High Assurance Models for Secure Systems

Hussain M. J. Almohri

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science and Applications

April 3, 2013
Blacksburg, Virginia

Keywords: Systems Security, Authentication, Mathematical Programming, Optimization, Security Risk

High Assurance Models for Secure Systems

Hussain M. J. Almohri

ABSTRACT

Despite the recent advances in systems and network security, attacks on large enterprise networks consistently impose serious challenges to maintaining data privacy and software service integrity. We identify two main problems that contribute to increasing the security risk in a networked environment: (i) vulnerable servers, workstations, and mobile devices that suffer from vulnerabilities, which allow the execution of various cyber attacks, and, (ii) poor security and system configurations that create loopholes used by attackers to bypass implemented security defenses.

Complex attacks on large networks are only possible with the existence of vulnerable intermediate machines, routers, or mobile devices (that we refer to as network components) in the network. Vulnerabilities in highly connected servers and workstations, that compromise the heart of today's networks, are inevitable. Also, modern mobile devices with known vulnerabilities cause an increasing risk on large networks. Thus, weak security mechanisms in vulnerable network components open the possibilities for effective network attacks.

On the other hand, lack of systematic methods for an effective static analysis of an overall complex network results in inconsistent and vulnerable configurations at individual network components as well as at the network level. For example, inconsistency and faults in designing firewall rules at a host may result in enabling more attack vector. Further, the dynamic nature of networks with changing network configurations, machine availability and connectivity, make the security analysis a challenging task.

This work presents a hybrid approach to security by providing two solutions for analyzing the overall security of large organizational networks, and a runtime framework for protecting individual network components against misuse of system resources by cyber attackers. We observe that to secure an overall computing environment, a static analysis of a network is not sufficient. Thus, we couple our analysis with a framework to secure individual network components including high performance machines as well as mobile devices that repeatedly enter and leave networks. We also realize the need for advancing the theoretical foundations for analyzing the security of large networks.

To analyze the security of large enterprise network, we present the first scientific attempt to compute an optimized distribution of defensive resources with the objective of minimizing the chances of successful attacks. To achieve this minimization, we develop a rigorous probabilistic model that quantitatively measures the chances of a successful attack on any network component. Our model provides a solid theoretical foundation that enables efficient computation of unknown success probabilities on every stage of a network attack. We design an algorithm that uses the computed attack probabilities for optimizing security configurations of a network. Our optimization algorithm uses state of the art sequential linear programming to approximate the solution to a complex single objective nonlinear minimization problem that formalizes various attack steps and candidate defenses at the granularity of attack stages.

To protect individual network components, we develop a new approach under our novel idea of *process authentication.* We argue that to provide high assurance security, enforcing authorization is necessary but not sufficient. In fact, existing authorization systems lack a strong and reliable process authentication model for preventing the execution of malicious processes (i.e., processes that intentionally contain malicious goals that violate integrity and confidentiality of legitimate processes and data). Authentication is specially critical when malicious processes may use various system vulnerabilities to install on the system and stealthily execute without the user's consent.

We design and implement the Application Authentication (A2) framework that is capable of monitoring application executions and ensuring proper authentication of application processes. A2 has the advantage of strong security guarantees, efficient runtime execution, and compatibility with legacy applications. This authentication framework reduces the risk of infection by powerful malicious applications that may disrupt proper execution of legitimate applications, steal users' private data, and spread across the entire organizational network.

Our process authentication model is extended and applied to the Android platform. As Android imposes its unique challenges (e.g., virtualized application execution model), our design and implementation of process authentication is extended to address these challenges. Per our results, process authentication in Android can protect the system against various critical vulnerabilities such as privilege escalation attacks and drive by downloads.

To demonstrate process authentication in Android, we implement DroidBarrier. As a runtime system, DroidBarrier includes an authentication component and a lightweight permission system to protect legitimate applications and secret authentication information in the file system. Our implementation of DroidBarrier is compatible with the Android runtime (with no need for modifications) and shows efficient performance with negligible penalties in I/O operations and process creations.

Dedication

To that who is unique, most merciful, and is the source of all knowledge.

To my parents, my wife and kids, and my siblings who provided me with unlimited support and love throughout my graduate carrier.

Acknowledgments

I would like to thank my advisor Danfeng (Daphne) Yao who is a teacher, a supporter, and an academic advisor for me. She was always available for providing help, answering my questions, and facilitating my research. She taught me technical writing skills and most importantly how to think positively. Danfeng always encouraged me to pursue useful and practical research. An important lesson that I learned from her is: a researcher *should never waist a valuable piece of work.*

Among the few people whose inspirations changed my academic carrier is Eli Tilevich. I first met Eli Tilevich in his research course and found out about his teaching philosophy: *a graduate student should teach him/herself.* He never hesitated answering my questions and providing me with inspirations. He introduced me to state of the art research in distributed software systems, some technical writing tricks, and the art of conducting useful research.

The work presented in this dissertation could not be completed without collaboration with Dennis Kafura, Layne T. Watson, and Simon Ou. Dennis Kafura was very helpful in providing critical feedback, brainstorming ideas, and teaching me his golden rules of writing. Similarly, Layne T. Watson was an amazing collaborator who introduced me to state of the art techniques in the world of optimization and some technical writing guidelines. My M.S. advisor, Simon Ou, provided help and support especially for using MulVAL.

Finally, I would like to thank Michael Hsiao and Chris North who accepted to be part of my committee. Michael and Chris provided useful and productive feedback during my proposal and research defense exams that positively impacted the content of this dissertation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Today's computing systems are growing in size and complexity. As more functionality is demanded from systems and networks, the complexity and interconnection, and as a result, the security risks increase. Two main problems contribute to increasing the risk in a networked environment: (i) vulnerable devices, and, (ii) poor security and system configurations.

Complex attacks on a large network often target specific goals and comprise several several steps to reach the goal. These attacks are only possible with the existence of vulnerable intermediate hosts, routers, or mobile devices (that we refer to as network components in this dissertation) in the network. In fact, modern mobile devices (i.e., devices that repeatedly join untrustworthy cellular and wireless networks) impose an increasing risk on large networks. These devices are vulnerable enough to be under the control of attacks, yet, mobile devices are granted access to wireless networks within organizations, without appropriate defense plans.

On the other hand, lack of systematic methods for an effective static analysis of an overall complex network results in inconsistent and vulnerable configurations at individual network components as well as at the network level. For instance, inconsistency and faults in designing firewall rules at a host may result in enabling more attack vector.

This work presents a hybrid approach to security by providing two solutions for analyzing the overall security of large organizational networks, and a runtime framework for protecting individual systems against malicious code execution. We realize the need for advancing the theoretical foundations for analyzing the security of large networks. We also observe that to secure an overall computing environment, a static analysis of a network is not sufficient. Thus, we couple our analysis with a framework to secure individual network components including high performance machines as well as mobile devices that repeatedly enter and leave networks.

In the remaining of this chapter, we first present a discussion on both problems, research challenges, and the shortcomings of current approaches. Then, we discuss our technical contributions and approaches to the problems.

1.1 Motivations and Challenges

1.1.1 Quantitative Risk Analysis

Analysis of security risk requires modeling of attack scenarios and reasoning about the dependencies among attack paths. Having a comprehensive model about possible attack scenarios and their dependencies, we can develop a quantitative measurement of the risk associated with a particular network configuration. The results of a quantitative measurement of the risk can be used in systematic methods for optimizing the security configurations in a network.

A reliable quantitative measurement of security risk faces several technical challenges:

1. Developing a rigorous model that can express various network configuration elements at a high degree of granularity.
2. The results of any quantitative measurement
   (a) must be theoretically sound with clear semantics. This is necessary for developing deterministic algorithms for computing the results.
   (b) must provide the basis for computing an *optimized network configuration.*
3. In many environments, sampling security sensitive data is difficult due to regulations and technical difficulties (e.g., correlating log data to specific attacks). Thus, quantitative analysis of security risk must be able to provide useful conclusions when lacking observable attack data.
4. Large networks are highly dynamic. System configurations change, network components' connectivities vary, and availability of the devices changes, especially in case of mobile devices.

Attack graphs are well known formalisms that provide the required reasoning about attack scenarios and their dependencies. An attack graph is a dependency directed graph in which an edge between two nodes represents the causality relationship between the two nodes. Attack graphs show the logical reasoning and dependencies between two attack steps. For instance, what technical conditions and steps are required to compromise a particular host using a specific vulnerable network service?

Quantitative assessment of a network based on attack graphs has been previously researched [44, 47, 48, 63, 64, 78]. However, we observe that the current state of the art on attack graph analysis lack several important components.

First, an attack graph provides a limited view of the security of a network. This limited view shows the possible ways one (or more) attackers may pursue a specific attack goal. An intuitive extension of this analysis is to provide a quantitative assessment on the nodes of an attack graph. While several mechanisms exist for providing a quantitative assessment of attack graphs (for example, using Bayesian belief propagation [71, 31, 35, 89], basic laws of probability [63, 85], and ranking algorithms [59, 75]), these existing method lack rigorous theoretical foundations.

Second, existing methods to quantify attack graphs do not consider dynamic aspects of a networked environments. For instance, when modeling an attack scenario, it is always assumed that all network properties are *available* to attackers. However, the availability of network components is always subject to problems such as connection downtimes and systems unavailabilities. Thus, virtually all the existing attack graph models assume a network with known and fixed configurations in terms of the connectivity, availability and policies of the network services and components.

Third, quantitative assessment of attack graphs can be practical with a systematic approach to use the computed risk values for hardening the security of a network. For example, network administrators may choose an *optimal placement* of candidate security solutions in order to minimize the risk associated with a network. Such an optimization problem requires *i)* the ability to model placement options of a component in an attack graph and *ii)* the ability to compute the optimal impact of various options on the protected resources. Neither requirement is addressed in existing attack graph models. Our work fills in this gap. One of the contributions of our work is to model and solve this optimal placement problem in attack graphs.

As discussed in Section 1.2, we address these challenges by developing a rigorous risk measurement model. Our model takes into consideration the dynamic aspects of a network. As a novel contribution, we develop a mathematical programming model to find an optimized network configuration according to the results of our quantitative risk measurement.

In the following section, we discuss the challenges associated with the design and development of a runtime system to secure individual network components.

1.1.2 Application Authentication

Mainstream operating system kernels (such as Linux/Unix) often enforce minimal restrictions on the applications permitted to execute, resulting in the ability of malicious programs to abuse system resources. Stealthy malware running as stand-alone processes, once installed, usually can freely execute privileges provided to the user account running the process.

A well-known approach to protecting systems from malicious activities is through the deployment of mandatory access control (MAC) systems. Such systems often provide the kernel with access monitoring mechanisms as well as policy specification platforms. The user decides on the policies and the various access rights on system resources. Existing MAC systems such as SELinux [55], grsecurity [1], and AppArmor [40] enable the user (or the system administrator) to express detailed and powerful policies. These solutions are often implemented using the Linux Security Modules [88] to monitor access to selected system resources, and apply the specified policies to the corresponding processes.

The above security solutions belong to the category of authorization. However, authorization mechanisms alone are not sufficient for achieving system assurance. Our thesis is to argue and demonstrate that the kernel must also have secure mechanisms for authenticating processes where the identity of a process can be proved. User authentication through techniques such as password or public-key cryptography is common in multi-user system or network environments. Many user authentication techniques exist in the literature. Yet, process authentication, i.e., how to prove a process is indeed what it claims to be, has never been systematically reported in the literature.

Because of the complexity of the operating system's tasks in managing a large number of diverse applications, ensuring the authenticity of the basic operating data is becoming increasingly important. With modern attack models, system information whose security is usually taken for granted needs to be re-examined and re-evaluated. The cryptographic provenance verification work in [90] points out the need for the kernel to ensure the authenticity of origins of data flows that are consumed by the system. For example, the data may be user inputs or traffic flows. Recent assured digital signing work in [20] describes methods for the integrity protection and authenticity verification of a signing agent on a host for creating digital signatures. It points out the differences between a human signer and a program signer and the system challenges associated with realizing a trustworthy program signer. Their solution extends the attestation service of the hardware trust platform module (TPM). Our work demonstrates another case of hardening the system by re-examining the fundamental process identification mechanism.

In this work, we introduce the novel idea of process authentication. Our process authentication model recognizes applications as individuals that are represented as processes in the memory. In this model, the operating system enforces a mandatory authentication on all user processes.

Process authentication is different and independent from process identification and requires stronger properties, for example preventing spoofing application identities. In contrast, identification is a way to describe a principal. Process IDs and process names are identifiers for processes in an OS environment. Typically, these process identifiers are generated by the system after examining the executable file names and installation paths of processes. This examination of executable file names and installation paths is the simplest form of process authentication. These simple authentication procedures are insecure against existential forgery attacks by malicious software, which we explain in details in Table 4.1. AppArmor (based on the Linux Security Modules) recognizes processes through the application's installation path, based on which access rights are enforced. However, process authentication based on the installation path is weak. Without secure process authentication, malware may impersonate legitimate applications and abuse system resources, thus violating system assurance.

Our process authentication model is primarily designed for Linux-based systems, and thus, we extend our work to apply the same model in Android. In the following section, we describe an overview of the challenges facing the Android platform and how our process authentication can address the security problems.

1.1.3 Authentication in Mobile Systems

An important feature of the Android operating system is that it relies on mature technologies such as the Linux kernel. The Linux kernel as the underlying layer in the Android software stack enables efficient system services such as process and memory management. In particular, Android's Dalvik runtime system relies on Linux process creation when launching an application or a service, making the runtime system as the parent process of all user application processes in Android.

With the assistance of the Linux kernel, Android implements a fundamental security feature called application sandboxes. Android's approach is to install each application with an isolated sandbox to protect its data from unauthorized accesses by other applications, using file system permissions. Another aspect of Android's application sandboxes is creating separate application processes at runtime to isolate the execution of applications. This isolation, in principle, provides a way to protect application's data, ensure its execution without malicious interference, and protect interprocess communications based on permissions designated by application developers.

Similar to other systems that use the Linux kernel, Android suffers from numerous vulnerabilities that cause the Android's application sandboxes to fail. For example, the `Gingerbreak` exploit affected the popular Android version 2.3 that enabled a malicious application to gain `root` privileges. When the malicious application escalates its privileges, it completely bypasses Android's application sandboxes. Malicious applications can then exploit other vulnerabilities above the Linux kernel level such as the ones in Android's inter-component communications [16, 29]. A malicious application may launch attacks to misuse system resources with the goal of spying on users, stealing private user data, and causing financial loss [30].

To combat the security vulnerabilities in Android, current state of the art focuses on a wide range of approaches. There have been proposals for using virtual layers to separate execution domains [4], exploiting control-flow capabilities to limit malicious access to data [27], and protecting inter-component communications in Android's Binder [25]. Despite that Android heavily relies on Linux to provide a security sandbox mechanism, with a few exceptions (e.g., [70]), existing security solutions barely attempt to enhance the security capabilities of the Linux kernel in Android. Moreover, traditional security solutions for the Linux kernel, that are used in other platforms (e.g., [45, 74]), are not directly usable in Android. Hardware and software architectural differences, for example, require current Linux security solutions to be revised for porting to Android.

Our key observation is that critical vulnerabilities in Android share a root cause: *malicious applications that are installed and executed without the user's consent.* Neither the Linux kernel nor Android's runtime incorporate a mechanism to authenticate the origin and authenticity of application processes. As a result, malicious applications can run in the context of independent Linux processes and misuse system resources to achieve their attack goals.

We extend the idea of process authentication (Section 1.1.3) to defend against kernel level attacks on Android. Our extended process authentication model regards Android application processes as individuals that must be authenticated before using system resources. In this model, legitimate applications are given credentials that are used for authentication at runtime. When enforcing process authentication, unauthorized processes that do not possess credentials fail to authenticate. This failure results in denying access to critical system services provided by the kernel.

The main property of our process authentication model is enabling the detection of unauthorized processes at runtime. To demonstrate this property, we design and implement DroidBarrier, a runtime system, that uses process authentication to detect and prevent the execution of malicious applications without compromising the openness of the Android platform. DroidBarrier enforces a mandatory authentication on all processes for any application in Android. Using this mandatory authentication enforcement, DroidBarrier *guarantees* the detection of processes that fail to authenticate, and, prevents their subsequent attacks.

To evaluate the performance DroidBarrier, we conducted several experiments that required porting selected Unix and Linux benchmark suites to Android. According to our experiments on a physical Android tablet, DroidBarrier performs efficiently with negligible performance penalties in process creation, and, a maximum and minimum I/O performance penalty of 12.92% and 3.76%, respectively.

1.2 Contributions

As discussed in Section 1.1, we address two major problems in this dissertation. First, we study the problem of providing an analytical framework to quantitatively measure the security risk in large networks, and, to provide systematic optimized recommendations to harden the security against various attacks.

Second, the multi step attack scenarios that can occur in networks share a main cause: vulnerable machines in the network that allow execution of code with arbitrary privileges. We believe to have a comprehensive solution to important security challenges that face today's modern computing systems, both analytical and protection tools at various levels (network or individual hosts) is needed. We realize the importance of providing a security solution for both conventional desktop operating systems and mobile platforms.

In the following we present a highlights of the contributions by this work.

1. Quantitative analysis and security hardening.
    (a) **Success measurement model.** We propose a rigorous probabilistic model for measuring the chances of successful attacks on various components of large dynamic corporate networks. Our success measurement model provides ways for expressing dynamic network configurations and uncertainties in attack decisions. This ability to model dynamic networks with uncertainties allows our model to substantially improve existing static attack graph analysis work (e.g., [71, 24, 86]).
    (b) **Computation of the values.** Our success measurement model introduces the novel idea of transforming an attack graph into a (i) linear system of equations for simulating various attack scenarios, and, a (ii) a mildly nonlinear mathematical programming model that aims for computing the maximum chance of success for a particular attack goal. We solve the first system with existing linear system solving algorithms (such as LU factorization) and the second nonlinear model with an algorithm based on the sequential linear programming for approximating the solution for nonlinear models.
    (c) **Measuring availability and considering mobile devices.** We formalize the availability and connectivity of devices when analyzing attacks. Capturing availability and connectivity is a key abstraction that allows our success measurement model to also quantify the existence of mobile devices in a network.
    (d) **A systematic approach for optimizing security configurations.** Analysis of large networks is importance in understanding the major network vulnerabilities and misconfiguration. We extend this analysis by developing an mathematical optimization model that extends our success measurement model to consider security improvement options (such as installing security defenses and reconfiguration of the network) with the objective of minimizing the security risk in the network.
    (e) **Evaluation.** We evaluate our models for the purpose of verifying the accuracy of our results. Our evaluation is performed based on the data gathered from an open large network. The gathered data is based on network scanning tools without using administrative or intrusion detection logs.
2. Host-based security defense.
    (a) **Process authentication model.** We present a process authentication model that is compatible with desktop Linux systems and the Android's runtime system. We discuss the security requirements and guarantees of our model and present the general operations needed to implement the model.

(b) **Process monitoring and runtime protection.** We design a runtime system that is capable of detecting unauthenticated processes, and, mediate the authentication between a process and the kernel. Our runtime system is compatible with Android's software stack. Our runtime system is also capable of authenticating applications that can implement our challenge-response protocol. By implementing our protocol the application needs to be slightly modified, however, it can authenticate itself to the operating system directly.

(c) **Applicability.** Our process authentication mechanism has important applications in preventing system access from being abused by malware. It can be either used alone in the OS or integrates with existing system authorization solutions such as SELinux [55] to support fine-grained process-level access control. We demonstrate its practical application in preventing unauthorized system calls.

(d) **DroidBarrier.** DroidBarrier implements our process authentication protocol for the Android platform. We implemented and tested DroidBarrier for a physical Android device. Our implementation consists of patches to the kernel and a set of tools for process monitoring, authentication, and a lightweight access control system in the kernel. DroidBarrier is fully compatible with the Android runtime and does not require any modifications to existing applications or the Dalvik virtual machine.

(e) **Evaluation.** We evaluate the performance for both of our implementations for desktop Linux and the Android platform. Our implementation causes negligible performance penalties, which makes them practically usable.

In the rest of this dissertation, in Chapter 2, we discuss the necessary background for discussing our research approach, and a highlight of related work on network security analysis and security of desktop and mobile platforms. In Chapter 3, we present our approach for quantitative analysis of large enterprise networks, with an emphasis on probabilistic modeling, computational algorithms, and optimization models. Chapter 4 presents a general process authentication model, discusses an abstract design of a corresponding runtime environment, and demonstrates the architecture, implementation, and evaluation of the Authentication Application (A2) system for general Linux-based desktop systems. Continuing the discussion on process authentication, the model is extended in Chapter 5, and a design, implementation, and evaluation of DroidBarrier for Android is discussed. Finally, in Chapter 6, we present our concluding remarks and discuss potential future research.

Chapter 2

Background and Related Work

In this chapter, we present an overview on the concept of attack graphs and discuss some of the previous work that involve quantitative analysis of networks using attack graphs. In addition, we describe the related work on securing systems in general and securing the Android platform specifically.

2.1 Quantitative Security Analysis

In this section, we first describe the idea of an attack graph and its applications in computer security problems. Then, we present two major categorizes of security assessment models, namely probabilistic metrics and attack graph ranking, along with a discussion on existing work for providing security improvement and mitigation. In the following, we compare our work with a highlight of existing solutions in quantifying and analyzing network security threats.

2.1.1 Attack Graphs

Attack graphs are directed graphs that are also known as dependency graphs. Nodes of an attack graph usually refer to attack goals. A rooted attack graph has a single ultimate attack goal (also referred to as attack target in this dissertation). This attack target is preceded by other intermediate attack goals that are necessary for the attack to succeed. In this work, work on a category of attack graphs referred to as logical attack graphs [65]. Logical attack graphs include two more types of nodes. Fact nodes represent ground facts corresponding to particular network configuration elements. For instance, the fact that MySQL is running on port 3306 on machine A, can be represented by a single fact node. The other type of nodes in attack graphs is rule nodes. Rule nodes are first order predicates that describe how a particular action can be undertaken in an attack. For instance, a rule node may refer to the predicate

$$\text{attack}(A, T) \leftarrow \text{exploit}(A, S, ...) \wedge \text{connected}(A, P, ...), \tag{2.1}$$

meaning that if there is a certain vulnerable software S running on machine A, and machine A is accessible through port P, then attack T is possible.

Attack graphs are useful to show if an attack target is reachable and the possible ways to reach the target. This is a useful information when analyzing the security of network. However, attack graphs provide limited utility if not coupled with additional analytical elements that make the analysis of large networks a reality.

An attack graph can be utilized as a visualization tool. However, this visualization becomes unusable as soon as the network grows larger than a few connected machines. In practice, corporate networks contain hundreds of machines with complex firewall, machine, and routing configurations, which makes the manual inspection of the corresponding attack graph nearly impossible.

Attack graphs are also lack an important analytical element, which is a method of quantification with rich semantics. A quantified attack graph enables a comparison among the various nodes in the attack graphs. This comparison assists the system administrators to better understand the bottlenecks of the network and point out the most vulnerable components in it. Nevertheless, this quantified analysis itself must provide a basis for computing optimized network security configurations.

In Sections 2.1.2 and 2.1.3, we argue that the existing methods for quantifying attack graphs lack (i) rigorous theoretical foundations, and, (ii) are not scalable to be used as a foundation for optimizing the security of the network.

2.1.2 Probabilistic Metrics

A work by Wang et al. [85] considers a probabilistic model for computing a security risk metric using attack graphs. The work in [85] discusses an interpretation of the metric and a heuristic to compute the metric. Our success measurement model generalizes this work by capturing the uncertainty in attacker's choices (discussed as a random selector in Section 3.2).

Bayesian analysis of the security of an enterprise network has been previously investigated [71, 35, 89]. Bayesian analysis of attack graphs differs with our success measurement model in that our model does not require the knowledge of conditional probabilities. In [35], a dynamic Bayesian network model was proposed that is capable of incorporating temporal factors. Bayesian threat probability based on security and organization-specific knowledge as well as attacker profile is discussed in [31]. Xie et al. [89] introduced a Bayesian model that adds a node to the Bayesian network indicating whether or not an attack has happened. Although this extension improves the models in [35], it does not capture the various possibilities of attack paths taken by an attacker before reaching an intermediate attack goal.

None of the previous work considers the effect of device availability on open networks. Furthermore, optimized network configurations and improvement in our work has not been previously studied. Bayesian methods are powerful in computing unobserved facts, such as predicting possible threats. It remains unclear how Bayesian methods can be used to support variability in attacker's decisions, device availabilities, and the effect of mobile devices.

