A conflict-free approach to application integrity using SELinux

Yizhou Ye

School of Science

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 3.12.2021

Supervisor

Dr. Lachlan J. Gunn

Advisor

Dr. Lachlan J. Gunn

Aalto University
School of Science

Aalto University
School of Science

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the master's thesis

Author Yizhou Ye

Title A conflict-free approach to application integrity using SELinux

Degree programme Computer, Communication and Information Sciences

Major Security and Cloud Computing (Security) **Code of major** SCI3084

Supervisor Dr. Lachlan J. Gunn

Advisor Dr. Lachlan J. Gunn

Date 3.12.2021 **Number of pages** 42 **Language** English

Abstract

Confidentiality, integrity and availability are all important factors to consider in information security. The subject of this thesis is the integrity of applications in Linux operating systems. Application integrity can be protected by enforcing an access control policy that prevents potentially malicious entities from influencing important application resources. Such policies are known as integrity policies. An access control policy consists of rules that can be enforced via access control. Linux includes support for SELinux, a Linux Security Module that provides Mandatory Access Control based on a policy for access control enforcement. However, SELinux is highly complicated due to its flexibility, and thus policy design and validation is often done with assistance from analysis tools.

Previous analysis tools require a user-defined Trusted Computing Base (TCB) to model entities that must be assumed to function correctly for the application to function correctly. The analysis tools compute conflicts between the integrity policy and the SELinux policy. The user is then expected to manually resolve said conflicts (e.g., modifying the policies). Once all conflicts have been resolved, the SELinux policy is proven to enforce the integrity policy, thereby protecting application integrity.

In this thesis, we have created a prototype application integrity analysis tool that implements a novel approach of achieving the desired result of application integrity. Unlike previous work, we automatically generate a TCB proposal. Compared to previous work, our approach significantly reduces the required level of user intervention.

Keywords selinux, integrity, tcb, policy analysis, conflict-free, information flow

Aalto University
School of Science

Aalto-yliopisto, PL 11000, 00076 AALTO
www.aalto.fi
Diplomityön tiivistelmä

Tekijä Yizhou Ye

Työn nimi Konflikti-vapaa lähestymistapa sovellusten eheyteen SELinux:n avulla

Koulutusohjelma Computer, Communication and Information Sciences

Pääaine Security and Cloud Computing (Security) **Pääaineen koodi** SCI3084

Työn valvoja Dr. Lachlan J. Gunn

Työn ohjaaja Dr. Lachlan J. Gunn

Päivämäärä 3.12.2021 **Sivumäärä** 42 **Kieli** Englanti

Tiivistelmä

Tietoturvan tärkeimmät tekijät ovat luottamuksellisuus, eheys ja saatavuus. Tämä diplomityö käsittelee sovellusten eheyttä Linux-käyttöjärjestelmässä. Sovellusten eheys voidaan turvata noudattamalla pääsynvalvontasääntöjä, joka estää mahdollisesti haitallisten komponenttien vaikutuksen tärkeisiin sovelluksen resursseihin. Tämän kaltaiset pääsynvalvontasäännöt tunnetaan myös eheyssääntöinä. Linux-käyttöjärjestelmässä tämä voidaan toteuttaa käyttämällä SELinux-tietoturvamoduulia, joka sisältää tuen pakolliseen pääsynvalvontamalliin (Mandatory Access Control) perustuen pääsynvalvontasääntöjen noudattamiseen. SELinux on erittäin monimutkainen joustavuutensa takia. Tämän takia SELinux-sääntöjen suunnittelu ja tarkistus suoritetaan usein analyysityökalujen avulla.

Aikaisemmat analyysityökalut vaativat käyttäjän määrittelemän luotettavan tietojenkäsittelyn (Trusted Computing Base). Oletus on, että tähän sisältyvien komponenttien täytyy toimia oikein, jotta sovellus toimii oikealla tavalla. Analyysityökalut laskevat konfliktit eheyssääntöjen ja SELinux-sääntöjen välillä. Käyttäjältä odotetaan kykyä ratkaista nämä konfliktit (esim. muuttamalla sääntöjä) manuaalisesti. Kun kaikki konfliktit on ratkaistu, SELinux-säännöt on todistettu noudattavan eheyssääntöjä suojaten sovelluksen eheyttä.

Tässä diplomityössä loimme prototyypin uudenlaisesta eheyssääntöjen analyysityökalusta, joka analysoi sovellusten eheyttä käyttäen uutta lähestymistapaa. Toisin kuin edelliset analyysityökalut, meidän työkalumme luo automaattisesti ehdotuksen luotettavalle tietojenkäsittelylle, mikä vähentää merkittävästi käyttäjältä vaadittua väliintuloa edellisiin työkaluihin verrattuna.

Avainsanat selinux, eheys, luotettava tietoturva, sääntöjen analyysi, konflikti-vapaa, tietovirta

Preface

Working on this master's thesis has been a wonderful journey. This marks the end of a chapter in my life of studying the program of Security and Cloud Computing at Aalto University. I am truly grateful for all the memories made along the way.

I would like to thank my supervisor **Lachlan J. Gunn** for always providing advice and support. You have done everything I've ever hoped for and more. I would like to thank the other members of the Secure Systems Group for providing valuable feedback during early stages of work.

I would like to thank our industry partners from Huawei for the motivation and funding for this work. Personal thanks goes to **Roberto Sassu** and **Silviu Vlasceanu** for constructive discussion and feedback.

Lastly, I would like to thank my girlfriend **Jenny** for unwavering support and our dog **Kinder** for reminding me to take breaks.

Espoo, Nov 25, 2021,

Yizhou Ye

Contents

Glossary

CC Cyclomatic Complexity. 37

CDI Constrained Data Item. 11, 17

DAC Discretionary Access Control. 10–12, 34, 35

GUI Graphical User Interface. 9, 26, 27, 34

IMA Integrity Measurement Architecture. 9

IVP Integrity Verification Procedure. 11

LOC Lines of Code. 37

LSM Linux Security Module. 11, 12, 38

MAC Mandatory Access Control. 10

MLS Multilevel Security. 10, 12

NSA National Security Agency. 12

RBAC Role-Based Access Control. 12

SELinux Security Enhanced Linux. 9, 12–17, 19–21, 23–27, 29, 31–39

SLOC Source Lines of Code. 37

TCB Trusted Computing Base. 9, 11, 17, 19, 21–25, 27–31, 33, 34, 36, 37, 39

TE Type Enforcement. 12, 14, 15, 17, 33

TP Transformation Procedure. 11, 17

UDI Unconstrained Data Item. 11, 17

1 Introduction

Our goal is to protect the integrity of user applications, more specifically the immutable and mutable files of the application. In this context, integrity is defined as the ability of software to behave as intended by the developer (Sassu, 2021). Immutable files can be integrity-protected using cryptographic hash functions, for instance, using the Linux Integrity Measurement Architecture (IMA). IMA computes the digest for these files, which can then be compared with reference digests stored as a predefined list. Mutable files, however, are expected to change dynamically during run-time and thus result in digests that are incompatible with reference values. If the mutable files are in a valid state initially, the validity of these files guarantees that the application behaves as intended, resulting in an initial state of integrity. If operations during run-time do not leave these files in a non-valid state, we say that the integrity of these files has been preserved. To prove the preservation of integrity, we use *access control*.

