nodes prototype.rels
```

After the completion of these commands, the `graph.db` directory will contain the generated graph database that can be imported into Neo4j for query.

6.6.1.3 Basic Information

Some basic information about the prototype WoA is listed below. 1. Overall: 3,620,120 nodes and 9,380,346 edges. 2. Fiber skeletons: 954 `SigningKey` nodes, 5,485 `Apk` nodes, 3,506 `Dex` nodes, 13,178 `Component` nodes, 116,840 `Callback` nodes, 230,392 `Method` nodes invoked by the `Callback` nodes (of which 220,336 are explicit invocations and 17,103 are implicit invocations), 475,678 `CallbackSignature` nodes, 7,633 `Package` nodes, 2,653,260 `CallGraphNode` nodes. 3. Peripherals: 295 `Permission` nodes, 92 `IntentFilterCategory` nodes, 1,241 `IntentFilterAction` nodes, 49 `MalwareFamily/Tag` nodes (based on AGMP labels), 30,941 `VirusTotal/Tag` nodes (obtained from VirusTotal).

6.6.2 Support for Declarative APK Analysis

In this section, we evaluate WoA's utility in supporting APK analysis through a series of explorations on the prototype WoA, each starting from a different angle.

6.6.2.1 APKs Connected by Dex Reusing

Different APK binaries can contain the same executable code (Dex). Therefore, we can begin our inquiry by finding APKs that contain the same Dex.

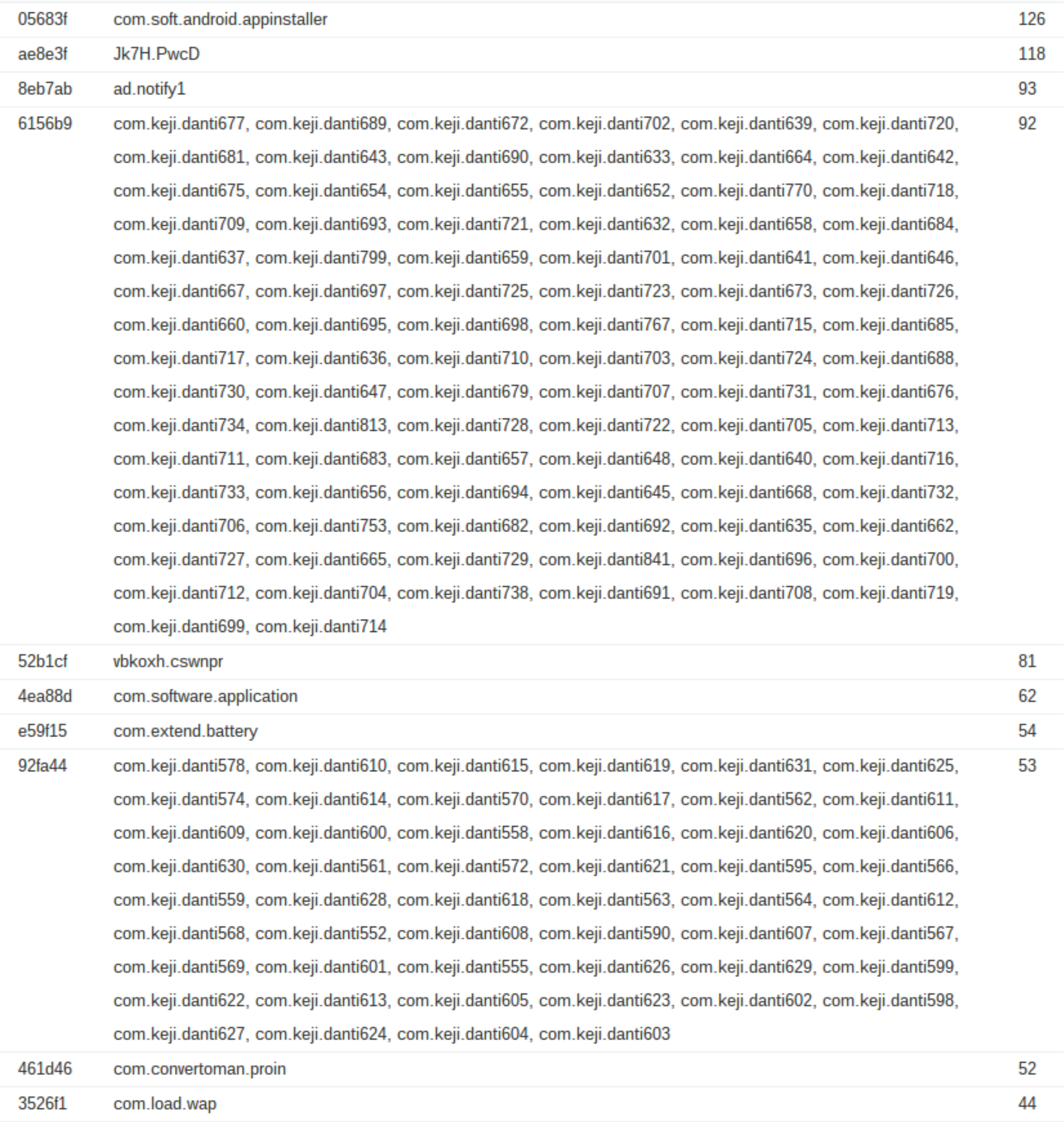
Figure 6.40 shows the result of the following WoA query:

```
MATCH (d:Dex) <-- (a:Apk)
WITH substring(d.sha256, 0, 6) as d, collect(DISTINCT a.package) as pa,
  count(DISTINCT a) AS ca
RETURN d, pa, ca
ORDER BY ca DESC LIMIT 10
```

which lists the top 10 repackaged DEX binaries (identified by the first 6 hexadecimal digits of their SHA-256 checksum), the package names of the APKs that contain them, and the number of these APKs.

The result in Figure 6.40 shows that:

- The mostly reused Dex has the SHA-256 identifier `05683f` and the package name `com.soft.android.appinstaller`



05683f	com.soft.android.appinstaller	126
ae8e3f	Jk7H.PwcD	118
8eb7ab	ad.notify1	93
6156b9	com.keji.danti677, com.keji.danti689, com.keji.danti672, com.keji.danti702, com.keji.danti639, com.keji.danti720, com.keji.danti681, com.keji.danti643, com.keji.danti690, com.keji.danti633, com.keji.danti664, com.keji.danti642, com.keji.danti675, com.keji.danti654, com.keji.danti655, com.keji.danti652, com.keji.danti770, com.keji.danti718, com.keji.danti709, com.keji.danti693, com.keji.danti721, com.keji.danti632, com.keji.danti658, com.keji.danti684, com.keji.danti637, com.keji.danti799, com.keji.danti659, com.keji.danti701, com.keji.danti641, com.keji.danti646, com.keji.danti667, com.keji.danti697, com.keji.danti725, com.keji.danti723, com.keji.danti673, com.keji.danti726, com.keji.danti660, com.keji.danti695, com.keji.danti698, com.keji.danti767, com.keji.danti715, com.keji.danti685, com.keji.danti717, com.keji.danti636, com.keji.danti710, com.keji.danti703, com.keji.danti724, com.keji.danti688, com.keji.danti730, com.keji.danti647, com.keji.danti679, com.keji.danti707, com.keji.danti731, com.keji.danti676, com.keji.danti734, com.keji.danti813, com.keji.danti728, com.keji.danti722, com.keji.danti705, com.keji.danti713, com.keji.danti711, com.keji.danti683, com.keji.danti657, com.keji.danti648, com.keji.danti640, com.keji.danti716, com.keji.danti733, com.keji.danti656, com.keji.danti694, com.keji.danti645, com.keji.danti668, com.keji.danti732, com.keji.danti706, com.keji.danti753, com.keji.danti682, com.keji.danti692, com.keji.danti635, com.keji.danti662, com.keji.danti727, com.keji.danti665, com.keji.danti729, com.keji.danti841, com.keji.danti696, com.keji.danti700, com.keji.danti712, com.keji.danti704, com.keji.danti738, com.keji.danti691, com.keji.danti708, com.keji.danti719, com.keji.danti699, com.keji.danti714	92
52b1cf	vbkoxh.cswnpr	81
4ea88d	com.software.application	62
e59f15	com.extend.battery	54
92fa44	com.keji.danti578, com.keji.danti610, com.keji.danti615, com.keji.danti619, com.keji.danti631, com.keji.danti625, com.keji.danti574, com.keji.danti614, com.keji.danti570, com.keji.danti617, com.keji.danti562, com.keji.danti611, com.keji.danti609, com.keji.danti600, com.keji.danti558, com.keji.danti616, com.keji.danti620, com.keji.danti606, com.keji.danti630, com.keji.danti561, com.keji.danti572, com.keji.danti621, com.keji.danti595, com.keji.danti566, com.keji.danti559, com.keji.danti628, com.keji.danti618, com.keji.danti563, com.keji.danti564, com.keji.danti612, com.keji.danti568, com.keji.danti552, com.keji.danti608, com.keji.danti590, com.keji.danti607, com.keji.danti567, com.keji.danti569, com.keji.danti601, com.keji.danti555, com.keji.danti626, com.keji.danti629, com.keji.danti599, com.keji.danti622, com.keji.danti613, com.keji.danti605, com.keji.danti623, com.keji.danti602, com.keji.danti598, com.keji.danti627, com.keji.danti624, com.keji.danti604, com.keji.danti603	53
461d46	com.convertoman.proin	52
3526f1	com.load.wap	44

Figure 6.40.: The top 10 repackaged DEX binaries (identified by the first 6 hexadecimal digits of their SHA-256 checksum), the package names of the APKs that contain them, and the number of these APKs in the prototype WoA.

- The Dex with SHA-256 identifier `ae8e3f`, which is contained in the running example `Jk7H.PwcD`, is reused in 118 APK samples in the dataset, all of which have the same package name.

- The Dex with SHA-256 identifiers `6156b9` and `92fa44`, which are reused in 92 and 53 APKs respectively, have similar package names of the form `com.keji.danti*`, with the * part being 3 decimal digits.

To further investigate the APKs that have the package name pattern `com.keji.danti*`, we use the following query,

```
MATCH (a:Apk) WHERE a.package=~"^com.keji.danti.*"
MATCH (a)<--(s:SigningKey)
RETURN DISTINCT s.sha256
```

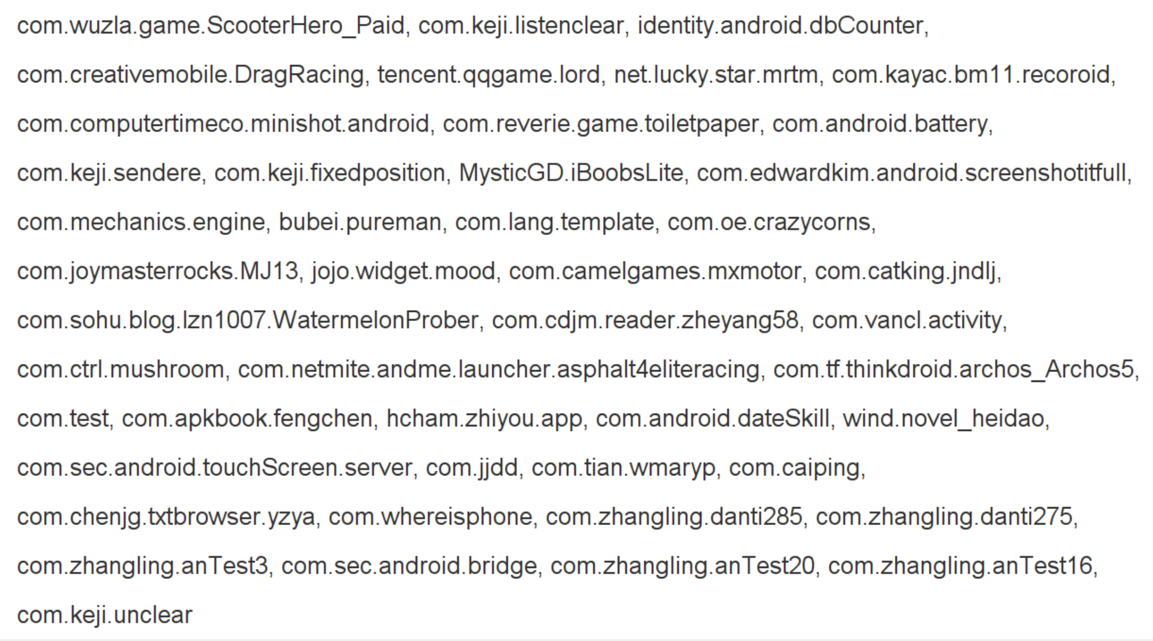
which shows the keys that are used to sign APKs with package name pattern `com.keji.danti.*`, and get the following 4 signing keys.

```
D9:25:1B:BD:D3:53:EA:59:4F:29:65:54:2F:7A:29:BC:0E:0A:19:44:4C:9B:58:F1:90:A9:73:2D:A6:FB:6F:B1
EC:A6:72:F7:07:C5:DD:4E:A1:C6:5F:7B:DA:18:82:B9:3D:94:81:BD:92:34:4A:3A:1D:43:BA:B6:82:DD:E9:DC
BB:99:04:D8:AA:8B:FE:B7:22:FE:A9:F9:39:3C:0D:CD:55:1D:A0:98:C1:BB:43:92:2F:54:1D:84:34:CF:D2:66
DB:2B:DF:A6:C4:1C:B4:B1:1B:E6:7B:71:51:1E:21:85:AC:88:C6:5E:D2:52:5C:4D:77:C0:F7:5B:F5:06:96:1B
```

To see the other APKs that are signed by one of these keys but do *not* have the package name pattern `com.keji.danti.*`, we use the following query:

```
MATCH (a:Apk) WHERE a.package=~"^com.keji.danti.*"
MATCH (a)<--(s:SigningKey)
WITH DISTINCT s
MATCH s-->(a:Apk)
WHERE NOT a.package=~"^com.keji.danti.*"
RETURN collect(DISTINCT a.package)
```

and get the result shown in Figure 6.41. The result indicates that the keys that sign one of `com.keji.danti.*` APKs are used to sign other APKs.



com.wuzla.game.ScooterHero_Paid, com.keji.listenclear, identity.android.dbCounter, com.creativemobile.DragRacing, tencent.qqgame.lord, net.lucky.star.mrtm, com.kayac.bm11.recoroid, com.computertimeco.minishot.android, com.reverie.game.toiletpaper, com.android.battery, com.keji.sendere, com.keji.fixedposition, MysticGD.iBoobsLite, com.edwardkim.android.screenshotitfull, com.mechanics.engine, bubei.pureman, com.lang.template, com.oe.crazycorns, com.joymasterrocks.MJ13, jojo.widget.mood, com.camelgames.mxmotor, com.catking.jndlj, com.sohu.blog.lzn1007.WatermelonProber, com.cdjm.reader.zheyang58, com.vancl.activity, com.ctrl.mushroom, com.netmite.andme.launcher.asphalt4eliteracing, com.tf.thinkdroid.archos_Archos5, com.test, com.apkbook.fengchen, hcham.zhiyou.app, com.android.dateSkill, wind.novel_heidao, com.sec.android.touchScreen.server, com.jjdd, com.tian.wmaryp, com.caiping, com.chenjg.txtbrowser.yzya, com.whereisphone, com.zhangling.danti285, com.zhangling.danti275, com.zhangling.anTest3, com.sec.android.bridge, com.zhangling.anTest20, com.zhangling.anTest16, com.keji.unclear

Figure 6.41.: The package names of the APKs that are signed by the authors who have signed at least one APK with the package name pattern `com.keji.danti.*`.

Although the exploration can continue into those packages, we choose to stop here. The point is that WoA facilitates such exploration by explicitly connecting syntactically similar elements semantic in APKs in a single graph.

6.6.2.2 APKs Connected by Identical Authorship

A single signing key can be used to sign different APKs. The last query in the previous section shows one example. APK analysis can begin by finding APKs connected by authorship.

We begin the inquiry with following WoA query:

```
MATCH (signingKey: SigningKey) --> (apk:Apk)
WITH signingKey, count(apk) as ca
ORDER BY ca DESC
LIMIT 10
MATCH (signingKey) --> (a:Apk)
WITH signingKey, count(DISTINCT a) as cnt_all
```

```
MATCH (signingKey) --> (a:Apk) <-- (m:Malware)
WITH signingKey, cnt_all, a, count(DISTINCT m) as cm
WHERE cm>=5
WITH signingKey, cnt_all, count(DISTINCT a) as cnt_malware
RETURN signingKey.sha256, cnt_all, cnt_malware,
  cnt_all-cnt_malware AS discrepancy
ORDER BY discrepancy DESC, cnt_malware DESC
```

which finds, among the 10 most prolific authors in the dataset: 1. the number of APKs that they have authored, 2. the number of samples among those APKs that are flagged by at least 5 AV vendors, 3. the discrepancies between the two numbers. We get the following result (sorted in descending order by the discrepancies and then the AV flags):

```
D8:24:59:01:9E:8A:82:CA:36:0E:49:A7:0D:B2:7D:E9:A9:4A:1B:81:9F:FB:3B:CE:E7:91:C7:FB:68:60:C9:01,219,215,4
A4:0D:A8:0A:59:D1:70:CA:A9:50:CF:15:C1:8C:45:4D:47:A3:9B:26:98:9D:8B:64:0E:CD:74:5B:A7:1B:F5:DC,365,363,2
24:2C:B4:8C:BF:CB:60:F5:A5:F0:AE:3E:44:35:DE:8F:08:E3:89:28:85:84:F7:66:39:B4:9C:8A:19:0A:06:98,495,494,1
6C:51:D9:0F:CA:DE:D4:68:1F:B9:68:92:22:2C:FE:EC:28:EB:30:ED:3E:DC:D9:E4:10:C6:DC:23:40:66:6E:70,326,325,1
EC:A6:72:F7:07:C5:DD:4E:A1:C6:5F:7B:DA:18:82:B9:3D:94:81:BD:92:34:4A:3A:1D:43:BA:B6:82:DD:E9:DC,170,169,1
50:C7:37:6D:76:72:24:ED:0C:2E:6B:43:14:55:EE:21:29:A7:8C:99:5A:E6:B5:D8:CA:E0:0E:96:48:1A:23:53,187,187,0
FD:FA:6E:8A:CB:6A:98:AC:30:76:BB:95:DC:B1:6C:74:F2:B7:87:B8:BE:36:EA:8C:20:D4:8F:BA:99:88:39:AA,171,171,0
B2:86:63:4C:26:14:28:D2:38:F7:16:0B:10:1B:E2:0C:C6:52:8C:D3:25:C0:4C:DA:11:D4:17:FF:C0:F3:4E:8A,147,147,0
17:67:7A:86:64:60:0A:EE:36:BF:77:51:56:60:34:9A:36:A1:4C:01:A4:36:7A:B4:EE:15:81:A0:50:0A:CC:CD,131,131,0
F0:90:C5:8C:4B:36:79:CB:1D:9A:97:2F:F0:68:C3:77:6D:70:55:05:3D:9F:6D:FA:29:34:B3:DC:22:7F:20:D6,128,128,0
```

in which we observe that only 5 out of the 10 authors have apps not detected by at least 5 AV vendors, and the largest discrepancy is only 4 out of a total of 219 APKs. This indicates that, months after these datasets were released (the AGMP dataset was released in 2012; the Drebin dataset should be available online on February 2014 after the NDSS Symposium '14), most malware samples are integrated into AV vendor's database with a few exceptions.

Alternatively, with the following WoA query:

```
MATCH (signingKey: SigningKey) --> (apk:Apk)
WITH signingKey, count(apk) as ca
ORDER BY ca DESC
LIMIT 50
MATCH (signingKey) --> (a:Apk)
WITH signingKey, count(DISTINCT a) as cnt_all
MATCH (signingKey) --> (a:Apk) <-- (m:Malware)
WITH signingKey, cnt_all, a, count(DISTINCT m) as cm
```

```
WHERE cm>=5
WITH signingKey, cnt_all, count(DISTINCT a) as cnt_malware
WHERE cnt_malware>0
WITH signingKey, cnt_all, cnt_malware,
  cnt_all-cnt_malware AS discrepancy
WHERE discrepancy>0
RETURN signingKey.sha256, cnt_all, cnt_malware, discrepancy
ORDER BY discrepancy DESC, cnt_malware DESC
```

we find, among the 50 most prolific authors in the dataset, the following ones that have discrepancies between “the number of APKs that they have authored” and “the number of samples among those APKs that are flagged by at least 5 AV vendors.”

```
D8:24:59:01:9E:8A:82:CA:36:0E:49:A7:0D:B2:7D:E9:A9:4A:1B:81:9F:FB:3B:CE:E7:91:C7:FB:68:60:C9:01,219,215,4
A4:0D:A8:0A:59:D1:70:CA:A9:50:CF:15:C1:8C:45:4D:47:A3:9B:26:98:9D:8B:64:0E:CD:74:5B:A7:1B:F5:DC,365,363,2
AF:DF:E7:6D:5C:8B:D9:94:F6:92:29:0C:DC:F8:EC:4E:96:EE:BC:05:E0:2C:EF:C0:69:59:D1:86:8E:32:A8:94,71,69,2
24:2C:B4:8C:BF:CB:60:F5:A5:F0:AE:3E:44:35:DE:8F:08:E3:89:28:85:84:F7:66:39:B4:9C:8A:19:0A:06:98,495,494,1
6C:51:D9:0F:CA:DE:D4:68:1F:B9:68:92:22:2C:FE:EC:28:EB:30:ED:3E:DC:D9:E4:10:C6:DC:23:40:66:6E:70,326,325,1
EC:A6:72:F7:07:C5:DD:4E:A1:C6:5F:7B:DA:18:82:B9:3D:94:81:BD:92:34:4A:3A:1D:43:BA:B6:82:DD:E9:DC,170,169,1
9F:56:98:1A:85:ED:E5:20:45:BA:2F:DE:53:82:9D:E3:9C:1D:75:02:BD:74:04:83:C6:CC:2A:7C:06:CA:6A:90,68,67,1
2F:19:FA:07:81:2F:33:42:A9:4C:8E:AF:16:66:B5:24:4A:EE:BF:9A:3D:F9:92:46:73:61:92:2F:21:F4:85:A6,61,60,1
```

We can proceed to investigate the discrepancies, for example, for the first author, whose signing key has a SHA-256 checksum that begins with `D8:24:59`. Using the following query:

```
MATCH (signingKey: SigningKey {sha256:
  "D8:24:59:01:9E:8A:82:CA:36:0E:49:A7:0D:B2:7D:E9:A9:4A:1B:81:9F:FB:3B:CE:E7:91:C7:FB:68:60:C9:01"
})
MATCH signingKey --> (apk:Apk)
WHERE NOT apk <-- (:Malware)
WITH signingKey, apk
MATCH (perm:Permission)<--apk-->(dex:Dex)
RETURN *
```

we find how the 4 non-flagged APKs out of the 219 APKs from author `D8:24:59` relate to each other by permission and Dex. Figure 6.42 shows a visualization of the result, in which we can observe that:

- Among the 4 APKs, two of them share the same package name (`com.gp.tiltmazes`) and the same Dex (SHA-256 identifier starts with `7164be`).

- The 4 APKs share some common permission requests, such as INTERNET and CHANGE_WIFI_STATE.

- Some permissions are only requested by one of the APKs. For example, READ_SMS and WRITE_SMS are only requested by com.ps.pushbox among the 4 APKs.

By their package name and requested permission, the 4 apps are not the variants of the same one. We can proceed to find what common packages they use:

```
MATCH (signingKey: SigningKey {sha256:
  "D8:24:59:01:9E:8A:82:CA:36:0E:49:A7:0D:B2:7D:E9:A9:4A:1B:81:9F:FB:3B:CE:E7:91:C7:FB:68:60:C9:01"
})
MATCH signingKey --> (apk:Apk)
WHERE NOT apk <-- (:Malware)
WITH apk
MATCH apk-->(:Dex)-->(:Component)<--(p:Package)
WHERE NOT p.name=~’com.google.*’
WITH p, count(distinct apk) as ca
WHERE ca>=3
RETURN ca, p.name
ORDER BY ca DESC
LIMIT 10
```

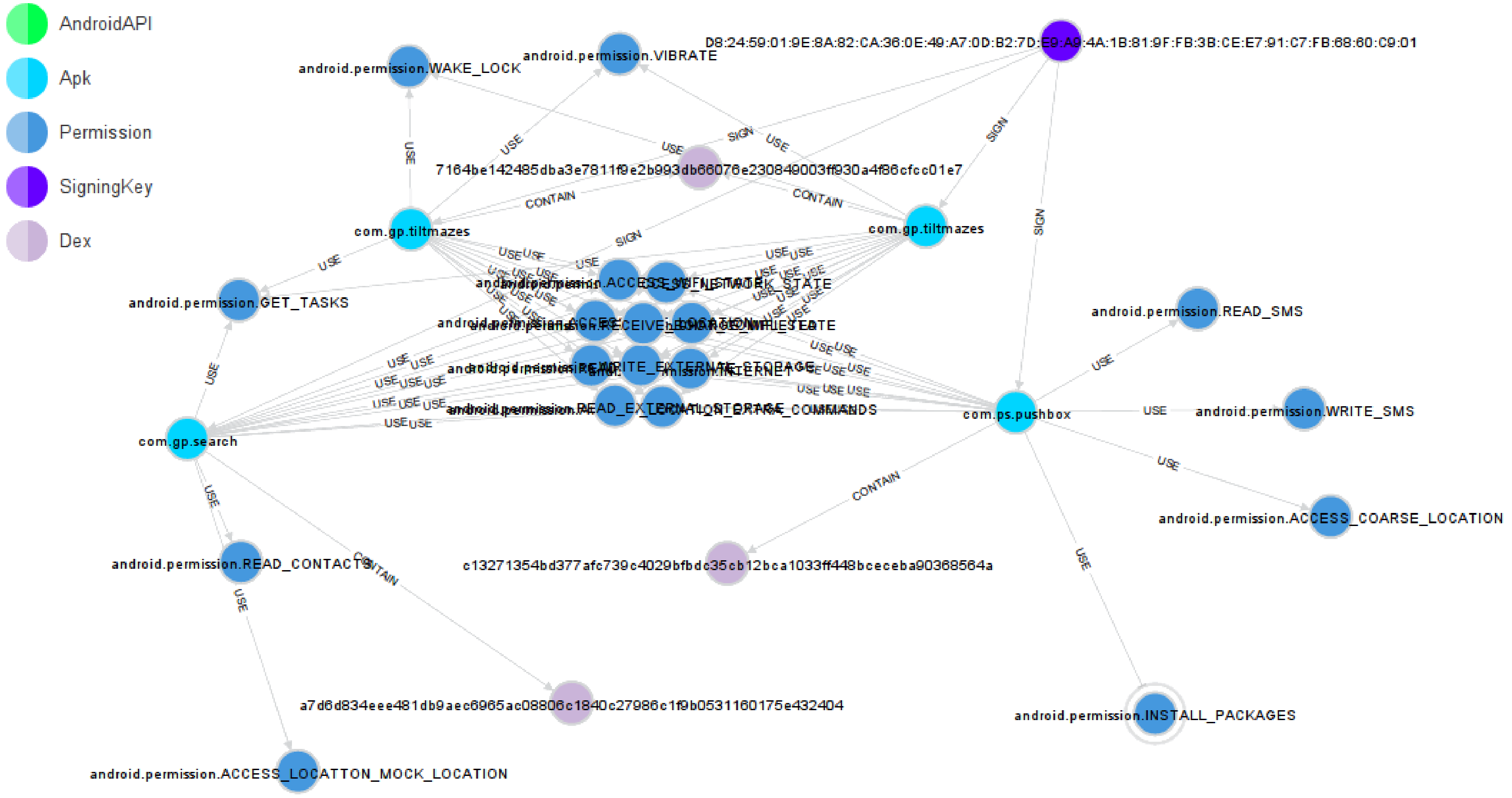
which lists non-Google packages that are used in at least 3 out of the 4 apps. This is the result:

```
4,cn.domob.android.ads
4,com.adwo.adsdk
3,com.waps
```

From the name of these packages, we can see that all 4 apps use two *ad packages* cn.domob.android.ads and com.adwo.adsdk.

We can verify this finding on other apps from the same author:

```
MATCH (signingKey: SigningKey {sha256:
  "D8:24:59:01:9E:8A:82:CA:36:0E:49:A7:0D:B2:7D:E9:A9:4A:1B:81:9F:FB:3B:CE:E7:91:C7:FB:68:60:C9:01"
})
MATCH signingKey --> (apk:Apk)
WITH apk
MATCH apk-->(:Dex)-->(:Component)<--(p:Package)
```



AndroidAPI
Apk
Permission
SigningKey
Dex
D8:24:59:01:9E:8A:82:CA:36:0E:49:A7:0D:B2:7D:E9:A9:4A:1B:81:9F:FB:3B:CE:E7:91:C7:FB:68:60:C9:01
android.permission.WAKE_LOCK
android.permission.VIBRATE
7164be142485dba3e7811f9e2b993db66076e230849003ff930a4f86cfcc01e7
com.gp.tiltmazes
com.gp.tiltmazes
android.permission.GET_TASKS
android.permission.READ_SMS
android.permission.WRITE_SMS
com.ps.pushbox
com.gp.search
android.permission.ACCESS_COARSE_LOCATION
android.permission.READ_CONTACTS
c13271354bd377afc739c4029bfbdc35cb12bca1033ff448bceceba90368564a
a7d6d834eee481db9aec6965ac08806c1840c27986c1f9b0531160175e432404
android.permission.INSTALL_PACKAGES
android.permission.ACCESS_LOCATTON_MOCK_LOCATION
SIGN
USE
CONTAIN

Figure 6.42.: The permission/Dex relationship between the 4 non-flagged APKs from author D8:24:59 (out of 219 APKs from the author).