2.1.3 Attack Graph Ranking

PageRank is an algorithm proposed by Page et al. [66], which is used to rank important web pages. The idea of page rank is based on a random web surfer that follows the links on web pages and compute a priority rank. Due to the similarity between link graphs and attack graphs, a variety of successful research has proposed the use of a modified version of PageRank to rank attack graphs. A ranking algorithm based on PageRank [66], AssetRank [75] was proposed to rank any dependency attack graph using a random walk model. AssetRank is a generalization of PageRank extending it to handle both conjunctive and disjunctive nodes. AssetRank is supported by an underlying probabilistic interpretation based on a random walk. Mehta *et al.* propose a ranking method using state enumeration attack graphs [59]. The idea of PageRank is applied to state enumeration attack graphs with a modified interpretation of the ranking. Attack graphs based on model checking have been proposed in [78] formalizing an intrusion attack in a finite state model. Authors in [78] do not propose a complete attack graph ranking method. Instead, a method to compute minimal critical attack assets based on user-specified metrics has been introduced.

Other approaches to security assessments include a goal-motivated attacker model based on a Markov decision process [92], a weakest-adversary approach to ranking attack graphs [68], a generic framework for an attack resistance metric [87], and an enterprise IT risk metric using CVSS scores [8].

The aforementioned techniques do not consider the effect of device availability in their vulnerability ranking algorithms. In addition, recommendation of security hardening options is not addressed. While we provide a systematic way to find optimal recommendation options, other researches have not provided such a mechanism.

2.1.4 Security Hardening

The authors in [71, 24] formulated the optimal security hardening problem as a multi-objective optimization problem. The method in [71, 24] computes a subset of security improvement options. We take a different approach by formulating the problem as a mathematical programming problem to find the best placement of a set of security improvement options on a subset of rule nodes of an attack graph. In contrast to a genetic algorithm used in [24], we use a state of the art technique, named sequential linear programming, which is scalable and efficient [7, 67, 13, 32].

The work in [2] defines a cost function to measure the effect of various network reconfigurations. The proposed method follows a forward search approach to assess the result of network hardening options.

In [62] Noel and Jajodia presented a greedy algorithm to solve the hard problem of the best placement of IDS sensors in a network using attack graphs. It is to find a minimal number of sensors that can cover all critical attack paths. The benefit of this method is to reduce the cost yet increase the effectiveness of IDS sensors. In comparison, our improvement recommendation method aims to find the placement that best reduces the expected chance of a successful attack. Both methods complement each others as they are under different optimization requirements and constraints.

Another method for hardening the security of a network was presented in [86]. The authors describe a method on finding the initial conditions that need to be removed to improve the network security. Our improvement recommendation method provides a more comprehensive solution by studying how adding improvement options helps hardening the security of the network.

Huang et al. proposed a method for distilling critical attack graph surface iteratively through minimum-cost SAT solving [41]. The presented method is useful in finding the most critical attack path, which can be considered later for hardening the security of the network. Such a result can be used to guide our improvement recommendation method to consider hosts found on a critical path.

In [63], a probabilistic metric was introduced. The core component of the proposed work is to simulate the attack scenario and provide recommendation options to find a better configuration of the network. Comparably, our improvement model is not limited to making an optimal choice between available configuration options. Our work goes further by considering additional security hardening options (such as installing an IPS) and finding an optimal recommendation accordingly. Our proposed model finds an optimal recommendation based on a nonlinear program and is not limited to simulation results. In [63] the authors provided a method to quantify the attack graph and simulate attackers' choices to compute an improved reconfiguration. While being a valuable approach, the proposed method does not take into account the the availability of machines and uncertainty in attackers' decisions.

Ingols et al. [43] extended attack graphs to model zero-day vulnerability attacks, client-side attacks, attack reachability, and firewall rules to mitigate the attack.

The authors in [63, 62, 86, 41] propose methods for hardening the security of networks. However, the recommendation of security improvement options is not studied there.

Our work is distinguished from the existing work presented above with three novel contributions. First, we provide a general mechanism for capturing network component availabilities (i.e., the variability in a device's network reachability), which also leads to quantifying and analyzing possible threats from mobile devices such as laptops, tablet computers, and cellphones. Second, our probability calculation scheme is general enough to allow performing various levels of success probability analysis by introducing variable attack steps as part of success probability computation. Third, we complete the analysis of network security threats by providing a sound and computationally efficient security improvement recommendation technique that is capable of finding optimal network configurations as well as optimal placement of security solutions in the network.

2.2 System Security

In this section, we present an overview of current practice for securing conventional desktop systems as well as the emerging mobile platforms. We point out the difference between our approach and the existing ones.

2.2.1 Access Control

Mandatory access control (MAC) systems specify fine-grained policies for the installed applications. These policies are typically administered by a power user (such as the `root` user in UNIX-based systems) to control the behavior of the applications. A well-known MAC system is SELinux [55]. SELinux assigns applications to domains and tags executable files with their appropriate domain information. At runtime, SELinux monitors the access by all processes and enforces the predefined access policies by binding the process to an appropriate domain and deciding on the right policies. An alternative to SELinux, grsecurity [34] provides sophisticated memory protection mechanisms such as enforcing read-only memory pages.

Policy-based systems such as SELinux are found to be difficult to use by end users [76], and lack a general application authentication mechanism. In A2, we provide the first process authentication mechanism. It is independent of a particular user identity and does not rely on dynamic features such as a process ID, yet (in its core functionality) does not depend on complex policy specification systems.

In [72, 73], the authors propose the use of message authentication code in monitoring system calls. By using an automated method binary rewriting, all the system calls functions calls are modified to include a message authentication code as extra arguments. The message authentication code is generated using a key that is available to the kernel. At runtime, the kernel uses the key to verify the code against the actual system call made by the application in order to detect possible modifications to the application's behavior. The presented work is limited to providing identities (the HMAC) to individual function calls to system calls in an application. Thus, it does not provide an identity to the application itself.

Systems like `ShellOS` [80] take a different approach by tracing memory reads and writes, using hardware virtualization techniques, with the goal of detecting malicious payloads. The techniques used in `ShellOS` are promising for mobile platforms provided the availability of proper hardware virtualization and higher performance capacities for mobile devices.

2.2.2 Remote Attestation

The Integrity Measurement Architecture (IMA) is a mechanism to provide attestations about the integrity of the kernel and the running programs for a trusted remote verifier [74]. In IMA, the kernel maintains an aggregation of user programs' and files' checksums (i.e., the hash of the file's contents) in the memory. The integrity of the list in the kernel's memory is maintained using TPM. The checksums of user programs' are communicated to the remote party to perform the necessary verification. In [61] a similar approach is taken to apply IMA on mobile operating systems.

The work in [74] is enhanced by PRIMA [45] to take advantage of information flow integrity for verifying and controlling user programs' inputs. Specifically, PRIMA forces the flow from a low integrity program to a higher integrity program to pass through a filter. ReDAS approaches the problem by providing attestation of dynamic program features to remote parties [52]. In the proposed methods, the integrity of the kernel is assumed to be established based on TPM. Then, the kernel keeps track of dynamic program features by a static analysis of the program binary. For instance, ReDAS makes sure that the return address of a function points to the instruction following the `call` instruction. In contrast, our authentication model uses the device's capabilities (such as the isolation provided by the kernel) and does not rely on third party attestation.

2.3 Android System Security

We start by presenting system security approaches for Android including systems to secure IPCs, use of virtual machines and isolation techniques, implementing access control systems within the Android operating system, and static analysis techniques for classifications of applications.

2.3.1 Protecting Interprocess Communications

Quire is a cryptographic solution that annotates interprocess communications (IPC) in Android to provide provenance assurance to the receivers of the IPC messages [25]. Quire provides authentication only at the IPC and remote procedure call (RPC) levels. Quire's strategy is to authenticate every IPC and RPC and enable the applications to use the provenance information provided by Quire. Our authentication model in DroidBarrier uses a different strategy. First, we authenticate the whole Linux processes. This authentication can provide provenance at any lower level such as function calls, IPCs, and RPCs. Second, our authentication policy is to only authenticate registered applications, and as a result, we consider an unauthenticated process potentially malicious.

XManDroid is a framework to monitor inter component communications (ICC) in Android [11]. XManDroid tries to protect the system from privilege escalation attacks by using a decision maker engine that queries a security policies database. The target of attacks in XManDroid is Android-level privileges that an unauthorized application tries to acquire. Our work presents a solution that targets privilege escalation attacks at the Linux kernel level to fundamentally isolate malicious applications at runtime.

2.3.2 Virtualization and Access Control

VMWare Mobile Virtualization Platform (MVP) provides a sophisticated virtualization environment for mobile operating systems [4]. MVP is a type 2 hypervisor and is capable of isolating restricted and normal execution environments. Bare Metal Hypervisor is a concept introduced in [38] to run security sensitive applications in trusted and isolated environments. The authors argue that formal verification of a small code-base in the hypervisor is feasible and provides formal security guarantees. Also Cells [21] runs multiple virtual phones using a shared underlying physical phone to provide isolation. Other systems such as TrustDroid [12] isolate applications in isolated logical domains. In DroidBarrier, we do not use virtualization to provide isolation at runtime. This helps in achieving better performance and usability.

Mandatory access control (MAC) systems can be integrated with our DroidBarrier to provide fine-grained application-level access rights enforcement. A well-known MAC system is SELinux [55]. SELinux assigns applications to domains and tags executable files with their appropriate domain information. The authors in [77] attempted to imported SELinux to Android. Moreover, TOMOYO Linux modified a vanilla Linux kernel to implement a Mandatory Access Control system for the Android OS based on behavioral analysis [69].

2.3.3 Static Analysis

Static analysis is a complement of our work that provides the user with classification information about the applications. Various indicators such as existence of dynamic class loaders can be checked prior to installation of applications to have a higher confidence that the application does not contain potentially malicious functionalities. DroidBarrier provides protections when the system or installed legitimate applications are compromised.

Kirin security service uses security requirement engineering techniques for finding fine-grained security rules based on which malware applications are classified [28]. The goal of this service is to inform the user about the nature of the applications at installation time. Another work, RiskRanker [37], provides an analysis of the Android applications for finding zero-day Android malware. General static analysis techniques such as [26] can also be used to provide information on installing applications.

Chapter 3

Success Measurement Model

3.1 Problem Statement and Overview

For hardening the security of the network, one important problem is to solve the problem of optimal placement, which is to minimize the security risk for the ultimate attack goal via the optimal placement of one or more security products.

The computation may be subject to placement constraints such as limiting the deployment of security products to a subset of machines in the network. One needs to be able to quantitatively compare all the possible configurations or placement options. Specifically, the problem is to compute the most effective placement of T security improvement products with K_τ placement options for each improvement option τ. We measure the effectiveness of a security product based on the percentage reduction in the chance of a successful attack on a particular target in the network. The direct search method for solving this combinatorial problem has a worst case factorial complexity in number of placements. There are two technical problems for hardening the security of a network: *i) modeling*, i.e., how to express and model the placement options in attack graph and *ii) computing*, i.e., how to efficiently compare these different configurations.

The ability to model configuration options is a general and powerful technique for representing uncertain properties (e.g., availability) in attack graphs. Some devices (e.g., laptops) may not connect to the organizational network all the time. Yet, these mobile devices may have vulnerabilities partly due to their mobility (e.g., infected in an outside network). The network administrator may want to realistically assess the effect of mobile devices on network security, i.e., to what degree mobile devices enable attacks. The analysis requires one to represent the uncertainty in the availability of devices. Existing attack graph ranking algorithms typically assume a static graph and static propagation of probabilities, which are not suitable to solve these problems.

We formulate the problem as a minimax problem and solve it. In the minimax problem, the objective function is the expected chance of success for the ultimate attack goal $\mathcal{G}$. We model the optimization problem as a nonlinear programming problem and solve it with a nontrivial linear approximation algorithm referred to as sequential linear programming (SLP) in Section 3.5.1.

Another motivation of our work is the need to quantitatively assess the impact of mobile devices on organizational security. We refer to such networks as dynamic networks (i.e., open networks with frequent changes in their configurations and properties), as the availability of machines is no longer fixed. Compared to the existing work, our success measurement model can analyze the effect of mobility through encoding possible attack paths from mobile devices. We use random variables with certain probability distributions to encode the availability of mobile devices in our computation.

We develop our success measurement model to assess the security of a network beyond the initial information provided by an attack graph. That is, we aim to calculate the *expected chance of a successful attack* (ECSA) for an attacker given our belief about the fact nodes in an attack graph. Our success measurement model is for quantifying the vulnerabilities of networked components and resources. It does so through computing the expected chances of successful attacks (ECSA) on attack graph nodes. We define *the ECSA of a node u* as the expected chance that a attacker successfully exploits the node u of the attack graph, given certain initial belief. There may be multiple attack paths (representing multiple attack choices) to reach u. Thus, the ECSA for u must represent an estimation of the success as a combined value of success for previous nodes.

The existing work in attack graph analysis is limited to a static analysis of attack patterns and a fixed view of network configurations. Moreover, no prior work has addressed the problem of finding the best placement of security products in dynamic organizational networks. In particular, our approach advances state of the art by

- introducing a rigorous probabilistic model that capture uncertain attack decisions and machine availability, which can use computational methods to provide detailed analysis of attack under various scenarios, and,
- being the first scientific attempt to develop a model for computing an optimal reconfiguration of the security policies, and potential security defenses.

Our probabilistic computation of attack success requires two types of inputs, and outputs the expected chances of successful attacks against network components and resources (Figure 3.1.2). One type of the input is an attack graph generated by a attack graph generator tool. The other input is a set of initial belief values associated with the ground facts which correspond to fact nodes of the attack graph. Facts related to network configurations and vulnerability data are associated with success probabilities and constitute our initial belief. Our model requires a minimal set of initial belief values (Section 3.4) that can be assigned

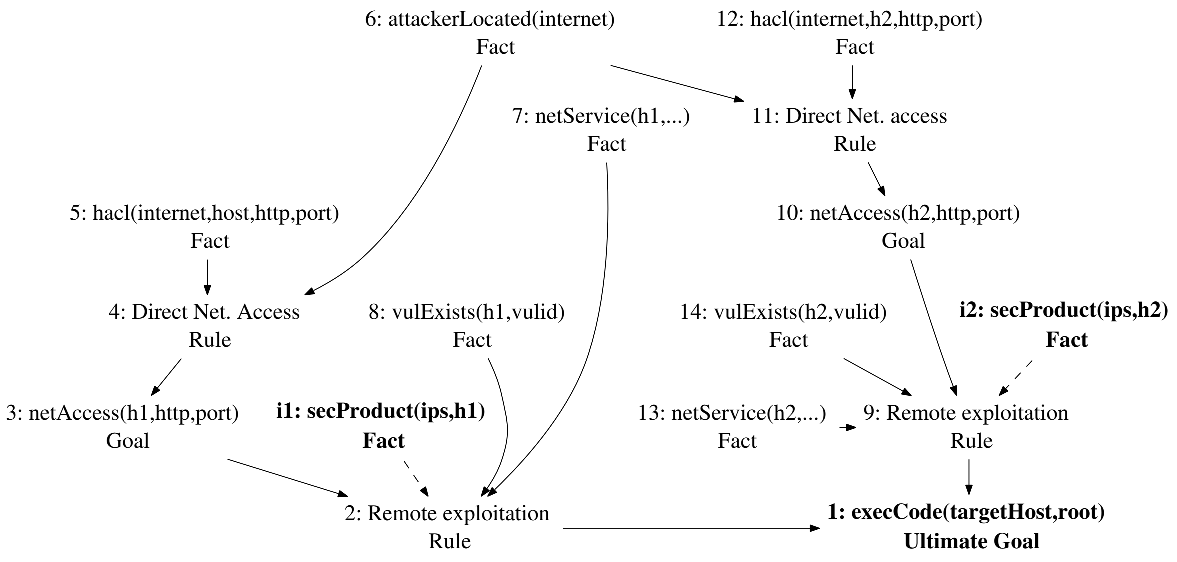


Figure 3.1.1: A simplified attack graph with goal nodes, rule nodes, fact nodes, and security improvement options. Node 1 is the ultimate attack goal of executing code on the target host, Nodes 3 and 10 are intermediate attack goals. Nodes 2, 4, 9, and 11 are attack rules. Fact nodes indicate ground facts such as a piece of configuration or a software vulnerability. There are two candidate placements (indicated by the nodes i_1 and i_2) of a single intrusion prevention system (IPS) in the network. Either host1 or host2 can have the IPS. The problem is to minimize the probability of a successful in achieving the ultimate attack goal by choosing one placement between i_1 and i_2.

according to an estimation obtained from experts' knowledge and standard vulnerability scoring systems such as the common vulnerability assessment system (CVSS) [60].

The expected chance of successfully exploiting attack goals is computed by propagating the initial belief about the fact nodes (probabilities of success at the fact nodes) through possible attack routes in the graph. The propagation of the values formalizes a natural flow of attack steps throughout the network by considering uncertain attackers' decisions.

In Section 3.2, we provide a nonlinear programming problem with the objective of finding the most vulnerable network components in the network. We solve the nonlinear programming problem using a state-of-the-art technique called sequential linear programming.

Our success measurement model forms the basis for solving the problem of computing an optimal placement. Figure 3.1.1 shows two improvement Nodes i_1 and i_2 added to the attack graph. Fact Node i_1 corresponds to the placement of an intrusion prevention system (IPS) at Node 2, and fact Node i_2 corresponds to the placement of an IPS at Node 9. Each of the placements can make the target less vulnerable. The optimal placement is to find the placement of one or more improvements (i.e., security products) that best lowers

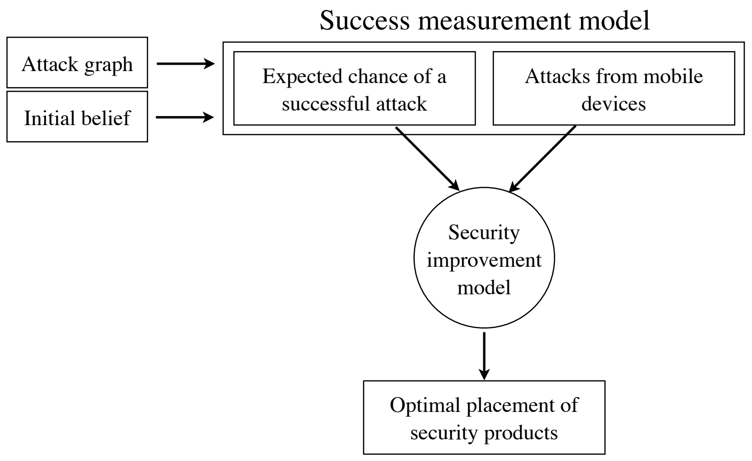

Success measurement model
Attack graph
Initial belief
Expected chance of a successful attack
Attacks from mobile devices
Security improvement model
Optimal placement of security products

Figure 3.1.2: We measure the success of an attack on the network given a set of initial belief values. The success measurement model is capable of computing the chance of a successful attack for various attack patterns. Our model includes dynamic aspects of a network (such as device availability) that are used to model the attacks from mobile devices and attack choices. Optimal placement of security products is formulated as an optimization problem to minimize the security threats.

the vulnerability of the target. We provide the model, formulation and computation for efficiently solving the problem for large attack graphs.

3.1.1 Assumptions

Our model computes a probabilistic measurement of successful attacks from a system administration perspective. Despite previous approaches (such as [75]), we do not simulate attackers' behavior and decisions. Our methodology is to provide estimation of *what an attacker or a group of attackers may decide to do.*

Our model analyzes attack steps based on the assumption of uncertain attack decisions. We assume that the defenders (that is, system administrators) do not have specific knowledge about how an attacker may approach the resources. The only available knowledge is all the possibilities of attack decisions that might be chosen during an attack.

To formalize realistic attack scenarios, we assume that an attack may be carried out by either a single or a group of attackers. In each case, the attack may have simultaneous attack paths approaching various resources in the network. Therefore, as a key strength of our model, we do not assume mutual exclusion of attack paths.

3.2 Expected Chance of Successful Attacks

In this section we present our success measurement model to compute the expected chance of a successful attack on a network with respect to the attack's ultimate goal. We first present the definitions of the expected chance of a successful attack (ECSA) followed by the description of two efficient algorithms to compute ECSA values. A list of symbols and their interpretations used throughout this section is given in Table 3.1.

We calculate the ECSA values for the nodes of an attack graph G based on intuitive attack scenarios. Developing a meaningful and rigorous probabilistic model is a technical challenge. The difficulty with defining a correct sample space and the corresponding random variables are two major issues with existing probabilistic approaches. Applying the basic laws of probability to compute a probability-based metric rather than the actual probability is a simplification. Therefore, even though plausible, previous probabilistic approaches lack a rigorous underlying foundation

The key component of our success measurement model is the probabilistic definition of the expected chance of a successful attack against any node in the attack graph.

We present an alternative approach to the Bayesian analysis discussed in [71, 89]. Our success measurement model computes probabilities as a function of initial belief probabilities without the need for specifying conditional probabilities required by Bayes' theorem. The set of initial belief values required by our model is small and can be obtained from standard vulnerability assessment systems (discussed in Section 3.4.1).

Our model measures the success of an attacker based on the attack dependencies determined by a logical attack graph.

Definition 1. *A logical attack graph $G = (V, E)$ is a digraph where $V = N_f \cup N_g \cup N_r$ and N_f, N_g, N_r are disjoint sets of nodes containing fact nodes, goal nodes, and rule nodes, respectively. E is the set of arcs, and $\mathcal{G} \in N_g$ is the attacker's goal.*

We define the sample space for a node and a corresponding random variable representing attack outcomes. The outcome of an attack attempt on a node can either by a success or a failure. Let $\Omega(u)$ be the sample space for a node $u \in V$ for an attack graph G. We define the random variable X_u for the node u as a Bernoulli random variable with $X_u(\omega) = 1$ denoting success in an attack and $X_u(\omega) = 0$ failure, where ω is an outcome.

Definition 2. *For any node $u \in V$ of an attack graph, the expected chance of a successful attack (ECSA) at a node u is given as $E[X_u] = P(X_u = 1)$, that is, the probability of success for the random variable X_u.*

Let $\phi(u) = \{v \mid (v, u) \in E\}$ be the set of predecessors (dependencies) of a node u. In the following, we define ECSA for the derived nodes based on the corresponding logical semantics (that is, conjunction for a rule node and disjunction for a goal node).

Symbol	Interpretation
G	An attack graph
N_g	Set of goal nodes in G
N_r	Set of rule nodes in G
N_f	Set of fact nodes in G
N'_r	Rule nodes with a fact predecessor
$\mathcal{G}$	Ultimate attack goal node
V	Set of all nodes in G
$\phi(u)$	Set of predecessors of u
X_u	A Bernoulli random variable on node u
$E[X_u]$	Expected value of X_u
$P(X_u = 1)$	Probability of success in X_u
Y_i	An auxiliary Bernoulli random selector
x	A vector of all random variables X_u
y	A vector of all random variables Y_i
$f_u(x, y)$	A constraint function for a node $u \in V$
N_{ra}	A set of candidate rule nodes $N_{ra} \subseteq N_r$
τ	An improvement option representing a security product
$f_u(T, x, y)$	Modified definition of $f_u(x, y)$ to include improvement options
f	A vector valued function to hold the constraints f_u

Table 3.1: A table of symbols and their interpretations.

ECSA value of a rule node. Let $u \in N_r$ be a rule node and $\phi(u) = \{v_1, v_2, \ldots, v_t\}$. The random variable X_u — corresponding to the success or failure of the attacker at node u — is defined as the product of the random variables for all predecessor nodes $v \in \phi(u)$:

$$X_u = \prod_{v \in \phi(u)} X_v, \tag{3.1}$$

and

$$E[X_u] = \prod_{v \in \phi(u)} E[X_v], \tag{3.2}$$

assuming independence of the predecessor random variables.

The independence assumption indicates that for any events $X_{v_i} = 1$ and $X_{v_j} = 1$, the occurrence of one does not affect the probability of the other. While assuming that the variables are dependent is possible, such an assumption does not substantially impact the results of the computation and has no clear predictive benefit. For instance, consider a rule with two fact nodes as predecessors, one indicating a software vulnerability (v_i) and another indicating a host access control firewall rule (v_j). Assuming fact nodes are represented by random variables, the event that a software vulnerability is exploitable by an attacker ($X_{v_i} = 1$) does not depend on the event that the target host is accessible through another host ($X_{v_j} = 1$).

ECSA value of a goal node. An attack graph has several goal nodes. A goal node either depends on a single exploitation rule (represented by a rule node) or multiple exploitation rules such as u_1 in Figure 3.2.1.

A goal node with multiple rule node dependencies is a logical disjunction. In reality, this disjunction indicates that there are multiple attack choices for an attacker towards a specific attack goal. For instance, consider a server with a local privilege escalation vulnerability (which is exploitable remotely in a multi-step attack) and runs a network service with multiple remote vulnerabilities. An attacker must exploit one (or more) of these vulnerabilities to gain privileges on the target server. In the lack of observable evidence, one needs to compute the ECSA of a goal node with a function that correctly captures the probabilities of such attack choices.