With access control we can define rules that only grant trusted components (e.g., components that maintain the integrity) access to the files that we want to protect. The sum of these rules forms an *integrity policy*, which refers to a set of rules that have to be enforced to preserve the integrity of the software (Sassu, 2021). We define the Trusted Computing Base (TCB) for a given application as the set of components that must be trusted not to violate the rules in the integrity policy for the application to function correctly. With this definition, we can extend the goal of preserving the integrity of important mutable files to preserving the integrity of the TCB.

We develop an integrity policy that is able to preserve the integrity of the application TCB. Once an integrity policy has been realized, it is approximated for compatibility with an Security Enhanced Linux (SELinux) policy. This SELinux policy enforces access control in the operating system, which preserves the integrity of the TCB. To achieve this goal, we have implemented a prototype analysis application that is capable of providing the SELinux rule changes necessary for the desired SELinux policy that enforces the same rules as the integrity policy, thus preserving application integrity.

We introduce the necessary background in Chapter 2. In Chapter 3, we present an assumed system model that forms the basis for all of our analyses. Furthermore, we describe the main objectives that we seek to accomplish. Chapter 4 describes the design choices we made for our analysis application, notably the discovery of a novel approach we refer to as the *conflict-free* method, which significantly reduces the required level of user intervention compared with previous works (Jaeger et al., 2003; Shankar et al., 2006; Xu et al., 2013; Zhai et al., 2015). Chapter 5 describes the implementation of the design choices in our application. For example, we implement information visualization to improve user understanding of abstract subjects (Herman et al., 2000) with a Graphical User Interface (GUI) that displays information flows of interest as a graph with interactive nodes and edges. In Chapter 6, we evaluate our work based on the objectives presented in Chapter 3. Chapter 7 presents past works of research that are closely related to our subject. Chapter 8 discusses the strengths and limitations of our work along with potential future work.

2 Background

2.1 Access control

Access control is a security mechanism used to constrain what an entity can do. Constraining entities in a computer system translates to constraining user executed programs by limiting the permissions necessary to perform operations. Therefore, access control determines how processes interact with other processes and files.

To enforce access control is to enforce a *security policy*, composed of access control rules. The security policy dictates the access subjects have on objects. Access decisions are made on the basis of *security attributes* associated with each subject and object. Furthermore, a *security model* is used to provide a formal representation of a security policy (Samarati and de Vimercati, 2001).

Most operating systems implement access control with an access control model known as Discretionary Access Control (DAC). In a DAC system, at their own discretion, a user may pass their privileges to other users in the system. Anderson describes DAC systems as systems in which protection is a task for the machine operator (Anderson, 2020, p. 209). Because of vast options for customization, one of the greatest strengths of DAC is the flexibility that it offers. This is apparent when DAC is compared to an alternative access control model, Mandatory Access Control (MAC).

In a MAC system, the basis for access control decisions is not at the discretion of individual users. Instead, the focus is on a central policy-driven approach (Eaman et al., 2017). Compared to DAC, MAC is less flexible but offers higher levels of data protection by forbidding the act of sharing access to data. Whereas DAC is more suitable for systems in which it is essential for users to manage content that they own, when the flexibility of DAC is not required, MAC can offer a central policy to manage the content instead. As a result, MAC is typically a more secure choice when the system has extremely sensitive data.

A prominent early MAC policy is Multilevel Security (MLS). In MLS, subjects and objects are assigned a security level. Typically, implementations of MLS are based on a security model known as the *Bell-LaPadula model* (Bell and LaPadula, 1975), where subjects are allowed to read objects on an equal or lower security level ("read down") and write to objects on an equal or higher level ("write up"). However, a subject is not allowed to read objects on a higher security level ("no read up") nor is it allowed to write to objects on a lower security level ("no write down"). The objective is to preserve the confidentiality of higher-level data by preventing information flows from higher levels to lower levels (Bell and LaPadula, 1975).

MLS, however, is limited to enforcing a singular security goal, confidentiality. Different systems prioritize different security goals, requiring different security models.

2.2 Integrity Policies

Integrity policy refers to a set of rules that have to be enforced to preserve the integrity of the software (Sassu, 2021). In this context, to preserve the integrity of

the software is to confirm that integrity-protected data items are kept valid when the system transitions from one state to another.

An early integrity policy, the Biba model, is fulfilled if higher-integrity processes do not depend on lower-integrity processes in any manner (Biba, 1974). In terms of information flows, Biba forbids flows from lower-integrity processes to higher ones. This implies two rules: Higher-integrity processes are not allowed to read lower-integrity data ("no read down") and lower-integrity processes are not allowed to write to data on a higher integrity ("no write up"). These rules are the inverse of what is allowed in the Bell-LaPadula model that prioritizes confidentiality as opposed to integrity, concepts that ask the questions of "who can read" and "who can write", respectively.

Biba is a strict integrity policy with little flexibility. As such, some applications are unable to function properly while the operating system is enforcing Biba. For instance, a modern web application implementing a client-server architecture has to allow information flows from lower-integrity components (client) to higher-integrity components (e.g., the application daemon). Therefore, it is impossible to enforce Biba unless the server is considered low-integrity.

A potential alternative is the Clark-Wilson model (Clark and Wilson, 1987). As opposed to Biba, the Clark-Wilson model allows information to flow upwards (low to high) through programs labeled as Transformation Procedures (TPs). Clark-Wilson defines Constrained Data Items (CDIs) as high-integrity data and Unconstrained Data Items (UDIs) as low-integrity data. The validity of CDIs at a given state is verifiable using what is known as Integrity Verification Procedures (IVPs).

TPs must be formally verified to transform all input data (whether UDI or CDI) to CDIs when writing to a CDI object. This way TPs always transition the system from one valid state to another. To summarize, Clark-Wilson only allows subjects to write to objects if the transaction is performed through a certified TP.