```
WHERE NOT p.name=~'com.google.*'
WITH p, count(distinct apk) as ca
RETURN ca, p.name
ORDER BY ca DESC
LIMIT 5
```

which does the same thing as above for all apps from the same author. We get the following result:

```
218,com.adwo.adsdk
113,cn.domob.android.ads
110,com.waps
16,com.ps.keepaccount
16,com.ps.keepaccount.activity
```

which shows that the two ad packages, especially `com.adwo.adsdk`, are indeed commonly used by the author.

Back to the 4 non-flagged of the author, with the following WoA query:

```
MATCH (signingKey: SigningKey {sha256:
  "D8:24:59:01:9E:8A:82:CA:36:0E:49:A7:0D:B2:7D:E9:A9:4A:1B:81:9F:FB:3B:CE:E7:91:C7:FB:68:60:C9:01"
})
MATCH signingKey --> (apk:Apk)
WHERE NOT apk <-- (:Malware)
WITH apk
MATCH apk--(d:Dex)--(a:Apk)
OPTIONAL MATCH a<--(m:Malware)
RETURN *
```

which find APKs that share Dex with them and, optionally, malware flags attached to those APKs, we get the result shown in Figure 6.43.

The result indicates that 3 out of the 4 non-flagged apps, 1 `com.ps.pushbox` and 2 `com.gp.tiltmazes`, should actually be flagged—the missed malware labels may be caused by incomplete results when the labels were collected.

The above exploration shows a benefit of using the declarative APK analysis approach supported by WoA: It can complement other approaches (e.g., machine learning based approach) by completing partial training datasetet. Machine-learning-based

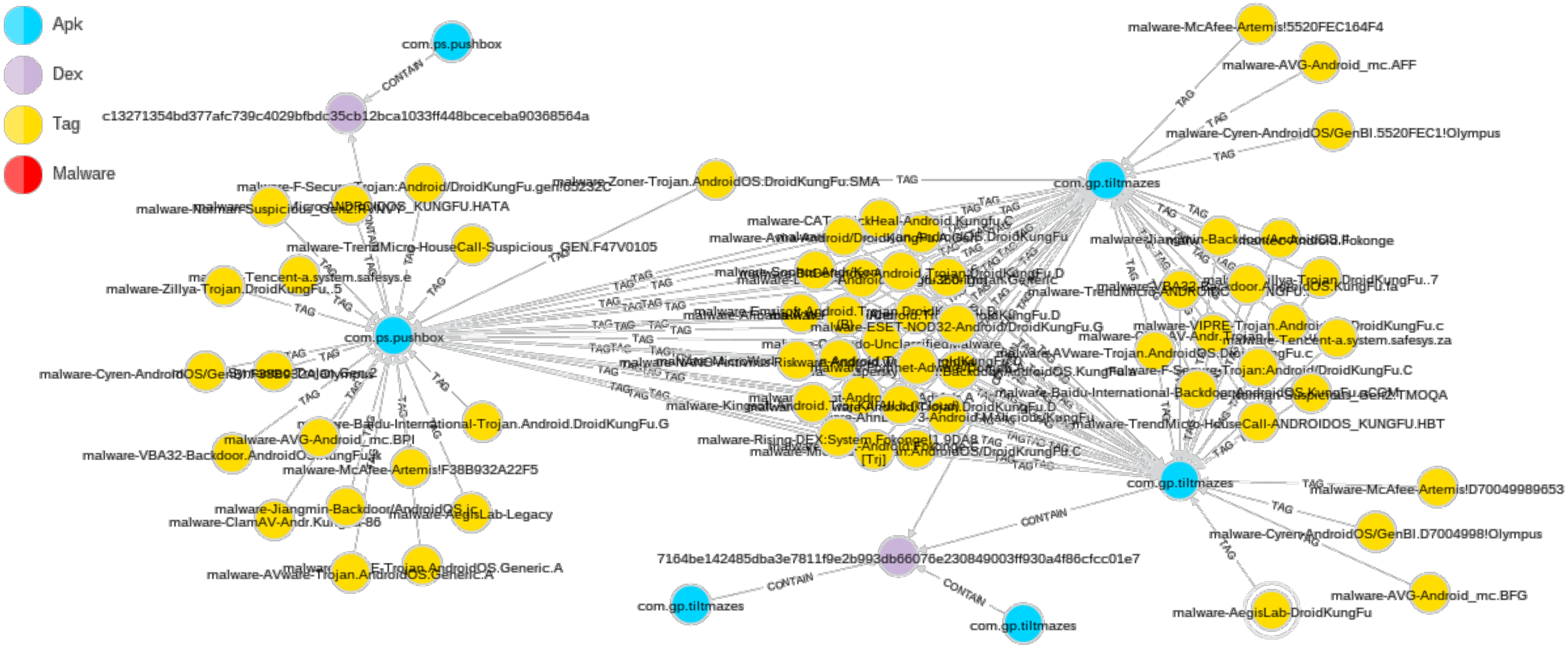

Apk
Dex
Tag
Malware
com.ps.pushbox
CONTAIN
c13271354bd377afc739c4029bfbdc35cb12bca1033ff448bceceba90368564a
com.gp.tiltmazes
7164be142485dba3e7811f9e2b993db66076e230849003ff930a4f86cfcc01e7
malware-McAfee-Artemis!5520FEC164F4
malware-AVG-Android_mc.AFF
malware-Cyren-AndroidOS/GenBl.5520FEC1!Olympus
malware-Zoner-Trojan.AndroidOS.DroidKungFu.SMA
malware-TrendMicro-HouseCall-Suspicious_GEN.F47V0105
malware-Zillya-Trojan.DroidKungFu..5
malware-Tencent-a.system.safesys.e
malware-Baidu-International-Trojan.Android.DroidKungFu.G
malware-AVG-Android_mc.BPI
malware-VBA32-Backdoor.AndroidOS.KungFu.k
malware-McAfee-Artemis!F38B932A22F5
malware-Jiangmin-Backdoor/AndroidOS.jc
malware-AegisLab-Legacy
malware-ClamAV-Andr.KungFu-86
malware-AVware-Trojan.AndroidOS.Generic.A
malware-F-Secure-Trojan.AndroidOS.Generic.A
malware-ESET-NOD32-Android/DroidKungFu.G
malware-VIPRE-Trojan.AndroidOS.DroidKungFu.c
malware-Tencent-a.system.safesys.za
malware-AVware-Trojan.AndroidOS.DroidKungFu.c
malware-F-Secure-Trojan:Android/DroidKungFu.C
malware-Baidu-International-Backdoor.AndroidOS.KungFu.aGCM
malware-TrendMicro-HouseCall-ANDROIDOS_KUNGFU.HBT
malware-Zillya-Trojan.DroidKungFu..7
malware-Rising-DEX:System.Fokonge!1.9DAB
malware-McAfee-Artemis!D70049989653
malware-Cyren-AndroidOS/GenBl.D7004998!Olympus
malware-AVG-Android_mc.BFG
malware-AegisLab-DroidKungFu
TAG

Figure 6.43.: APKs that share Dex with the 4 non-flagged APKs from author D8:24:59 (out of 219 APKs from the author) and their malware flags.

approaches are sensitive to quality of training dataset; suppose the two com.ps.pushbox APKs in Figure 6.43, one flagged as malware and the other not flagged, are included the training dataset, it is not clear how the resulting classifier should classify new com.ps.pushbox samples. In this case, the declarative approach as illustrated above can be used to recover the missing flags in the training dataset.

6.6.3 Robustness Against APK Transformation

To evaluate WoA's ability to withstand APK transformations that include repackaging and obfuscation, we obtain the ADAM APK transformation framework [231] from its authors (Zheng et al.) and test WoA on the transformed APKs generated by ADAM. Since ADAM's license requires explicit permission from the authors to acquire the software implementation, the reader is referred to the authors' site[12] for instructions on acquisition.

Nevertheless, for result reproducibility, we have prepared transformed versions of the running example Jk7H.PwcD generated by ADAM for download. Section 6.6.3.1 has the preparation instruction; Section 6.6.3.2 shows the transformations applied by ADAM in detail.

6.6.3.1 Preparation

Execute the following Bash commands to obtain the transformed versions of Jk7H.PwcD generated by ADAM.

```
(
PREFIX="https://github.com/pw4ever/web-of-apks/releases/download/apk-samples/";
NAMES=(
"Jk7H.PwcD-1-a00f2b489dac150e513526ab285141d41a127133cd3be2115046e22e189ff2a3-1_insert.apk"
"Jk7H.PwcD-1-a00f2b489dac150e513526ab285141d41a127133cd3be2115046e22e189ff2a3-2_ChangeName.apk"
"Jk7H.PwcD-1-a00f2b489dac150e513526ab285141d41a127133cd3be2115046e22e189ff2a3-3_ChangeCFG.apk"
"Jk7H.PwcD-1-a00f2b489dac150e513526ab285141d41a127133cd3be2115046e22e189ff2a3-4_StringEncrypt.apk"
```

[12] http://ansrlab.cse.cuhk.edu.hk/software/adam/

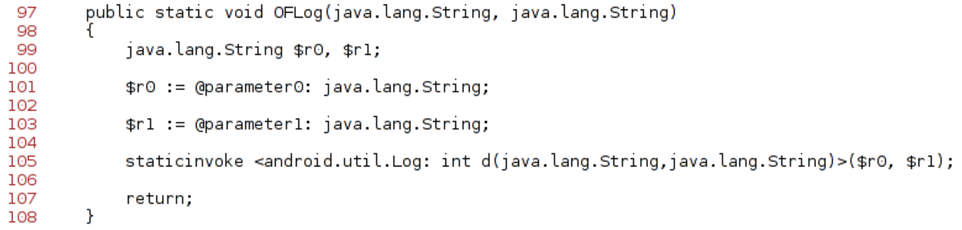
```
"Jk7H.PwcD-1-a00f2b489dac150e513526ab285141d41a127133cd3be2115046e22e189ff2a3-rebuild.apk"
"Jk7H.PwcD-1-a00f2b489dac150e513526ab285141d41a127133cd3be2115046e22e189ff2a3-resigned.apk"
"Jk7H.PwcD-1-a00f2b489dac150e513526ab285141d41a127133cd3be2115046e22e189ff2a3-zipaligned.apk"
"Jk7H.PwcD-1-a00f2b489dac150e513526ab285141d41a127133cd3be2115046e22e189ff2a3.apk"
);
TARGET="$HOME/woa-samples/";
mkdir -p "${TARGET}"; cd "${TARGET}";
for i in ${NAMES[@]}; do
  wget -nc -nd "${PREFIX}/${i}"
done
)
```

Afterwards, the `woa-samples` directory will contain the transformed APK samples generated by ADAM.

6.6.3.2 APK Transformations in ADAM

Two types of APK transformations are provided by ADAM: repackaging and obfuscation.

- Repackaging.
  - Rebuild: The input APK is disassembled and re-assembled (baksmali/smali in Android's term) with the tool `apktool` [2]. The idea is to exploit the binary-non-idempotent of the rebuilding process (i.e., the rebuilt APK binary does not necessarily equal to the original binary) to transform the APK.
  - Resign: The input APK is stripped of its original signature, and a new signature is used to generate a resigned APK. The idea is that resigning APK can evade APK-signature-based detection.
  - Realign: Android's `zipalign` is used to realign uncompressed data within the APK (such as images or raw files) from the Android's default 4-byte alignment to 8-byte alignment boundaries. The idea is that realignment changes the APK binary without changing APK program logic.

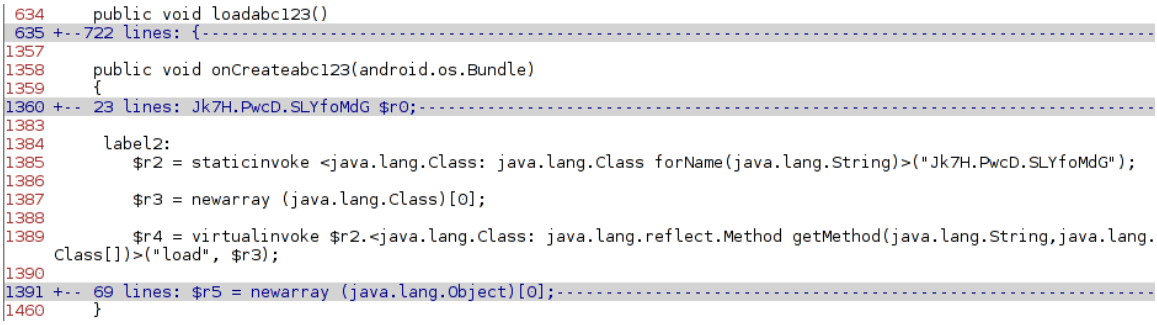


```
public static void OFLog(java.lang.String, java.lang.String)
{
    java.lang.String $r0, $r1;

    $r0 := @parameter0: java.lang.String;

    $r1 := @parameter1: java.lang.String;

    staticinvoke <android.util.Log: int d(java.lang.String,java.lang.String)>($r0, $r1);

    return;
}
```

Figure 6.44.: Effect of ADAM's defunct-method-insertion obfuscation on `Jk7H.PwcD.SLYfoMdG.onCreate`. A defunct method `OFLog` is inserted into each user-defined class, as shown here for the class `Jk7H.PwcD.SLYfoMdG`. File: `Jk7H.PwcD.SLYfoMdG.jimple`.

- Obfuscation.
  - Insert: A defunct method is into each user-defined class. This changes the structure of these classes.
  - Change method name: The string `abc123` is appended to every internal method, i.e., the method that is defined within the APK. However, as will be discussed shortly in the next section, the implementation of ADAM does not properly handle implicit control flow such as Java reflection—this shows the importance of handling implicit control flows.
  - Change intra-procedural control flow graph (CFG): A pair of `goto` instruction is inserted into the bytecode of each internal method. This changes the CFGs of these methods.
  - String encryption: Each string constant in the input APK's bytecode is replaced by an obfuscated one that can is recovered by calling a `DecryptString` method. To the best of our knowledge, no existing work on static APK analysis can handle this case.

While repackaging does not change the Dex inside the APK, all 4 types of obfuscation do change the executable code. Figures 6.44, 6.45, 6.46, 6.48, and 6.47 show the effect of applying ADAM's obfuscation on the running example `Jk7H.PwcD`.