Our approach is to computationally determine attack choice probabilities according to various attack patterns (Section 3.3). Per our knowledge, no previous work has modeled this reality.

In the the attack graph of Figure 3.2.1, Node u_1 has three predecessors (rule Nodes u_2, u_3, and u_4). To compute $E[X_{u_1}]$, we introduce auxiliary Bernoulli random variables Y_i (referred to as the random selectors) to capture the random selection of an attack path. The values of Y_i are multiplied with the computed ECSA for the predecessor nodes to reflect the attack choices. In Section 3.3, we show how the values of Y_i variables are computed.

Let $\phi(u) = \{v_1, v_2, \dots, v_t\}$ be the set of dependencies of u. Then we define the random variable X_u for a goal node $u \in N_g$ as

$$X_u = \sum_{k=1}^{t-1} \left[Y_k X_{v_k} \prod_{i=1}^{k-1} (1 - Y_i) \right] + X_{v_t} \prod_{i=1}^{t-1} (1 - Y_i), \tag{3.3}$$

for which

$$E[X_u] = \sum_{k=1}^{t-1} \left[E[Y_k] E[X_{v_k}] \prod_{i=1}^{k-1} (1 - E[Y_i]) \right] + E[X_{v_t}] \prod_{i=1}^{t-1} (1 - E[Y_i]). \tag{3.4}$$

Observe that the Definition (3.3) selects $X_u = X_{v_i}$ by the event $Y_i = 1$, $Y_j = 0$ for $j < i < t$. Note that the Bernoulli variables Y_i in general depend on the node u, but this dependence is not reflected with the notation $Y_i^{(u)}$ for simplicity.

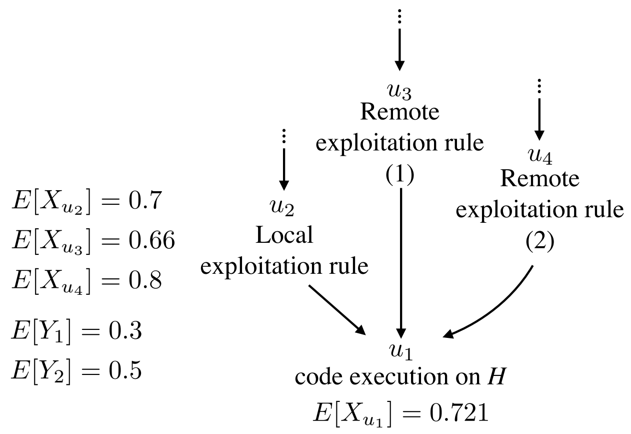


Figure 3.2.1: A goal node for an attack on host H with three attack choices: a local exploitation and two methods of remote exploitation. The expected values for the three rule nodes u_2, u_3, and u_4 are computed based on Equation 3.2. The variables Y_1 and Y_2 measure the probability of attack choices. $E[Y_1] = P(Y_1 = 1)$ is the probability of choosing the path u_2 to u_1. Its complement, $1 - P(Y_1 = 1)$, is the probability of choosing either u_3 or u_4. $E[Y_2] = (Y_2 = 1)$ is the probability of choosing u_3 and its complement is the probability of choosing u_4. We assume $E[Y_1]$ and $E[Y_2]$ are not available, and thus, we computationally determine their values based on Equation 3.4.

While we generally assume that the random variables are independent, we do not assume they are mutually exclusive. That means multiple concurrent attacks (more than one Y_i) can lead to a goal node from various attack paths.

Equation (3.3) forms a summation over the predecessor nodes of a goal node u, with each component of the summation multiplied with a combination of the random selectors Y_i. For example, the equation for Node u_1 in the graph of Figure 3.2.1 is

$$E[X_1] = E[Y_1]E[X_2] + (1 - E[Y_1])E[Y_2]E[X_3] + (1 - E[Y_1])(1 - E[Y_2])E[X_4]. \tag{3.5}$$

3.3 Computation

Attack choices are uncertain, and, various attack scenarios are possible. Existing work such as [78, 85, 75], have provided ways to compute a static view of the security risk corresponding to specific attack scenarios. In this section we describe two methods for computing ECSA values of an attack graph to model various attack patterns.

1. **Simulation approach.** This method simulates attack choices (using a random distribution of the attack choice values $E[Y_i]$) to estimate an average value of the ECSA.

We form a linear system of equations based on our Equations (3.4) and (3.2) that we efficiently solve by LU factorization and show that a solution always exists. The results give the vulnerability of the network under an average attack pattern.

2. **Optimization approach.** This method solves a constrained nonlinear programming problem in order to find the expected values of all the random variables that maximizes $E[X_{\mathcal{G}}]$. Attack choices, represented by $E[Y_i]$, are computed to maximize the objective function as opposed to being randomly chosen. The computation involves a mildly nonlinear optimization problem and is more complex than the simulation above. We use a standard state of the art algorithm called sequential linear programming that efficiently approximates and finds a solution that is close to the optimal solution. The results show the most vulnerable components of the network under a specialized attack pattern.

3.3.1 Simulation Approach

To estimate the average attack pattern, we examine various assignments of the random selectors Y_i for a goal node u. To accomplish this, our general strategy is first to compute values of all random selectors of all goal nodes with a random distribution of probabilities (for example, $E[Y_i] = 0.5$). We then fix the values of the random selectors in the Equations (3.4) and (3.2) to compute $E[X_v]$ for all nodes $v \in V$. With fixed values of the random selectors Y_i, the Equations (3.4) and (3.2) are linear and can be solved, in polynomial time, using standard direct methods for linear systems.

In each iteration, the randomly generated values of the random selectors represent a particular attack pattern. Although our model does not require historical data about attacks, in the presence of high evidence about a likely preferred attack path, this preference can be regarded as a bias when generating the values of Y_i.

A linear system of equations for goal and rule nodes is defined next. For notational convenience, let $V = \{1, 2, \ldots, L\}$, and $N_r \cup N_g = \{1, \ldots, M\}$. Let $x = (x_1, x_2, \ldots, x_M)^T$ be a vector of unknowns. Each variable x_i in the vector x corresponds to the expected value $E[X_i]$ of the random variable X_i for a goal or rule node $i \in (N_r \cup N_g)$. Therefore, the vector x is of size $M = |N_r \cup N_g|$ where N_r is the set of all rules and N_g is the set of all goals in the attack graph.

The relationship between the variables x_i is defined by a linear system of equations $Ax = b$, where $A = (a_{ij})$ is a real $M \times M$ matrix and b is a real M-vector.

If i is a goal node, for all predecessors $j_1 < j_2 < \cdots < j_t$ of goal node i, set

$$a_{ij_k} = \begin{cases} -P(Y_k = 1) \prod_{m=1}^{k-1} P(Y_m = 0), & 1 \leq k \leq t-1, \\ -\prod_{m=1}^{t-1} P(Y_m = 0), & k = t. \end{cases} \tag{3.6}$$

Also set $a_{ii} = 1$ and $a_{ik} = 0$ for $k \neq i$ and $k \notin \phi(i)$, and let $b_i = 0$.

For a rule node i, there is at most one goal node and at least one fact node in $\phi(i)$. If the rule node i has one goal node $j \in \phi(i)$, set

$$a_{ij} = -\prod_{\substack{k\in\phi(i)\\ k\in N_f}} P(X_k = 1), \tag{3.7}$$

$a_{ii} = 1$, $a_{ik} = 0$ for $k \neq i$ and $k \neq j$, and $b_i = 0$. If the rule node i has no goal node in $\phi(i)$, set $a_{ik} = \delta_{ik}$ (the Kronecker delta), and

$$b_i = \prod_{\substack{k\in\phi(i)\\ k\in N_f}} P(X_k = 1). \tag{3.8}$$

The linear system formed by (3.6), (3.7), and (3.8) represents the ECSA equations for rule (3.2) and goal nodes (3.4). Solving for the variables in x, we compute the ECSA for the attack graph nodes that measure the chances of successful attacks on network components.

The linear system of equations $Ax = b$ formed based on (3.6), (3.7), and (3.8) is solvable using either sparse LU or stationary iterative methods. A unique nonnegative solution for x exists: Under the reasonable assumption that every goal node has at least one rule node as predecessor with the selection coefficient $a_{ij} < 0$, and all the initial beliefs $P(X_u = 1)$ are not one, the matrix A is an invertible M-matrix (nonpositive off diagonal elements and eigenvalues in the open right half plane). Therefore A^{-1} is a nonnegative matrix (all elements nonnegative). Since $b \geq 0$, it follows that $x = A^{-1}b \geq 0$ also. (The proof that A is an invertible M-matrix consists of constructing a positive diagonal matrix D such that AD is strictly row diagonally dominant, which is a characterization of invertible M-matrices.)

3.3.2 Optimization Approach

The computation method described in this section allows one to find the ECSA values such that the ECSA of the attack target is maximized. The results represent a specific attack pattern, as opposed to the averaged attack pattern obtained using the simulation method introduced in the previous section. Both computation methods are useful and provide complementary information about the vulnerability of the network. The result of this computation is in particular important for optimal placement of security hardening products described in Section 3.5.1.

In the following we present the steps to formulate the maximization problem. It is no longer possible to reduce (3.2) and (3.4) to a linear system of equations, because both $E[X_u]$ and $E[Y_i^{(u)}]$ are unknowns, and thus, the maximization problem is a nonlinear programming problem. To find the most vulnerable components, we formulate a maximization problem with a nonlinear objective function subject to linear and nonlinear equality constraints. The decision variables are the nodes of an attack graph. The equations for computing ECSA, (3.2) and (3.4), form the constraints of the maximization problem.

Let x_i be a decision variable for a node $i \in N_r \cup N_g$ corresponding to $E[X_i]$, and $x = (x_1, x_2, \ldots, x_M)^T$ be the vector of unknown ECSA values for all nodes. Let y_i be a decision variable for a random selector Y_i corresponding to $E[Y_i]$, and $y = (y_1, y_2, \ldots, y_P)^T$ be the vector of unknown expected values of the random selectors. For a rule node $u \in N_r$ with predecessors $\phi(u)$, the constraint function is

$$f_u(x, y) = \begin{cases} x_u - x_j \displaystyle\prod_{\substack{k \in \phi(u) \\ k \in N_f}} P(X_k = 1), & j \in \phi(u) \cap N_g, \\ x_u - \displaystyle\prod_{\substack{k \in \phi(u) \\ k \in N_f}} P(X_k = 1), & \phi(u) \cap N_g = \emptyset. \end{cases} \tag{3.9}$$

Note that Equation 3.9 has two cases. The first case is for rule nodes with one goal node as a predecessor and the second case is for rule nodes with no goal nodes as predecessors. For a goal node $u \in N_g$ with predecessors $\phi(u) = \{v_1, v_2, \ldots, v_t\}$, the constraint function is

$$f_u(x, y) = x_u - \sum_{k=1}^{t-1} \left[y_{m_u+k} x_{v_k} \prod_{i=1}^{k-1} (1 - y_{m_u+i}) \right] - x_{v_t} \prod_{i=1}^{t-1} (1 - y_{m_u+i}). \tag{3.10}$$

Recall that all the selector variables for all the goal nodes are numbered consecutively, so that the y_i for node u are $y_{m+1}, y_{m+2}, \ldots, y_{m+t-1}$ for some $m = m_u$ depending on u.

Let $f(x, y) = (f_1, f_2, \ldots, f_M)^T$ be a vector-valued function. The nonlinear program for finding the most vulnerable components is

$$\begin{aligned} &\text{maximize } f_{\mathcal{G}}(x, y) \\ &\text{subject to } f(x, y) = 0, \\ &0 \le x_i \le 1 \text{ , } i = 1, \ldots, M, \\ &0 \le y_i \le 1 \text{ , } i = 1, \ldots, P. \end{aligned} \tag{3.11}$$

In (3.11), the vector-valued function $f(x, y)$ holds all the constraint functions (that is, (3.9) and (3.10)) for all rule and goal nodes in the attack graph. Note that the constraints in $f(x, y)$ are the ECSA equations (3.2) and (3.4) equalized to zero.

To solve (3.11), we use a technique called sequential linear programming (SLP) [7, 67]. SLP has been widely applied in engineering, and efficient algorithms for solving nonlinear programs using SLP are known [67]. The worst-case complexity of nonlinear global optimization is exponential. SLP is a standard technique for finding a close approximate solution for nonlinear optimization problems. SLP is computationally efficient and converges to an optimal solution [7, 67, 13].

The idea is to sequentially approximate 3.11 by a linear program, replacing each nonlinear function by its first order Taylor series expansion. We solve the linear approximation iteratively until the change in the solution is modest. The iteration is defined by limiting the change in the value of the decision variables (as move limit constraints) and slowly decreasing the move limit.

3.4 Determination of Initial Belief Values

In this section we describe the techniques and concrete examples for choosing initial belief values for fact nodes and improvement nodes.

3.4.1 Initial Belief for Fact Nodes

An *initial belief value* is a given probability of success, $P(X_{u_i} = 1)$, at a fact node $u_i \in N_f$. Our success measurement model relies on a relatively small set of initial beliefs that provide an *estimation* of expected chance of success for specific attacks on network services. In an attack graph, these network service vulnerabilities are formalized as fact nodes. The methods for obtaining initial belief values may vary. We illustrate some specific approaches next.

The probability value $P(X_{u_i} = 1)$ is based on experts' estimations. For documented software vulnerabilities, the value of standard vulnerability scores (assigned by experts, often from software providers) can be used as an estimation of the expected chance of success of exploiting the vulnerability. For example, common vulnerability scoring system (CVSS) includes three categories of vulnerability scores (base, temporal, and environmental), which can be used to estimate the ECSA at a vulnerability node.

In the following, we present three steps for obtaining the initial belief values for vulnerabilities and network configurations.

Analyzing the network configuration. Consider a machine A running Ubuntu server. A includes a MySQL database server (version 5.5) listening on port 3306, which allows remote

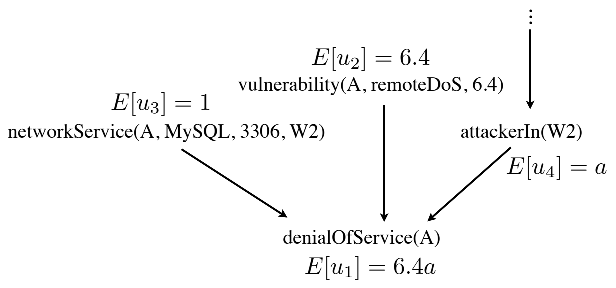


Figure 3.4.1: u_1 is a denial of service targeting server A, u_2 is a vulnerability, u_3 is a network service and firewall connection rule, and u_4 is a goal indicating that the attacker has reached the workstation W_2 that can access A. we compute the initial belief for $E[u_2]$ based on CVSS. We also set the initial belief for $E[u_3]$ to indicate that the connectivity on port 3306 is reliable. The ECSA of the goal node u_4 is computed based on its dependencies.

connections. To protect A from basic attacks, iptables rules are set to allow tcp/udp connections either locally or to specific IP addresses inside a NAT subnet. These IP addresses belong to workstations from which the database administrators and developers connect to the server A, and a web server that runs the web applications.

Analyzing attacks and vulnerabilities. The server A may be attacked through a multi-step attack. An attacker can exploit a remote privilege escalation vulnerability from a workstation W_1 to a database developer workstation W_2. Since A accepts MySQL connections from W_2, the attacker can use one of multiple remote denial of service vulnerabilities to launch a denial of service on the MySQL server in A. One such vulnerability is CVE-2012-3147 that allows unauthenticated users to establish a denial of service attack. The CVSS base score of CVE-2012-3147 is 6.4 (on a 10 point scale).

Assigning initial belief values. When modeling the configuration of the network, we create a vulnerability fact node u_2 as a dependency of a remote denial of service rule node u_1 for the MySQL server. When multiple documented vulnerabilities with similar effects exist for u_2, we compute the value $P(X_{u_2} = 1) = max(s_1, s_2, \cdots, s_K)$, where s_j is a value in $[0, 1]$ based on the CVSS base score for a vulnerability j (for example, the score divided by 10), with K number of documented vulnerabilities. The computation of $P(X_{u_2} = 1)$ can be done in alternative ways, such as $P(X_{u_2} = 1) = \mu(s_1, s_2, \cdots, s_K)$, where μ is the mean of the score values.

We create another fact node as a dependency of the rule node u_1, denoted u_3, to indicate that incoming traffic on port 3306 is allowed from host W_2. We choose the probability value $P(X_{u_3} = 1) = 1$, indicating that the connection to the port 3306 is reliable and the attacker is knowledgeable about the port 3306 when attacking a MySQL database server. Otherwise, depending on the network configurations, we can set $P(X_{u_3} = 1) < 1$, with a reasonable value. The example rule node u_1 is shown in Figure 3.4.1.

3.4.2 Initial Belief for Improvement Nodes

Initial belief values for improvement nodes correspond to the reliability of the security solution represented by the nodes. There are several assessment factors for computing the initial belief values. We categorize these factors into two main groups: (i) effectiveness and (ii) deployment. Effectiveness is measured by detection accuracy and the rate of false positive/negative decisions. The deployment factor includes measurements for memory consumption, CPU utilization, library dependencies, maintenance, and financial cost.

To compute an estimated initial belief value for a security product, we use a weighted function of effectiveness and deployment parameters. Let $Z_k^{(u_i)}$ a Bernoulli variable for an assessment factor u_i, and let L be the total number of assessment factors. We define the expected value for X_{u_i} as

$$E[X_{u_i}] = \frac{\sum_k E[Z_k^{(u_i)}]}{L}. \tag{3.12}$$

For an effectiveness factor k, the value of $E[Z_k^{(u_i)}]$ indicates the accuracy of improvement option u_i. For a deployment factor l, a higher value of $E[Z_l^{(u_i)}]$ indicates lower deployment overhead.

In the example scenario of Section 3.4.1, we create an improvement node for additional iptables rules to improve security. For instance, we modify the firewall rules on server A to allow connection to the database server on an unusual port p other than the default 3306, and also change MySQL socket configuration to listen on port p. Then we create an improvement node u_5 for an iptables rule dropping ICMP requests and limiting TCP ACK packets to already established connections to prevent the attacker from easily finding the port number p through a port scanner such as nmap. We expect that the firewall rule of the node u_5 has an average effectiveness (some attacks may bypass this rule) with virtually no deployment overheads. Thus, we compute the initial belief value for u_5 as $P(X_{u_5} = 1) = 0.5 * E[Z_1^{(u_5)}] + 0.5 * E[Z_2^{(u_5)}]$ with a value of $E[Z_1^{(u_5)}] \geq 0.5$ for the effectiveness factor and $E[Z_2^{(u_5)}] = 1$ for the deployment factor.

When computing an optimal placement of improvement nodes, we face several choices for placing the security products and solutions. Our model computes an optimal placement according to the initial belief values associated with the improvement nodes.

The computation of initial belief values can be customized by system administrators to suit the needs of an organization. Our success measurement model is flexible to accommodate various setups, and provides rigorous quantitative threat analysis and optimized security mitigations.

3.5 Optimization Model and Machine Availability

In this section we present two extensions of our success measurement model. In the first extension, we demonstrate the usefulness of the success measurement model by developing an optimization model based on it. In our optimization model, we aim for computing an optimized network security configuration with respect to potential security defense mechanisms. In the second extension, we present the modeling of machine availability, which is used to capture the availability of mobile devices in the network. A highlight of both extensions is given below.

- **Optimization.** Given a set of security hardening products (e.g., IDS/IPS, firewall), to find their placements in the organizational network such that the vulnerability of the target is minimized. The optimal placement may be subject to other constraints imposed by the network configuration.
- **Machine availability.** Current attack graph literature does not address how to assess the (additional) vulnerabilities brought by mobile devices such as laptops and smartphones to the network. One needs to be able to represent the *availability* (i.e., periods of on and connected to the organizational network) of any machine[1]. Our work is the first to show how to represent and assess devices with variable availability, which is one of the characteristics of mobile devices. Due to their mobility, these devices differ from workplace desktops or servers in that they may connect to (untrusted) networks outside the organization, resulting in possible infection or information leak.

Our success measurement model and its computational techniques naturally yield the solutions to these problems, which are described below.

3.5.1 Optimized Security Configuration

With limited resources for hardening an organizational network, it is important to install a single or a combination of security hardening products so that the expected chance of a successful attack on the network is minimized. To find the best placement of a set of security products in a network, we extend the attack graph to define a security product as a special fact node referred to as an *improvement node.*

Definition 3. *An improvement node is a fact node that represents a security hardening product, service, practice, or policy.*

[1]The ability to represent and evaluate the availability of any machine in attack graph analysis is useful in general for non-mobile ones such as servers and desktops.

The objective of solving the problem of optimal placement of security products is *to compute and compare the effects of various placements of one or more improvement nodes while subject to certain constraints, and choose the placement that minimizes the attack goal's ECSA value.*

Compared to existing work (discussed in Section 2.1, such as [24]), we introduce a novel mathematical programming problem to find the best placement of a set of security improvement options on a subset of rule nodes of an attack graph.

In Section 3.4.2, we discuss how the initial belief value for an improvement node is computed. The following describes computing the best place to deploy a single security product (that can be generalized to multiple security products) in the network. We formulate this optimal placement problem as a minimax problem — finding the best placement of the improvement option that minimizes $\hat{x}_{\mathcal{G}}$, where $\hat{x}_{\mathcal{G}}$ is the maximum of $E[X_{\mathcal{G}}]$ with respect to X_u and $Y_i^{(u)}$.

We consider a single improvement option for rule nodes given deployment constraints. We define the set of admissible rule nodes $N_{ra} \subseteq N_r$ as a subset of all rule nodes. Let $P(X_\tau = 1)$ be the initial belief of some improvement option τ. The problem is to find a configuration that minimizes $\hat{x}_{\mathcal{G}}$. That is, we aim to find a rule node $u \in N_{ra}$ such that if $\tau \in \phi(u)$, the value of $\hat{x}_{\mathcal{G}}$ is minimized.

Let $\mathcal{A} = \left|N_{ra}\right|$ and $j_1 < j_2 < \ldots < j_{\mathcal{A}}$ be the nodes in N_{ra}. Define 0-1 variables t_{j_i} for $i = 1, \ldots, \mathcal{A}$ and let $T = (t_{j_1}, \ldots, t_{j_{\mathcal{A}}})$. A single improvement corresponds to the constraint

$$t_{j_1} + t_{j_2} + \cdots + t_{j_{\mathcal{A}}} = 1,$$

and the generalization to multiple improvements is obvious.

We modify the definition of $f_u(x, y)$ for a rule node given in Equation (3.9) to include the effect of the improvement option τ. For a rule node $u \in N_{ra}$, define

$$f_u(T, x, y) = \begin{cases} x_u - (P(X_\tau = 1))^{t_u} x_j \prod\limits_{\substack{k \in \phi(u) \\ k \in N_f}} P(X_k = 1), & j \in \phi(u) \cap N_g, \\ x_u - (P(X_\tau = 1))^{t_u} \prod\limits_{\substack{k \in \phi(u) \\ k \in N_f}} P(X_k = 1), & \phi(u) \cap N_g = \emptyset. \end{cases}$$

This modified definition adds the improvement node at exactly one rule node in N_{ra}. Note that the definition of f_u for a goal node is identical to Equation 3.10. The minimax problem to find the best placement of security products is

$$\begin{aligned} &\underset{T \in \{0,1\}^{\mathcal{A}}}{\text{minimize}}\ \hat{x}_{\mathcal{G}} \\ &\text{subject to } t_{j_1} + \cdots + t_{j_{\mathcal{A}}} = 1, \end{aligned} \tag{3.13}$$

where $\hat{x}_{\mathcal{G}}$ is the solution to

$$\begin{aligned}
&\underset{x,y}{\text{maximize}}\, f_{\mathcal{G}}(T,x,y) \\
&\text{subject to } f(T,x,y)=0, \\
&0 \le x_i \le 1, i=1,...,M, \\
&0 \le y_i \le 1, i=1,...,P.
\end{aligned} \tag{3.14}$$

The minimax problem (3.13) maximizes the ECSA value of the attack's goal ($E[X_{\mathcal{G}}]$) to find the highest chance of success in attacking a specific network component (such as a server). The result of the inner maximization problem (3.14) is then used in the outer minimization problem (3.13) to find the best placement of the security product such that the maximized ECSA is minimized.

The inner maximization problem is solved using SLP as before. The outer minimization problem is a limited combinatorial problem for one improvement. For multiple improvements, the outer problem can be solved by an LP relaxation (change $t_i \in \{0,1\}$ to $0 \le t_i \le 1$) with branch and bound. For k improvements, the complexity is $\binom{\mathcal{A}}{k}$.

3.5.2 Availability and Mobile Devices

To capture the increase of security threats due to the inclusion of mobile devices (such as laptops, smartphones, and tablet computers) in the network, our approach is to extend an original attack graph for a network to include attack paths from mobile devices. Specifically, we define special rules to represent the uncertain availability of mobile devices in an attack graph, as well as the corresponding ECSA formulation and computation. The ability to model the availability of machines in attack graphs is general and useful beyond the specific mobile devices studied.