CW-Lite is defined as a weaker version of the Clark-Wilson integrity model (Shankar et al., 2006). The authors specify two observations of the Clark-Wilson model that led to the development of CW-Lite. First, the formal verification of TPs is a difficult task that has led to Clark-Wilson being too heavyweight for practical use. This requirement is a separate problem that is application-specific. For these reasons, CW-Lite does not require formal verification of TPs. Second, Clark-Wilson requires TPs in both scenarios of information flows from low or high-integrity subjects to high-integrity objects. However, in practice, this is often not necessary for TCB integrity, since the TPs can take advantage of the constraints imposed on CDIs and so need only filter UDI data.

2.3 SELinux

By default, the Linux kernel only provides support for DAC using traditional UNIX permissions. Implementing other security models is possible using Linux Security Modules (LSMs). LSMs are a framework that integrate various security modules into the Linux kernel. An LSM mediates access to Linux kernel objects using *hooks*. Hooks are placed in the kernel code such that they call a function provided by a

module just before the kernel accesses an object (Wright et al., 2002). LSMs check an internal policy for allowed operations if the traditional DAC has allowed the operation. Based on the policy, the module can decide to deny access if necessary.

SELinux is an LSM that supports various access control mechanisms, including Type Enforcement (TE), MLS, and Role-Based Access Control (RBAC). The TE model bases its access control decisions on what is known as the type attributes of subjects and objects. Using TE, various integrity policies, such as *Biba* (Biba, 1974) and *Clark-Wilson* (Clark and Wilson, 1987), can be implemented.

SELinux enforces a collection of rules on files and processes in the system. This collection of rules that is loaded into the kernel and subsequently enforced is called an *SELinux policy*.

SELinux policy development began with the *SELinux example policy* that was first introduced by the National Security Agency (NSA). Based on this policy, another policy was created known as the *SELinux reference policy*. The objective here was to create a general policy which can then be used as a starting point for further policy development. According to their documentation (SELinuxPolicy, 2021), some of the design goals of the reference policy include: strong modularity, security goals, a flexible base policy, and documentation. However, the flexible nature of SELinux results in complexity. This makes policy development and analysis a difficult task (Eaman et al., 2017; Said and Mohamed, 2011; Shankar et al., 2006; Xu et al., 2013; Zhai et al., 2015). The resulting complexity is also seen as a potential threat to the overall security of the policies: a policy that is too complex to understand is difficult to make secure.

SELinux labels all processes and files in the system with a *security context*, which is of the following form:

```
user:role:type:level
```

The first component of the context, `user`, refers to an SELinux user. Each Linux user is mapped to an SELinux user in the policy. This results in Linux users inheriting the restrictions placed on SELinux users (Jahoda et al., 2021, p. 10). The second component, `role`, refers to an attribute for users that specifies the groups of types that the user has access to. Inclusion of the `role` component allows SELinux to implement RBAC. In RBAC, the focus shifts from the identity of a user to the capabilities of the user. This is especially useful for directly mapping capabilities assigned to personnel with different roles in a real-world organization to a security policy (Samarati and de Vimercati, 2001).

The third component, `type`, is the primary component in TE. Type is a label assigned to each entity in the system. At the most granular level, SELinux rules refer to types. The type of a process (subject) is also referred to as a domain.

Finally, `level` is an optional component only used in MLS for sensitivity classification.

SELinux operates on a *closed-world assumption*. This results in SELinux denying all access by default. To access an object, the subject's type must be authorized for the object's type (Mayer et al., 2006). Authorization is determined based on rules with the following syntax:

```
allow source target:class permission; [ boolean ]
```

Consider the following rule:

```
allow user_t bin_t:file { read, execute, getattr };
```

The `allow` rule type is the only rule type that grants access. This is followed by the source type `user_t` (also referred to as subject type or domain type) and the target type `bin_t` (object type). Next, `file` refers to the name of the object class, such as `file`, `dir` and `tcp_socket`. Third, `read`, `execute`, and `getattr` are actions that subjects with the source type are allowed to take on objects with a combination of target type and object class. Finally, `boolean` refers to an optional rule component that exists if the rule is enabled by an SELinux *Boolean*. Booleans are settings in the policy that can be turned on or off to customize parts of the policy.

2.4 SELinux policy analysis

The sheer volume of rules (on the order of tens of thousands) included in a SELinux policy introduces a significant amount of complexity to the policy. In order to make sense of the policy, policy analysis tools are used to answer questions such as "How does the policy behave?" or "How is the policy written?". For the purposes of this thesis, we use *information flow analysis* as the method for policy analysis. This is because the relevant integrity policies (e.g. Biba or CW-Lite) are all information flow policies, and because it provides a way to represent relationships, direct and indirect, between components in the policy.

Xu *et al.* describe information flow as the reachability of a resource from another resource (Xu et al., 2013). Direct information flows between subjects and objects involve operations by the subject on the object that can be classified as *read-like* or *write-like* (Guttman et al., 2003). Given a subject S and an object O, read-like operations introduce information flows in the direction of $O \rightarrow S$ while write-like operations introduce flows in the direction of $S \rightarrow O$. Furthermore, the combination of multiple rules can lead to transitive information flows, where information flows indirectly (Ahn et al., 2008).

Let $G = (V, \rightarrow)$ be an information flow graph consisting of vertices (V) and edges ($\rightarrow$). We represent a direct information flow from `a` to `b`, where $a, b \in V$, by an edge $a \rightarrow b$. Similarly, we represent a transitive information flow from `a` to `c`, where $a, c \in V$, by the notation $a \rightsquigarrow c$. This transitive information flow relation $\rightsquigarrow$ is the transitive closure of the direct information flow relation $\rightarrow$.

Consider the following SELinux rules:

1. `allow a b : file write;`

2. `allow c b : file read;`

Rule 1 results in a direct information flow $a \rightarrow b$, because the operation `write` is write-like. Likewise, rule 2 results in a direct information flow $b \rightarrow c$, because the

operation `read` is read-like. The combination of multiple rules leads to transitive information flows. For example, subject `a` does not directly interact with object `c`. Nonetheless, the flow $a \rightarrow b$ combined with the flow $b \rightarrow c$ allows information to flow indirectly from `a` to `c`, leading to a transitive information flow $a \rightsquigarrow c$. Figure 2.1 shows the information flows between the components.

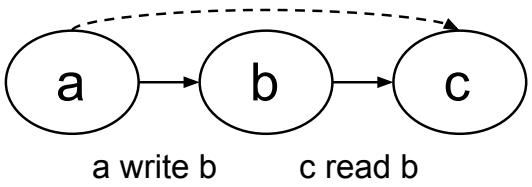

Figure 2.1: Information flows resulting from SELinux rules.