```
634      public void loadabc123()
635 +--722 lines: {-----------------------------------------------------------------------------------------
1357
1358     public void onCreateabc123(android.os.Bundle)
1359     {
1360 +-- 23 lines: Jk7H.PwcD.SLYfoMdG $r0;----------------------------------------------------------------------
1383
1384      label2:
1385         $r2 = staticinvoke <java.lang.Class: java.lang.Class forName(java.lang.String)>("Jk7H.PwcD.SLYfoMdG");
1386
1387         $r3 = newarray (java.lang.Class)[0];
1388
1389         $r4 = virtualinvoke $r2.<java.lang.Class: java.lang.reflect.Method getMethod(java.lang.String,java.lang.
     Class[])>("load", $r3);
1390
1391 +-- 69 lines: $r5 = newarray (java.lang.Object)[0];-------------------------------------------------------
1460     }
```

Figure 6.45.: Effect of ADAM's method name obfuscation on `Jk7H.PwcD.SLYfoMdG.onCreate`. Unfortunately, as shown here, it does not properly handle implicit invocation through Java reflection. This is an example of the challenge and importance of handling implicit control flow, which is an important part of WoA's C3GE. File: `Jk7H.PwcD.SLYfoMdG.jimple`.

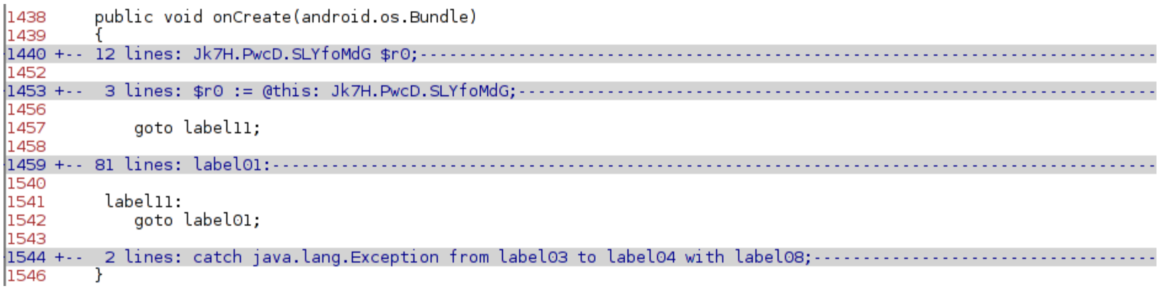
Figure 6.44 shows the effect of ADAM's defunct-method-insertion obfuscation on `Jk7H.PwcD.SLYfoMdG.onCreate`. A defunct method `OFLog`, which is not called by any other methods, is created and inserted into each user-defined class, as shown here for the class `Jk7H.PwcD.SLYfoMdG`. APK signatures that rely on APK method table will be affected by this obfuscation.

Figure 6.44 shows the effect of ADAM's method name obfuscation on `Jk7H.PwcD.SLYfoMdG.onCreate`. As previously mentioned, the string `abc123` is appended to every internal method, as shown here for `load` and `onCreate` of the class `Jk7H.PwcD.SLYfoMdG`. Unfortunately, Figure 6.44 indicates that ADAM's method name obfuscation fails to preserve app semantics in the following two cases:

- Component callback methods (e.g., `onCreate`) should not be renamed; otherwise, Android runtime would be able to find and call them when the corresponding event happens.

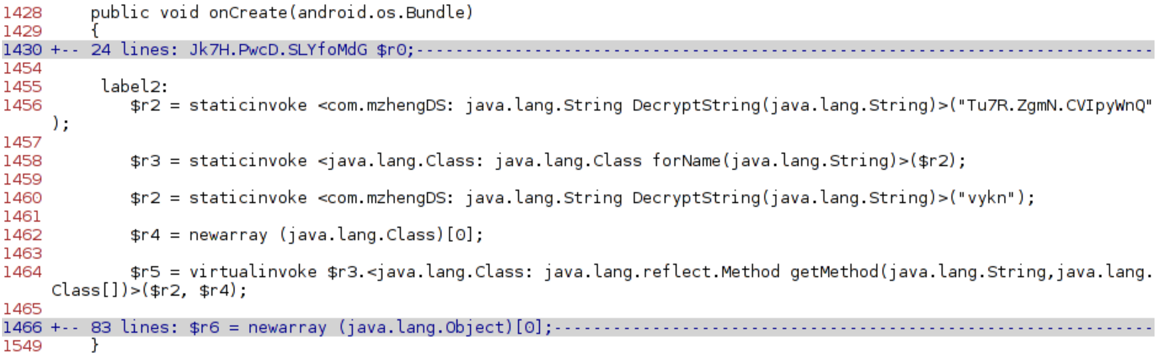
- Implicit control flows are not detected and renamed, as shown here on line 1389, in which the string `load` does not match the renamed method `loadabc123`.

This shows the need for proper implicit control flow handling, which is an important aspect of WoA.



```
    public void onCreate(android.os.Bundle)
    {
+-- 12 lines: Jk7H.PwcD.SLYfoMdG $r0;--------------------------------------------------------------------------

+--  3 lines: $r0 := @this: Jk7H.PwcD.SLYfoMdG;----------------------------------------------------------------

        goto label11;

+-- 81 lines: label01:-------------------------------------------------------------------------------------------

     label11:
        goto label01;

+--  2 lines: catch java.lang.Exception from label03 to label04 with label08;-----------------------------------
    }
```

Figure 6.46.: Effect of ADAM's `goto`-insertion obfuscation on `Jk7H.PwcD.SLYfoMdG.onCreate`. File: `Jk7H.PwcD.SLYfoMdG.jimple`.



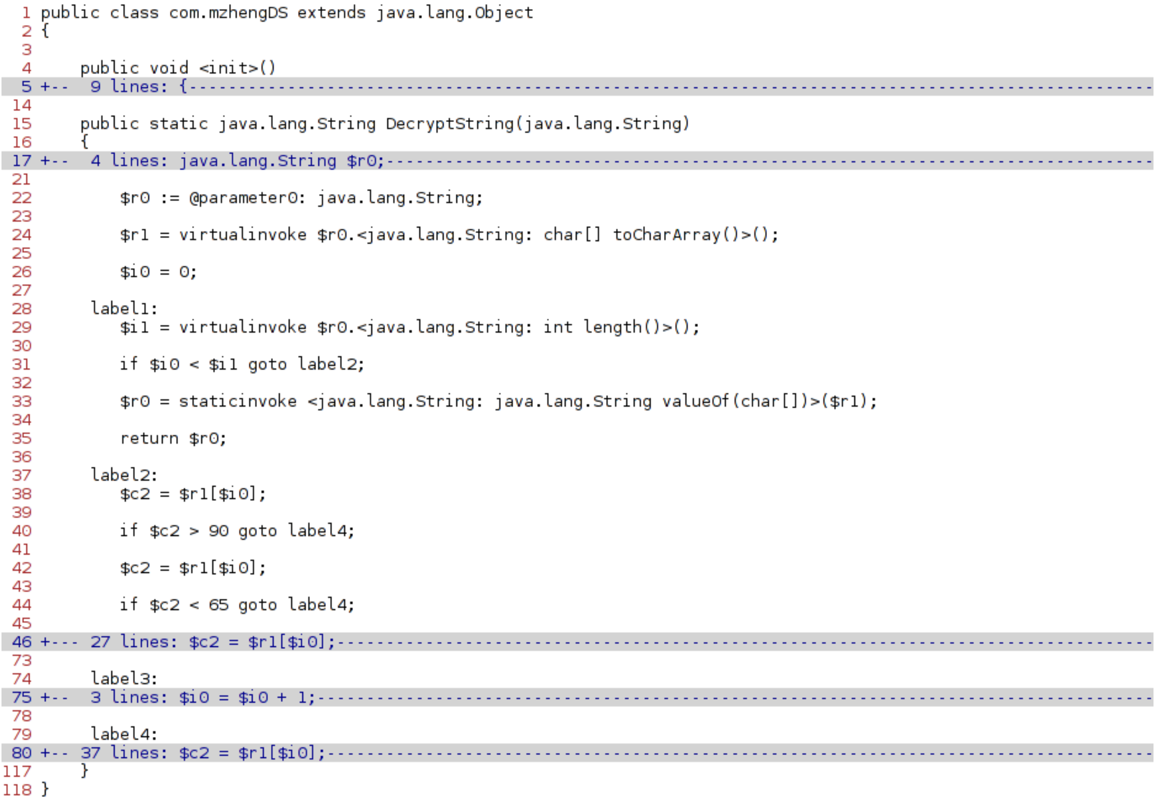
```
    public void onCreate(android.os.Bundle)
    {
+-- 24 lines: Jk7H.PwcD.SLYfoMdG $r0;--------------------------------------------------------------------------

     label2:
        $r2 = staticinvoke <com.mzhengDS: java.lang.String DecryptString(java.lang.String)>("Tu7R.ZgmN.CVIpyWnQ"
);

        $r3 = staticinvoke <java.lang.Class: java.lang.Class forName(java.lang.String)>($r2);

        $r2 = staticinvoke <com.mzhengDS: java.lang.String DecryptString(java.lang.String)>("vykn");

        $r4 = newarray (java.lang.Class)[0];

        $r5 = virtualinvoke $r3.<java.lang.Class: java.lang.reflect.Method getMethod(java.lang.String,java.lang.
Class[])>($r2, $r4);

+-- 83 lines: $r6 = newarray (java.lang.Object)[0];-------------------------------------------------------------
    }
```

Figure 6.47.: Effect of ADAM's string encryption obfuscation on `Jk7H.PwcD.SLYfoMdG.onCreate`. Strings are encrypted by the method `com.mzhengDS.DecryptString`. File: `Jk7H.PwcD.SLYfoMdG.jimple`.

Figure 6.44 shows the effect of ADAM's `goto`-insertion obfuscation on `Jk7H.PwcD.SLYfoMdG.onCreate`. A `goto` instruction is inserted at the beginning of the method, which jumps to another `goto` instruction at the end of the method, which then jumps to the instruction after the first `goto`. This does not change the externally observable behavior of the method; however, it changes the control flow within the method.

Lastly, Figures 6.48 and 6.47 show the effect of ADAM's string encryption obfuscation.

As shown in Figure 6.48, a new class `com.mzhengDS` with a method `DecryptString` is created. `DecryptString` transforms its sole `String` argument to a `char` array (line 24), and if the `String` argument is non-empty (line 31), `DecryptString`



```
  1 public class com.mzhengDS extends java.lang.Object
  2 {
  3
  4     public void <init>()
  5 +--  9 lines: {-------------------------------------------------------------------------------------------
 14
 15     public static java.lang.String DecryptString(java.lang.String)
 16     {
 17 +--  4 lines: java.lang.String $r0;-----------------------------------------------------------------------
 21
 22         $r0 := @parameter0: java.lang.String;
 23
 24         $r1 = virtualinvoke $r0.<java.lang.String: char[] toCharArray()>();
 25
 26         $i0 = 0;
 27
 28      label1:
 29         $i1 = virtualinvoke $r0.<java.lang.String: int length()>();
 30
 31         if $i0 < $i1 goto label2;
 32
 33         $r0 = staticinvoke <java.lang.String: java.lang.String valueOf(char[])>($r1);
 34
 35         return $r0;
 36
 37      label2:
 38         $c2 = $r1[$i0];
 39
 40         if $c2 > 90 goto label4;
 41
 42         $c2 = $r1[$i0];
 43
 44         if $c2 < 65 goto label4;
 45
 46 +--- 27 lines: $c2 = $r1[$i0];---------------------------------------------------------------------------
 73
 74      label3:
 75 +--  3 lines: $i0 = $i0 + 1;----------------------------------------------------------------------------
 78
 79      label4:
 80 +-- 37 lines: $c2 = $r1[$i0];---------------------------------------------------------------------------
117     }
118 }
```

Figure 6.48.: Effect of ADAM's string encryption obfuscation: a new class `com.mzhengDS` with method `DecryptString` is inserted. File: `com.mzhengDS.jimple`.

starts de-obfuscating the string through `char` arithmetics (lines 38–116; abbreviated in Figure 6.48).

Figure 6.47 shows the effect on `Jk7H.PwcD.SLYfoMdG.onCreate`. All the string constants are first de-obfuscated by `DecryptString` before being used. On line 1456, the obfuscated string, `Tu7R.ZgmN.CVIpyWnQ`, of the original string `Jk7H.PwcD.SLYfoMdG` (refer to line 1385 in Figure 6.5) is first de-obfuscated by `DecryptString` before being used on line 1458 to get the class reference. Similarly, on line 1460, the obfuscated string, `vykn`, of the original string `load` (refer to line 1389 in Figure 6.5) is first de-obfuscated by `DecryptString` before being used on line 1464 to get the method reference.

To the best of our knowledge, no publicly available static APK analysis tool is designed to handle string encryption like this. This is a major motivation to the design of WoA's C3GE, which we evaluate next.

6.6.3.3 WoA C3GE's Robustness on Transformed APKs

To evaluate WoA C3GE's robustness in processing transformed APKs, executing the following Bash commands (refer to Section 6.3.2 on bootstrapping WoA's command-line interface `woa`):

```
(
cd "$HOME/woa-samples/";
ls *.apk | woa --prep-tags "" | JVM_HEAP=5g woa -sv -d /dev/null \
  --soot-basic-block-simulation-budget 20 \
  --soot-method-simulation-depth-budget 8 \
  --soot-simulation-conservative-branching
woa -L -l <(ls *.model-dump) --debug-cgdfd | tee cgdfd.txt
sha256sum *.apk > sha256.txt
)
```

which saves the models extracted by WoA in `*.model-dump` files and the C3GDFD signatures (Section 6.4.4.2) in the file `cgdfd.txt`.

Table 6.3 shows the effect of the APK transformations implemented in ADAM on the component callback call graph (C3G) of `Jk7H.PwcD.SLYfoMdG.onCreate` extracted by WoA C3GE (this can be obtained from the result in the aforementioned file `cgdfd.txt`). Except for two cases, the extracted call graphs are identical. The explanations for the result is:

- Since "rebuild," "realigned," and "resigned" do not change the executable, the extracted call graph is not affected.
- "defunct method insertion" only changes the method table of the APK by associating `OFLog` with each class, but does not change the C3G rooted at `Jk7H.PwcD.SLYfoMdG.onCreate`.

Table 6.3.: Effect of the APK transformations implemented in ADAM on the component callback call graph (C3G) of `Jk7H.PwcD.SLYfoMdG.onCreate` extracted by WoA C3GE. “0:54” in the C3GDFD column means there are 54 nodes with an out-degree of 0. C3GE simulation parameters: BBB = 20, IDB = 8, CSB = 10000, conservative branching.

APK Transformation	C3GDFD	C3GDFD-based Signature
no transformation	0:54 1:4 2:1 3:3 4:7 5:3 7:1 8:2 9:2 14:1 17:1	[79 1.646 3.266 2.521 6.990]
rebuild	0:54 1:4 2:1 3:3 4:7 5:3 7:1 8:2 9:2 14:1 17:1	[79 1.646 3.266 2.521 6.990]
realigned	0:54 1:4 2:1 3:3 4:7 5:3 7:1 8:2 9:2 14:1 17:1	[79 1.646 3.266 2.521 6.990]
resigned	0:54 1:4 2:1 3:3 4:7 5:3 7:1 8:2 9:2 14:1 17:1	[79 1.646 3.266 2.521 6.990]
change method name	0:9 9:1	[10 0.900 2.846 2.277 3.570]
defunct method insertion	0:54 1:4 2:1 3:3 4:7 5:3 7:1 8:2 9:2 14:1 17:1	[79 1.646 3.266 2.521 6.990]
`goto` insertion	0:54 1:4 2:1 3:3 4:7 5:3 7:1 8:2 9:2 14:1 17:1	[79 1.646 3.266 2.521 6.990]
string encryption	0:57 1:3 2:1 3:5 4:3 5:5 6:2 8:2 9:2 10:1 15:1 17:1	[83 1.747 3.400 2.378 6.024]

- "goto insertion" only changes intra-procedural control flow, but does not change the C3G rooted at Jk7H.PwcD.SLYfoMdG.onCreate.

As for the two exceptions, "change method name" and "string encryption," the explanation for the result is:

- As previously discussed in Section 6.6.3.2, ADAM's implementation of "change method name" does not match reflection calls to the renamed methods—which changes app's semantics. After reload is renamed to reloadabc123, C3GE cannot resolve reflection call "reload."

- As for "string encryption," the inserted DecryptString method and the methods it invokes (e.g., java.lang.String.toCharArray in Figure 6.48) are added to the C3G at appropriate places, which changes the C3GDFD and the signature. Note that, although a non-trivial method DecryptString is inserted, the changes to C3GDFD and the signature is minor.

To confirm the C3G generated for "string encryption"-obfuscated APK by WoA C3GE matches the original ones, using the following WoA query:

```
MATCH
(a:Apk {sha256:"ed050a1fce94acb4b07ea88630f1ec2d5cb690378734e97b23e01d08cf243350"})
  -[:CALLGRAPH]->(cg:CallGraphNode)
MATCH (cg)-[:INVOKE*..4]->(m:CallGraphNode)
RETURN *
```

we get the 4-hop C3G of the callback method Jk7H.PwcD.SLYfoMdG.onCreate for the obfuscated APK shown in Figure 6.49, which can be compared with Figure 6.23. Note the inserted com.mzheng.DecryptString method are invoked from methods that use String constants.

Furthermore, to investigate the details of the de-obfuscation process, we can trace C3GE in action during DecryptString by the following invocation of woa on command line:

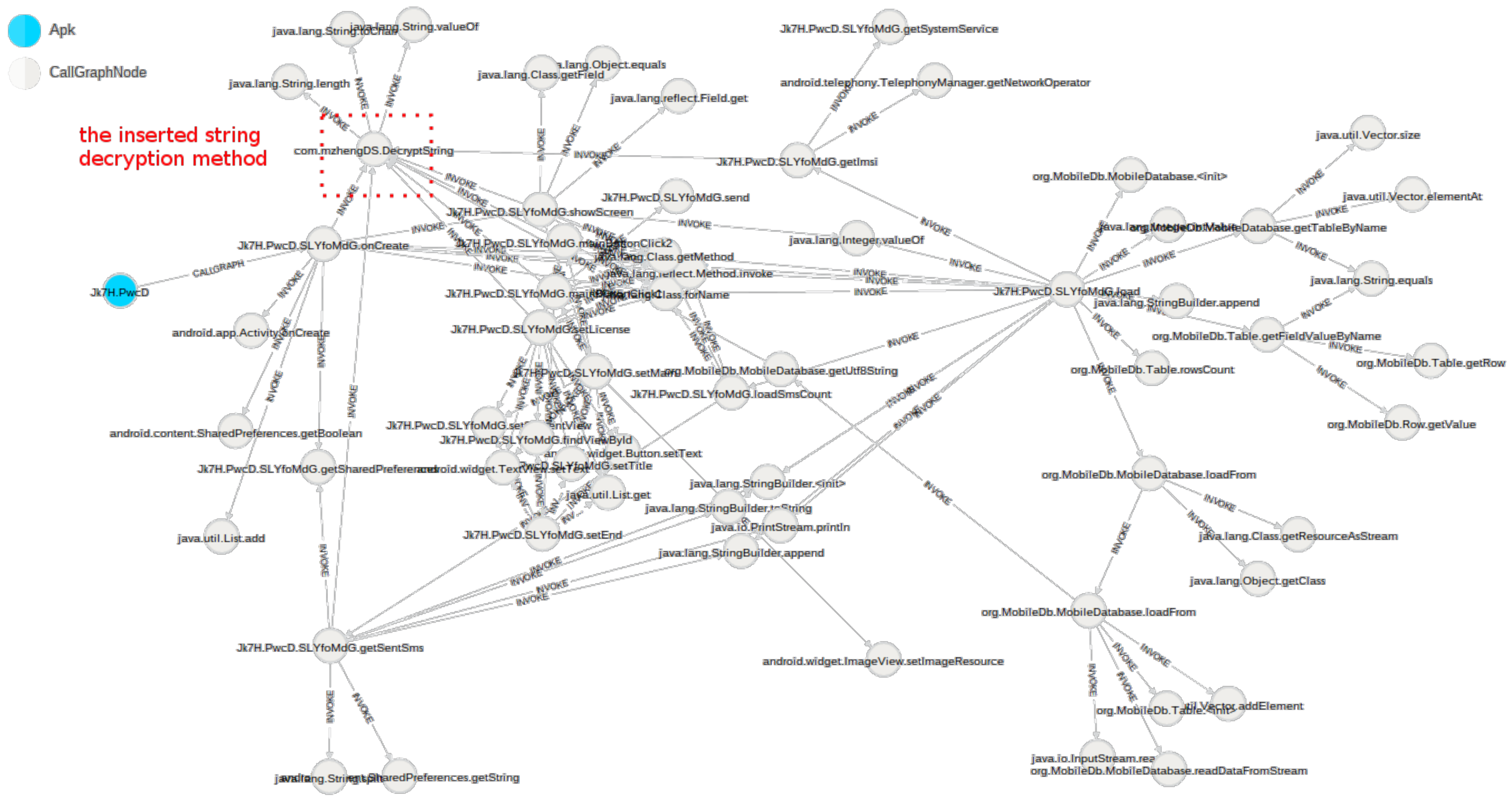
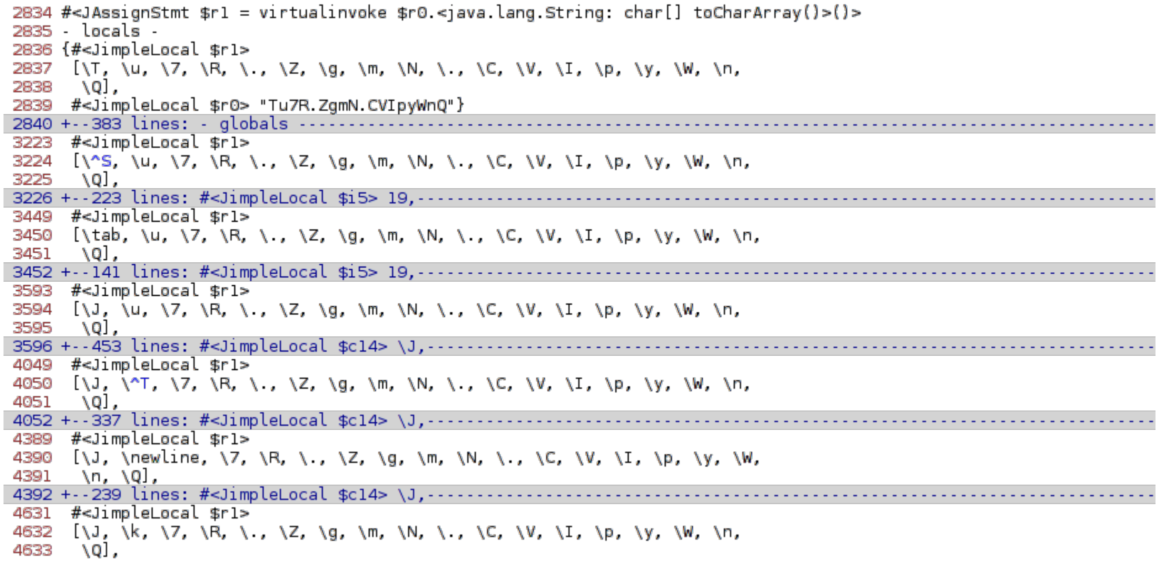

Apk
CallGraphNode
the inserted string decryption method
com.mzhengDS.DecryptString
Jk7H.PwcD
CALLGRAPH
Jk7H.PwcD.SLYfoMdG.onCreate
android.app.Activity.onCreate
android.content.SharedPreferences.getBoolean
Jk7H.PwcD.SLYfoMdG.getSharedPreferences
java.util.List.add
Jk7H.PwcD.SLYfoMdG.getSentSms
java.lang.String.length
java.lang.String.valueOf
java.lang.Class.getField
java.lang.Object.equals
java.lang.reflect.Field.get
Jk7H.PwcD.SLYfoMdG.showScreen
Jk7H.PwcD.SLYfoMdG.send
Jk7H.PwcD.SLYfoMdG.getSystemService
android.telephony.TelephonyManager.getNetworkOperator
Jk7H.PwcD.SLYfoMdG.getImsi
java.lang.Integer.valueOf
java.lang.Class.getMethod
java.lang.reflect.Method.invoke
java.lang.Class.forName
Jk7H.PwcD.SLYfoMdG.setLicense
Jk7H.PwcD.SLYfoMdG.setMain
org.MobileDb.MobileDatabase.getUtf8String
Jk7H.PwcD.SLYfoMdG.loadSmsCount
Jk7H.PwcD.SLYfoMdG.findViewById
widget.Button.setText
android.widget.TextView.setText
Jk7H.PwcD.SLYfoMdG.setTitle
java.util.List.get
Jk7H.PwcD.SLYfoMdG.setEnd
java.lang.StringBuilder.<init>
java.io.PrintStream.println
java.lang.StringBuilder.append
android.widget.ImageView.setImageResource
Jk7H.PwcD.SLYfoMdG.load
org.MobileDb.MobileDatabase.<init>
org.MobileDb.MobileDatabase.getTableByName
java.util.Vector.size
java.util.Vector.elementAt
java.lang.String.equals
org.MobileDb.Table.getFieldValueByName
org.MobileDb.Table.getRow
org.MobileDb.Row.getValue
org.MobileDb.Table.rowsCount
org.MobileDb.MobileDatabase.loadFrom
java.lang.Class.getResourceAsStream
java.lang.Object.getClass
org.MobileDb.Table.<init>
java.util.Vector.addElement
java.io.InputStream.read
org.MobileDb.MobileDatabase.readDataFromStream
android.content.SharedPreferences.getString
INVOKE

Figure 6.49.: 4-hop C3G of the callback method `Jk7H.PwcD.SLYfoMdG.onCreate` after `Jk7H.PwcD` has been transformed by ADAM's string-encryption obfuscation. Note the inserted `com.mzheng.DecryptString` method are invoked from methods that use `String` constants.



```
#<JAssignStmt $r1 = virtualinvoke $r0.<java.lang.String: char[] toCharArray()>()>
- locals -
{#<JimpleLocal $r1>
 [\T, \u, \7, \R, \., \Z, \g, \m, \N, \., \C, \V, \I, \p, \y, \W, \n,
  \Q],
 #<JimpleLocal $r0> "Tu7R.ZgmN.CVIpyWnQ"}
+--383 lines: - globals ---------------------------------------------------------------------------------------------
 #<JimpleLocal $r1>
 [\^S, \u, \7, \R, \., \Z, \g, \m, \N, \., \C, \V, \I, \p, \y, \W, \n,
  \Q],
+--223 lines: #<JimpleLocal $i5> 19,--------------------------------------------------------------------------------
 #<JimpleLocal $r1>
 [\tab, \u, \7, \R, \., \Z, \g, \m, \N, \., \C, \V, \I, \p, \y, \W, \n,
  \Q],
+--141 lines: #<JimpleLocal $i5> 19,--------------------------------------------------------------------------------
 #<JimpleLocal $r1>
 [\J, \u, \7, \R, \., \Z, \g, \m, \N, \., \C, \V, \I, \p, \y, \W, \n,
  \Q],
+--453 lines: #<JimpleLocal $c14> \J,-------------------------------------------------------------------------------
 #<JimpleLocal $r1>
 [\J, \^T, \7, \R, \., \Z, \g, \m, \N, \., \C, \V, \I, \p, \y, \W, \n,
  \Q],
+--337 lines: #<JimpleLocal $c14> \J,-------------------------------------------------------------------------------
 #<JimpleLocal $r1>
 [\J, \newline, \7, \R, \., \Z, \g, \m, \N, \., \C, \V, \I, \p, \y, \W,
  \n, \Q],
+--239 lines: #<JimpleLocal $c14> \J,-------------------------------------------------------------------------------
 #<JimpleLocal $r1>
 [\J, \k, \7, \R, \., \Z, \g, \m, \N, \., \C, \V, \I, \p, \y, \W, \n,
  \Q],
```

Figure 6.50.: A snippet of simulator trace for the string decryption process of `com.mzheng.DecryptString`, in which C3GE is de-obfuscating the obfuscated string `Tu7R.ZgmN.CVIpyWnQ` into `Jk7H.PwcD.SLYfoMdG` for the Java reflection call in `Jk7H.PwcD.SLYfoMdG.onCreate`. The snippet shows the de-obfuscation of the first two letters.