We extend the rules of the MulVAL attack graph generator [64] to include exploitation rules that capture the availability of mobile devices. An identified mobile device may not always appear in the network. Mobile devices rarely include a server software. The majority of Internet-based mobile applications are clients to the outside world, requiring interaction with malicious input to execute a successful exploit. For instance, most of the vulnerabilities that we studied for the Android platform involved an interaction with a malicious code (i.e. a malicious website) and exploiting a local vulnerability. Thus, we define basic exploitation rules for mobile devices in Figure 3.5.1.

We capture the availability of a device with the node `deviceOnline(H,Platform)`. In the success measurement model, these nodes are dynamic nodes with no fixed initial belief. The availability of a device may be measured as the percentage of the time that the device is

```
execCode(H,Perm) :-
        compromised(H),
        vulExists(H,Vulid
                 ,localExploit
                 ,privEscalation).

compromised(H) :-
        deviceOnline(H,Platform),
        vulExists(H,Vulid
                 ,remoteClient
                 ,codeExecution),
        maliciousInteraction(H,_,App).
```

Figure 3.5.1: Rule nodes according to Datalog [14] syntax. The first predicate specifies the condition that if the device was compromised (i.e., through the second rule), using a local exploit with privilege escalation consequences, root privileges could be gained. The second rule describes a remote client exploit involving interaction with malicious content. The predicate deviceOnline(H,Platform) indicates the availability of the device H.

connected within the target network (e.g., through a wireless connection) in a certain period. This data may be collected or estimated for the target network.

Our rules are general enough to be applicable to any logical attack graph generator. In addition to the extended rules, an input of mobile devices data is given to the attack graph generator. The data includes access information to other devices, presence of vulnerabilities, and information about the platform. Once the attack graph is generated, we compute the ECSA values for the network, which is described next.

For mobile device fact nodes, the availability of the device cannot be deterministically specified. Thus, fact nodes similar to `deviceOnline(H,Platform)` cannot have a precomputed value for all instances of ECSA computation. A fixed value of $E[X_u]$ (for a fact node u) does not accurately reflect the device's availability. In order to solve this issue, we first define the stochastic fact node below to represent the device availability.

Definition 4. *A stochastic fact node represents a dynamic ground fact that is not associated with a fixed initial belief.*

We define a random variable for a stochastic fact node (such as `deviceOnline(H,Platform)`). Based on our success measurement model, the variable X_u, for a stochastic fact node u, is a Bernoulli random variable. For the node `deviceOnline(H,Platform)`, $P[X_u = 1]$ is the probability of the event that the device is online.

Using the random variable for the stochastic fact nodes, we can generate random instances corresponding to the probability of success at those nodes for our computation. Both computation methods in Sections 3.3.1 and 3.3.2 can be modified to analyze networks with mobile devices. In the average attack pattern simulation (the first method), the average behavior of the device is measured by randomly instantiating the availability values. When computing to find the most vulnerable component (the second method), the value of the device availability that creates the most risk to the target can be found.

Using our success measurement model we accurately capture the effect of mobile devices as part of the network. We believe this analysis is valuable in order to make better decisions on the policies that determine mobile interactions with the network.

3.6 Experiments

In this section we present several experiments to verify the mathematical models introduced in Sections 3.3 and 3.5.1. We compute ECSA values for a network configuration obtained from a functioning real world network. The goal of our experiments is to demonstrate that the computation of our mathematical programming model is feasible. The results of our experiments are complementary to the established theoretical foundations of the sequential linear programming (discussed in Section 3.3.2).

We implemented the two computational methods and the security improvement model in Java (approximately 3500 lines of code). We use GLPK [15] for solving linear programs. Our implementation parses an attack graph input file (obtained from MulVAL [64]) and an initial belief file, computes the ECSA values according to various parameters, and performs security improvement analysis based on a set of improvement options and constraints.

In the following we describe our results for computing ECSA values, assess improvement options on our example network, and discuss the effect of mobile devices in the network.

3.6.1 Experiments Setup

Our experiments are based on network configuration and vulnerability data from a real dynamic organizational network that is open to a large number of users and contains several servers and workstations. To simplify the discussion, our experiments include a subset of machines in the network.

We used network security scanning tools (such as nmap), online vulnerability repositories, and information provided by system administrators to create a network topology (depicted in Figure 3.6.1) and the attack graphs that represent the real network. We performed wireless network scanning to confirm the connectivity of wireless devices in the network.

In the following sections we describe four sets of experiments for which we generate two

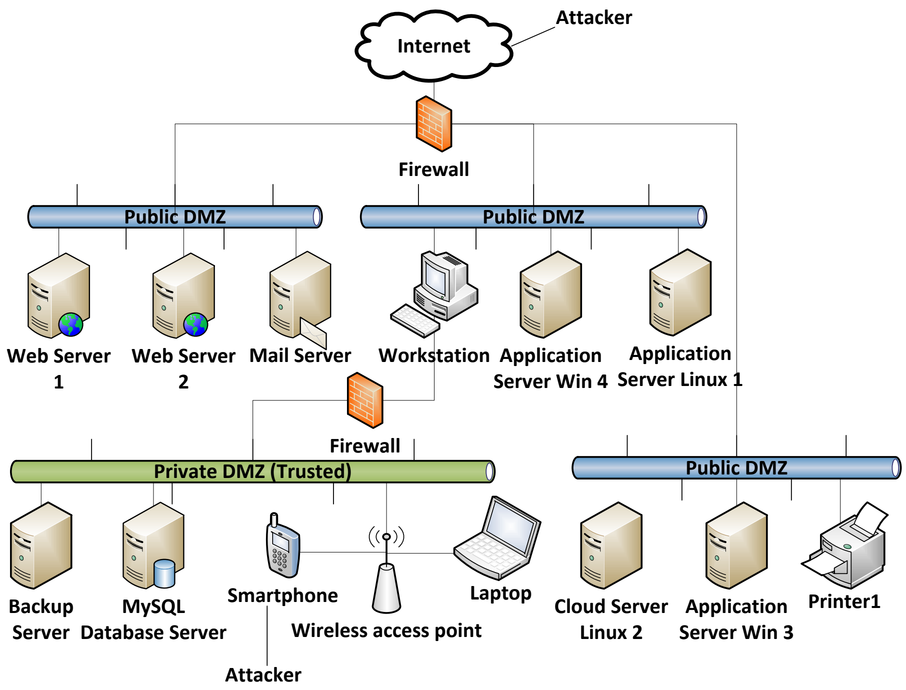


Figure 3.6.1: The topology of a dynamic organizational network with four subnetworks. Each machine on the three public DMZ subnetworks runs at least a network service with an open port. Data servers are on a NAT subnetwork and can only be accessed through the workstation. Mobile devices are connected to the wireless access point. The attacker either attacks remotely or uses a phone to crack the wireless password and attack the servers.

attack graphs (Table 3.2) with slight variations. We generate attack graph A (483 nodes) with no mobile devices in the network (i.e., availability of mobile devices is 0%) and attack graph B (549 nodes) that includes attack scenarios from untrusted mobile devices. Our experiments are computing ECSA for the attack graphs, computing an optimal placement for an intrusion prevention system, assessing the effect of mobile devices, and computing a network reconfiguration to minimize ECSA.

To assign initial belief values for the fact nodes, we followed the steps discussed in Section 3.4.1. Our program scans vulnerability score values and configuration settings to compute the initial belief values for computing the ECSA values.

For our experiments, we did not have access to data from attack incidents such as system logs to perform a validation of our results.

As described in Section 3.3.2, manual analysis of attack graphs A and B requires several iterations of computation according to the sequential linear programming. Moreover, by

Attack Graph	Hosts	Nodes	Edges	Placement Options
A: No mobile	13	483	663	206
B: With mobile	13	549	757	235

Table 3.2: Attack graph A is generated with no mobile devices in the network and attack graph B is generated with two mobile devices. Placement options refers to the nodes that can be considered for the addition of an improvement node.

manually inspecting the network, through configurations, network topology, and software vulnerabilities, it is difficult to find reasonable conclusions about chances of successful attacks or the effect of particular devices on the network.

3.6.2 Chances of a Successful Attack

Using the maximization method (Section 3.3.2), we performed an experiment on attack graph A (with no mobile devices) for which the database server is the attack's target. A sample of our results is depicted in Figure 3.6.2. The x-axis shows the targets of exploitation goal nodes achievable by the attacker. Notice that the maximization method computes the maximum ECSA values for the nodes of the attack graph. The maximized ECSA values show which network components (e.g., servers) are the most vulnerable.

The results of this experiment suggest that both application servers 3 and 4 (denoted server3 and server4 in Figure 3.6.2) have high ECSA values for their goal node, indicating high chances of successful attacks. This is because application servers 3 and 4 have highly scored software vulnerabilities, a high number of open ports, and thus they are relatively more exposed to the outside world. However, the chances of successful attacks on the target database server is the lowest, which is due to a better network configuration to protect it. This result is expected as the database server is less exposed to the outside and runs fewer vulnerable network services.

Figure 3.6.2 also shows the results for the ECSA computed based on attack graph B, which are discussed in Section 3.6.5.

3.6.3 Optimal Placement of Security Products

We tested our security improvement method introduced in Section 3.5.1 with a single improvement applied to a comprehensive set of rules in attack graph B. We used the results from the previous section to find the best placement of an improvement option for the network of Figure 3.6.1. Our improvement option is the installation of an intrusion prevention system (IPS) on a single device to minimize the risk on the target host (the database server).

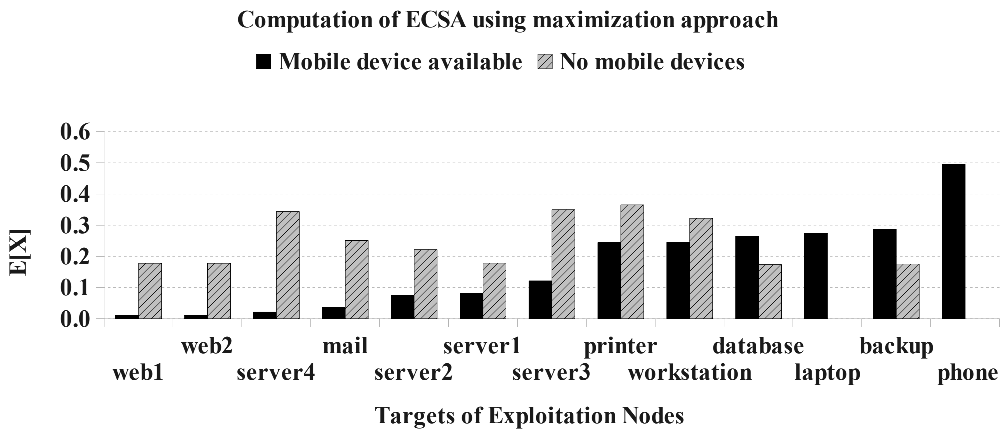


Figure 3.6.2: ECSA values attack graph A (no mobile devices) and B (with mobile devices). In the experiment with mobile devices, the availability of a mobile device is captured with a random variable and is not assumed to be fixed.

Our choice of IPS has some deployment overhead because of memory and CPU usage. After testing its effectiveness, we believe that this IPS has a low false negative rate. Using Equation 3.12, the initial belief for each improvement fact node for the IPS is $E[X_\tau] = 0.3$.

According to our method (described in Section 3.5.1), we add all the exploitation rules, to the set of applicable placement nodes N_{ra} (i.e., 206 nodes for attack graph A and 235 nodes for attack graph B[2]). Then we modify the original attack graphs to include improvement fact nodes as predecessors to each $u \in N_{ra}$.

We computed the improvement for the attack graph with no mobile devices and with the mobile devices present in the network. Table 3.3 shows the improvement results, for each attack graph configuration, ordered based on the percentage decrease in $E[X_\mathcal{G}]$. Third column shows the best placement of the IPS. $E'[X_\mathcal{G}]$ and $E[X_\mathcal{G}]$ denote the expected chances of a successful attack for $\mathcal{G}$ (i.e., the database server) in the improved attack graph and the original attack graph, respectively.

The results of the Table 3.3 demonstrate significant decrease in $E[X_\mathcal{G}]$ when considering the improvement option for attack graphs A and B. Our results indicate that installing the IPS on application server 3 has the best effect in minimizing the ECSA of the attack's goal. The reason is that the target server can be attacked from a number of ports indicated by goal nodes. Based on the computed values of the random selectors Y_i, a particular port p_1 receives a high chance of attacking the database server.

In the results, attacking the database server from p_1 has a lower ECSA compared to attacking

[2]Note that one can choose fewer rule nodes for solving the optimal placement problem, depending on possible placement constraints.

Rank	Attack Graph	Machine	$E'[X_\mathcal{G}]$	$E[X_\mathcal{G}]$	% $\downarrow$
1	A: No mobile	App Server 3	0.0520	0.1739	70.09
	B: With mobile	Database	0.0521	0.2651	70.04
2	A: No mobile	Database	0.0552	0.1739	79.18
	B: With mobile	Workstation	0.0791	0.2651	70.16

Table 3.3: Optimal selection for IPS installation for attack graphs A and B. The attack target is code execution on the database server. The results are compared against the original ECSA values without the improvement option. $E'[X_\mathcal{G}]$ is the ECSA value of the improved model. Note that % $\downarrow$ refers to percentage decrease.

application server 3. In the attack graph, attacking application 3 is a predecessor of attacking the database server on port p_1. Thus, the improvement option multiplied with ECSA of attacking application server 3 reduces the value of $E[X_\mathcal{G}]$ more, and installing the IPS on application server 3 yields a slightly lower value of $E[X_\mathcal{G}]$.

3.6.4 Large Scale Attack Graphs

To perform a large-scale improvement experiment, we extended the network in Figure 3.6.1 by adding another trusted subnet (denoted as trusted subnet 2) only accessible from the trusted subnet shown in Figure 3.6.1 (denoted trusted subnet 1). The trusted subnet 2 has three database servers (enumerated as database 1, 2, and 3) with similar configurations as the database server in trusted subnet 1. The database servers 1, 2, and 3 are accessible from the database server in trusted subnet 1, the laptop, and the workstation.

MulVAL (an attack graph generator) generates a graph of 1148 nodes for the extended network. The attack goal was set to execute code on database 1 in trusted subnet 2. The ECSA value for the goal was computed as $E[X_\mathcal{G}] = 0.078$. We considered an improvement option with $P(X_\tau = 1) = 0.2$. The results for the improvement options are ranked and given in Table 3.4.

Our computation shows that the optimal placement of the IPS is not on the target host (i.e., database 1). Instead, it is on the database server in trusted subnetwork 1. These results show the power of our technique for identifying the most effective placement of security products.

3.6.5 Effect of Mobile Devices

The network architecture presented in Figure 3.6.1 is also vulnerable to threats from mobile devices. For example, in the network of Figure 3.6.1, the system administrators have allowed mobile devices to join the wireless access point that is set up for internal purposes in the

Rank	Machine	$E'[X_{\mathcal{G}}]$	$E[X_{\mathcal{G}}]$	$\% \downarrow$
1	Database (TS1)	0.0087	0.078	88.85
2	Database 1 (TS2)	0.0091	0.078	88.33
3	Workstation	0.0215	0.078	72.44
4	App Server 3	0.0405	0.078	48.07

Table 3.4: Optimal selection for placing a security product in the extended network for attack graph C with 1148 nodes. Note that $\% \downarrow$ refers to percentage decrease, and TS stands for trusted subnet.

private DMZ region. Also, the laptop (connected to the wireless access point) is directly accessible from the workstation and the printer. Such configurations increase the attack surface. We assessed the security of the network by computing the ECSA on attack graph B that includes the attack vectors from mobile devices.

From the results of the experiment with mobile devices, we can conclude that the presence of highly vulnerable mobile devices in the network increases the chance of attack on the target machine. Using attack graph B, the most vulnerable components are the workstation, the printer (which has vulnerable server software), and the mobile devices (i.e., the laptop and the smartphone). In this experiment, the chance of success on exploiting the database server is increased by 52.44%.

Comparison of two computational methods. In Figure 3.6.3, we compared the computation of ECSA using our simulation method against the maximization method, described in Section 3.3.2. We used the attack graph with available mobile devices to highlight the conclusions we can draw from these two methods. As it is expected, in the simulation results $E[X_{\mathcal{G}}]$ (ECSA of the database server) is lower than that of the maximization approach.

3.6.6 Improving Network Configuration

Our optimal recommendation method is capable to compute an improved network configuration with no extra security products (such as an IPS) added to the network. In particular, we find a port p (amongst all open ports on all machines) such that if it is disabled, the value of $E[X_{\mathcal{G}}]$ is minimized. That is, for any other port p', if p' is disabled in the network (for which we obtain $E'[X_{\mathcal{G}}]$), then $E'[X_{\mathcal{G}}] \geq E[X_{\mathcal{G}}]$.

We used our method to examine the option on every possible open port that appears in the attack graph. The results of our experiments on attack graphs A (no mobile) and B (with mobile) are summarized in Table 3.5.

To verify the accuracy of our method, we considered open ports on the target database server that if disabled would eliminate the chance of attack. Although it is a common practice to

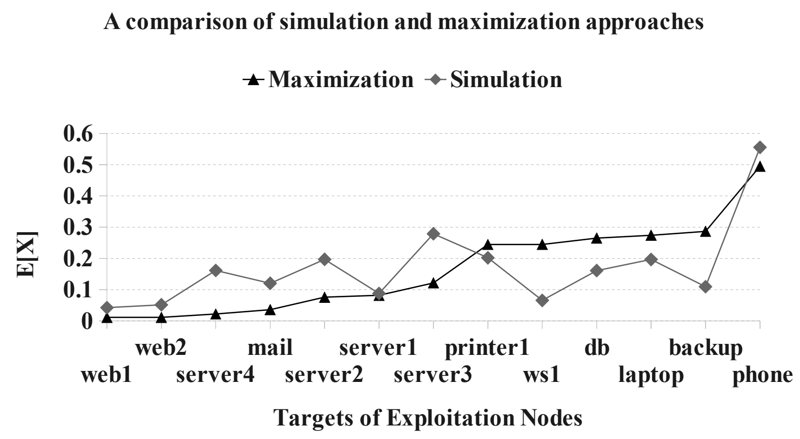


Figure 3.6.3: Computed ECSA values for attack graph B (with mobile device) attack using simulation and maximization approaches. The results show the ECSA for goal nodes. The maximization method computes a maximized ECSA on the attack's goal ($E[X_{\mathcal{G}}]$), which indicates the highest chance of a successful attack.

Rank	Attack Graph	Machine, Port	$E'[X_{\mathcal{G}}]$	$E[X_{\mathcal{G}}]$	% ↓
	A: No mobile	Database, 2200	0.0	0.1739	100
1	B: With mobile	Database, 2200	0.0	0.2651	100
	A: No mobile	App Server 3, 22	0.0	0.1739	100
2	B: With mobile	Backup, 2200	0.12	0.2651	53.8

Table 3.5: Optimal selection for closing a single port with the best effect on the security of the network. Note that % ↓ refers to percentage decrease.

eliminate straightforward attacks on well known ports, some of the servers in the target network did have open ports with minimum firewall rules.

The results in Table 3.5 show that the best recommendation is to disable the port 2200 and that would achieve a zero expected chance of successful attack. The second ranked recommendations are to close ports on the application server 3 and the backup server. Notice that both recommendations achieved a lower value of $E[X_{\mathcal{G}}]$, thus improving the security of the network.

3.6.7 Summary of Experiments

A summary of our experiments is given in Table 3.6. We refied that our success measurement model (Section 3.2) is capable of capturing various levels of success of an attacker in a network. The ECSA values assist system administrators in analyzing the security of the network in a number of ways including

1. finding the most vulnerable components of the network via direct comparison of ECSA values for goal nodes of an attack graph,
2. determination of the magnitude of a particular threat to the network by examining various network configurations (such as including mobile devices),
3. assessment of a list of candidate improvement products according to the lowest ECSA achieved, and
4. computing improved network configurations.

Attack Graph	Attack goal	$E[X_{\mathcal{G}}]$	μ	σ
A: No mobile	Database (TS1)	0.1739	0.3105	0.1169
B: With mobile	Database (TS1)	0.2651	0.1704	0.1464

Table 3.6: A summary of the experimental results for three attack graphs A and B showing the ultimate attack goal, the corresponding ECSA, the average of ECSA (denoted μ) for all goal nodes in each attack graph, and the standard deviation of ECSA (denoted σ) for all goal nodes in each attack graph.

3.7 Summary

In this work we presented a new probabilistic model and several computation methods for measuring the security threats in large enterprise networks. *The novelty of our work is our ability to quantitatively analyze the chances of successful attacks in the presence of uncertainties about the configuration of a dynamic network and routes of potential attacks.* We formalized, implemented, and evaluated our method. To demonstrate the importance of our technique, we showed the use of our success measurement in solving two open problems in network security: i) How to optimally deploy security products and services across the network? ii) How to formally analyze the vulnerability of dynamic networks with mobile devices?

We extended attack graphs analysis by including the threat from mobile devices, and probabilistically represented the availability of devices. Mobile devices cause increased threat to the security of a network, which our model can accurately quantify. For the optimal deployment of security products, we formulated a minimax problem, approximated it with the SLP technique, and evaluated the solution. The optimal improvement mechanism reveals considerable potentials for effectively hardening the security in large-scale networks.

Chapter 4

Authenticated Application (A2)

4.1 Model and Overview

We give the models and definitions used in our work. We discuss the design choices and general requirements for the authentication of applications and their processes in the operating system.

4.1.1 Motivations

We motivate our work through discussing and distinguishing four pairs of concepts related to authentication below.

Process identification vs. process authentication A process identifier may be the process ID, process name, etc. In the context of our A2 work, we define process identification as a naming convention to describe a process. Process authentication, on the other hand, is for a process to prove its identity to the operating system. It needs to prevent identity spoofing. There is no process authentication mechanism in the systems security literature, even though almost all access control solutions for OS make access decisions based on who the processes are. In these systems, installation paths may be used to distinguish among processes. However, such a simple mechanism is weak. We summarize their differences from our A2 solution in Table 4.1.

Process authentication vs. user authentication Unlike the conventional user authentication (e.g., password authentication) in a client-server architecture, process authentication imposes new and unique system and security challenges.

- *Storage of the secret* The user's secret (e.g., password or passphrase) can be memorized by the user. A process' secret is stored on the host and needs to be kept confidential

Identity	Weakness	Comparison to A2
PID and executable path	Executable integrity compromised	Registering executables with secret credentials
SELinux domains	Subject to spoofing and replay attack	SELinux does not authenticate processes or application binaries
Executable checksum	Need to be recomputed at runtime; verifies integrity but does not preserve it	Hashes may be used as credentials in A2
Developer signature	May be spoofed using fake certificates, e.g., Flame and Flashback malware; impractical implementation	A2 implements a kernel level certification without using developer public keys

Table 4.1: A comparison between the security in conventional process identification mechanisms and our application authentication solution.

with the help of the kernel to prevent it from being accessed by other processes.

- *Compatibility* An authentication mechanism needs to be compatible with legacy applications without requiring customization.

Application authentication vs. process authentication The authentication of applications is realized through the authentication of processes of that application. By process authentication, we refer to that the process of an application needs to prove the application's identity. (We choose to perform this authentication at the start of the process in our prototype.) We use the terms application authentication and process authentication interchangeably in this paper.

One-time authentication vs. runtime authentication status One-time authentication refers to the authentication of a process, which can be done at its creation. The authentication status of a process needs to be recorded and maintained by the system. At runtime, when the process makes requests to access system resources (e.g., system call requests), the authentication status can be queried and used for deciding whether or not to grant the request. Our A2 system supports both mechanisms.

4.1.2 Security Models

Security goals and assumptions Our security goal is to ensure the system assurance, which is to verify that a system enforces a desired set of security goals [46], more specifically, to ensure that the operating system correctly authenticates processes of applications at the runtime and malware cannot impersonate the identities of legitimate processes.

Our basic trusted components are the kernel code, kernel data structure (e.g., PIDs), and kernel's memory region. The kernel's code is trustworthy and does not contain any malicious code. The confidentiality and integrity of the kernel's memory are preserved. (Such a trust can be partly established using existing techniques such as the Trusted Platform Module (TPM) [79, 82] at boot time, assuming the exclusion of hardware attacks.)

Attack model Stealthy malicious code on the system may attempt to run itself as a stand-alone user-level process. Malware may be downloaded to the victim computer through a crafted malicious web page (e.g., drive-by download). Malware stores and attempts to execute at the user space. Malware may attempt to impersonate other (legitimate) applications, e.g., by spoofing the names of other processes. Thus, process names alone are not reliable for distinguishing processes. Malicious code running within the boundary of a legitimate process (such as a malicious browser script or extension) is out of the scope of our attack model.