2.5 Approximating integrity policies using SELinux

In this section, we discuss the act of integrity policy approximation using SELinux. Because high-level integrity properties are not readily apparent from the TE policy, we seek to approximate an integrity policy using SELinux. Our approach is based on two assumptions:

1. The permissions in integrity policies are a superset of those in the SELinux policy.

2. An integrity policy is easier to comprehend than the SELinux policy.

Because integrity policies that are a superset of the SELinux policy grant more permissions, they are less strict than the SELinux policy. We can then use a less complex integrity policy to reason about the integrity of software. If this integrity policy can be approximated using the underlying SELinux policy, we can remain confident that the SELinux policy is at least as strict.

Using the components of a web application as an example, we examine the implementations of three integrity policies: Biba, Clark-Wilson, and CW-Lite. This web application consists of a web frontend, application server, a database server, and a database. The application server is connected to both the frontend and the

database server, while the database server is responsible for the database as stored on disk. Figure 2.2 illustrates the information flows between these components.

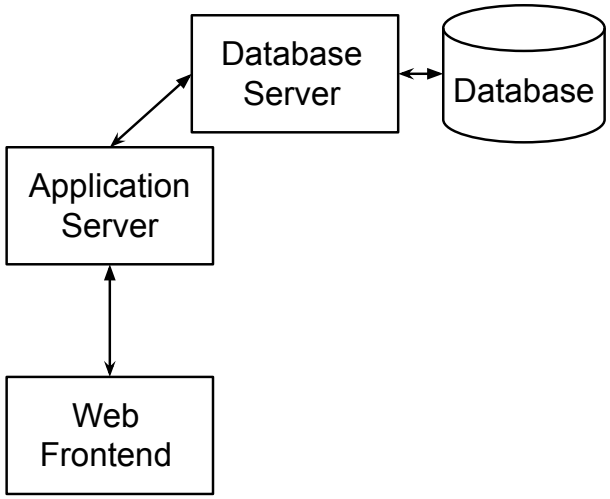

Figure 2.2: A graph representing the information flows between the components of a web application.

2.5.1 Biba

In order to define a Biba policy, we have to assign integrity levels to each component in the system. In our example web application, let us suppose that the database is the key component whose integrity we want to protect. For this reason, we classify it as high-integrity. In contrast, we classify the web frontend as low-integrity. This is due to it being directly influenced by external clients that are possibly untrustworthy. What remains to be classified is the application server. If we classify the application server as high-integrity, the rule "no read down" is violated upon requests on the server from the frontend. Conversely, if we classify the application server as low-integrity, the same rule is also violated when the application server sends requests to the database server. This highlights the main limitation of the Biba model: Only a Biba model with one integrity level would allow this example. Figure 2.3 illustrates the information flow graph from Figure 2.2, updated to enforce the Biba model.

The Biba model can be implemented using the TE capabilities of SELinux. SELinux types associated with components considered high-integrity are protected from the influence of low-integrity components through rule enforcement. Let $G = (V, \rightarrow)$ be the information flow graph consisting of vertices (V) and edges ($\rightarrow$). In this context, vertices represent SELinux types in the policy while edges represent information flows enabled by rules in the policy. For each vertex, we define $\text{int}(v)$ as the integrity level of the given vertex. For example, $\text{int}(a) \geq \text{int}(b)$ translates to "the integrity level of a is greater than or equal to the integrity level of b". In order

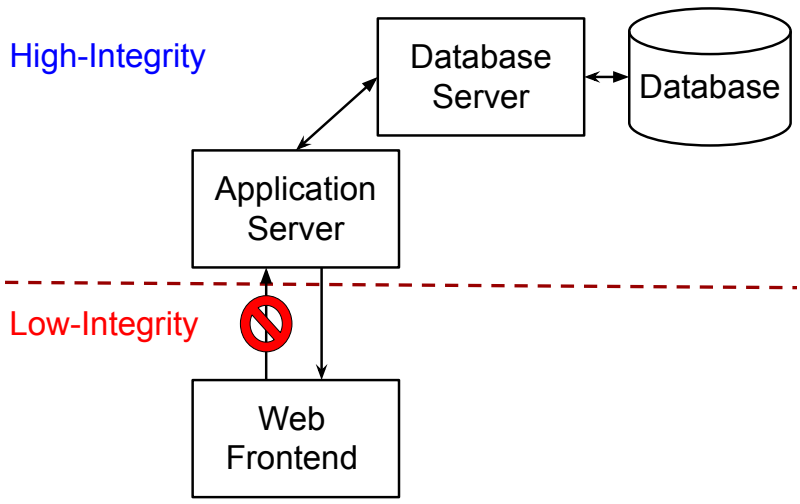

Figure 2.3: This graph visualizes the information flows in the web application with a Biba model enforced. The application server is classified as high-integrity. This causes the information flow *web frontend* $\rightarrow$ *application server* to violate Biba integrity. Classifying the application server as low-integrity would result in a similar violation by the flow *application server* $\rightarrow$ *database server*.

to fulfill a Biba integrity policy, the information flow graph must satisfy

$$\forall a, b \in V : a \rightarrow b \Longrightarrow \text{int}(a) \geq \text{int}(b),$$

which states that there are no information flows from lower-integrity components to higher-integrity components.

2.5.2 Clark-Wilson

The implementation of a Clark-Wilson model using SELinux is examined by Jaeger *et al.* (Jaeger et al., 2003). The requirements of the Clark-Wilson model are (Clark and Wilson, 1987):

- **R-1:** Each TP operates on a particular list of CDIs and CDIs are only manipulated by a TP.
- **R-2:** The system must maintain a list of subjects, TPs, and the CDIs those TPs may reference, and only those references are permitted.
- **R-3:** The system must authenticate the identity of each user that attempts to execute a TP.
- **R-4:** The list of TPs and IVPs can only be changed by a subject permitted to certify those TPs and IVPs.

The first requirements, R-1 and R-2, can be partially fulfilled by the TE capabilities of SELinux. We can identify operations between components using SELinux types. However, SELinux does not distinguish between CDIs and UDIs, nor does it define TPs. These requirements must be addressed during policy development. R-3 and R-4 are fulfilled by a well-crafted SELinux policy. We define a TCB, which is the set of components that do not violate the integrity policy. In other words, the components in the TCB operate in a way that does not compromise the integrity of the system. In this case, the TCB is the set of TPs and CDIs.

An *interface* for a subject refers to a distinct information input channel that enables input for the subject during calls, such as `open()` or `connect()` (Shankar et al., 2006). In the context of TCB integrity, TPs that read UDI data in the Clark-Wilson model are also referred to as *filtering interfaces* or *filters* for short. This name is more descriptive because we require filters to be able to correctly filter UDI data that is read by a TCB subject. Following this convention, TPs that do not read UDI data should be referred to as non-filtering interfaces. However, Clark-Wilson does not distinguish between TPs or interfaces. Therefore, all interfaces of a TP must be certified filtering interfaces.