```
(
cd "$HOME/woa-samples/"
ls *StringEncrypt.apk | woa --prep-tags "" | JVM_HEAP=5g woa -s -d /dev/null \
  --overwrite-model \
  --soot-basic-block-simulation-budget 20 \
  --soot-method-simulation-depth-budget 8 \
  --soot-simulation-conservative-branching \
  --soot-debug-show-all-per-statement > trace-se.txt
)
```

which saves an execution trace of the simulator in the file `trace-se.txt`, a snippet of which is shown in Figure 6.50. Figure 6.50 shows the de-obfuscation of the first two letters in the obfuscated string `Tu7R.ZgmN.CVIpyWnQ` into `Jk7H.PwcD.SLYfoMdG` for the Java reflection call in `Jk7H.PwcD.SLYfoMdG.onCreate` (line 1456 in Figure 6.47).

The string de-obfuscation example above justifies the additional complexity of C3GE's simulator logic beyond simple replacement-based implicit control flow resolution. To the best of our knowledge, this capability is a novel contribution of WoA that no existing static APK analysis solution provides.

6.7 Summary and Future Work

The central theme of this chapter is a novel declarative approach to Android APK analysis based on the proposed Web of APKs (WoA) model. Towards this objective, we examine various aspects of our publicly available reference implementation and evaluate the approach on a prototype WoA that is built from real app samples. The core contributions are:

- Propose, implement, and evaluate a declarative graph analysis approach, Web of APKs, to analyze the relationship between APKs.
- Design and implement a call graph extraction algorithm that can handle Android-specific implicit control flows and value obfuscation, and provide means to handle adversary scenarios.

This is only a beginning. Future extensions to the present work include:

- Integrate resource references [229, 205] in the WoA model to facilitate the discovery of APKs that are connected by usage of common resources.
- Integrate bytecode-based method signatures [97, 233] in the WoA model to facilitate the discovery of APKs that have similar methods.
- Collect a library of APK analysis recipes based on WoA's declarative graph analysis support.
- Design new features that can be extracted with WoA for machine-learning-based APK analysis approaches.
- Integrate WoA with a dynamic APK analysis tool to form a hybrid APK analysis platform. A current candidate is our `Figurehead` tool, also publicly available online[13], which provides remote access to *full* Android API (including the classes that used by Android system tools such as `am` and `pm`, and are usually not available to app developers) with root privilege.

[13] `https://github.com/pw4ever/tbnl`

It should also be pointed out that there are several APK analysis scenarios that are beyond the capability of present work, which include: 1. use of non-Java native libraries, 2. dynamic code loading through Java's class loading facility, 3. dynamic value loading from external sources such as over Internet or from files/databases. These are currently open problems that require further research to address. Nevertheless, through collaboration and open access to implementation (in addition to ideas), we hope this work will contribute to future mobile security research.

7 CONCLUSION

This dissertation presents a study of several problems regarding the application of opportunistic proximate links as a cost-effective alternative to persistent cellular links in smartphone networks.

- Application scenarios that are considered include prioritized defense deployment (Chapter 2) and mobile data offloading (Chapter 3).
- For distribution of useful content using opportunistic proximate links, while sharing the same constraint of "no central coordination/planning through cellular channel," the different models that are considered include: full content delivery coverage (Chapter 2), topical content delivery coverage (Chapter 3), and full content delivery coverage with partial cellular coverage (Chapter 4).
- For preventing malware propagating over opportunistic proximate links, both an abstract probabilistic behavioral characterization (Chapter 5) and concrete cases on the Android platform (Chapter 6) are investigated.

With future work for each problem summarized in its respective chapter, the following ideas are the essence of this dissertation in retrospect.

- A temporally dominating/covering subset of a smartphone network that is connected by opportunistic proximate links, elected by a distributed algorithm using a local temporal quality metric (the reachability metric in Chapter 2 and the KDE-based temporal coverage quality metric in Chapter 4), is a virtual content delivery backbone that balances between delivery effectiveness and costs.
- In offloading topical cellular content, the virtue of patience (Chapter 3) is to allow the more capable to have better chances of serving the common good

through situation awareness that adapts to opportunistic neighborhood interest profile, temporal topological importance, and content receival status.

- The KDE-based temporal coverage metric (Chapter 4) and “look-ahead” (Chapter 5) are two way to handle uncertainty in temporal channel quality estimation and evidence collection, respectively.

- As manifested by dogmatic filtering and adaptive look-ahead (Chapter 5), a way to aggregate sequential indirect evidence without knowing the authenticity of the evidence is to restrict how much the current opinion could be swayed by the indirect evidence.

- The declarative graph query approach implemented in Web of APKs (Chapter 6) provides an effective way to discover syntactic/semantic similarity between mobile apps, and provides support for both interactive exploration and further automated analysis based on the extracted features.

REFERENCES

REFERENCES

[1] US patent 20110051665: Location histories for location aware devices, 2011.

[2] android-apktool. `https://code.google.com/p/android-apktool/`, 2015.

[3] The Drebin dataset. `http://user.informatik.uni-goettingen.de/~darp/drebin/`, 2015.

[4] Serge Abiteboul and Victor Vianu. Datalog extensions for database queries and updates. *Journal of Computer and System Sciences*, 43(1):62–124, 1991.

[5] J.M. Agosta, C. Diuk-Wasser, J. Chandrashekar, and C. Livadas. An adaptive anomaly detector for worm detection. In *Proceedings of USENIX Workshop on Tackling Computer Systems Problems with Machine Learning Techniques (SysML)*, pages 1–6. USENIX, 2007.

[6] P. Akritidis, WY Chin, VT Lam, S. Sidiroglou, and KG Anagnostakis. Proximity breeds danger: Emerging threats in metro-area wireless networks. In *Proceedings of USENIX Security*. USENIX, 2007.

[7] Open Handset Alliance. Overview. `http://www.openhandsetalliance.com/oha_overview.html`, 2015.

[8] Amazon. Amazon Appstore. `https://amazon.com/appstore`, 2015.

[9] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *Proceedings of IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 459–468. IEEE, 2006.

[10] Android. Managing the Activity lifecycle. `https://developer.android.com/training/basics/activity-lifecycle/index.html`, 2015.

[11] Android. Android app components. `https://developer.android.com/guide/components/index.html`, 2015.

[12] Android. Android developers. `https://developer.android.com/index.html`, 2015.

[13] Android. Android emulator. `https://developer.android.com/tools/help/emulator.html`, 2015.

[14] Android. ProGuard. `https://developer.android.com/tools/help/proguard.html`, 2015.

[15] I. Androutsopoulos, J. Koutsias, K.V. Chandrinos, and C.D. Spyropoulos. An experimental comparison of naive bayesian and keyword-based anti-spam filtering with personal e-mail messages. In *Proceedings of ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 160–167. ACM, 2000.

[16] Daniel Arp, Michael Spreitzenbarth, Malte Hübner, Hugo Gascon, and Konrad Rieck. DREBIN: Effective and explainable detection of Android malware in your pocket. In *Proceedings of ISOC Network and Distributed System Security Symposium (NDSS)*. 2014.

[17] Steven Arzt, Siegfried Rasthofer, E Bodden, A Bartel, J Klein, Y Le Traon, D Octeau, and P McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Proceedings of ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 259–269, 2014.

[18] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. Pscout: Analyzing the Android permission specification. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, pages 217–228. ACM, 2012.

[19] Phil Bagwell. Fast functional lists. In *Implementation of Functional Languages*, pages 34–50. Springer, 2003.

[20] Yameng Bai, Xingming Sun, Guang Sun, Xiaohong Deng, and Xiaoming Zhou. Dynamic k-gram based software birthmark. In *Proceedings of IEEE Australian Conference on Software Engineering (ASWEC)*, pages 644–649. IEEE, 2008.

[21] A. Balasubramanian, R. Mahajan, and A. Venkataramani. Augmenting mobile 3G using WiFi. In *Proceedings of ACM International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 209–222. ACM, 2010.

[22] David Barrera, H Güneş Kayacik, Paul C van Oorschot, and Anil Somayaji. A methodology for empirical analysis of permission-based security models and its application to Android. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, pages 73–84. ACM, 2010.

[23] U. Bayer, P.M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda. Scalable, behavior-based malware clustering. In *Proceedings of ISOC Network and Distributed System Security Symposium (NDSS)*. ISOC, 2009.

[24] Brain Bergstein. IBM faces the perils of “bring your own device”, 2012. URL `http://goo.gl/Dnlpe`.

[25] Matthias Böhmer, Brent Hecht, Johannes Schöning, Antonio Krüger, and Gernot Bauer. Falling asleep with Angry Birds, Facebook and Kindle: A large scale study on mobile application usage. In *Proceedings of International Conference on Human Computer Interaction with Mobile Devices and Services*, pages 47–56. ACM, 2011.

[26] P. Bonacich. Power and centrality: A family of measures. *American Journal of Sociology*, pages 1170–1182, 1987.

[27] JR Bookwater. Google Nexus 7 review, 2012. URL `http://goo.gl/yigHn`.

[28] S.P. Borgatti. Centrality and network flow. *Social Network*, 27(1):55–71, 2005.

[29] A. Bose and K. Shin. On mobile viruses exploiting messaging and Bluetooth services. In *Proceedings of IEEE Securecomm and Workshops*, pages 1–10. IEEE, 2006.

[30] U. Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25(2):163–177, 2001.

[31] Ivan Bratko. *Machine learning: Between accuracy and interpretability.* Springer, 1997.

[32] Leo Breiman et al. Statistical modeling: The two cultures (with comments and a rejoinder by the author). *Statistical Science*, 16(3):199–231, 2001.

[33] Andrei Z Broder, Moses Charikar, Alan M Frieze, and Michael Mitzenmacher. Min-wise independent permutations. In *Proceedings of ACM Symposium on Theory of Computing (STOC)*, pages 327–336. ACM, 1998.

[34] S. Buchegger and J. Boudec. Performance analysis of the CONFIDANT protocol. In *Proceedings of ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc)*, pages 226–236. ACM, 2002.

[35] S. Buchegger and J.Y. Le Boudee. Self-policing mobile ad hoc networks by reputation systems. *IEEE Communication Magazine*, 43(7):101–107, 2005.

[36] Horst Bunke and Janos Csirik. An improved algorithm for computing the edit distance of run-length coded strings. *Information Processing Letters*, 54(2): 93–96, 1995.

[37] J. Burgess, B. Gallagher, D. Jensen, and B. Levine. MaxProp: Routing for vehicle-based disruption-tolerant networks. In *Proceedings of IEEE International Conference on Computer Communications (INFOCOM)*, pages 1–11. IEEE, 2006.

[38] Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani. Crowdroid: Behavior-based malware detection system for Android. In *Proceedings of ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 15–26, 2011.

[39] A. Carzaniga, M. Rutherford, and A. Wolf. A routing scheme for content-based networking. In *Proceedings of IEEE International Conference on Computer Communications (INFOCOM)*, pages 1–11. IEEE, 2004.

[40] Lei Cen, Chris Gates, Luo Si, and Ninghui Li. A probabilistic discriminative model for Android malware detection with decompiled source code. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, online pre-release, 2014.

[41] IBM T.J. Waston Research Center. T.J. Watson Libraries for Analysis (WALA). `http://wala.sourceforge.net/wiki/index.php/Main_Page`, 2015.

[42] D. Chakrabarti, J. Leskovec, C. Faloutsos, S. Madden, C. Guestrin, and M. Faloutsos. Information survival threshold in sensor and P2P networks. In *Proceedings of IEEE International Conference on Computer Communications (INFOCOM)*, pages 1316–1324, 2007.

[43] Saurabh Chakradeo, Bradley Reaves, Patrick Traynor, and William Enck. Mast: Triage for market-scale mobile malware analysis. In *Proceedings of ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, pages 13–24. ACM, 2013.

[44] Kai Chen, Peng Liu, and Yingjun Zhang. Achieving accuracy and scalability simultaneously in detecting application clones on Android markets. In *Proceedings of IEEE International Conference on Software Engineering (ICSE)*, pages 175–186. IEEE, 2014.

[45] S.M. Cheng, W.C. Ao, P.Y. Chen, and K.C. Chen. On modeling malware propagation in generalized social networks. *IEEE Communication Letter*, 15 (1):25–27, 2011.

[46] Shigeru Chiba. Load-time structural reflection in Java. In *Object-Oriented Programming*, pages 313–336. Springer, 2000.

[47] Shyam R Chidamber and Chris F Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[48] Rudi Cilibrasi and Paul MB Vitányi. Clustering by compression. *IEEE Transactions on Information Theory*, 51(4):1523–1545, 2005.

[49] William D Clinger. Proper tail recursion and space efficiency. In *ACM SIGPLAN Notices*, volume 33, pages 174–185. ACM, 1998.

[50] William Clocksin and Christopher S Mellish. *Programming in PROLOG*. Springer Science & Business Media, 2003.

[51] Christian Collberg, Ginger Myles, and Andrew Huntwork. Sandmark—a tool for software protection research. *IEEE Security & Privacy*, 1(4):40–49, 2003.

[52] Luigi P Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. A (sub) graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(10):1367–1372, 2004.

[53] The Oracle Corporation. Java SE. `http://www.oracle.com/technetwork/java/javase/downloads/index.html`, 2015.

[54] Enrique Costa-Montenegro, Ana Belén Barragáns-Martínez, and Marta Rey-López. Which app? A recommender system of applications in markets: Implementation of the service for monitoring users' interaction. *Expert Systems with Applications*, 39(10):9367–9375, 2012.

[55] Jonathan Crussell, Clint Gibler, and Hao Chen. Attack of the clones: Detecting cloned applications on Android markets. In *Computer Security—ESORICS*, pages 37–54. Springer, 2012.

[56] Jonathan Crussell, Clint Gibler, and Hao Chen. Andarwin: Scalable detection of semantically similar Android applications. In *Computer Security—ESORICS*, pages 182–199. Springer, 2013.

[57] E.M. Daly and M. Haahr. Social network analysis for information flow in disconnected delay-tolerant MANETs. *IEEE Transactions on Mobile Computing (TMC)*, 8(5):606–621, 2009.

[58] D. Dash, B. Kveton, J.M. Agosta, E. Schooler, J. Chandrashekar, A. Bachrach, and A. Newman. When gossip is good: Distributed probabilistic inference for detection of slow network intrusions. In *Proceedings of National Conference on Artificial Intelligence*, volume 21, page 1115, 2006.

[59] Chris J Date and Hugh Darwen. *A Guide to the SQL Standard*, volume 3. Addison-Wesley New York, 1987.

[60] Anindya Datta, Kaushik Dutta, Sangar Kajanan, and Nargin Pervin. Mobilewalla: A mobile application search engine. In *Mobile Computing, Applications, and Services*, pages 172–187. Springer, 2012.

[61] William HE Day and Herbert Edelsbrunner. Efficient algorithms for agglomerative hierarchical clustering methods. *Journal of Classification*, 1(1):7–24, 1984.

[62] Anthony Desnos. Android: Static analysis using similarity distance. In *Proceedings of IEEE Hawaii International Conference on System Science (HICSS)*, pages 5394–5403, 2012.

[63] Anthony Desnos. Androguard APK reverse engineering suite. `https://code.google.com/p/androguard/`, 2015.

[64] John Donovan. Wireless Data Volume on AT&T's Network Continues to Double Annually, 2012. URL `http://goo.gl/JtNKT`.

[65] James R Driscoll, Neil Sarnak, Daniel Dominic Sleator, and Robert Endre Tarjan. Making data structures persistent. In *Proceedings of ACM Symposium on Theory of Computing (STOC)*, pages 109–121. ACM, 1986.