4.1.3 Application Credentials

We define *secret application credential* next and explain the requirements for realizing a specific credential scheme in the operating system environment.

Definition 5. *A secret application credential (SAC) is a unique secret information issued to a trustworthy application by the operating system. SAC is used for processes of the application to prove their identities to the OS kernel during the authentication procedure.*

There are various approaches to instantiate secret application credentials, but they need to satisfy the following requirements. Some requirements are common in other credential systems, whereas some are new and unique to the process authentication problem.

- *Unique credential set.* For every legitimate application, there is exactly one credential $\gamma \in \Gamma$, where Γ is a unique set of credentials.
- *Protected credential set.* Access to the set Γ (in memory or on the file system) is restricted to the verifier process π and the registrar process ρ.
- *Hard to regenerate.* It is computationally hard to regenerate an application credential $\gamma \in \Gamma$ created by ρ.
- *Preventing replay attacks.* Eavesdropping and replaying of a credential shall not be possible.

4.1.4 Operations For Process Authentication

The authentication operation requires userland processes to demonstrate the possession and knowledge of kernel-issued application credentials. Processes without valid credentials are restricted from accessing system resources (e.g., making system calls) and considered potentially malicious. This mechanism provides a secure sandbox that isolates malware from system resources. We describe the general operations needed for process authentication solutions, including Credential generation, Process authentication, and Runtime monitoring.

1. Credential generation This is a one-time operation run by the kernel to issue the secret application credential to a (trusted) application. This operation may be performed at the time of application installation.

   Determining whether an application should be given a credential is a classification procedure, which is independent of our focus on process authentication. A classifier analyzing the trustworthiness of the executable code can be deployed for this purpose to complement A2, e.g., using the static programming analysis tools described in [26, 17].
2. Process authentication This is a protocol run by the kernel and a process for the process to authenticate itself to the kernel. The authentication is through the process proving the possession of the required SAC value.
3. Runtime monitoring The kernel monitors the execution of processes so that processes that have not been properly authenticated are caught.

   A system administrator may also choose to enforce fine-grained access control policies at the process level (e.g., specifying what system calls can be performed by applications), which can be integrated with this operation.

A process is not allowed to inherit its authentication status from its parent process in our model. Programs that are executed as part of other programs (e.g., through `execve`) can be recognized by A2 and need to authenticate themselves. Next, we describe details of our design for a system that supports the authentication of applications.

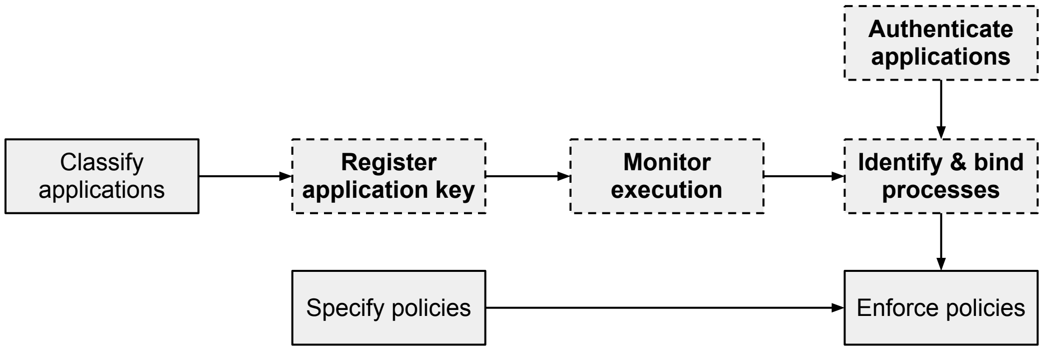


Figure 4.1.1: A diagram showing how A2 components/operations (in dashed line blocks) work with each other and how they may be integrated with complementary security elements.

4.2 Design

Our Authenticated Application (A2) design enables the authentication of applications. It consists of three main components: *Credential Registrar*, *Authenticator* and *Service Access Monitor (SAM).* (We implement the Authenticator and SAM as Linux kernel modules without modifying the kernel). We describe the functions of our components in the following, and describe in details how each of three operations Credential generation, Process authentication, and Runtime monitoring are realized.

Credential Registrar is for generating a credential for the application and registering the application with the kernel.

Authenticator is for authenticating a process when it first starts.

Service Access Monitor (SAM) is for verifying the authentication status of a process at runtime, i.e., whether the process has been successfully authenticated by the Authenticator.

There are two important lists in A2, the *credential list* and the *status list*. The credential list is the registrar's copy of the all the current valid credentials generated for registered applications. The status list is the Authenticator's record of the currently running processes that have been successfully authenticated.

4.2.1 Credential Generation and Storage

The kernel, more specifically the credential registrar, generates the secret credentials for legitimate applications. In A2, the registration operation for trustworthy applications can be done any time between the time of installation and the time of first execution. The registrar and the application each maintain a copy of the credential. The credential is no longer valid if the application is removed, reinstalled, or modified; and a new credential needs to be issued. There are many algorithms to instantiate the credential value. A simple method is to use a secret value of sufficient length as the SAC value that is generated by a pseudorandom number generator controlled by the kernel. The random values should be hard to guess.

A key problem in credential storage is how to protect the secrecy of application credentials that is stored by the application. (Kernel side of credential storage is assumed to be secure in our model.) To address that problem, we introduce a protection mechanism referred to as the *code capsule*. The application's copy of the secret credential is stored along with the application's code capsule (e.g., appending to the end of the executable). We define the code capsule as follows:

Definition 6. *A code capsule is a piece of executable code along with a secret application credential that is unique and verifiable by the kernel. A code capsule is not read or write accessible by any user process except by necessary kernel helper processes.*

Code capsules serve two major purposes. One purpose is to protect the secret application credential from being revealed to unauthorized processes through the file system. The other purpose is to bind a credential with the executable file, which is later used to verify the identities of the running processes by the kernel. Code capsules are accessible and maintained by a kernel helper, namely the credential registrar. When the application is executed, the credential is not loaded into the memory.

The Registrar runs a registration function as defined below.

Definition 7. *The registration function $\psi : E, s \to C_s$ where E is a string containing the executable code and s is the secret application credential generated by the credential Registrar. The function ψ produces the string $C_s = E||s$, which is a code capsule protected by the kernel. $||$ represents string concatenation.*

We describe the steps for a credential Registrar to generate an application credential next. Applications with the credentials issued by the Registrar are referred to by us as the registered applications. For malicious applications that bypass this registration phase, they cannot succeed in the authentication next due to the lack of valid registered credentials. Denote the current credential list maintained by the Registrar by `L`. The list consists of (`name`, `credential`) pairs of registered applications. The list needs to be kept confidential with restricted read and write access.

1. The Registrar runs an external checking mechanism (e.g., a classification method [26]) to verify that the application with name `app.name` is trustworthy.
2. If the external verification fails (indicating that the application may be malicious), reports it and halts. Otherwise, the Registrar generates a random value of required length n as the new credential s, and generates a code capsule C_s using the function $\psi(E, s)$. The Registrar writes C_s to the file system.
3. The Registrar appends the tuple (`app.name`, s) to the credential list `L`.
4. When the application is uninstalled, the Registrar is notified and deletes the entry (`p.name`, s) from the credential list `L`.

The credential generation operation is fully compatible with legacy applications and does not require any customization. Large applications may consist of several executable files. We register each executable file that may create at least one independent process with a unique credential. The purpose of having a unique s for each executable is to be able to correctly identify each running process and bind it to its executable code. The registrar may also need to ensure that the application has not been previously issued a credential, e.g., by checking whether a credential already exists at the end of its executable.

We point it out that the trusted registrar itself can be given a credential (e.g., manually installed by the system administrator). Then it can engage in the authentication procedure with the kernel as a regular process once it starts every time. The detailed process authentication protocol via challenge and response is described next.

4.2.2 Process Authentication

The process authentication protocol is to authenticate individual processes based on the credentials of the corresponding applications. We first discuss several design choices for realizing process authentication, and justify our approach next. One simple design choice is that the kernel directly accesses the application's credential and verifies its identity provided that the credential is stored in a predefined location. However, this method does not provide the security level that is needed in order to establish a strong identification. The location of the credential can be either defined in memory or the file system. Defining the credential in the memory imposes additional risk to stealing the credential as well as causing complexity of maintaining the credential location. An alternative design is to isolate all credentials in a restricted storage. The kernel retrieves the credential of corresponding process at the authentication time. However, this design is clearly inadequate because it does not bind a running process to the corresponding credential file at the runtime.

In order for a process to prove its identity to the kernel using the application's secret credential, our approach is for the process and the kernel to engage in a challenge-and-response protocol. The challenge-and-response concept is common in network security. We tailor it for the operating system environment. *i)* The kernel sends a random nonce to the application process. *ii)* The process produces the hash-based message authentication code (HMAC) with the nonce and the secret credential and returns the hash value to the kernel. *iii)* The kernel recomputes the HMAC and compares it to the value submitted by the process. We describe the three technical challenges and our approaches for addressing them next.

- One technical challenge is the implementation of an efficient and reliable communication channel between the process and the kernel. Our authentication protocol is executed on a socket file between the process and the kernel. This method is realized using a memory-based socket or shared memory, e.g., `/proc` file system [50]. The advantage of using the shared memory is that it is conveniently accessible by kernel device drivers and is under the complete control of the kernel.

  Throughout A2 design, this communication method via shared memory with restrictions is used for all the communications between the userspace and kernel space processes. The userspace processes may be applications or A2 components.

- Another technical challenge is that because the authentication protocol requires additional operations by processes, one needs to avoid having to modify and customize existing applications. We design and implement a piece of middleware that assists processes with the authentication operations. As a result, A2 is completely compatible with existing applications for process authentication.

- The third technical issue is how to minimize the authentication overhead while ensuring the runtime system assurance (e.g., at the system call level of granularity). Requiring the process to authenticate itself at every request of system call incurs excessive runtime overhead. We choose to perform the authentication at the time of process creation. We have a lightweight mechanism to maintain the *authenticated status* of a process during all subsequent requests, which is conceptually similar to session IDs in the web.

4.2.3 Authentication Protocol

The authentication protocol is run between the Authenticator and a process at the time of process creation. The description below requires the application to be customized to follow our protocol. The goal of A2 authentication protocol is two-fold: *i)* to securely authenticate running processes and *ii)* to record and maintain the authentication status of processes. The authentication status information is stored as a kernel data structure by the Authenticator, which is referred to as the *status list*. It is shared with the Service Access Monitor (SAM) at runtime for SAM to determine the legitimacy of processes submitting system requests.

Let `A` represent the Authenticator module. We denote the `A`'s credential list by `L`. It is a list of (`app-name`, `app-cred`) pairs, where `app-name` is an application name, and `app-cred` is its corresponding application credential generated by `A`. The Authenticator (and no one else) can submit queries to the Registrar in the form of `query(app-name)`, which takes as the input an application name and returns its corresponding credential on `L`. The Authenticator `A` maintains a status list `T` consisting of process IDs of successfully authenticated running processes. The list `T` is made readable by the Service Access module (SAM). Let `p` be a user process; `p.pid` is `p`'s process identification; and `p.name` is `p`'s application name. We denote `p`'s copy of its secret credential (stored in the code capsule) by `p.cred`. Let `auth-request(p.app)` is the function used by `p` to send an authentication request to `A`. Let HMAC be a secure hash-based message authentication code function.

1. `p`: Sends `auth-request(p.name)` to `A`, i.e., the application claiming to be `p.name` requests to be authenticated.
2. `A`: Does the following.
   (a) `p.cred'` $\leftarrow$ `query(p.name)`, that is, queries the Registrar with `p.name` to retrieve its credential `p.cred'` on the credential list `L`. If the returned `p.cred' is null` i.e., the application does not have a registered credential, reports `p` as suspicious.
   (b) Generates a random nonce and sends it to `p`. `A` also sets a timer t for the string to expire if there is no future response from `p`. The time frame to expire t is short (e.g., in milliseconds).

3. `p`: Computes $h \leftarrow$ HMAC(`nonce`, `p.pid p.cred`), where `p.cred` is `p`'s secret credential obtained from its code capsule. h is sent to the Authenticator `A`.

4. `A`: If the delay associated with the received HMAC exceeds the required threshold t, the authentication request is discarded and the authentication of `p` fails.

   `A` computes $h' \leftarrow$ HMAC(`nonce`, `p.pid p.cred'`). If $h == h'$, then the authentication is successful. Otherwise, it fails.

   `A` checks to see whether `p.pid` $\in$ `T`. If yes, reports `p` as suspicious. Otherwise, `A` appends `p.pid` to the list `T`.

5. When `p` terminates, `A` is notified, deletes `p.pid` from the status list `T`.

The ability for an application to succeed in the authentication protocol depends on its knowledge of the required secret application credential. E.g., if a process claims to be the Mozilla Firefox browser (i.e., with `p.name` being Mozilla Firefox), then it needs to prove its knowledge of the registered credential value of that application. Our security guarantee is hinged on the confidentiality of the application credentials, both the copy on the credential list and the application's copy in the code capsule. The PID that is used as an identifier for querying the status list belongs to kernel data, which is assumed to be trustworthy and unforgeable in our security model.

4.2.4 Runtime Verification

At runtime, whenever a process makes a request for accessing system resources through system calls, the Secure Access Monitor (SAM) intercepts the request and verifies the authentication status of the process, i.e., whether the process has successfully passed the authentication. This verification is accomplished by SAM through looking up the process's PID on the status list of the Authenticator, and verifies the PID's existence on the list. Our experiments show that this runtime verification of authentication status of processes is lightweight. PID is used an identifier for looking up the status list. Because of our assumption on kernel's code and data integrity, PID values used for this runtime verification are trustworthy, specifically unforgeable.

Our authentication system can be conveniently integrated with existing policy based access control systems for strong system assurance. We have demonstrated it by modifying AppArmor to incorporate strong process authentication and runtime verification of authentication status (not described). SAM can also be integrated with a policy specification language to benefit from existing work in policy specification such as [39] that uses an abstracted logical language to specify SELinux policies and Polymer [6], a runtime policy specification framework.

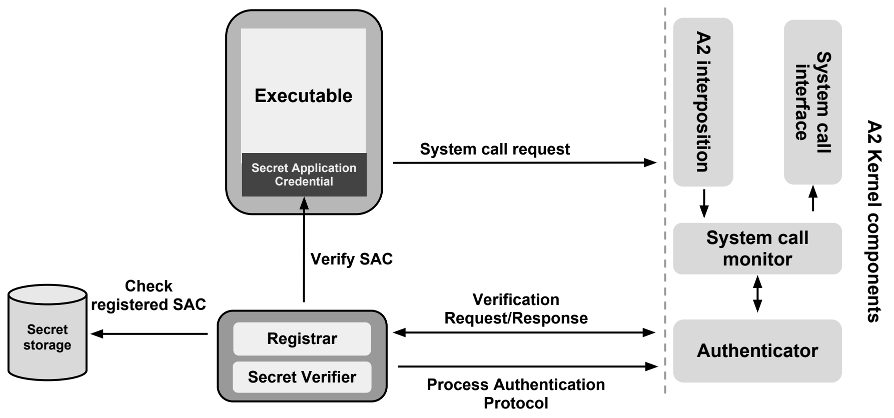


Figure 4.2.1: For compatibility with legacy applications, A2 provides a middleware that mediates the authentication between an application and the kernel. When an application creates a process and interacts with the kernel, the Authenticator requests from a trusted verifier process for verifying the authenticity of the application. The authenticity is determined by checking application credentials in their secure code capsules on the disk.

4.2.5 Mediating the Authentication

Our process authentication protocol described as in Section 4.2.3 requires the modification of legacy applications to support the interaction with the Authenticator, raising a compatibility issue.

We deign a middleware to perform the authentication on behalf of the application. The credential generation operation is unchanged. By performing the authentication on behalf of the application, the credential is no longer available to the application itself. As a benefit of this design, the problem of leaking the credential from the application is resolved.

We authenticate legacy applications using a helper program referred to as the *Verifier*. The Verifier has the read access to the credential list `L` maintained by the registrar, but not the write access.

1. To authenticate a newly started process `p` with process name `p.name`, process ID `p.pid`, and the path to the code capsule `p.path` (all obtained from the kernel), the Authenticator checks if the process has already been verified by looking its `p.pid` up in the status list `T`. If `p.pid` $\notin$ `T`, the Authenticator sends to the Verifier (`p.path`, `p.name`).

2. The Verifier reads `p.path` to retrieve the application's copy of the credential at the end of its code capsule. This credential is denoted by `p.cred`. It throws an error if the credential cannot be found.

3. The Verifier looks up the credential list `T` by `p.name` to retrieve the corresponding credential, which is denoted by `p.cred'`. It throws an error if `p.cred'` is null.

4. The Verifier checks if `p.cred'` == `p.cred` [1]. If yes, then the authentication succeeds. Otherwise, fails. The Verifier notifies the Authenticator with the authentication result.

5. The Authenticator updates the status list with `p.pid`.

The Verifier's main task is to access the code capsule of the application on behalf of the application. The security guarantee of the authentication protocol using the Verifier is equivalent to the one without it (discussed next). In our A2 prototype, the Verifier is implemented as a userspace application. It has a shared memory region with the Authenticator to exchange verification messages.

The Verifier is equipped with a manually installed credential, so that itself can be authenticated as a bootstrapping procedure. When the Verifier's process starts, the Authenticator authenticates its identity to prevent identity spoofing.

4.3 Security Analysis

The security of A2 relies on the confidentiality of the application credentials. Thus, we analyze our security guarantees by discussing the confidentiality of the application credentials and the integrity of A2 components.

Unforgeable credentials. Forging existing secret credentials (that appear on the credential list) by attackers is computationally hard, as long as a strong pseudorandom number generator is used to generate the credential. Besides existential forgery, a malware process using a self-generated arbitrary value as its credential cannot successfully pass the authentication, because that self-generated credential is not registered with the kernel and does not appear on the credential list.

Confidentiality of code capsules and credential list. To protect the secret credential from being revealed to other applications, A2 restricts the read access to applications' binaries, namely code capsules (where the application's copy of credential is stored). Malware may attempt to steal a credential from application's or A2 components' memory at runtime, which is prevented by the standard process memory isolation mechanism of the system. Similarly, A2 restricts the access to the credential list (owned by registrar) by other processes, thus ensuring its confidentiality. More specifically, only the registrar and the verifier have the direct access, and the Authenticator can (indirectly) query the list.

[1]Instead of the direct comparison of the two copies of credentials, an equivalent-yet-less-straightforward approach is for the Verifier to engage in the authentication protocol of Section 4.2.3 with the Authenticator (on behalf of the application process).

Resistance to replay attacks and denial of service. Malware may attempt to intercept the challenge-and-response communication between the Authenticator and a process during the authentication protocol. Because the `proc` file is only readable by A2 components, messages exchanged cannot be intercepted preventing replay. (This read restriction is enforced in kernel by A2.) Denial of service attacks are also prevented since our protocol limits the number of requests that may be received from a single process in Step 2.3. Further, code injection attacks are avoided in our protocol. An attacker can write a malicious code to the shared memory socket. The code may be read either by `p` or `A`. However, since the communication between `p` and `A` has a definite length (either the length of the request, or the nonce or the HMAC), exploiting a buffer overflow is avoided by verifying the string length. Moreover, the token generation protocol enables transparent and secure communication between applications and the kernel relying on the properties of the cryptographic hash functions.

Integrity of A2 components and data. A2 components span both the kernel space and the user space. The kernel-level components include the Authenticator and Service Access Module. Because of our assumption on the kernel integrity, these two components are trustworthy. In contrast, there is no assumption on the integrity of the user-level Registrar and Verifier, which may be targets of malware tampering and spoofing.

For anti-spoofing, A2 requires these two programs (namely the Registrar and Verifier) to authenticate to the Authenticator through our A2's authentication protocol as they start. The authentication procedure slightly differs from the one described in Section 4.2.3 in that *i)* their credentials are manually generated, and *ii)* the kernel's copies of their credentials are hardcoded in the Authenticator, as opposed to be stored on the credential list. For anti-tampering, A2 forbids the write access to the code capsules of the Registrar and Verifier by other userland processes.

A2 data includes the credential list, status list, as well as the intermediate communication messages in the authentication protocols via (`proc` file based) shared memory. We have discussed the confidentiality of the credential list, which is a user-space file. In contrast, the status list is a kernel data structure in the memory. Thus, its integrity and confidentiality are guaranteed as a result of our kernel integrity assumption.

The shared memory approach is used for all the communication between the kernel modules and user-level processes, including: *i)* the authentication protocol messages between the Authenticator and the requesting process, *ii)* credential queries between the Authenticator and the Registrar, and *iii)* authentication status update between the Authenticator and the Verifier. A2 secures the confidentiality and integrity of the shared memory based communication channels by preventing the read and write access to the shared memory by other non-related user-level processes.

Application isolation and access rights. In the A2 framework, we fully sandbox undesired processes. Our sandbox relies on the fact that malicious applications fail to authenticate to the kernel and thus are prevented from using most critical system calls. Moreover, such processes are exposed to the kernel when trying to interact with it. This makes A2 a powerful tool to find malware that was dropped by other applications by various means such as drive-by download. Although a legitimate application such as a web browser may allow malware to be downloaded, A2 prevents the downloaded malicious code to execute.

Limitations. A2 is capable of identifying interpreted programs running as stand-alone processes. For instance, a Java executable runs as a separate process. The program can be given a unique credential at registration. Each program can authenticate itself independently using our framework. Other interpreted languages such as JavaScript, Adobe Action Script, and Word document macros are out of our security model, because the program runs in a container process as opposed to a separate process ID. For a similar reason, the detection of malicious code injected into (vulnerable) authenticated processes, as opposed to running as a stand-alone process, is out of the scope of A2's attack model.

4.4 Implementation

We have developed a prototype of the A2 framework in C in Debian Linux (version 3.2.0-36), including the implementation of the credential registrar, the Authenticator, the Service Access Module, and the verifier. (The verifier is introduced in Section 4.2.5 to improve the compatibility of A2.)

- The Authenticator and SAM are kernel device drivers (modules). These two modules are loaded at boot time.
- The credential registrar and the verifier are implemented as kernel helper programs that run in the user mode for efficiency and compatibility considerations. We avoid the unnecessary context switches to register an application, and can use standard user-level libraries.

  As a bootstraping procedure, we require the registrar and the verifier to authenticate themselves to the kernel (namely the Authenticator) with their respective secret credentials at the time of the process creation.

Credential List

Each secret credential is a 128-bit random value generated by the AES key generation algorithm during the Credential generation operation by the registrar. We describe how the credential list can be accessed by each of the A2 components.

- The registrar can read and write to the credential list, which is a file. The verifier can read the file as well. No other processes can access that file. This restriction is realized as monitoring the `open` system call requests to the file. This system call monitoring is performed by SAM.

- Because Linux kernel components do not access file system directly, in our prototype the Authenticator and SAM do not have the direct read access to the file storing the credential list. They need to communicate with the registrar to retrieve a credential via the `proc` file mechanism.

The verifier process is similarly authenticated. Both the credential registrar and the verifier are stored in code capsules that are protected by the kernel. The trusted credential registrar has access to application executables to respond to the kernel's requests. These requests are sent to the verifier process through the `/proc` file system. We also secure the communication channel by restricting the `open` system call to all other processes.

Authenticator

To carry out the authentication protocol, the Authenticator communicates with the user space applications using the `/proc` file system, which is a memory-based file system controlled by the kernel. A protocol file is created by the Authenticator in the `/proc` file system. We support two functions for reading and writing operations to the protocol file in `/proc` file system. The `read_protocol_file` function is executed when the user reads the file. For writing to the challenge file, we define the function `write_protocol_file`. In this function, our module reads the data that is written by the user-level process. The Authenticator module uses the Linux kernel Cryptographic API [18] to perform the HMAC operations using a number of supported hashing algorithms. Our implementation of the Authenticator can accept multiple requests from multiple processes using the same `/proc` protocol file. For each process, only one request is served at a time.

Status List and Secure Access Monitor

Secure Access Monitor (SAM) and the Authenticator communicate via a shared data structure in the memory that holds the status list, i.e., a list of PIDs of successfully authenticated running processes. This data structure is maintained by the Authenticator and visible to SAM (but no other processes). To verify a process' identity, SAM searches through the status list.

We implement the status list as a sequential dynamic array. In our experiments, under a normal use, the number of running processes was under 100. As it is shown in the evaluations, searching the status list did not have a significant overhead in a normal usage. However, in order to improve the overhead, one can implement the list as a red-black tree (a special type of balanced binary search tree [9]) that has a search complexity of O(log n) where n is the size of the status list in the memory.

To avoid the need to modify the kernel, SAM uses the `kprobe` API to hook into system calls and monitor process activities. Although the probes introduce extra overhead, the produced overhead does not cause considerable latencies to applications' functionality, limited by an average of 3 times more overhead (see Section 4.5).