2.5.3 CW-Lite

CW-Lite is a relaxed version of Clark-Wilson that is specifically tailored towards systems where integrity of the TCB is a security goal (Shankar et al., 2006).

Similar to Biba, CW-Lite policies can also be described in terms of an information flow graph. Given a CW-Lite policy (V, T, F), we describe the policy by its components V, its TCB $T \subset V$, and its filtering interfaces $F \subset V \times T$. An information flow graph is said to fulfill CW-Lite if

$$\forall a \in V, b \in T : a \to b \implies a \in T \vee (a, b) \in F.$$

This necessitates that each edge into a TCB component must be either a TCB component or a designated filtering interface.

This policy model is more practical than Biba in many applications, as it allows for information flows between the TCB and its complement, the N-TCB. However, it is still simple enough to be interpreted by application developers, who can understand which components of the system are critical to the integrity of a given resource.

Implementing a CW-Lite policy requires identification of candidates for the TCB and filtering interfaces. This task requires a combination of heuristics and domain knowledge. Figure 2.4 illustrates the information flow graph updated to fit the CW-Lite model.

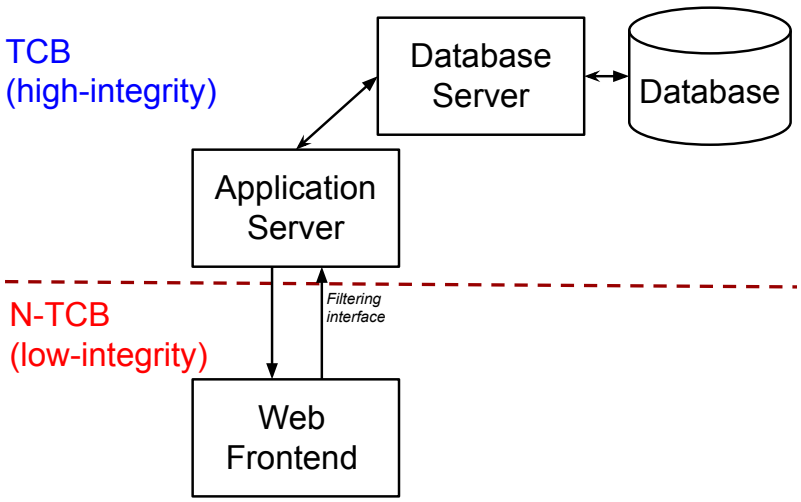

Figure 2.4: As opposed to the graph in Figure 2.3, where the information flow *web frontend* $\rightarrow$ *application server* violates Biba integrity, the CW-Lite model permits this information flow. This is because the application server now executes a filtering interface that is capable of filtering low-integrity output from the frontend to the database server.

3 System model and objectives

3.1 System model

Our system model comprises a Linux system that is enforcing a SELinux policy. The Linux system runs applications whose integrity we want to protect. We assume the following of the protected applications:

1. Protected applications can maintain their integrity when no outside influence is present.
2. Protected applications are secure to the extent that they can not directly be compromised by an adversary.

Although an adversary can not directly compromise a protected application, we model the adversary by assuming the following:

1. An adversary can compromise certain other non-secure services in the system.
2. An adversary may be able to influence the integrity of protected applications indirectly through information flows from a compromised service.

The rest of this chapter describes the main objectives of this thesis. We then evaluate our work based on these objectives in Chapter 6.

3.2 Objectives

O-1 Semi-automated analysis
Contemporary policy analysis tools require significant manual user intervention for policy integrity analysis. Some tools expect the user to query the policy for answers, others inspect information flows to a user-specified TCB. We aim to automate this process by eliminating the need for a prespecified TCB.

O-2 Establish and preserve TCB integrity
Given a user application and a set of components in the application that the user wants to protect, we derive a TCB. The TCB is the set of SELinux types that must be assumed to be non-compromisable for the application to function correctly.

We aim to identify a way to ensure the integrity of the TCB. In doing so, we can affirm that the integrity of protected components is preserved, and the application behaves as intended by the developer.

O-3 Applying application domain knowledge
Our tool is able to apply application domain knowledge provided by the user for a more precise analysis. We believe this is necessary, since the behaviour and requirements for each application is unique. The main challenge is to determine what types of input can be useful, and subsequently, how to incorporate this input into the analysis.

O-4 Usable by non-experts
Application developers typically have more specific domain knowledge related to their applications, as opposed to security analysts. Because of this, we aim to make our application usable for application developers who do not necessarily have significant SELinux expertise. If changes have to be made in the SELinux policy, the application developer can use our tool to communicate the necessary changes to a security expert.

4 Semi-automated TCB analysis

We seek to assist application developers in protecting the integrity of their applications. To achieve this, we create an integrity policy that is less strict and complex than the SELinux policy, in this case a CW-Lite policy. The CW-Lite policy is less strict than the SELinux policy because it grants more permissions. This results in less complexity which in turn leads to easier human comprehension. If the CW-Lite policy can be approximated using the SELinux policy, we prove that the SELinux policy provides protection that is at least as great as the integrity policy. Therefore, we protect applications by approximating a CW-Lite policy using SELinux.

In the CW-Lite policy, the SELinux types associated with the application are in the TCB. Furthermore, a potential adversary exists outside the TCB, in the N-TCB. The task for the CW-Lite policy is to prevent information flows from the adversary from reaching the TCB, with the exception of information flows classified as filtering interfaces. Recall from Section 2 that a CW-Lite policy is satisfied if each information flow into a TCB component is either from another TCB component or via a filtering interface.

A common approach employed by SELinux policy analysis tools in the past is to view the task of policy approximation as a problem of conflict-resolution. To achieve a security goal, a set of requirements are imposed on the SELinux policy. For instance, Gokyo (Jaeger et al., 2003) imposes the requirement that the SELinux policy implements a proposed CW-Lite policy. Other tools, such as SCIATool (Zhai et al., 2015) and PVA (Xu et al., 2013), do not impose a particular policy model, but instead expect the user to define their own requirements outside the tool according to their desired security goals. The analysis tools then proceed to identify conflicts between these requirements and the underlying SELinux policy. These conflicts can be resolved by either modifying the SELinux policy or the requirements (e.g., a CW-Lite policy). Once all conflicts have been resolved the SELinux policy is then proven to provide protection that is at least as great as the integrity policy.

Instead of employing a conflict-based approach and requiring a prespecified TCB and set of allowable filters as in Gokyo (Jaeger et al., 2003) for a CW-Lite policy, we propose a conflict-free approach where one specifies a set of *protected resources* along with allowable filters that form the border of the TCB. The TCB is then derived by graph traversal. Figure 4.1 shows an example of a conflict-based approach applied to a CW-Lite policy.