[66] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification*. Wiley-Interscience, 2 edition, November 2001.

[67] N. Eagle and A. Pentland. CRAWDAD data set MIT/reality (v. 2005-07-01). http://goo.gl/V3YKc, July 2005.

[68] William Enck, Machigar Ongtang, and Patrick McDaniel. On lightweight mobile phone application certification. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, pages 235–245. ACM, 2009.

[69] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A study of Android application security. In *Proceedings of USENIX Security Symposium (Security)*, page 2. USENIX, 2011.

[70] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. TaintDroid: An information flow tracking system for real-time privacy monitoring on smartphones. *Communications of the ACM*, 57(3):99–106, 2014.

[71] Vassiliy A Epanechnikov. Non-parametric estimation of a multivariate probability density. *Theory of Probability & Its Applications*, 14(1):153–158, 1969.

[72] V. Erramilli, M. Crovella, A. Chaintreau, and C. Diot. Delegation forwarding. In *Proceedings of ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc)*, pages 251–260. ACM, 2008.

[73] M. Everett and S.P. Borgatti. Ego network betweenness. *Social Networks*, 27 (1):31–38, 2005.

[74] S. Fahl, M. Harbach, T. Muders, M. Smith, L. Baumgärtner, and B. Freisleben. Why Eve and Mallory love Android: An analysis of android SSL (in) security. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, pages 50–61. ACM, 2012.

[75] Kevin Fall. A delay-tolerant network architecture for challenged internets. In *Proceedings of ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 27–34. ACM, 2003.

[76] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, pages 627–638. ACM, 2011.

[77] Adrienne Porter Felt, Matthew Finifter, Erika Chin, Steve Hanna, and David Wagner. A survey of mobile malware in the wild. In *Proceedings of ACM Sorkshop on Security and Privacy in Smartphones and Mobile Devices*, pages 3–14. ACM, 2011.

[78] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. Apposcopy: Semantics-based detection of Android malware through static analysis. In *Proceedings of ACM Symposium on the Foundations of Software Engineering (FSE)*, pages 576–587. ACM, 2014.

[79] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

[80] Jay Freeman. JailbreakMe. `http://goo.gl/iqk7`, 2015.

[81] Christian Fritz, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves le Traon, Damien Octeau, and Patrick McDaniel. Highly precise taint analysis for Android applications. Technical report, TU Darmstadt, 2013.

[82] Mark Gabel, Lingxiao Jiang, and Zhendong Su. Scalable detection of semantic clones. In *Proceedings of IEEE International Conference on Software Engineering (ICSE)*, pages 321–330. IEEE, 2008.

[83] C. Gao and J. Liu. Modeling and restraining mobile virus propagation. *IEEE Transactions on Mobile Computing (TMC)*, preprint(99):1, 2012. ISSN 1536-1233.

[84] Gartner. Gartner says tablet sales continue to be slow in 2015. `http://www.gartner.com/newsroom/id/2954317`, 2015.

[85] Hugo Gascon, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Structural detection of Android malware using embedded call graphs. In *Proceedings of ACM Workshop on Artificial Intelligence and Security*, pages 45–54, 2013.

[86] Christopher S Gates, Jing Chen, Ninghui Li, and Robert W Proctor. Effective risk communication for Android apps. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 11(3):252–265, 2014.

[87] Christopher S Gates, Ninghui Li, Hao Peng, Bhaskar Pratim Sarma, Yuan Qi, Rahul Potharaju, Cristina Nita-Rotaru, and Ian Molloy. Generating summary risk scores for mobile applications. *IEEE Transactions on Dependable and Secure Computing (TDSC)*, 11(3):238–251, 2014.

[88] Gerald Gazdar and Christopher S Mellish. *Natural language processing in Prolog: An introduction to computational linguistics*, volume 148. Addison-Wesley Wokingham, 1989.

[89] Genymobile. Genymotion Android emulator. `https://www.genymotion.com/`, 2015.

[90] Peter Gilbert, Byung-Gon Chun, Landon P Cox, and Jaeyeon Jung. Vision: Automated security validation of mobile apps at app markets. In *Proceedings of ACM International Workshop on Mobile Cloud Computing and Services (MCS)*, pages 21–26, 2011.

[91] M.C. Gonzalez, C.A. Hidalgo, and A.L. Barabási. Understanding individual human mobility patterns. *Nature*, 453(7196):779–782, 2008.

[92] M. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic detection of capability leaks in stock android smartphones. In *Proceedings of ISOC Symposium on Network and Distributed Systems Security (NDSS)*. ISOC, 2012.

[93] Paul Graham. Better Bayesian filtering. URL `http://goo.gl/AgHkB`.

[94] B. Han, P. Hui, VS Kumar, M.V. Marathe, G. Pei, and A. Srinivasan. Cellular traffic offloading through opportunistic communications: A case study. In *Proceedings of ACM Workshop on Challenged Networks (CHANTS)*, pages 31–38. ACM, 2010.

[95] B. Han, P. Hui, and A. Srinivasan. Mobile data offloading in metropolitan area networks. *ACM SIGMOBILE Mobile Computing and Communication Review (MC2R)*, 14(4):28–30, 2011.

[96] B. Han, P. Hui, V. Kumar, M. Marathe, J. Shao, and A. Srinivasan. Mobile data offloading through opportunistic communications and social participation. *IEEE Transactions on Mobile Computing (TMC)*, 99(5):821–834, 2012.

[97] Steve Hanna, Ling Huang, Edward Wu, Saung Li, Charles Chen, and Dawn Song. Juxtapp: A scalable system for detecting code reuse among Android applications. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 62–81. Springer, 2013.

[98] Erin Harrison. Five enterprise tech trends for 2013: BYOD, VPNs, AaaS, Big Data and Business Intelligence, 2012. URL `http://goo.gl/T1JqO`.

[99] Rich Hickey. The Clojure programming language. In *Proceedings of ACM Symposium on Dynamic Languages (DLS)*, page 1. ACM, 2008.

[100] Shohei Hido and Hisashi Kashima. A linear-time graph kernel. In *Proceedings of IEEE International Conference on Data Mining (ICDM)*, pages 179–188. IEEE, 2009.

[101] Jochen Hipp, Ulrich Güntzer, and Gholamreza Nakhaeizadeh. Algorithms for association rule mining—a general survey and comparison. *ACM SIGKDD Explorations Newsletter*, 2(1):58–64, 2000.

[102] Ting-Wei Hou, Hsiang-Yang Chen, and Ming-Hsiu Tsai. Three control flow obfuscation methods for java software. *IEE Proceedings-Software*, 153(2):80–86, 2006.

[103] W. Hsu, T. Spyropoulos, K. Psounis, and A. Helmy. Modeling time-variant user mobility in wireless mobile networks. In *Proceedings of IEEE International Conference on Computer Communications (INFOCOM)*, pages 758–766. IEEE, 2007.

[104] Cuixiong Hu and Iulian Neamtiu. Automating GUI testing for Android applications. In *Proceedings of ACM International Workshop on Automation of Software Test (AST)*, pages 77–83, 2011.

[105] Heqing Huang, Sencun Zhu, Peng Liu, and Dinghao Wu. A framework for evaluating mobile app repackaging detection algorithms. In *Trust and Trustworthy Computing*, pages 169–186. Springer, 2013.

[106] Jianjun Huang, Xiangyu Zhang, Lin Tan, Peng Wang, and Bin Liang. AsDroid: Detecting stealthy behaviors in Android applications by user interface and program behavior contradiction. In *Proceedings of IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 1036–1046, 2014.

[107] Pan Hui, Jon Crowcroft, and Eiko Yoneki. Bubble rap: Social-based forwarding in delay-tolerant networks. *IEEE Transactions on Mobile Computing*, 10(11): 1576–1589, 2011.

[108] S. Ioannidis, A. Chaintreau, and L. Massoulié. Optimal and scalable distribution of content updates over a mobile social network. In *Proceedings of IEEE International Conference on Computer Communications (INFOCOM)*, pages 1422–1430. IEEE, 2009.

[109] E.T. Jaynes. Information theory and statistical mechanics. *Physics Review*, 108 (2):171–190, 1957. ISSN 0031-899X.

[110] Yoon-Chan Jhi, Xinran Wang, Xiaoqi Jia, Sencun Zhu, Peng Liu, and Dinghao Wu. Value-based program characterization and its application to software plagiarism detection. In *Proceedings of ACM International Conference on Software Engineering (ICSE)*, pages 756–765. ACM, 2011.

[111] Bin Jiang, Junjun Yin, and Sijian Zhao. Characterizing the human mobility pattern in a large street network. *APS Physical Review E*, 80(2):021136, 2009.

[112] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of IEEE International Conference on Software Engineering (ICSE)*, pages 96–105. IEEE, 2007.

[113] D.B. Johnson. A note on Dijkstra's shortest path algorithm. *Journal of ACM (JACM)*, 20(3):385–388, 1973.

[114] David Jurgens. The VF2 subgraph isomorphism algorithm implementation in The S-Space Package. `https://github.com/fozziethebeat/S-Space/tree/master/src/main/java/edu/ucla/sspace/graph/isomorphism`, 2015.

[115] S. Kamvar, M. Schlosser, and H. Garcia-Molina. The eigentrust algorithm for reputation management in P2P networks. In *Proceedings of ACM International Conference on World Wide Web (WWW)*, pages 640–651. ACM, 2003.

[116] Alexandros Karatzoglou, Linas Baltrunas, Karen Church, and Matthias Böhmer. Climbing the app wall: Enabling mobile app discovery through context-aware recommendations. In *Proceedings of ACM International Conference on Information and Knowledge Management*, pages 2527–2530. ACM, 2012.

[117] D. Kempe, J. Kleinberg, and É. Tardos. Maximizing the spread of influence through a social network. In *Proceedings of ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 137–146. ACM, 2003.

[118] H. Kim and R. Anderson. Temporal node centrality in complex networks. *Physics Review E*, 85(2):026107, 2012.

[119] C. Kolbitsch, P.M. Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X.F. Wang. Effective and efficient malware detection at the end host. In *Proceedings of USENIX Security*. USENIX, 2009.

[120] Jesse Kornblum. Identifying almost identical files using context triggered piecewise hashing. *Digital investigation*, 3:91–97, 2006.

[121] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. The Soot framework for Java program analysis: A retrospective. In *Proceedings of Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, 2011.

[122] Adriana Lee. FBI warns: New malware threat targets travelers, infects via hotel Wi-Fi, 2012. URL `http://goo.gl/D8vNU`.

[123] Kyunghan Lee, Joohyun Lee, Yung Yi, Injong Rhee, and Song Chong. Mobile data offloading: How much can WiFi deliver? *IEEE/ACM Transactions on Networking (TON)*, 21(2):536–550, 2013.

[124] F. Li and J. Wu. MOPS: Providing content-based service in disruption-tolerant networks. In *Proceedings of IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 526–533. IEEE, 2009.

[125] F. Li, Y. Yang, and J. Wu. CPMC: An efficient proximity malware coping scheme in smartphone-based mobile networks. In *Proceedings of IEEE International Conference on Computer Communications (INFOCOM)*, pages 1–9. IEEE, 2010.

[126] Y. Li, P. Hui, L. Su, D. Jin, and L. Zeng. An optimal distributed malware defense system for mobile networks with heterogeneous devices. In *Proceedings of IEEE Conference on Sensor, Mesh and Ad Hoc Communications and Networks (SECON)*, pages 314–322. IEEE, 2011.

[127] Jovian Lin, Kazunari Sugiyama, Min-Yen Kan, and Tat-Seng Chua. New and improved: Modeling versions to improve app recommendation. In *Proceedings of ACM International Conference on Research and Development in Information Retrieval*, pages 647–656. ACM, 2014.

[128] Martina Lindorfer, Matthias Neugschwandtner, Lukas Weichselbaum, Yanick Fratantonio, Victor van der Veen, and Christian Platzer. ANDRUBIS—1,000,000 apps later: A view on current Android malware behaviors. In *Proceedings of International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, pages 1–15, 2014.

[129] Cong Liu and Jie Wu. An optimal probabilistic forwarding protocol in delay tolerant networks. In *Proceedings of ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHOC)*, pages 105–114. ACM, 2009.

[130] John W Lloyd. Practical advantages of declarative programming. In *Joint Conference on Declarative Programming, GULP-PRODE*, volume 94, page 94, 1994.

[131] Douglas Low. Protecting Java code via code obfuscation. *Crossroads*, 4(3): 21–23, 1998.

[132] Bin Lu, Fenlin Liu, Xin Ge, Bin Liu, and Xiangyang Luo. A software birthmark based on dynamic opcode *n*-gram. In *Proceedings of IEEE International Conference on Semantic Computing (ICSC)*, pages 37–44. IEEE, 2007.

[133] Aravind MacHiry, Rohan Tahiliani, and Mayur Naik. Dynodroid: An input generation system for Android apps. In *Proceedings of ACM Joint Meeting on Foundations of Software Engineering*, pages 224–234, 2013.

[134] Kanti V Mardia. Measures of multivariate skewness and kurtosis with applications. *Biometrika*, 57(3):519–530, 1970.

[135] S. Marti, T.J. Giuli, K. Lai, M. Baker, et al. Mitigating routing misbehavior in mobile ad hoc networks. In *Proceedings of ACM International Conference on Mobile Computing and Networking (MobiCom)*, pages 255–265. ACM, 2000.

[136] Thomas J McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, (4):308–320, 1976.

[137] Glen McCluskey. Using Java reflection. *Java Developer Connection*, 1998.

[138] R. McGill, J.W. Tukey, and W.A. Larsen. Variations of box plots. *The American Statistician*, 32(1):12–16, 1978.

[139] M. McNett and G.M. Voelker. Access and mobility of wireless PDA users. *ACM SIGMOBILE Mobile Computing and Communication Review (MC2R)*, 9 (2):40–55, 2005.

[140] P. Michiardi and R. Molva. Core: A collaborative reputation mechanism to enforce node cooperation in mobile ad hoc networks. In *Proceedings of IFIP TC6/TC11 Joint Working Conference on Communications and Multimedia Security*, pages 107–121, 2002.

[141] K.W. Miller, J. Voas, and G.F. Hurlburt. BYOD: Security and privacy considerations. *IT Professional*, 14(5):53–55, 2012.

[142] Ginger Myles and Christian Collberg. Detecting software theft via whole program path birthmarks. In *Information security*, pages 404–415. Springer, 2004.

[143] S. Nanda and D. Kotz. Localized bridging centrality for distributed network analysis. In *Proceedings of IEEE International Conference on Computer Communications and Networks (ICCCN)*, pages 1–6. IEEE, 2008.

[144] Inc. Neo Technology. The Neo4j graph database. `http://neo4j.com/`, 2015.

[145] Inc. Neo Technology. Neo4j Cypher refcard (version 2.1.7). `http://neo4j.com/docs/stable/cypher-refcard/`, 2015.

[146] NFC Forum. About NFC. URL `http://goo.gl/zSJqb`.

[147] Doug Orleans and Karl Lieberherr. DJ: Dynamic adaptive programming in Java. In *Metalevel Architectures and Separation of Crosscutting Concerns*, pages 73–80. Springer, 2001.

[148] Rahul Pandita, Xusheng Xiao, Wei Yang, William Enck, and Tao Xie. WHYPER: Towards automating risk assessment of mobile applications. In *Proceedings of USENIX Security*, pages 527–542, 2013.

[149] Hao Peng, Chris Gates, Bhaskar Sarma, Ninghui Li, Yuan Qi, Rahul Potharaju, Cristina Nita-Rotaru, and Ian Molloy. Using probabilistic generative models for ranking risks of Android apps. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, pages 241–252. ACM, 2012.

[150] Roger D Peng. Reproducible research in computational science. *Science*, 334 (6060):1226, 2011.

[151] W. Peng, F. Li, X. Zou, and J. Wu. Behavioral detection and containment of proximity malware in delay tolerant networks. In *Proceedings of IEEE International Conference on Mobile Adhoc and Sensor Systems (MASS)*. IEEE, 2011.

[152] W. Peng, F. Li, X. Zou, and J. Wu. A privacy-preserving social-aware incentive system for word-of-mouth advertisement dissemination on smart mobile devices. In *Proceedings of IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks (SECON)*, pages 596–604. IEEE, 2012.

[153] Wei Peng, Feng Li, Keesook J. Han, Xukai Zou, and Jie Wu. *T*-dominance: Prioritized defense deployment for BYOD security. In *Proceedings of IEEE Conference on Communications and Network Security (CNS)*. IEEE, 2013.

[154] Wei Peng, Feng Li, Xukai Zou, and Jie Wu. The virtue of patience: Offloading topical cellular content through opportunistic links. In *Proceedings of IEEE International Conference on Mobile Ad-hoc and Sensor Systems (MASS)*. IEEE, 2013.

[155] Wei Peng, Feng Li, Chin-Tser Huang, and Xukai Zou. Temporal coverage based content distribution in heterogeneous smart device networks. In *Proceedings of IEEE International Conference on Communications (ICC)*. IEEE, 2014.