To provide a more efficient alternative realization of SAM, we modified the Linux kernel to implement a faster system call tracing method. In this implementation, we modify kernel's entry assembly code to perform the verification of identities before the system call takes place. As the kernel prepares for jumping to the address of the requested system call, we place a jump to the address of our kernel function that implements SAM. We store all the necessary information before the jump and send the system call number and the process information to SAM's kernel function. The system call may be allowed or disallowed according to the value returned from our kernel function. After the return, we check the return value and either jump to the desired system call function or execute an exit code to user mode.

4.5 Performance Evaluation

The strong security guarantees provided by our A2 framework require additional computational and management overheads in the operating system. In order to assess the efficiency of our framework, we answer the following questions in our experiments:

- How does A2 impact the overall system performance?
- What is the performance penalty caused by A2 on various I/O and system call functions?
- What is the process creation performance under A2?

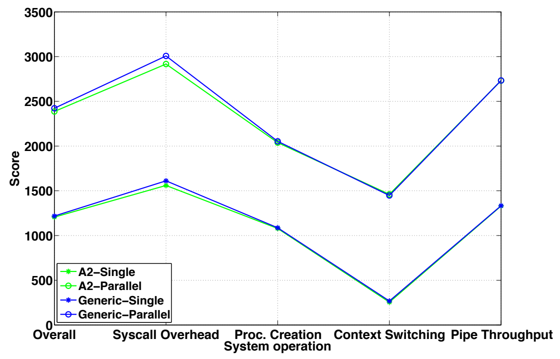
Overview and Setup. We conducted extensive performance evaluations using various system benchmark suites. An overall measurement of A2's performance is calculated using UnixBench[2], and, Phoronix test suite[3]. To measure the performance of critical system functions, such as I/O and system calls, we use lmbench [58] and UnixBench system benchmark suites.

Our experiments reveal efficient performance of A2 in various system operations. The highest performance downgrade is in open system calls (with A2 being about two times slower than the generic stock Linux). Our overall performance measurements, process creation and general I/O operations show reasonably fast performance of A2.

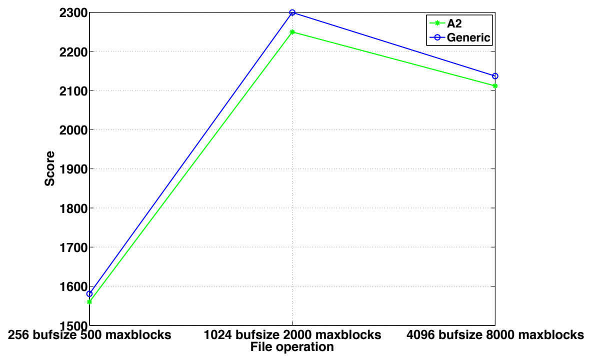
Our experiments executed on directly on a physical Intel Core Duo 2 machine with two cores of 2.99 Mhz speed and 3 GB of ram. The kernel version on the testing machine was Debian Linux 3.2.0-36. At the time of each test, there was no user interaction with the machine except for execution of the benchmark programs. All benchmark results show an average of several iterations.

We tested the system's performance twice for each experiment. First, we experimented with the stock generic Linux kernel (referred to as generic in Figures 4.5.1a, 4.5.1b, 4.5.2a, 4.5.2b

[2]http://code.google.com/p/byte-unixbench/
[3]http://www.phoronix-test-suite.com/



(a)



(b)

Figure 4.5.1: (a) UnixBench benchmark operations and results. The values are calculated according to a base score method. For each kernel (A2 and the generic Linux kernel) two sets of experiments are performed: single and two parallel copies, one for each core. (b) UnixBench benchmark file copy with various sizes. The file copy is slightly affected by our permissions checking in A2 with an average downgrade of 1.55%.

Benchmark	A2	Generic	Decrease
PHP compile time (seconds)	75.43	74.75	0.91%
Apache throughput (requests/sec)	12324.62	12531.00	1.65%

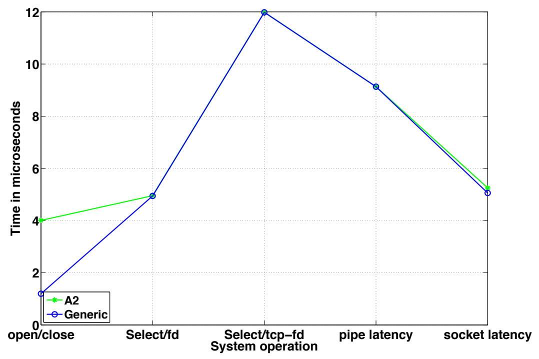
Table 4.2: Comparison of PHP compile time and Apache throughput (using Phoronix test suite) in A2 vs the generic Linux.

and Table 4.2) installed on the machine. Second, we tested the system with all A2 modules loaded. Also, we developed a daemon to simulate the authentication, by sending authentication requests to the kernel and performing the authentication protocol, in the intervals of 60 seconds during the course of each experiment.

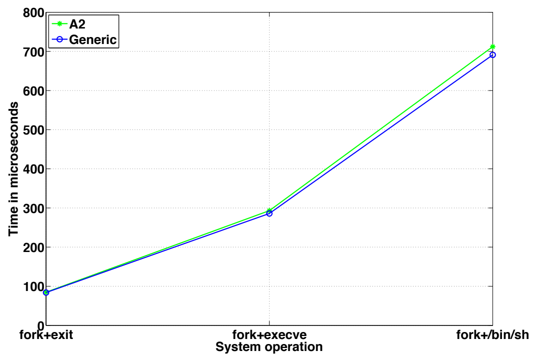
Overall system performance. For an overall measurement of A2's performance, we used UnixBench and two benchmarks from Phoronix test suite, that is PHP compile, and Apache throughput. UnixBench performs various system level tests and calculates an overall performance index relative to a base score, referred to as the BYTE index. A higher value of the BYTE index indicates a relatively better performance.

Figure 4.5.1a shows the results form UnixBench tests. The overall performance penalty with a single processor is 0.977%. For two parallel executions, the performance downgrade is 1.534%. Our tests with PHP compile and Apache throughput show an efficient overall performance of A2. As described in Table 4.2, PHP compile time had a downgrade of 0.91% under A2, and, Apache's requests per second downgraded 1.65% when using A2.

I/O and system call performance. UnixBench and lmbench perform extensive I/O and system call performance measurements. The benchmarks involve continuous calls to a system function and measuring the total processing time. For UnixBench results (in Figure 4.5.1b),



(a)



(b)

Figure 4.5.2: (a) lmbench benchmark operations and results. The values show time to execute the operation in microseconds. Most operations perform efficiently and do not suffer major performance downgrades. The performance downgrade in open/close is due to our permission checking before granting a file descriptor in the open system call. (b) lmbench process creation results. The values show time to execute the operation in microseconds. The results show process creation using fork without running external code, process creation with a call to execve, and process creation with running shell scripts.

we show the benchmark scores, whereas for lmbench results (in Figure 4.5.2a) we show the actual processing time.

The file copy operations in UnixBench had an average decrease of 1.55% in performance with maximum decrease of 2.161%. lmbench measures calls to open followed by a close, as depicted in Figure 4.5.2a. These calls had the most downgrade of about 235% in A2. This major downgrade is mainly due to various checks that we perform at the open system call to make sure proper access rights on protected regions that contain application credentials (e.g., checking access to code capsules). Other I/O operations such as select on file descriptors, select on tcp file descriptors, and sockets demonstrated statistical ties between A2 and the generic kernel.

Process creation performance. We directly monitor calls to fork and execve for monitoring process creations and activities by all processes to enforce our mandatory authentication protocol. Our monitoring involves a check to see if the process requires authentication. Thus, expect modest performance penalties by A2. To measure process creation, we used lmbench (Figure 4.5.2b) and UnixBench (Figure 4.5.1a) process creation benchmarks.

The results from lmbench show an average performance downgrade of 2.1627% when using A2 modules and a maximum downgrade of 2.949% for the fork, execve and shell execution results. UnixBench measures process creation with fork with a performance downgrade of 0.686% in the single execution experiment and a performance downgrade of 0.701% in the parallel execution experiment. The corresponding experiment in lmbench (calls only to fork) has a similar performance downgrade of about 1%.

4.6 Summary

Our work is the first to formally design application and process authentication in the operating environments. We have demonstrated its feasibility by presenting our architecture, implementation, and evaluation of a prototype Linux system supporting process authentication. We explained how process authentication can isolate malicious processes and thus prevent them from abusing and accessing system resources. The authentication model of A2 is highly portable and can be made compatible with legacy applications without any customization. Our evaluation results indicate that the overhead of performing process authentication at the system call level is acceptable. Future work of ours will be focused on porting our A2 design to Android operating system for mobile devices in order to support the authentication of apps. Such a solution will significantly improve the system assurance of Android devices that may be targets of malicious and stealthy apps.

Chapter 5

DroidBarrier

5.1 Background

5.1.1 Android System

In this section we describe the Android's runtime, sandbox, and security challenges. Our description of the Android system and security is based on our study of the internals of Android, and, some of the previous work [16, 10, 29, 30, 84, 93].

Android's runtime. Android uses Linux kernel for managing memory, file system, and account permissions. Applications are written in Java and run in Dalvik virtual machines. At boot time, the Linux kernel initializes a process called `zygote`, which is responsible for creating Android application processes [10]. To run an application, `zygote` forks a new process and loads the application code into the memory from application's Dalvik class file (a byte code format used in Android's runtime system). Applications may include native code that directly uses Linux system services.

Application sandbox. Android's security model is based on application sandboxes that are created at install time and application permissions that are enforced at runtime.

Android application installer requires developers to sign each application bundle with a certificate that is generated using the developer's private key [36]. Android allows self-signed certificates without the need for a certificate authority. The reason to have an application certificate is for the Linux kernel to issue a unique user identification (UID) for each application. No two applications can share a UID unless they are signed with the same developer certificate. Android's application sandbox mechanism uses the Linux file system permissions. For each application, Android's application installer, creates a new user identification (UID). To create a sandbox, Linux file system permissions are set for the newly created application's UID. Android uses an application's sandbox to isolate application resources and its processes. For instance, the application's sandbox protects the files owned by each application to be only accessible to permitted third parties, thus forming a protected file system sandbox for each application.

Android introduces inter-component communication (ICC), which is a mechanism for communications among applications using *intents*. Intents are messages that are delivered by *Binder*. Binder uses Linux kernel's memory-based interprocess communication (IPC) capabilities to implement and deliver intents.

5.1.2 Security of Android

Powerful attacks occur in two stages. First, the attack tries to compromise the Android's sandbox system through numerous privilege vulnerabilities in Android. This allows for absolute flexibility of further attacks on the system. Second, the attacker attempts to install malicious applications stealthily, and, runs the applications in background by registering service processes.

An attacker may compromise Android's sandbox in a number of ways. For example, the vulnerability `CVE-2011-2357` allows a crafted website to execute a JavaScript payload and bypass the browser's sandbox. This allows the website to download a malicious application and install it on the system.

Privilege escalation attacks in Android target the underlying Linux kernel. For example, the RACT exploit [93] uses a vulnerability in Android's `adb` daemon (that is a `setpid` program) to gain root privileges in the device. Several other privilege escalation attacks are discussed in [84].

Attackers attempt to install malicious applications on victims' devices to benefit from further vulnerabilities in the Android's runtime permission system. Android's application and security models have a number of vulnerabilities [16, 29] that, for example, allow unprivileged attacks [84] with the goal of spying on users, stealing valuable information, or causing financial loses [30]. Many of these vulnerabilities share a root cause of *malicious executables installed stealthily on the system*, which further motivates us in developing a model to fundamentally prevent the execution of malicious code in Android without compromising the openness of the Android platform.

Android's application sandbox mechanism has been shown to be vulnerable to privilege escalation scenarios [5, 22, 93]. For example, a malicious website could download in the document vulnerability `CVE-2011-2357`, a malicious website can misuse the browser to bypass its sandbox by executing a JavaScript payload.

Another vulnerability `CVE-2011-1149` allows an application to bypass its sandbox by misusing android's shared memory mechanism `ashmem`. This shows that Android simply relies on the unique UID mechanism without post-verifying a process's accesses to disallow such situations. Therefore, once an application (or a piece of malicious payload) could bypass the application sandbox, the operating system (OS) is unable to detect and disallow unauthorized accesses.

Application code signing does not prove the identity of an application at runtime. That is, to enforce application-level access rights, the kernel needs to have a proof of identity of an application's code. Further, the authors in [5] find out that Android's signing architecture has security flaws due to limitations of Android's UID sharing method. The identity can only be proved if the kernel is capable of securely registration application's identity and use it for performing runtime identity verification. With a third-party certification, this is only achievable if the kernel can verify the certificate and securely register the key to use it at runtime.

Android uses a interprocess communication mechanism called Intents. Android's Intents can be used both in inter-application and intra-application communications [16]. Even though permissions may be setup by the developer and enforced by the operating system, poor use of the Intents can cause several problems. These problems occur due to the extra functionalities provided by Android, such as implicit Intents where an application sends an Intent with no explicit recipient. Thus, any application that has announced the requested service can receive the message [16].

In this section we present the problem and an overview of the solution. We also discuss our security model and present the process authentication model.

5.1.3 Problem Statement and Overview of the Solution

We address the problem of *preventing stealthy installation and execution of malicious applications in an Android-enabled device.* This is a root cause of many malicious attacks that exploit other vulnerabilities in Android, with attack goals such as stealing private data and incurring financial loss.

Motivating example attack. We conducted a runtime analysis of three sets of Android malware `DroidKungFu`, `BaseBridge`, and `AnserverBot`. Each of these malware sets have a number of variants with nearly identical malicious activities and variant user interface elements. According to our findings, and the results in [91] and [93], these malware sets rely on malicious shell scripts (running in independent processes) to perform a privilege escalation exploit. To demonstrate the problem, we construct an example attack scenario that is common to these (and other) malware categories.

In the remote attack scenario, depicted in Figure 5.1.1, the attack exploit two critical vulnerabilities that lead to execution of malicious applications without the user's permission. One is a vulnerable client that allows limited execution of a payload and enables downloading a native binary (other tricks include sending a SMS message to propagate malware [19]) that can launch the attack. The goal of this phase of the attack is to connect to another malicious

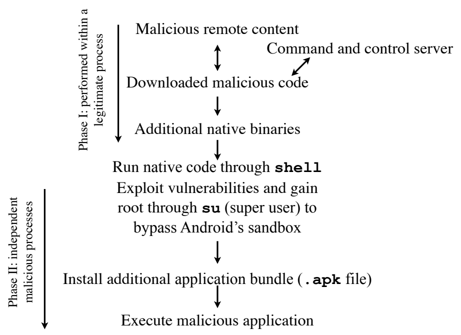

Phase I: performed within a legitimate process
Malicious remote content
Command and control server
Downloaded malicious code
Additional native binaries
Run native code through shell
Exploit vulnerabilities and gain root through su (super user) to bypass Android's sandbox
Phase II: independent malicious processes
Install additional application bundle (.apk file)
Execute malicious application

Figure 5.1.1: A remote attack sequence to bypass Android's sandbox and install malicious applications without user's permissions.

service (on a command and control server) and download a complete malicious application package. This downloaded application package will be installed with `root` privileges in the second phase of the attack. The second important vulnerability is in a system task (e.g., Android debugging bridge daemon) that runs as `root`. The downloaded native binary exploits a system vulnerability to gain `root` privileges and install the malicious package that was downloaded from a command and control server. Having `root` privileges, the native binary can bypass Android's sandbox and install applications on the file system without asking for the user's permission.

Once the malicious application is installed with all the requested permissions defined in its manifest file, it can conduct further attacks. The application can intercept inter-component communications, replace other legitimate applications with fake ones, consistently update itself with dynamic class loading, or cause financial charges, for instance, by sending premium SMS messages. The key element to this attack strategy is the ability to install applications without the user's explicit authorization. DroidBarrier is designed to prevent such installations, thereby foiling this form of attack.

Challenges and Goals. To develop a solution for protecting the system from execution of unauthorized malicious applications, we face several technical challenges:

- Assuming that the Android sandboxes are compromised, it is no longer possible to rely on any security guarantees provided by Android.
- It is difficult for the user to manually control the installation of new applications.
- The kernel only enforces file system permissions, and provide memory isolation for processes. In Android, the kernel lacks advanced capabilities to detect possible misuse of `root` privileges.
- An Android application bundle contains a Java bytecode class that runs in Android's virtual machine called Dalvik. Dalvik forks new processes on behalf of applications.

When forking a new process, as opposed to creating processes with `execve`, the kernel does not have immediate knowledge about the application file path that must be loaded into the memory. Therefore, authentication of processes requires further investigation from the kernel.

To address these challenges and develop a mechanism that can detect installation and execution of unauthorized applications, this work has the following goals:

- Providing a mechanism for the kernel to authenticate processes and unveil the existence of malicious applications.
- Detecting malicious processes at runtime with high confidence about their origins.
- Enabling the user to designate legitimate applications that are trusted to run.

These goals set the road map for developing a model based on our novel idea of authenticating a process, which is discussed in Section 5.1.5. In this model, the kernel enforces a mandatory authentication on every process. The user designates, in advance, which applications are allowed to run, and thus, has full control on the execution. Our runtime system, DroidBarrier (Section 5.2.2), implements our process authentication model. DroidBarrier continuously monitors low level process creation functions in the kernel to effectively enforce the mandatory authentication policy.

In the remaining, we present our security model, describing our trust assumptions and attack vectors, followed by a detailed description of our process authentication model.

5.1.4 Security Model

Security assumptions and trust model. We trust the kernel's code and the isolation of memory provided by the kernel used in Android. We assume that the integrity and confidentiality of kernel's memory are preserved. Further, we assume that Android's system software and processes do not intentionally contain malicious functionality.

Attack model. We target the following methods of attacks. We first categorize the attacks according to their installation approach.

- **Remote attacks.** As shown in the example attack in Figure 5.1.1, remote exploitation attacks start by remotely exploiting a vulnerability in an application. For example, attackers can exploit the many vulnerabilities in Android's WebView API (that applications use to show specific web pages) to trigger drive-by-downloads [57] and install malicious applications. These attacks can download the application, and, use system vulnerabilities to bypass user's permission for installing the application.

- **Physical attacks.** Using a physical communication channel such as Android Debug Bridge (adb) daemon to install malicious applications [84]. In this case, the attacker has physical access to the device and the attack goal is to install malicious applications without the user's knowledge. These attacks require that the device is not password protected, or, there are vulnerabilities that can bypass the password protection.

We mainly focus on the remote attacks that we believe are more likely to occur. The remote attacks are further categorized into two main classes, according to their execution models:

- **Dependent attacks**. The malicious code runs in a compromised application's processes. Thus, the malicious code depends on another legitimate application to continue execution.
- **Independent attacks**. The malicious code needs to run in at least one independent process that is created by a native code or a Dalvik application.

In case of dependent attacks (such as attacks on Adobe Flash [83] or return-oriented programming [23]) , the malicious payload may be limited to achieve all its functionality within the boundary of a legitimate application. We realize the importance of such attacks, however, these attacks are out of the scope of our current model.

In this paper, we specifically target independent (physical or remote) attacks that require installation of malicious applications with full capabilities. independent attacks are widely seen in Android malware samples. As pointed out in [93], there are powerful malware species (such as `DroidKungFu`) that cannot achieve their effects if they execute within the context of a compromised legitimate process. These attacks need (i) additional functionality that is not provided in the context of a legitimate application, and, (ii) to continue execution even when the compromised legitimate process is terminated.

To detect independent attacks, the main challenge is to find a reliable method to distinguish malicious processes from legitimate ones with high confidence. In the following section, we describe our authentication model that is capable of achieving this goal.

5.1.5 Process Authentication Model

To achieve our goals for protecting Android from malicious applications that could be installed without the user's consent, we use a process authentication model that can detect malicious application executions. Our process authentication model (also referred to as authentication model in this paper) regards application processes as individual principals that must be authenticated at runtime. Our mandatory authentication provides legitimate applications with valid *application credentials*, and, detects malicious applications that lack such credentials.

A related previous work, Quire [25], uses cryptographic annotations to authenticate interprocess communications in Android. Another work, A2 [3], uses a challenge-response protocol to authenticate applications. We provide a generalized authentication model compared to [25]. Our model is similar to [3] in considering processes as principals, however, we do not require the use of cryptographic protocols for authenticating user processes. Further, A2 [3] is designed to work with conventional Linux processes. We focus on authenticating Dalvik application processes that need monitoring by the Dalvik runtime (the `zygote` process) to achieve secure authentication.

Our goal is to authenticate individual application processes at the time of creation. Since applications are loaded in separate processes at runtime, our authentication model requires each process (that is created by an Android application) to be authenticated by the operating system. Inspired by authentication mechanisms in communications protocols, we assign a credential to each application. A credential must be a secure piece of information that can strongly authenticate a process and bind it to its application code at runtime. We define a secure application credential below.

Definition 1. *A secure application credential (SAC) is a unique secret issued to an application by a trusted process. Each SAC is associated with exactly one installed Dalvik or native application.*

Our authentication model's policy is to enforce a mandatory authentication for each process. The authentication is based on using the secure application credential (also referred to as the credential in this paper) provided to the application that created the process. We also define two special processes called the *verifier process* that represents the authenticator (i.e., the operating system), and, the *registrar process* that is responsible for registering application credentials.

Definition 2. *The verifier process π is a trusted process that has the authority of authenticating other processes.*

Definition 3. *The* registrar *process ρ is authorized to perform the registration of an application by associating an application bundle with a unique SAC. The registration of the application is performed at install time with explicit user permission.*

The goal of providing applications with secure application credentials is to properly authenticate their processes at runtime. Based on this authentication, the kernel distinguishes authorized applications from unauthorized ones. To implement a practical authentication method and prevent attacks on the credentials, our process authentication model follows the following *credential registration requirements*, which are first presented in Section 4.1.3.

- *Unique credential set.* For every legitimate Android application bundle, there is exactly one credential $\gamma \in \Gamma$, where Γ is a unique set of credentials.

- *Protected credential set.* Access to the set Γ (in memory or on the file system) is restricted to the verifier process π and the registrar process ρ.
- *Hard to regenerate.* It is computationally hard to regenerate an application credential $\gamma \in \Gamma$ created by ρ.
- *Preventing replay attacks.* Eavesdropping and replaying of a credential shall not be possible.

These requirements provide the basis for detecting and preventing the execution of malicious processes through a number of operations and properties that we discuss next.

5.1.6 Authentication Properties and Operations

The operating system is responsible for using application credentials to authenticate processes. The rationale behind our authentication model is determined by the following *Process Authentication (PA)* properties:

1. User processes (other than π and ρ) may not create or modify credentials, or, assign credentials to other processes and applications.
2. If a process fails to authenticate itself with a valid secure application credential, then the process is potentially malicious.
3. A process may not be authenticated with more than one credential.
4. A process may not inherit its *authentication status* from parent processes or any other process. Sibling processes are authenticated with a shared application credential.
5. A Dalvik application process is always a child of the `zygote` process. Native processes must either be a child of an Android system process or a Dalvik application process.

The PA properties ensure that processes are bound to proper application credentials such that a malicious process (under our attack model in Section 5.1.4) does not bypass the authentication, or, spoof other legitimate processes.

Operations. Our authentication model has three core operations: credential registration, process authentication, and runtime detection.

1. *Credential registration.* The user of a mobile device requests the registration of an application bundle, which is performed by the registrar process ρ. A credential is *valid* if (i) it is generated according the credential registration requirements, (ii) it is registered in an application bundle, and (iii) it is recorded in a credential database maintained by the registrar.

2. *Process authentication.* In this operation, the operating system exchanges an authentication request and response with a verifier process π (defined earlier) to validate the authenticity of a process and bind it to a registered credential. Process authentication fails if (i) the process has no application credential, or, (ii) the process possesses an invalid credential.
3. *Runtime detection.* A runtime detection operation monitors process creations and enforces the mandatory authentication on all processes. A process that fails in process authentication is flagged as malicious and is denied execution rights.

These operations ensure proper authentication of all processes and detection of unauthorized ones. In the following section, we present the design of DroidBarrier that implements these operations in the Android operating system.

Our model may complement (i) a classification procedure (for example, [17, 26, 28]) that precede the credential registration operation, which provides the user more information about the application, and, (ii) a sophisticated access control system (such as Android's SELinux [77]) that uses our strong authentication and detection to properly identify application process and enforce fine-grained access rights based on the specified policies.

5.2 Design

We design *DroidBarrier* (Figure 5.2.1) to realize our authentication model. Our goal is to detect malicious processes and prevent them from achieving their goals. We follow a proactive approach in DroidBarrier by registering the applications that the user desires to run. This registration enables the user to control which applications are legitimate and allowed to run. Also, we include a detection capability in DroidBarrier to monitor process creations and activities and authenticate processes at runtime. In the following sections, we present the core components of DroidBarrier for implementing the operations of our authentication model described in Section 5.1.5.