The conflict-free approach is based on two intuitions:

1. Specifying a set of resources that are to be integrity-protected and a set of initial filtering interfaces is an easier task than specifying an entire TCB with filtering interfaces.

2. A conflict-based approach is time-consuming and error-prone as the process of resolving a conflict may lead to new conflicts.

Instead of focusing on every individual conflict, our proposed conflict-free approach offers improved automation, thus reducing manual user intervention.

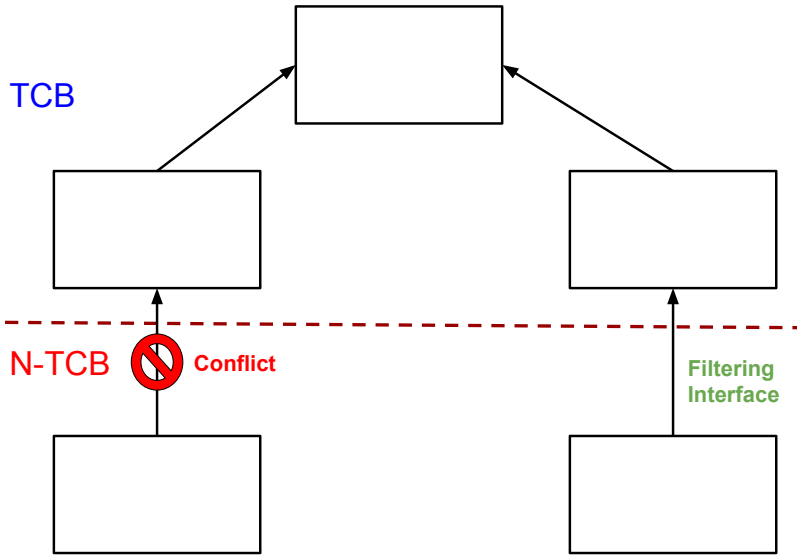

Figure 4.1: The system has a prespecified TCB (everything else is in the N-TCB) and a filtering interface. In CW-Lite, information flows from N-TCB to TCB lead to integrity conflicts unless a filtering interface is present. The focus is then on resolving these conflicts.

4.1 TCB identification

We construct a TCB around the protected resources. This is done by traversing the information flow graph $G = (V, \rightarrow)$ from the protected resources in reverse (inflows instead of outflows). If an edge is one of the allowable filters, it is excluded from the graph before traversal. The set of visited nodes is the TCB of the system. Figure 4.2 shows an example of our proposed approach.

Choosing the protected resource is a task that requires domain knowledge from the user. Consider a database application, such as PostgreSQL, as an example. The user knows that the PostgreSQL daemon is the service responsible for responding to queries. If the security goal is to protect the integrity of queries, the user chooses their protected resource accordingly, in this case type `postgresql_t`. Potentially malicious queries are received from the type `postgresql_port_t`. As such, this information flow needs to be a filtering interface. On the other hand, the information flow $postgresql\_db\_t \rightarrow postgresql\_t$ is not filtered, because both components are application subject types which are expected to function correctly. Figure 4.3 shows this example using PostgreSQL.

4.2 TCB optimization

Because the user is not expected to know the optimal filters for the CW-Lite policy in advance, the initial TCB identified by our graph traversal algorithm is unlikely to be optimal. In cases where the TCB is too large, perhaps even encompassing the entire system, the main purpose of the TCB is defeated. A minimal TCB is

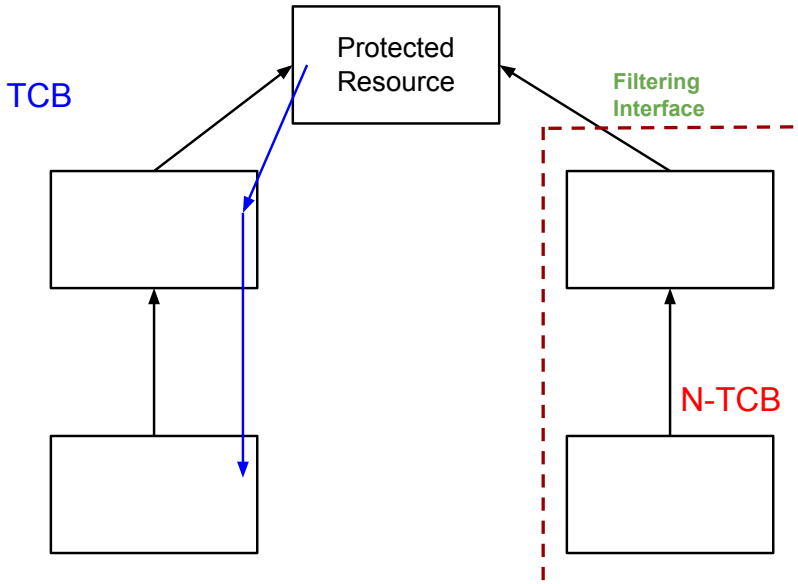

Figure 4.2: We identify the TCB by traversing the graph in reverse from the protected resource (blue arrows). Edges that implement a filtering interface are not visited. The resulting set of visited nodes form the TCB.

generally preferable as a smaller TCB is easier to verify (Jaeger et al., 2003). A smaller TCB also reduces the probability of a potentially compromisable service (recall assumptions on the adversary in Section 3) belonging to the TCB. As such, the number of components in the TCB has to be reduced. We do this by modifying the policy and adding additional filtering interfaces.

We propose semi-automated changes to the SELinux policy by excluding certain information flows (edges) in an effort to reduce the size of the TCB. We propose a combination of methods based on heuristics and optimization.

4.2.1 SELinux Booleans

We observe information flows enabled by SELinux *Booleans*, settings in the policy that can be turned on or off. Booleans allow the user to customize parts of the policy at runtime without recompilation of the policy (Jahoda et al., 2021). For example, enabling the Boolean

```
httpd_can_network_connect
```

allows the rule

```
allow httpd_t port_type:tcp_socket name_connect;
```

which allows the HTTP daemon to connect to all TCP-socket types that possess the port_type attribute. If an information flow is only enabled by a Boolean and that Boolean can be disabled on the system, we can exclude the information flow.