[156] Wei Peng, Feng Li, Xukai Zou, and Jie Wu. Behavioral malware detection in delay tolerant networks. *IEEE Transactions on Parallel and Distributed Systems,*, 25(1):53–63, Jan 2014.

[157] A-K Pietiläinen and C. Diot. Dissemination in opportunistic social networks: The role of temporal communities. In *Proceedings of International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc)*, pages 165–174, June 2012.

[158] A-K Pietiläinen, E. Oliver, J. LeBrun, G. Varghese, and C. Diot. Mobiclique: Middleware for mobile social networking. In *Proceedings of ACM SIGCOMM Workshop on Online Social Networks (WOSN)*, pages 49–54, August 2009.

[159] M. Potts. The state of information security. *Network Security*, 2012(7):9–11, 2012.

[160] B Aditya Prakash, Deepayan Chakrabarti, Nicholas C Valler, Michalis Faloutsos, and Christos Faloutsos. Threshold conditions for arbitrary cascade models on arbitrary networks. *Knowledge and information systems*, 33(3):549–575, 2012.

[161] The GNU Project. Bash: The Bourne Again Shell. `https://www.gnu.org/software/bash/`, 2015.

[162] The GNU Project. GNU Wget. `https://www.gnu.org/software/wget/`, 2015.

[163] Mykola Protsenko and Tilo Muller. Pandora applies non-deterministic obfuscation randomly to android. In *Proceedings of IEEE International Conference on Malicious and Unwanted Software(MALWARE)*, pages 59–67, 2013.

[164] Mykola Protsenko and Tilo Müller. Android malware detection based on software complexity metrics. In *Trust, Privacy, and Security in Digital Business*, pages 24–35. Springer, 2014.

[165] N. Provos. A virtual honeypot framework. In *Proceedings of USENIX Security*. USENIX, 2004.

[166] Jenny Rains. Bring your own device (BYOD): Hot or not?, 2012. URL `http://goo.gl/AV7mj`.

[167] V. Rastogi, Yan Chen, and Xuxian Jiang. Catch me if you can: Evaluating Android anti-malware against transformation attacks. *IEEE Transactions on Information Forensics and Security*, 9(1):99–108, Jan 2014.

[168] Vaibhav Rastogi, Yan Chen, and William Enck. Appsplayground: Automatic security analysis of smartphone applications. In *Proceedings of ACM Conference on Data and Application Security and Privacy (CODASPY)*, pages 209–220, 2013.

[169] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. Droidchameleon: Evaluating Android anti-malware against transformation attacks. In *Proceedings of ACM SIGSAC Symposium on Information, Computer and Communications Security*, pages 329–334. ACM, 2013.

[170] Alessandro Reina, Aristide Fattori, and Lorenzo Cavallaro. A system call-centric analysis and stimulation technique to automatically reconstruct Android malware behaviors. In *Proceedings of European Workshop on System Security (EuroSec)*, 2013.

[171] John C Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of ACM Annual Conference*, pages 717–740. ACM, 1972.

[172] I. Rhee, M. Shin, S. Hong, K. Lee, S.J. Kim, and S. Chong. On the Levy-walk nature of human mobility. *IEEE/ACM Transactions on Networks (ToN)*, 19 (3):630–643, 2011.

[173] Matthew Richardson and Pedro Domingos. Mining knowledge-sharing sites for viral marketing. In *Proceedings of ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 61–70. ACM, 2002.

[174] Rene Ritchie. iPhone 5 rumored to be getting low power, Wi-Fi Direct enabled chipset... and AirDrop?, 2012. URL `http://goo.gl/pJhTI`.

[175] Ian Robinson, Jim Webber, and Emil Eifrem. *Graph databases*. O'Reilly Media, 2013.

[176] Yusuke Sakabe, Masakazu Soshi, and Atsuko Miyaji. Java obfuscation approaches to construct tamper-resistant object-oriented programs. *IPSJ Digital Courier*, 1:349–361, 2005.

[177] K. Sakai, S.C.H. Huang, W.S. Ku, M.T. Sun, and X. Cheng. Timer-based CDS construction in wireless ad hoc networks. *IEEE Transactions on Mobile Computing (TMC)*, 10(10):1388–1402, 2011.

[178] Bhaskar Pratim Sarma, Ninghui Li, Chris Gates, Rahul Potharaju, Cristina Nita-Rotaru, and Ian Molloy. Android permissions: A perspective combining risks and benefits. In *Proceedings of ACM Symposium on Access Control Models and Technologies*, pages 13–22. ACM, 2012.

[179] J. Scott, R. Gass, J. Crowcroft, P. Hui, C. Diot, and A. Chaintreau. CRAWDAD data set cambridge/haggle (v. 2006-09-15). http://goo.gl/RJrKN, September 2006.

[180] James Scott, Richard Gass, Jon Crowcroft, Pan Hui, Christophe Diot, and Augustin Chaintreau. CRAWDAD trace cambridge/haggle/imote/infocom2006 (v. 2009-05-29). Downloaded from `http://goo.gl/u58AG`, May 2009.

[181] C.E. Shannon. Communication theory of secrecy systems. *Bell System Technical Journal*, 28(4):656–715, 1949.

[182] C. Song, Z. Qu, N. Blumm, and A.L. Barabási. Limits of predictability in human mobility. *Science*, 327(5968):1018–1021, 2010.

[183] Frank Sposaro and Gary Tyson. iFall: an Android application for fall monitoring and response. In *Proceedings of IEEE Conference of Engineering in Medicine and Biology Society (EMBC)*, pages 6119–6122. IEEE, 2009.

[184] Frank Sposaro, Justin Danielson, and Gary Tyson. iWander: An Android application for dementia patients. In *Proceedings of IEEE Conference of Engineering in Medicine and Biology Society (EMBC)*, pages 3875–3878. IEEE, 2010.

[185] Michael Spreitzenbarth, Florian Echtler, Thomas Schrek, Felix C Freiling, and Johannes Hoffman. MobileSandbox: Looking deeper into Android applications. In *Proceedings of ACM Symposium on Applied Computing (SAC)*, pages 1808–1815. ACM, 2013.

[186] Thrasyvoulos Spyropoulos, Konstantinos Psounis, and Cauligi S Raghavendra. Efficient routing in intermittently connected mobile networks: The multiple-copy case. *IEEE/ACM Transactions on Networking (TON)*, 16(1):77–90, 2008.

[187] Thrasyvoulos Spyropoulos, Konstantinos Psounis, and Cauligi S Raghavendra. Efficient routing in intermittently connected mobile networks: The single-copy case. *IEEE/ACM Transactions on Networking (TON)*, 16(1):63–76, 2008.

[188] Avinash Srinivasan, Joshua Teitelbaum, and Jie Wu. DRBTS: Distributed reputation-based beacon trust system. In *IEEE International Symposium on Dependable, Autonomic and Secure Computing (DASC)*, pages 277–283. IEEE, 2006.

[189] Vikram Srinivasan, Mehul Motani, and Wei Tsang Ooi. CRAWDAD data set nus/contact (v. 2006-08-01). Downloaded from `http://goo.gl/KRtDE`, August 2006.

[190] S.H. Strogatz. Exploring complex networks. *Nature*, 410(6825):268–276, 2001.

[191] Jing Su, Kelvin KW Chan, Andrew G Miklas, Kenneth Po, Ali Akhavan, Stefan Saroiu, Eyal de Lara, and Ashvin Goel. A preliminary investigation of worm infections in a Bluetooth environment. In *Proceedings of ACM Workshop on Recurring Malcode (WORM)*, pages 9–16. ACM, 2006.

[192] Johan AK Suykens and Joos Vandewalle. Least squares support vector machine classifiers. *Neural processing letters*, 9(3):293–300, 1999.

[193] J. Tang, M. Musolesi, C. Mascolo, and V. Latora. Characterising temporal distance and reachability in mobile and online social networks. *ACM SIGCOMM Computing and Communication Review (CCR)*, 40(1):118–124, 2010.

[194] the dex2jar team. dex2jar: Tools to work with Android .dex and Java .class files. `https://code.google.com/p/dex2jar/`, 2015.

[195] G. Thomson. BYOD: Enabling the chaos. *Network Security*, 2012(2):5–8, 2012.

[196] Trend Micro Inc. SYMBOS_CABIR.A, 2004. URL `http://goo.gl/aHcES`.

[197] Trend Micro Inc. IOS_IKEE.A, 2009. URL `http://goo.gl/z0j56`.

[198] Trustwave Holdings, Inc. BeEF injection with MITM, 2012. URL `http://goo.gl/P97Au`.

[199] A. Vahdat and D. Becker. Epidemic routing for partially-connected ad hoc networks. Technical report, Duke University, 2002.

[200] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot—a Java bytecode optimization framework. In *Proceedings of Conference of the Centre for Advanced Studies on Collaborative Research*, page 13. IBM Press, 1999.

[201] Nicholas C Valler, B Aditya Prakash, Hanghang Tong, Michalis Faloutsos, and Christos Faloutsos. Epidemic spread in mobile ad hoc networks: Determining the tipping point. In *Networking*, pages 266–280. Springer, 2011.

[202] Luke VanderHart and Stuart Sierra. Macros and metaprogramming. In *Practical Clojure*, pages 167–178. Springer, 2010.

[203] Rob J VanGlabbeek, Scott A Smolka, and Bernhard Steffen. Reactive, generative, and stratified models of probabilistic processes. *Information and Computation*, 121(1):59–80, 1995.

[204] Timothy Vidas, Daniel Votipka, and Nicolas Christin. All your droid are belong to us: A survey of current Android attacks. In *Proceedings of USENIX Workshop on Offensive Technologies*, pages 81–90. USENIX, 2011.

[205] Nicolas Viennot, Edward Garcia, and Jason Nieh. A measurement study of Google play. In *Proceedings of ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 221–233. ACM, 2014.

[206] Ricardo Villamarín-Salomón and José Carlos Brustoloni. Bayesian bot detection based on DNS traffic similarity. In *Proceedings of ACM Symposium on Applied Computing (SAC)*, pages 2035–2041. ACM, 2009.

[207] VirusTotal. VirusTotal—free online virus, malware and URL scanner. `https://www.virustotal.com/`, 2015.

[208] Fei Wang, Zhe Zhang, Hailong Sun, Richong Zhang, and Xudong Liu. A cooperation based metric for mobile applications recommendation. In *Proceedings of IEEE International Joint Conferences on Web Intelligence and Intelligent Agent Technologies*, volume 3, pages 13–16. IEEE, 2013.

[209] Xinran Wang, Yoon-Chan Jhi, Sencun Zhu, and Peng Liu. Detecting software theft via system call based birthmarks. In *Proceedings of IEEE Annual Computer Security Applications Conference (ACSAC)*, pages 149–158. IEEE, 2009.

[210] Wi-Fi Alliance. Wi-Fi Direct. URL `http://goo.gl/fZuyE`.

[211] Wikipedia. Android (operating system): History. `https://en.wikipedia.org/wiki/Android_(operating_system)#History`, 2015.

[212] J. Wright. Access pricing under competition: An application to cellular networks. *Jrnl. Industrial Economics*, 50(3):289–315, 2002.

[213] J. Wu, F. Dai, and S. Yang. Iterative local solutions for connected dominating set in ad hoc wireless networks. *IEEE Transaction on Computers (TC)*, 57:702–715, 2008.

[214] Jie Wu. Extended dominating-set-based routing in ad hoc wireless networks with unidirectional links. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 13(9):866–881, 2002.

[215] Xin Xie, Fenlin Liu, Bin Lu, and Lin Chen. A software birthmark based on weighted *k*-gram. In *Proceedings of IEEE International Conference on Intelligent Computing and Intelligent Systems (ICIS)*, pages 400–405. IEEE, 2010.

[216] Qiang Xu, George Ibrahim, Rong Zheng, and Norm Archer. Toward automated categorization of mobile health and fitness applications. In *Proceedings of ACM MobiHoc Workshop on Pervasive Wireless Healthcare*, pages 49–54. ACM, 2014.

[217] Guanhua Yan, Hector D Flores, Leticia Cuellar, Nicolas Hengartner, Stephan Eidenbenz, and Vincent Vu. Bluetooth worm propagation: Mobility pattern matters! In *Proceedings of ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, pages 32–44. ACM, 2007.

[218] Lok-Kwong Yan and Heng Yin. DroidScope: Seamlessly reconstructing the OS and Dalvik semantic views for dynamic android malware analysis. In *Proceedings of USENIX Security*, pages 569–584, 2012.

[219] Chao Yang, Zhaoyan Xu, Guofei Gu, Vinod Yegneswaran, and Phillip Porras. DroidMiner: Automated mining and characterization of fine-grained malicious behaviors in Android applications. In *Computer Security—ESORICS*, pages 163–182. Springer, 2014.

[220] S. Yang, J. Wu, and F. Dai. Efficient directional network backbone construction in mobile ad hoc networks. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 19(12):1601–1613, 2008.

[221] Zhemin Yang, Min Yang, Yuan Zhang, Guofei Gu, Peng Ning, and X Sean Wang. Appintent: Analyzing sensitive data transmission in Android for privacy leakage detection. In *Proceedings of ACM Conference on Computer & Communications Security (CCS)*, pages 1043–1054, 2013.

[222] Peter N Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms*, pages 311–321. SIAM, 1993.

[223] Peifeng Yin, Ping Luo, Wang-Chien Lee, and Min Wang. App recommendation: A contest between satisfaction and temptation. In *Proceedings of ACM International Conference on Web Search and Data Mining*, pages 395–404. ACM, 2013.

[224] J.A. Zdziarski. *Ending spam: Bayesian content filtering and the art of statistical language classification*. No Starch Press, 2005.

[225] Fangfang Zhang, Heqing Huang, Sencun Zhu, Dinghao Wu, and Peng Liu. ViewDroid: Towards obfuscation-resilient mobile application repackaging detection. In *Proceedings of ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, pages 25–36. ACM, 2014.

[226] Xiaolan Zhang, Giovanni Neglia, Jim Kurose, and Don Towsley. Performance modeling of epidemic routing. *Computer Networks*, 51(10):2867–2891, 2007.

[227] Xuesong Zhang, Fengling He, and Wanli Zuo. An inter-classes obfuscation method for Java program. In *Proceedings of IEEE International Conference on Information Security and Assurance (ISA)*, pages 360–365. IEEE, 2008.

[228] Yuan Zhang, Min Yang, Bingquan Xu, Zhemin Yang, Guofei Gu, Peng Ning, X Sean Wang, and Binyu Zang. Vetting undesirable behaviors in Android apps with permission use analysis. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, pages 611–622. ACM, 2013.

[229] Yury Zhauniarovich, Olga Gadyatskaya, Bruno Crispo, Francesco La Spina, and Ermanno Moser. FSquaDRA: Fast detection of repackaged applications. In *Data and Applications Security and Privacy XXVIII*, pages 130–145. Springer, 2014.

[230] Cong Zheng, Shixiong Zhu, Shuaifu Dai, Guofei Gu, Xiaorui Gong, Xinhui Han, and Wei Zou. Smartdroid: An automatic system for revealing UI-based trigger conditions in Android applications. In *Proceedings of ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 93–104, 2012.

[231] Min Zheng, Patrick PC Lee, and John CS Lui. ADAM: An automatic and extensible platform to stress test Android anti-virus systems. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 82–101. Springer, 2013.

[232] Min Zheng, Mingshen Sun, and John Lui. Droid Analytics: A signature based analytic system to collect, extract, analyze and associate Android malware. In *Proceedings of IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 163–171. IEEE, 2013.

[233] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. Detecting repackaged smartphone applications in third-party Android marketplaces. In *Proceedings of ACM Conference on Data and Application Security and Privacy (CODASPY)*, pages 317–326, 2012.

[234] Wu Zhou, Yajin Zhou, Michael Grace, Xuxian Jiang, and Shihong Zou. Fast, scalable detection of piggybacked mobile applications. In *Proceedings of ACM Conference on Data and Application Security and Privacy (CODASPY)*, pages 185–196. ACM, 2013.

[235] Xiaoming Zhou, Xingming Sun, Guang Sun, and Ying Yang. A combined static and dynamic software birthmark based on component dependence graph. In *Proceedings of IEEE International Conference on Intelligent Information Hiding and Multimedia Signal Processing (IIHMSP)*, pages 1416–1421. IEEE, 2008.

[236] Yajin Zhou and Xuxian Jiang. Dissecting Android malware: Characterization and evolution. In *Proceedings of IEEE Symposium on Security and Privacy (S&P)*, pages 95–109. IEEE, 2012.

[237] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative Android markets. In *Proceedings of ISOC Network and Distributed System Security Symposium (NDSS)*, pages 1–13. ISOC, 2012.

[238] Xuejun Zhuo, Wei Gao, Guohong Cao, and Yiqi Dai. Win-Coupon: An incentive framework for 3G traffic offloading. In *Proceedings of IEEE International Conference on Network Protocols (ICNP)*, pages 206–215. IEEE, 2011.

[239] Gjergji Zyba, Geoffrey M Voelker, Michael Liljenstam, András Méhes, and Per Johansson. Defending mobile phones from proximity malware. In *Proceedings of IEEE International Conference on Computer Communications (INFOCOM)*, pages 1503–1511. IEEE, 2009.

VITA

VITA

Wei Peng received his B.S. in Mathematics from Zhejiang University, China, in 2007. He enrolled the doctoral program of Computer Science in Purdue University in 2010. During his graduate study, he published seven first-author conference papers, two first-author journal articles, and one book chapter. Further information about him, including his publication and software projects, can be found at his homepage `http://cs.iupui.edu/~pengw/index.html`.