5.2.1 Credential Registration and Protection

To perform the registration of credentials, DroidBarrier includes a component referred to as the *credential registrar* (also referred to as the registrar in this paper). Credential registrar's task is to generate credentials for applications that the user designates as legitimate. Although establishing the trustworthiness of applications is an important procedure that can be executed before registering an application, in practice users choose to install applications that they trust by means of prior experiences or based on experts' knowledge.

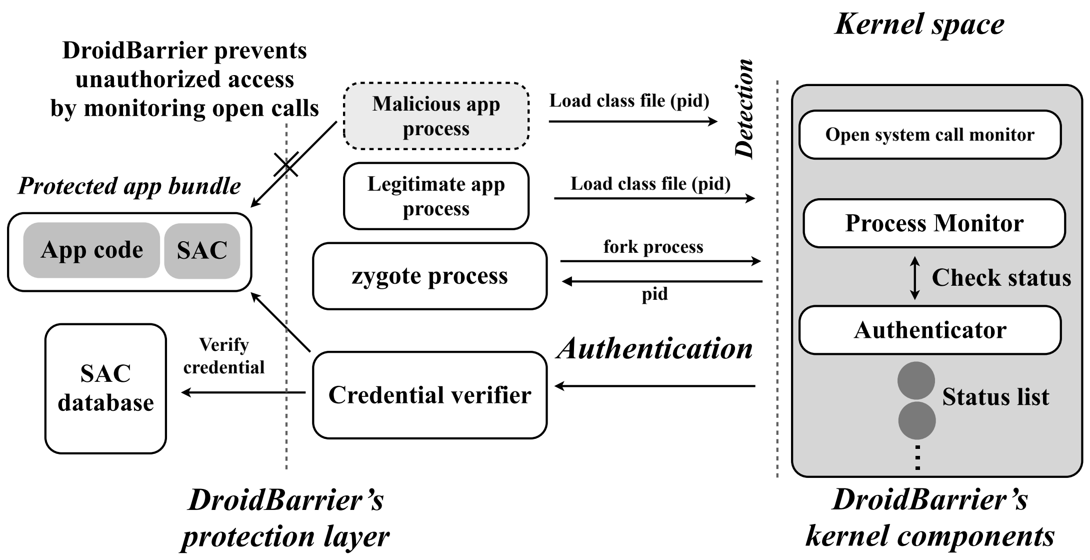


Figure 5.2.1: DroidBarrier performs three operations: credential storage and protection, runtime monitoring, and authentication of processes. Applications must obtain valid secure application credentials to ensure normal execution of their processes. An unauthenticated process is considered potentially malicious.

Credential generation. The registrar generates a credential that is computationally hard to guess. Among many ways to generate a credential, the registrar can use a strong pseudo random number generator.

There are two alternative approaches to our authentication mechanism based on secret credentials. First, registering a public checksum (e.g., a hash of application's class file) of an application bundle, and, recomputing the checksum at runtime to establish the authenticity and integrity of the application. However, using a *secret* credential, we eliminate the need for recomputing the checksum. Although, by checking the checksum one can determine if the application bundle's integrity was violated, we choose to protect application bundles (as described below) for verifying and *preserving* their integrity and disabling possible denial of service.

The second alternative approach is to use a form of developer signatures to establish some degree of trust. Using developer signatures unnecessarily complicates the design without security advantages. In fact, Android uses developer signatures, but, without an actual certification of the signature. Further, to perform a certification, either the kernel has to trust third party certification authorities, or, it has to establish the trust based on verifying developer public keys. The actual challenge is to prevent malicious applications from presenting fake or stolen signatures. Our design eliminates the need for third party certifications and verifications of public keys, yet, delivers the required level of trust.

To correctly establish the authenticity of an application, the registrar uses a two-way registration method. First, the registrar generates and stores a credential γ in A's bundle, forming a new application bundle A^*. We refer to an application bundle with an embedded credential, as a *protected application bundle*. This design choice is important to bind processes to specific executables in our runtime detection system, and, to protect the credential γ from attacks (described below).

Second, the registrar stores a copy of γ in a *SAC database*. The SAC database implements the unique set Γ, which is the set of reference for validating any credential. Maintaining a copy of γ in the SAC database prevents forgery and replay attacks on the credentials. Note that an application's credential γ is invalidated if γ is removed from Γ and if the application is reinstalled or deleted.

Credential protection. To fulfill the specification of credentials (Section 5.1.5) and the PA properties (Section 5.1.6), our design must fully protect the credentials generated by the registrar to preserve their integrity, without relying on file system permissions.

Our approach to this problem is to enforce access restrictions on the protected application bundles and the SAC database. To maintain integrity and confidentiality, we disable any process, other than the verifier process and the registrar, from write/read access to protected application bundles and the SAC database. We enforce this protection using DroidBarrier's kernel-side components by mediating the access to all open system calls and preventing unauthorized access to the SAC database.

5.2.2 Runtime System

DroidBarrier uses registered application credentials at runtime to detect and monitor processes and perform a mandatory authentication. In contrast to Quire [25] where interprocess communications are authenticated, our approach is to authenticate processes using pre-registered credentials. In our design, processes may fail to be authenticated, and thus, unauthenticated processes are considered potentially malicious.

DroidBarrier includes a *process monitor* to track the creation of processes by the Android's runtime. Process monitor relies on the authentication decision by DroidBarrier's *authenticator*. To maintain compatibility, we design these components as part of the Linux kernel without modifying Android's runtime. Our design strategy is to detect process creations, bind them to specific application bundles, and authenticate the processes according to registered credentials.

Runtime detection. Process monitor's policy is to regard every new process unauthorized until it is authenticated. The detection strategy of process monitor is to check the authentication status of applications at the time of creation. A process's status is either authenticated or unauthenticated.

The technical challenge in the design of DroidBarrier's runtime system is the semantic gap between the kernel and Dalvik virtual machine, which runs Android applications. Android's runtime system process, `zygote`, forks a new process when the user wants to run an application. At this point, it is not clear to the kernel, which application is loaded in the newly created process. To reconstruct the semantics, in addition to monitoring process creations, process monitor keeps track of file accesses by `zygote` to bind the loaded class file of the application to the newly forked process. When the newly forked process is bound to a Dalvik class file, the process monitor's runtime detection is complete and the process's identifier (PID) is sent for authenticator to proceed with the authentication.

Process authentication. DroidBarrier needs a reliable mechanism for authentication to prevent stealing credentials and spoofing legitimate processes. Android requires applications to be signed by developers [5]. However, as discussed earlier, developer signatures do not provide any reliable assurance about the authenticity of applications [5, 84]. We follow a design choice to mediate the authentication between DroidBarrier and user applications. Using our authentication mediation strategy, DroidBarrier performs the process authentication operation in three stages:

1. *Kernel-side checking.* A kernel-side component, authenticator, receives the authentication request from process monitor for a process P. Authenticator in DroidBarrier maintains a list of authentication status referred to as the *status list* L. L records the authentication status for each process, *authenticated*, or, *unauthenticated*. If P's status is unauthenticated, authenticator sends an authentication verification request to the verifier process. The format of the request is $(P.PID, P.path)$, where $P.PID$ is P's process ID and $P.path$ is the file path for the Dalvik class file that created P.

2. *Verification of credentials.* The verifier process loads the credential c from $P.path$ and the corresponding credential c' from the SAC database. If $c = c'$, then the verifier sends a success response message back to authenticator. Otherwise, the verifier sends a failure response message to authenticator.

3. *Status update.* When authenticator receives the response message from the verifier, the authenticator updates P's authentication status in L, accordingly.

Our runtime system adheres to the process authentication properties (Section 5.1.6) and ensures mandatory authentication of processes. In the next section, we analyze our design and describe the security guarantees and limitations.

5.2.3 Security Analysis

In this section we present the security guarantees provided by DroidBarrier, and, analyze the security of DroidBarrier from three perspectives. First, we provide an analysis of the security of the credentials. Second, we describe the security of DroidBarrier code and data. Third, we discuss the limitations of our model and design.

Security guarantees. DroidBarrier guarantees that all processes are authenticated. This property ensures that stealthy applications that were installed without the user's consent are detected. DroidBarrier also guarantees that the integrity and confidentiality of secure application credentials and protected application bundles are maintained.

Security of credentials. To secure application credentials, we study various attacks that can occur on the credentials and provide solutions accordingly. An important attack on DroidBarrier is to steal or corrupt application credentials. Attacks on credentials can occur in a number of ways:

- An attacker may include a credential in a malware's application bundle according to our credential specifications. This attack fails since the fake credential created by malware is not registered with the SAC database.
- Under our attack model (Section 5.1.4), an attacker may attempt to read the credentials from the disk by breaking Android's sandbox. We protect credentials by means of the protected application bundles in DroidBarrier that are not readable by user processes. Note that readability constraints on applications do not prevent loading class files. That is, we especially allow `zygote` to load class files, after authenticating `zygote`.
- We prevent memory attacks on credentials by mediating the authentication (Section 5.2.2). Thus, credentials are never stored in application heaps or stacks at runtime.
- Integrity violation attacks on credentials involve corrupting the credential on the disk or in memory, causing denial of service to legitimate applications. Such attacks are also prevented by disabling write access on protected application bundles and mediating the authentication.

Protecting DroidBarrier code and data. DroidBarrier's process monitor and authenticator execute in kernel mode. This guarantees the isolation of process monitor and authenticator data from user space attacks under the reasonable assumption of a trusted Linux kernel (Section 5.1.4). DroidBarrier is protected from malicious applications using `root` privileges by preventing dynamic patches to the kernel (through loadable device drivers) from disabling the functionality of DroidBarrier.

Process monitor and authenticator communicate directly on shared data structures. Authenticator and the verifier process communicate through a shared memory that is managed by the kernel. This shared memory is not accessible by any other process and may not be read from or written to. This protection is implemented in DroidBarrier's kernel-side components without relying on Linux kernel's permission system.

The verifier component is protected in two ways. The executable code is kept in a protected application bundle with an additional restriction of disallowing the Android application manager to remove or reinstall it. At runtime, the verifier process is automatically created by DroidBarrier's kernel-side components when the `zygote` process is created. Further execution of the verifier application is denied.

Limitations of DroidBarrier. DroidBarrier specifically targets processes that are created by malicious applications. This is a critical category of attacks that is used by modern Android malware [93]. Embedded attacks that run entirely within the boundaries of a legitimate process cannot be detected by our current mechanism. However, we envision extensions of our authentication model with more detailed inspections to deliver code-level authentication.

In principle, rootkit attacks (such as return-oriented rootkits [42] and system call obfuscation [81]) can cause kernel integrity and confidentiality violations. Although DroidBarrier is not designed to specially prevent rootkits, assuming that the kernel is initially free of malicious code, DroidBarrier prevents further malicious code executions. For instance, modern rootkits need to use return-to-user attacks [51] and run the code stealthily in user space. DroidBarrier detects this type of attack when the prerequisite of a successful attack is to first run a user space task that can receive the execution, for example, from a `NULL` dereferencing in the kernel.

5.3 Implementation

In this section we describe a prototype of our DroidBarrier. Our implementation involves a patch to the Linux kernel for process monitor and authenticator. We use Android Honeycomb 3.2 with the Linux kernel version 2.6.36.4 for our implementation platform. We choose to run and test our implementation on a physical Android device (Samsung Galaxy tab) instead of a virtual x86-based Android machine. Below, we present a highlight of our implementation features.

1. Kernel modification is minimal. DroidBarrier performs necessary check points and authentication for processes through manual check points that we insert in the beginning of monitored kernel functions. These check points transfer execution to DroidBarrier before completing the monitored kernel function.

2. DroidBarrier does not modify the original semantics of process management and creation functions in Linux. We develop our prototype based an asynchronous authentication design.

3. Our prototype is compatible with the stock Android runtime, Binder, and all other Android operating system components, without any modifications.

4. We fully implement DroidBarrier in C using Android's native APIs and Linux kernel libraries.

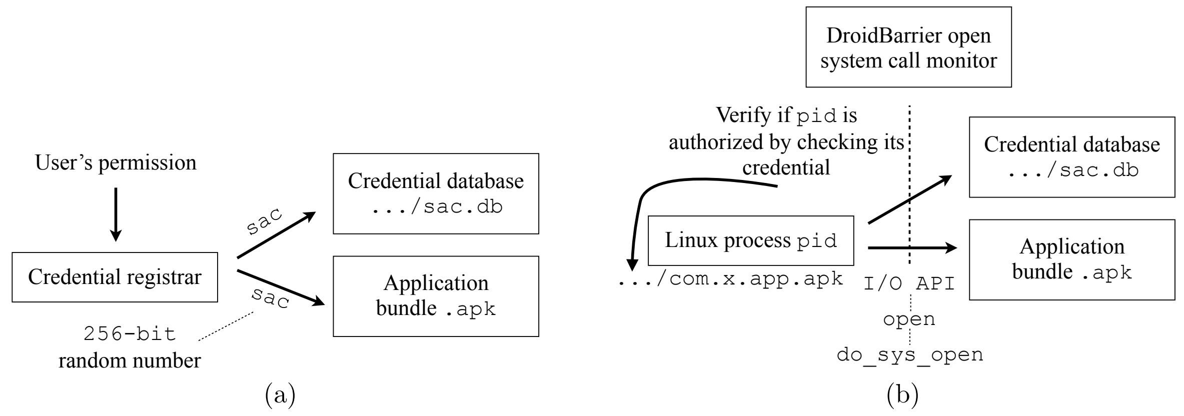


Figure 5.3.1: (a) The registrar creates new credentials for applications and uses the credentials to label the apk bundles. (b) DroidBarrier protects credentials by monitoring open system calls and checking for permissions.

In the following we first describe the implementation details of credential registrar followed by process monitor, authenticator, and verifier.

Credential registrar. We implement a credential registrar that install application credentials for an Android application bundle. The registrar uses a random number generation function to generate a credential, and, labels the application bundle (with the `apk` extension) with the newly created credential. The registrar also records the new credential in a sequential database of keys and application names for reference at authentication time. Our labeling of the `apk` bundle does not alter the functionality of the application in any way. The label only includes an additional header and the credential in a format that DroidBarrier can recognize.

Credential protection. We implement our credential protection mechanism (Section 5.2.1) by implementing a light-weight access control system in the kernel. We place checkpoints in the beginning of `do_sys_open`. Before returning a file descriptor `fd` to the requesting process, we check for two conditions. First, we check if the requested file path is a protected application bundle, by reconstructing the full file path from the process's current directory. Then, we verify if the requested file path is an executable. If the condition is true, we check if the requesting process is the registrar, the credential verifier or `zygote`. According to our policies (Section 5.1.6), DroidBarrier denies access to executables for all other processes. Second, we check the file path against our credential database. For the credential database, we only return an `fd`, it the calling process is either the credential verifier or the registrar.

Process monitor. We implement the process monitor by inserting check points in specific kernel functions. To maintain performance, we avoid using existing Linux APIs to trace kernel functions (such as `kprobe` or `ptrace`). To monitor the creation of processes by the Android's runtime, we insert check points in `do_fork`, for Android's Dalvik applications, and `do_execve`, for native applications loaded directly by the kernel. The `zygote` process calls `fork()` to create a new process and load a Dalvik class in it. We record the generated

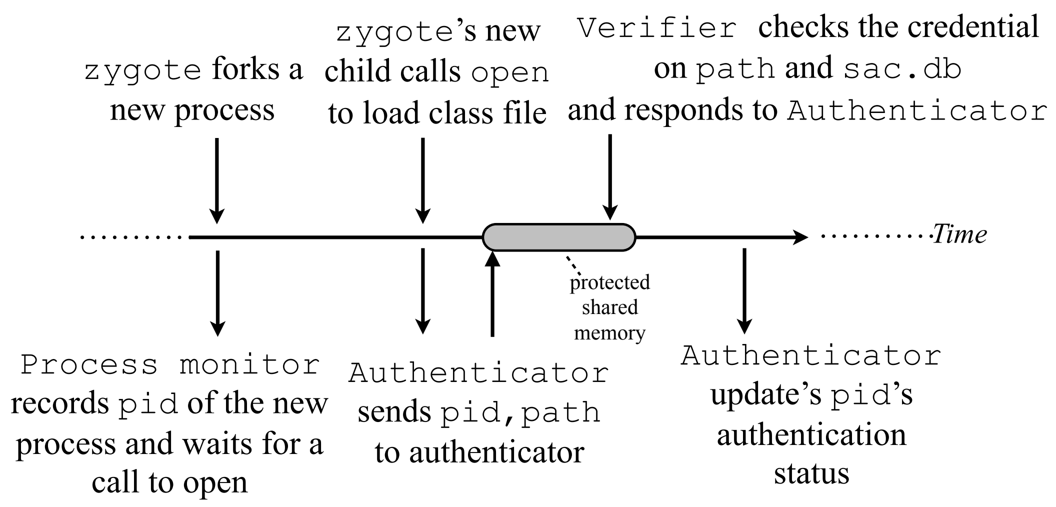

zygote forks a new process
zygote's new child calls open to load class file
Verifier checks the credential on path and sac.db and responds to Authenticator
Time
protected shared memory
Process monitor records pid of the new process and waits for a call to open
Authenticator sends pid,path to authenticator
Authenticator update's pid's authentication status

Figure 5.3.2: Monitoring of a process starts at the time zygote forks the process. Process monitor prepares an authentication pool for authenticator, which sends verification requests to the verifier process. The interactions above the time line are in user space.

`pid` in `do_fork` and track `zygote`'s subsequent system calls so that we bind the new `pid` to its application bundle.

DroidBarrier tracks the loading of Dalvik class files through our check point in the `do_sys_open` function. When the process `pid` (that must be the child of `zygote`) loads a class file from the file system, the authentication operation can proceed to verify `pid`'s credentials. Since the authentication is asynchronous and waits for the open system call, DroidBarrier maintains a dynamic pool of awaiting authentication processes (an array `requests` of type `struct auth_request`) for authenticator to perform the authentication.

Authenticator and verifier. The authenticator creates and manages a shared memory with the verifier process in user space. The shared memory is not visible to any process. We implement this restriction in the check point in `do_sys_open` function. DroidBarrier checks for permissions before the open system call returns a file descriptor. The permission check maintains a list of permissions for specific file descriptors. Access to the shared memory file descriptor is denied to all processes except the verifier (for which we record the `pid`).

The authenticator continuously checks the `requests` pool for fresh authentication requests. Authenticator sends each request in a special string format to the verifier using a first come first served schedule. Once the verifier checks the credentials for the process, it sends a response back on the same shared memory to the authenticator. The authenticator informs the process monitor about completed authentication via updating the status of the request in `requests`.

The verifier's task is to check the credentials using a file path that is sent by the authenticator. The verifier opens the corresponding file path, if that is a Dalvik class file and contains a credential, the verifier reads the credential for checking. The verifier checks if the credential exists in the SAC database. The SAC database is a sequential file on the disk containing application names, file paths, and credentials. If the credential matches the process file path, verifier creates a success message and sends it to the authenticator.

5.4 Evaluation

We evaluate the performance of our implementation prototype and investigate the following:

1. DroidBarrier performs permission checks prior to creating a file descriptor in the open system call. How these permission checks affect the I/O performance?
2. What is the performance penalty on process creations?
3. Measuring Dalvik runtime performance under DroidBarrier.

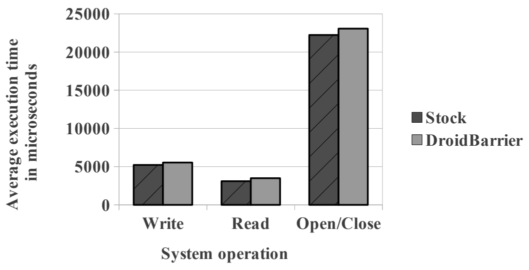
In the rest of this section, we describe our experimental setup followed by three sets of experiments for I/O and process creation. Our results show efficient performance of DroidBarrier with low extra overhead in I/O performance, and, negligible performance penalty in process creation.

Experimental setup. Our evaluation focuses on the effect of DroidBarrier on the performance of the Linux kernel in Android. We run all the experiments on a Samsung Galaxy Tab 10 (with model number P7510) running Android Honeycomb 3.2 with the Linux kernel version 2.6.36.4.

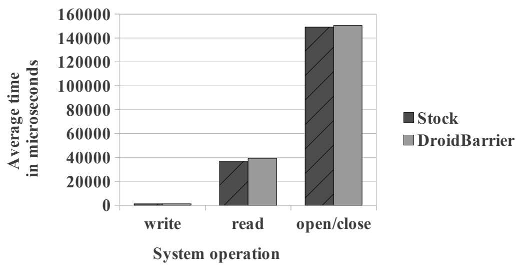
For our evaluations we needed a benchmarking suite that can accurately measure I/O and process creation. There are existing suites such as lmbench[1] and UnixBench[2] that conveniently run on Linux distributions for x86 machines. Per our research, these suites have not been ported to embedded Linux running on the ARM architecture. Thus, we modified some of the existing code in lmbench and UnixBench for compatibility with the ARM Linux. Our collection of benchmarking suite for native code in Android will be available as an open source project.

When running the modified kernel with DroidBarrier, we periodically simulate the authentication of processes as if a user is consistently launching new applications. When performing experiments, there was about a total of 130–150 running Linux processes. While experimenting, we monitored the memory usage. Either using DroidBarrier's kernel or the stock Android Linux kernel, the available free memory was always under 100 MB.

[1]http://www.bitmover.com/lmbench/
[2]http://code.google.com/p/byte-unixbench/



(a)



(b)

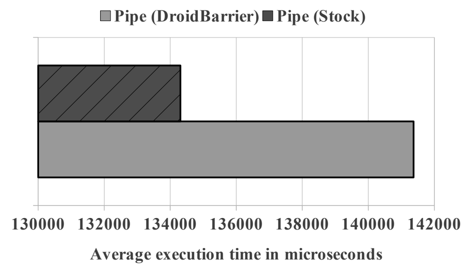
Figure 5.4.1: (a) Four I/O operations: write, read, open, and close. Maximum downgrade is 12.92%. (b) Four I/O operations: write, read, open, and close for Dalvik applications. Maximum downgrade is 6.26%. For better visibility, we divide the result for open/close by 10. The results show an average of several runs.

I/O performance. I/O performance experiments show a consistently efficient performance of DroidBarrierwith the performance penalty not exceeding 13%. For measuring I/O performance, we evaluate the performance of the I/O operations write, read, open, and close system calls as well as the performance of piping. Note that for protecting secure application credentials, DroidBarrier only includes a monitoring on the open system call. For each measurement, the I/O operation is repeated 10000 times continuously. We measure the time taken to perform the whole 10000 calls to the I/O function. The experiments on read and write include calls to open and close. For open and close, in each iteration we call open followed by a close. We performed 250 runs of each loop to collect an average performance value.

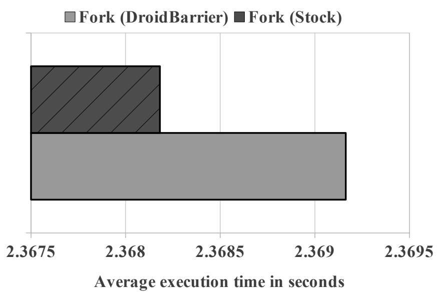
In the results of our experiments (depicted in Figure 5.4.1a), we have an average maximum of 12.92% downgrade for the read calls, an average minimum of 3.76% downgrade for open/close, and for write calls there is an average downgrade of 6.01%.

To examine the performance of Dalvik applications under DroidBarrier, we conducted an I/O experiment by making file writes, reads, open and close using the `BufferedWriter` and `BufferedReader`. The open and close calls involve opening a file using `FileWriter`, which is used to create a `BufferedWriter` and we close the `BufferedWriter` subsequently. We developed an application to perform 1000 iterations of each I/O operation (we call open and close calls together in one iteration). The results of Figure 5.4.1b shows the average performance for 200 runs of all experiments.

We measure performance on pipes by using the pipe system call. Figure 5.4.2a depicts the results of our experiments on pipes. After creating a pipe, a loop of 10000 iterations performs write system calls on the pipe followed by read system calls. We measure the total time for executing the loop. The average performance downgrade is 5.26% for a total of 140 runs, which is reasonably efficient.



(a)



(b)

Figure 5.4.2: (a) The average performance downgrade in this experiment is 5.26%. (b) An insignificant downgrade of 0.041%, in process creation.

Our experiments show an average maximum performance downgrade of 6.26% in `BufferedReader`. Calls to `BufferedWriter` has on average a performance downgrade of 1.48% under DroidBarrier. The minimum performance downgrade is for calls to open and close with an average of 0.927% downgrade.

Process creation performance. Android heavily uses fork for process creations. Since DroidBarrier performs the authentication asynchronously, we do not see major performance downgrade for forking a process. The average performance downgrade is 0.041% (depicted in Figure 5.4.2b), which is statistically insignificant. We measure the fork system call in loops of 1000 iterations.