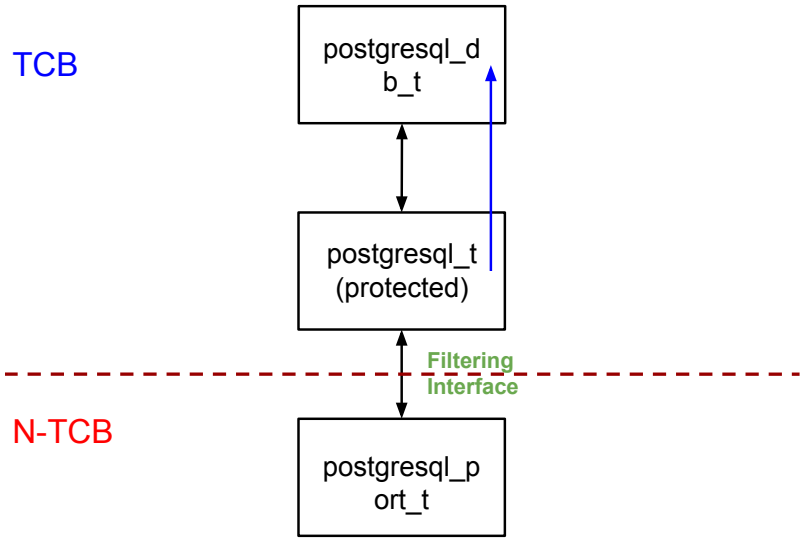

Figure 4.3: PostgreSQL as an example. The type associated with the PostgreSQL daemon service, `postgresql_t`, is provided as the protected resource and information flow *postgresql_db_t* → *postgresql_t* is a filtering interface. This information is then used as input for our graph traversal algorithm.

4.2.2 Unconfined processes

We observe information flows involving processes that run in `unconfined` domains. Unconfined domains are a special type of domain found in targeted SELinux policies. Processes that run in unconfined domains are unconfined by SELinux, although they are still confined by standard Linux security (Jahoda et al., 2021, p. 16). Because unconfined domains have unlimited access to application types, the resulting TCB is the entire system. While changes in the policy can be made to confine these domains, we argue that changing the policy this way is too significant and may alter the functionality of the system. Thus, we automatically exclude information flows involving unconfined domains in our input data.

4.2.3 Minimum-cut optimization

We use the minimum cut algorithm to identify the minimum number of information flows that have to be excluded in order to prevent the adversary from influencing the protected resources. This is essential for minimizing user effort as otherwise the user would need to analyze every information flow.

The algorithm takes as input a graph that is both directed and weighted, a source node, and a sink node. As output we receive two partitions of the input graph, one containing the source node and another containing the sink node. We then compute the suggested cuts as the set of edges from one partition to another. The suggested cuts are the set of edges with the minimum total weight that have to cut from the graph in order to separate the source and the sink into two partitions.

We apply the algorithm by creating an "Adversary" node that models a potential adversary. We also create a "TCB" node that models what is to be protected. These act as the source and the sink node, respectively. Information flows from the adversary node to compromised types and information flows from the protected types to the TCB node are assigned an infinite weight. This guarantees that these flows are not cut. Flows that are classified as filters are assigned a weight of zero, because they are assumed to be safe and thus need not be considered. This is because the filters have already been analyzed by the user. Every other flow in the graph is assigned a weight of one. Because these flows have an equal weight, computing the set of edges with the minimum total weight to be cut is equivalent to the minimum number of edges to be analyzed. These cuts can be implemented by changing the SELinux policy or alternatively they can be classified as filtering interfaces, thus avoiding the need to be cut. We then obtain a new TCB that is smaller and more optimized than the initial TCB. Figure 4.4 depicts how the minimum cut algorithm operates.

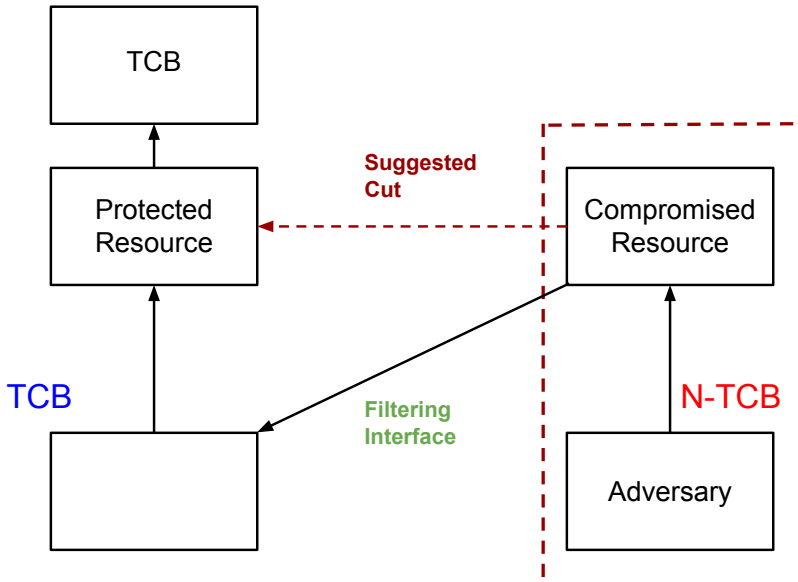

Figure 4.4: The objective of isolating the "Adversary" node from the "Protected Resource" is achieved by cutting the information flow (dashed line) labeled as the "Suggested Cut". The adversary has an alternative path to reach the protected resource. However, this information flow flows through a trustworthy filtering interface. As a result, the algorithm does not consider cutting it.

5 Implementation

5.1 Software architecture

Our application architecture consists of three main components:

1. A modified version of Apol used to extract the information flow graph from the SELinux policy.
2. An analysis backend.
3. A visual frontend.

We then take the following input from the user:

1. An information flow graph.
2. A list of protected SELinux types.
3. A list of potentially compromised SELinux types.
4. A list of information flows that serve as initial filtering interfaces.

5.2 Acquiring the input graph

To realize our approach, we require input in the form of an information flow graph. Apol, an open source analysis tool, is used to obtain this input. We use a modified version of Apol (SELinuxProject, 2021a) to perform an information flow query on the policy. Apol then constructs an information flow graph of the policy $G = (V, \rightarrow)$.

Graph G is filtered according to permission weight threshold that dictates which permissions lead to information flows. This yields a subgraph S. Apol classifies each permission as read-like, write-like, both, or neither. Additionally, Apol assigns a permission weight ranging from 1 to 10 to each permission. The weight is a qualitative value that suggests the amount of information that can be passed through a permission (SELinuxProject, 2021a). For instance, weights ranging from 1 to 2 are considered covert flows that are difficult to exploit (e.g., file:audit_access). On the other end, weights ranging from 8 to 10 are considered high-bandwidth overt flows (e.g. file:write and file:read have a weight of 10). For our purposes, we set the permission threshold to the maximum of 10 in order to only yield the most obvious information flows that can be considered information flows in most cases. Our modifications to Apol then allow us to extract the resulting information flow subgraph S into a text file.

5.3 Functionality

Our application allows a user to explore the cuts using a GUI. We display the suggested minimum cut both visually as a graph and in text. Figure 5.1 shows the GUI.