Computational performance. We use polybench suite of benchmarks to execute three computational intensive algorithms. The experiments show an efficient performance under DroidBarrier with a downgrade not exceeding 2%. The experiments include a recursive implementation of towers of Hanoi, an LU factorization algorithm, and a dynamic programming algorithm. We modified polybench to compile it for the ARM architecture. We run the three algorithms 60 times on the stock Android Linux and DroidBarrier's modified Linux.

The experiments' results are depicted in Figure 5.4.3. The average increase in execution time for the experiments is: towers of Hanoi 0.34%, LU factorization 1.31%, and dynamic programming 1.14%.

5.5 Summary

We presented a general model for authenticating Android application processes that is capable of providing high assurance proof of provenance about the Linux application processes running on an Android-enabled device. We achieve the high assurance by developing an authentication model that uses secure application credentials, maintained and protected by our runtime system, to authenticate processes and bind their identity to legitimate applications installed on the device. Our authentication approach guarantees protecting the system

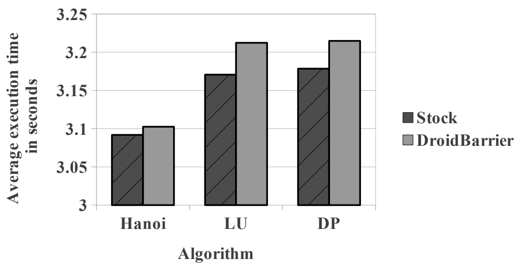


Figure 5.4.3: Maximum average performance downgrade in this experiment is 1.14%. The three measured algorithms are towers of Hanoi (10 disks), LU factorization, and dynamic programming. The results show an average of several runs.

from execution of malicious applications that may exploit the many system and application vulnerabilities to be installed on the device. Our runtime system, DroidBarrier, monitors every application process that either correspond to a Dalvik application or a native one to enforce a mandatory authentication, and therefore, isolation of unauthorized malicious applications.

Chapter 6

Conclusions

6.1 Impact and Advantages

In this dissertation, we introduced models and algorithms for systematic analysis of large enterprise networks. Our analysis takes a novel approach and extends state of the art by developing the problem of optimal distribution of defensive resources across a large network.

This work makes a key contribution by extending the analysis of attack graphs with a formal measurement of the threat from mobile devices that repeatedly join dynamic networks. Mobile devices cause increased threat to the security of a network, which our model can accurately quantify. This is an important step towards understanding the significance of the threats from mobile devices. Nevertheless, more elements of mobile devices may be formalized, which include a measurement of mobility with respect to wireless signal strength, reasoning about the location of the device in various subnetworks, and the volume of device communications with critical network components.

Understanding the effect of mobile devices in the context of a rigorous probabilistic measurement of attack success assisted in developing a fine grained model that captures attack steps and security defense plans simultaneously. Our optimization model recognizes attack steps generated in the form of an attack graph. The system administrators can designate potential defenses in the same model such that an overall optimized configuration subject to attack steps and defenses could be computed.

The results of our optimization model are important from two aspects. First, using our model, system administrators have a provable optimization result that accurately demonstrates the nearly best choices for distributing defense mechanisms (such as defense policies or solutions). Second, our model provides an efficient computational method for computing the results. The nonlinear minimax problem that we developed are solvable using state of the art techniques.

Our analysis shades light on security bottlenecks and configuration problems in a large network. Nevertheless, we realize the need for providing security solutions to address security problems in individual network component. In this work, we focused on an important aspect of security by introducing the idea of process authentication. In this security model, we organize applications as authenticated parties that require valid credentials to run. An application is represented by its processes, for which the authentication is carried out.

The importance and advantage of the idea of authentication within one computing unit is that it does not produce false results. In comparison to behavioral analysis, control flow methods, or other methods that employ a form of intelligence in detecting anomalies and malicious executions, our process authentication takes a firm binary approach towards application processes. A process is either created from a legitimate application or it is from an application whose legitimacy is unknown. Such an approach provides a flexible way for the user to designate applications are legitimate and thus have full control on what can execute in the system.

Our process authentication provides a suitable security platform for mobile operating systems. As demonstrated in this work, we applied our process authentication model to the Android platform. Our results indicate feasibility of the approach, efficiency of the execution, and powerful results. Using this model, the kernel is capable of discriminating applications according to their legitimacy. Thus, the kernel can overcome the security vulnerabilities from its permission systems and the weaknesses of Android's sandboxes.

6.2 Future Directions

We envision extending the work presented in this dissertation in two directions. First, we extend our analysis model towards including additional optimization features. In many scenarios it is desired to fine tune existing security configurations without introducing additional security defenses. The problem is to represent existing configurations in a formalized model, define security rules and policies that the configuration must adhere to, and computing a system and network level configuration setting that achieves a better security.

To develop an advanced optimization model, we need to advance our probabilistic reasoning about network attacks by considering available attack data (which may require additional efforts for interpretation and semantics). Moreover, a formalization of attacker utilities, behavior, and thinking is needed. In particular, a model must capture attack constraints outside network and system configurations settings, a behavioral model showing detailed predictable attack actions and steps, and understanding realistic attack goals such as multiple objective goals. In fact, our current analysis is limited by the ability of attack graphs to reason about a single objective of the attack. In some attack scenarios, attack objective may not even be deterministic. Thus, an attacker may decide to choose a particular goal that is unplanned.

Our second future direction involves extending our system security solution. Our model currently formalizes a novel authentication mechanism. We envision to complement our model with a solution to examine the legitimacy of applications before designation as legitimate. Such examination cannot be an exact science, and requires employing intelligent algorithms. The problem is to form a model specific to the target platform that is scalable and produces the least false results. In addition to determination of legitimacy of applications, our authentication mechanism needs a more advanced authorization with a smart engine to determine more access rights with less policy configurations. In particular, the authorization model must be able to analyze anomaly at runtime and have a more fine-grained control on the execution of applications.

Bibliography

[1] grsecurity. http://www.grsecurity.net/.

[2] M. Albanese, S. Jajodia, and S. Noel. Time-efficient and cost-effective network hardening using attack graphs. In *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*, pages 1–12, june 2012.

[3] H. M. Almohri, D. D. Yao, and D. Kafura. Identifying native applications with high assurance. In *Proceedings of the second ACM conference on Data and Application Security and Privacy*, pages 275–282, New York, NY, USA, 2012. ACM.

[4] K. Barr, P. Bungale, S. Deasy, V. Gyuris, P. Hung, C. Newell, H. Tuch, and B. Zoppis. The vmware mobile virtualization platform: is that a hypervisor in your pocket? *ACM SIGOPS Operating Systems Review*, 44(4):124–135, Dec. 2010.

[5] D. Barrera, J. Clark, D. McCarney, and P. C. van Oorschot. Understanding and improving app installation security mechanisms through empirical analysis of android. In *Proceedings of the second ACM workshop on security and privacy in smartphones and mobile devices*, SPSM '12, pages 81–92, New York, NY, USA, 2012. ACM.

[6] L. Bauer, J. Ligatti, and D. Walker. Composing expressive runtime security policies. *ACM Transactions on Software Engineering and Methodology*, 18(3):9:1–9:43, June 2009.

[7] M. S. Bazaraa, H. D. Sherali, and C. M. Shetty. *Nonlinear Programming.* John Wiley and Sons, Inc., 2005.

[8] S. Bhatt, W. Horne, and P. Rao. On computing enterprise IT risk metrics. In *Future Challenges in Security and Privacy for Academia and Industry*, volume 354 of *IFIP Advances in Information and Communication Technology*, pages 271–280. Springer Boston, 2011.

[9] D. P. Bovet and M. Cesati. *Understanding the Linux kernel.* O'Reilly, 2006.

[10] P. Brady. Anatomy & physiology of an android, 2008. https://sites.google.com/site/io/anatomy–physiology-of-an-android.

[11] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A.-R. Sadeghi. XManDroid: A New Android Evolution to Mitigate Privilege Escalation Attacks. Technical Report TR-2011-04.

[12] S. Bugiel, L. Davi, A. Dmitrienko, S. Heuser, A.-R. Sadeghi, and B. Shastry. Practical and lightweight domain isolation on android. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM'11, pages 51–62, New York, NY, USA, 2011. ACM.

[13] R. H. Byrd, N. I. M. Gould, J. Nocedal, and R. A. Waltz. On the convergence of successive linear-quadratic programming algorithms. *SIAM J. on Optimization*, 16(2):471–489, June 2005.

[14] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about Datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146 –166, 3 1989.

[15] R. Ceron. The GNU linear programming kit, part 1: Introduction to linear optimization, 2006. http://www.ibm.com/developerworks/linux/library/l-glpk1/.

[16] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 239–252, New York, NY, USA, 2011. ACM.

[17] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in Android. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services (MobiSys)*, June 2011.

[18] J.-L. Cooke and D. Bryson. Strong cryptography in the Linux kernel. In *Proceedings of the 2003 Linux Symposium*, pages 139–144, 2003.

[19] B. Coskun and P. Giura. Mitigating sms spam by online detection of repetitive near-duplicate messages. In *to appear in IEEE ICC'12 Symposium on Communication and Information Systems Security*, 2012.

[20] W. Dai, T. P. Parker, H. Jin, and S. Xu. Enhancing data trustworthiness via assured digital signing. *IEEE Transactions on Dependable and Secure Computing*, 9 (6):838 – 851, 2012.

[21] C. Dall, J. Andrus, A. Van't Hof, O. Laadan, and J. Nieh. The design, implementation, and evaluation of cells: A virtual smartphone architecture. *ACM Trans. Comput. Syst.*, 30(3):9:1–9:31, Aug. 2012.

[22] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Privilege Escalation Attacks on Android. In *Information Security*, volume 6531 of *Lecture Notes in Computer Science*, pages 346–360. Springer Berlin / Heidelberg, 2011.

[23] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Privilege escalation attacks on android. In *Proceedings of the 13th international conference on Information security*, ISC'10, pages 346–360, Berlin, Heidelberg, 2011. Springer-Verlag.

[24] R. Dewri, N. Poolsappasit, I. Ray, and D. Whitley. Optimal security hardening using multi-objective optimization on attack tree models of networks. In *Proceedings of the 14th ACM conference on Computer and Communications Security*, CCS '07, pages 204–213, New York, NY, USA, 2007. ACM.

[25] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. Quire: lightweight provenance for smart phone operating systems. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 23–23, Berkeley, CA, USA, 2011. USENIX Association.

[26] K. O. Elish, D. Yao, and B. G. Ryder. User-centric dependence analysis for identifying malicious mobile apps. In *Proceedings of the Workshop on Mobile Security Technologies (MoST)*, May 2012. In conjunction with the IEEE Symposium on Security and Privacy.

[27] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[28] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 235–245, New York, NY, USA, 2009. ACM.

[29] W. Enck, M. Ongtang, and P. McDaniel. Understanding Android Security. *IEEE Security and Privacy*, 7(1):50–57, Jan. 2009.

[30] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner. A survey of mobile malware in the wild. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, SPSM '11, pages 3–14, New York, NY, USA, 2011. ACM.

[31] S. Fenz. An ontology- and Bayesian-based approach for determining threat probabilities. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pages 344–354, New York, NY, USA, 2011. ACM.

[32] R. Fletcher, N. I. M. Gould, S. Leyffer, P. L. Toint, and A. Wächter. Global convergence of a trust-region sqp-filter algorithm for general nonlinear programming. *SIAM J. on Optimization*, 13(3):635–659, Aug. 2002.

[33] B. Ford and R. Cox. Vx32: lightweight user-level sandboxing on the x86. In *Proceedings of the 2008 USENIX Annual Technical Conference*, pages 293–306, Berkeley, CA, USA, 2008. USENIX Association.

[34] M. Fox, J. Giordano, L. Stotler, and A. Thomas. SELinux and grsecurity: A Case Study Comparing Linux Security Kernel Enhancements, 2003.

[35] M. Frigault, L. Wang, A. Singhal, and S. Jajodia. Measuring network security using dynamic Bayesian network. In *Proceedings of the 4th ACM Workshop on Quality of Protection*, QoP '08, pages 23–30, New York, NY, USA, 2008. ACM.

[36] Google, Inc. Android Dev. Guide, Security and Permissions. http://developer.android.com/guide/topics/security/security.html.

[37] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. Riskranker: scalable and accurate zero-day android malware detection. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, MobiSys '12, pages 281–294, New York, NY, USA, 2012. ACM.

[38] K. Gudeth, M. Pirretti, K. Hoeper, and R. Buskey. Delivering secure applications on commercial mobile devices: the case for bare metal hypervisors. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '11, pages 33–38, New York, NY, USA, 2011. ACM.

[39] B. Hicks, S. Rueda, L. St.Clair, T. Jaeger, and P. McDaniel. A logical specification and analysis for SELinux MLS policy. *ACM Transactions on Information and Systems Security*, 13(3):26:1–26:31, July 2010.

[40] Z. M. Hong Chen, Ninghui Li. Analyzing and comparing the protection quality of security enhanced operating systems. In *Proceedings of the 16th Annual Network & Distributed System Security Symposium*, 2009.

[41] H. Huang, S. Zhang, X. Ou, A. Prakash, and K. Sakallah. Distilling critical attack graph surface iteratively through minimum-cost sat solving. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 31–40, New York, NY, USA, 2011. ACM.

[42] R. Hund, T. Holz, and F. C. Freiling. Return-oriented rootkits: bypassing kernel code integrity protection mechanisms. In *Proceedings of the 18th conference on USENIX security symposium*, SSYM'09, pages 383–398, Berkeley, CA, USA, 2009. USENIX Association.

[43] K. Ingols, M. Chu, R. Lippmann, S. Webster, and S. Boyer. Modeling modern network attacks and countermeasures using attack graphs. In *Annual Computer Security Applications Conference*, pages 117 –126, dec. 2009.

[44] K. Ingols, R. Lippmann, and K. Piwowarski. Practical attack graph generation for network defense. In *Computer Security Applications Conference, 2006.*, pages 121 –130, 12 2006.

[45] T. Jaeger, R. Sailer, and U. Shankar. PRIMA: policy-reduced integrity measurement architecture. In *Proceedings of the 11th ACM symposium on Access control models and technologies*, SACMAT '06, pages 19–28, New York, NY, USA, 2006. ACM.

[46] T. Jaeger and R. Sandhu. *Operating System Security*. Morgan & Claypool, 2008.

[47] S. Jajodia, S. Noel, and B. O'Berry. Topological analysis of network attack vulnerability. In V. Kumar, J. Srivastava, and A. Lazarevic, editors, *Managing Cyber Threats: Issues, Approaches and Challanges*, chapter 5. Kluwer Academic Publisher, 2003.

[48] S. Jha, O. Sheyner, and J. Wing. Two formal analyses of attack graphs. In *Proceedings of the* 15th *IEEE workshop on Computer Security Foundations*, CSFW '02, Washington, DC, USA, 2002. IEEE Computer Society.

[49] X. Jiang, X. Wang, and D. Xu. Stealthy Malware Detection and Monitoring Through VMM-based "out-of-the-box" Semantic View Reconstruction. *ACM Transactions on Information Systems Security*, 13:12:1–12:28, March 2010.

[50] M. T. Jones. Access the Linux kernel using the `/proc` filesystem, 2006. *http://www.ibm.com/developerworks/linux/library/l-proc.html.*

[51] V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis. kguard: lightweight kernel protection against return-to-user attacks. In *Proceedings of the 21st USENIX conference on Security symposium*, Security'12, pages 39–39, Berkeley, CA, USA, 2012. USENIX Association.

[52] C. Kil, E. C. Sezer, A. M. Azab, P. Ning, and X. Zhang. Remote attestation to dynamic system properties: Towards providing complete system integrity evidence. In *Dependable Systems and Networks*, pages 115–124, 2009.

[53] T. Kim and N. Zeldovich. Making Linux protection mechanisms egalitarian with UserFS. In *Proceedings of the 19th USENIX conference on Security*, pages 13–27, Berkeley, CA, USA, 2010. USENIX Association.

[54] B. Li, J. Li, T. Wo, C. Hu, and L. Zhong. A VMM-based system call interposition framework for program monitoring. In *Proceedings of the 16th IEEE International Conference on Parallel and Distributed Systems*, ICPADS '10, pages 706–711, Washington, DC, USA, 2010. IEEE Computer Society.

[55] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the Linux operating system. In *Proceedings of the 2001 USENIX Annual Technical Conference*, Berkeley, CA, 2001. USENIX Association.

[56] L. Lu, V. Yegneswaran, P. Porras, and W. Lee. BLADE: an attack-agnostic approach for preventing drive-by malware infections. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 440–450, New York, NY, USA, 2010. ACM.

[57] T. Luo, H. Hao, W. Du, Y. Wang, and H. Yin. Attacks on webview in the android system. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, pages 343–352, New York, NY, USA, 2011. ACM.

[58] L. McVoy and C. Staelin. lmbench: portable tools for performance analysis. In *Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*, pages 23–23, Berkeley, CA, USA, 1996. USENIX Association.

[59] V. Mehta, C. Bartzis, H. Zhu, E. Clarke, and J. Wing. Ranking Attack Graphs. In *Recent Advances in Intrusion Detection*, volume 4219, pages 127–144. Springer Berlin, 2006.

[60] P. Mell, K. Scarfone, and S. Romanosky. Common Vulnerability Scoring System. *IEEE Security & Privacy*, 4(6):85–89, 11 2006.

[61] D. Muthukumaran, A. Sawani, J. Schiffman, B. M. Jung, and T. Jaeger. Measuring integrity on mobile phone systems. In *Proceedings of the 13th ACM Symposium on Access control Models and Technologies*, SACMAT '08, pages 155–164, New York, NY, USA, 2008. ACM.

[62] S. Noel and S. Jajodia. Optimal IDS sensor placement and alert prioritization using attack graphs. *Journal of Network and Systems Management*, 16(3):259–275, Sept. 2008.

[63] S. Noel, S. Jajodia, L. Wang, and A. Singhal. Measuring security risk of networks using attack graphs. *International Journal of Next-Generation Computing*, 1(1), July 2010.

[64] X. Ou, W. F. Boyer, and M. A. McQueen. A scalable approach to attack graph generation. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, pages 336–345, New York, NY, USA, 2006. ACM.

[65] X. Ou, S. Govindavajhala, and A. W. Appel. Mulval: A logic-based network security analyzer. In *Proceedings of the 14th Conference on USENIX Security Symposium*, pages 113–128, Berkeley, CA, USA, 2005. USENIX Association.

[66] L. Page, S. Brin, R. Motwani, and T. Winograd. The Pagerank citation ranking: Bringing order to the web. Technical report, Stanford Digital Library Technologies Project, September 1999.

[67] F. Palacios-Gomez, L. Lasdon, and M. Engquist. Nonlinear optimization by successive linear programming. *Management Science*, 28(10):1106–1120, 1982.

[68] J. Pamula, S. Jajodia, P. Ammann, and V. Swarup. A weakest-adversary security metric for network configuration security analysis. In *Proceedings of the 2nd ACM Workshop on Quality of Protection*, QoP '06, pages 31–38, New York, NY, USA, 2006. ACM.

[69] J. Park, B. Kim, S.-R. Kim, J. H. Yoon, and Y. Cho. Performance analysis of security enforcement on android operating system. In *Proceedings of the 2011 ACM Symposium on Research in Applied Computation*, RACS '11, pages 282–286, New York, NY, USA, 2011. ACM.

[70] Y. Park, C. Lee, C. Lee, J. Lim, S. Han, M. Park, and S.-J. Cho. Rgbdroid: a novel response-based approach to android privilege escalation attacks. In *Proceedings of the 5th USENIX conference on Large-Scale Exploits and Emergent Threats*, LEET'12, pages 9–9, Berkeley, CA, USA, 2012. USENIX Association.

[71] N. Poolsappasit, R. Dewri, and I. Ray. Dynamic security risk management using Bayesian attack graphs. *IEEE Transactions on Dependable and Secure Computing*, 9(1):61–74, Jan 2012.

[72] M. Rajagopalan, M. Hiltunen, T. Jim, and R. Schlichting. Authenticated system calls. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks*, pages 358–367, June 2005.

[73] M. Rajagopalan, M. A. Hiltunen, T. Jim, and R. D. Schlichting. System call monitoring using authenticated system calls. *IEEE Transactions on Dependable and Secure Computing*, 3:216–229, July 2006.

[74] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *Proceedings of the 13th USENIX Security Symposium*, SSYM'04, pages 16–16, Berkeley, CA, USA, 2004. USENIX Association.

[75] R. E. Sawilla and X. Ou. Identifying Critical Attack Assets in Dependency Attack Graphs. In *Proceedings of the 13th European Symposium on Research in Computer Security: Computer Security*, ESORICS '08, pages 18–34, Berlin, Heidelberg, 2008. Springer-Verlag.

[76] Z. C. Schreuders, T. McGill, and C. Payne. Empowering End Users to Confine Their Own Applications: The Results of a Usability Study Comparing SELinux, AppArmor, and FBAC-LSM. *ACM Trans. Inf. Syst. Secur.*, 14(2):19:1–19:28, Sept. 2011.

[77] A. Shabtai, Y. Fledel, and Y. Elovici. Securing android-powered mobile devices using SELinux. *Security Privacy, IEEE*, 8(3):36–44, may-june 2010.

[78] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J. M. Wing. Automated Generation and Analysis of Attack Graphs. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, Washington, DC, USA, 2002. IEEE Computer Society.

[79] S. W. Smith. *Trusted Computing Platforms: Design and Applications*. Springer-Verlag, Secaucus, NJ, USA, 2004.

[80] K. Z. Snow, S. Krishnan, F. Monrose, and N. Provos. Shellos: enabling fast detection and forensic analysis of code injection attacks. In *Proceedings of the 20th USENIX conference on Security*, SEC'11, pages 9–9, Berkeley, CA, USA, 2011. USENIX Association.

[81] A. Srivastava, A. Lanzi, J. Giffin, and D. Balzarotti. Operating system interface obfuscation and the revealing of hidden operations. In *Proceedings of the 8th international conference on Detection of intrusions and malware, and vulnerability assessment*, DIMVA'11, pages 214–233, Berlin, Heidelberg, 2011. Springer-Verlag.

[82] D. Stefan, C. Wu, D. Yao, and G. Xu. Knowing where your input is from: Kernel-level provenance verification. In *Proceedings of the 8th International Conference on Applied Cryptography and Network Security (ACNS)*, pages 71–87. Springer-Verlag, 2010.

[83] T. Van Overveldt, C. Kruegel, and G. Vigna. Flashdetect: actionscript 3 malware detection. In *Proceedings of the 15th international conference on Research in Attacks, Intrusions, and Defenses*, RAID'12, pages 274–293, Berlin, Heidelberg, 2012. Springer-Verlag.

[84] T. Vidas, D. Votipka, and N. Christin. All your droid are belong to us: a survey of current android attacks. In *Proceedings of the 5th USENIX conference on Offensive technologies*, WOOT'11, pages 10–10, Berkeley, CA, USA, 2011. USENIX Association.

[85] L. Wang, T. Islam, T. Long, A. Singhal, and S. Jajodia. An attack graph-based probabilistic security metric. In *Proceeedings of the 22nd annual IFIP WG 11.3 Working Conference on Data and Applications Security*, pages 283–296, Berlin, Heidelberg, 2008. Springer-Verlag.

[86] L. Wang, S. Noel, and S. Jajodia. Minimum-cost network hardening using attack graphs. *Computer Communications*, 29(18):3812–3824, Nov. 2006.

[87] L. Wang, A. Singhal, and S. Jajodia. Measuring the overall security of network configurations using attack graphs. In *Proceedings of the 21st annual IFIP WG 11.3 working conference on Data and applications security*, pages 98–112, Berlin, Heidelberg, 2007. Springer-Verlag.

[88] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman. Linux security module framework. In *Proceedings of the 11th Ottawa Linux Symposium*, 2002.

[89] P. Xie, J. H. Li, X. Ou, P. Liu, and R. Levy. Using Bayesian networks for cyber security analysis. In *The 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2010.

[90] K. Xu, H. Xiong, D. Stefan, C. Wu, and D. Yao. Data-provenance verification for secure hosts. *IEEE Transaction on Dependable and Secure Computing (TDSC)*, 9(2):173 – 183, March/April 2012.

[91] L. K. Yan and H. Yin. DroidScope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *Proceedings of the 21st USENIX conference on Security symposium*, Security'12, pages 29–29, Berkeley, CA, USA, 2012. USENIX Association.

[92] Z. Zhang, F. Naït-Abdesselam, X. Lin, and P.-H. Ho. A Model-based Semi-quantitative Approach for Evaluating Security of Enterprise Networks. In *Proceedings of the 2008 ACM Symposium on Applied Computing*, SAC '08, pages 1069–1074, New York, NY, USA, 2008. ACM.

[93] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 95–109, may 2012.