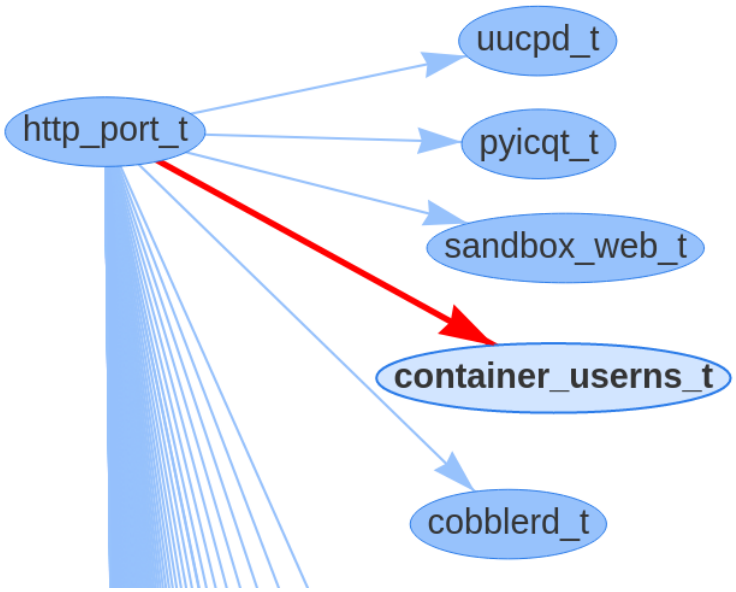

Figure 5.1: Our GUI displays suggested cuts as edges in a graph. In this example, the node `container_userns_t` has been selected, which highlights the information flows from and to that node, in this case the flow `http_port_t` → `container_userns_t`. When an edge is highlighted the rules that enable the information flow are also displayed.

We also display the corresponding rule changes that have to be made in the policy to accommodate the exclusion of the information flows. For instance, highlighting the flow `http_port_t` → `container_userns_t` shown in Figure 5.1 displays the corresponding SELinux rules that need to be edited in the policy.

```
allow sandbox_net_domain port_type:tcp_socket
{ name_bind name_connect recv_msg send_msg };

allow sandbox_net_domain port_type:udp_socket
{ name_bind recv_msg send_msg };
```

The user can inspect each element in the graph for additional information: Selecting a node highlights all edges to and from that node. Selecting an edge displays the SELinux rules that enable the information flow. If the user disagrees with the suggested cuts, the user can choose to classify edges as either necessary or filters and perform the minimum cut again with different parameters. Necessary edges and filtering edges are assigned an infinite weight and a zero weight, respectively. A new cut is then suggested. This process can be repeated until a satisfactory TCB has been found. Figure 5.2 shows the intended workflow when operating our application.

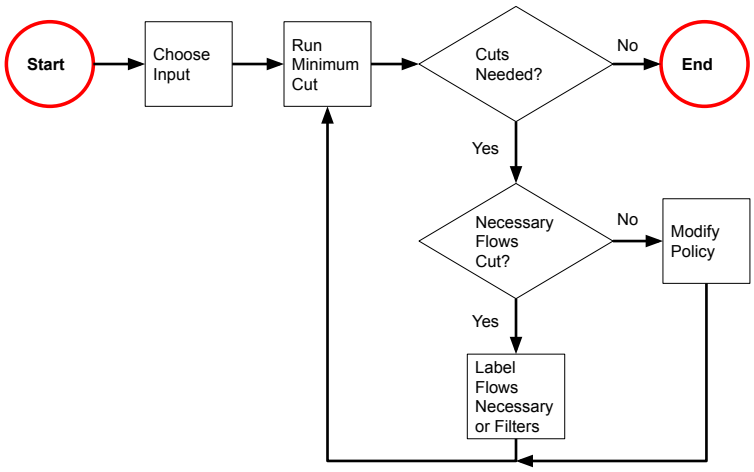

Figure 5.2: The first process is to identify input consisting of necessary types, compromised types, and initial filtering interfaces. With this input the minimum cut algorithm can be run. If no cuts are necessary, a satisfactory TCB has been obtained. If cuts are necessary, changes in the policy have to be made to facilitate the removal of said flows. However, if a flow that is suggested to be cut is deemed necessary, it can be labeled as either a necessary flow or a filtering flow. Subsequently, the minimum cut algorithm is executed again with new parameters.

5.4 Example with PostgreSQL

In this section we describe a sample analysis on a system that runs a web application with PostgreSQL. To begin, the user needs to identify the protected types. We choose type `postgresql_t` as the protected type. The reasoning for this is explained in Chapter 4. Next, we make an assumption on the adversary, where we assume they have the means to compromise http-ports (type `http_port_t`). These are ports that the web server can listen to by default. We then use these parameters as initial input for the application. The initial TCB we receive via graph traversal contains 4924 types out of a total of 4950 types in the graph. This TCB includes, for example, types for network ports. As a result, the TCB is unfeasible given the threat model and thus requires further optimization via the minimum cut.

The minimum cut suggests that we cut all information flows from `http_port_t`. However, thanks to our domain knowledge we know that doing this would stifle the functionality of the web application. Therefore, we designate the flow `http_port_t` → `httpd_t` as a necessary flow, one that must exist for purposes of functionality. Our tool shows that doing this means preserving the following rules:

```
allow httpd_t port_type:tcp_socket { recv_msg send_msg };
allow httpd_t port_type:udp_socket { recv_msg send_msg };
```

Conversely, removing these rules would cause the flow in question to be removed. Furthermore, we also label flows from `http_port_t` to other types with a prefix of `httpd` as necessary flows. We then execute the minimum cut again.

This time the suggested cuts are direct flows to `postgresql_t`. Because we told the algorithm that completely isolating `http_port_t`, the compromised type, was not an option, the next solution would be to directly address flows to `postgresql_t`. We observe that most flows to `postgresql_t` are from different port and temporary file types. For instance, we show that disabling the single permission of

```
allow postgresql_t port_type:tcp_socket { recv_msg send_msg };
```

leads to cutting 305 flows, for example:

- `qpasa_agent_port_t` → `postgresql_t`
- `sge_port_t` → `postgresql_t`
- `commplex_link_port_t` → `postgresql_t`
- `dnssec_port_t` → `postgresql_t`
- `gopher_port_t` → `postgresql_t`
- `agentx_port_t` → `postgresql_t`
- `milter_port_t` → `postgresql_t`
- `bacula_port_t` → `postgresql_t`

However, this analysis fails to consider the application architecture of PostgreSQL, thus leaving important PostgreSQL types out of the TCB, such as `postgresql_db_t`. For our next iteration of analysis, we incorporate the full domain knowledge of both the web application and the database application.

We make note of all types associated with PostgreSQL. These include: `postgresql_db_t`, `postgresql_etc_t`, `postgresql_exec_t`, `postgresql_lock_t`, `postgresql_log_t`, `postgresql_server_packet_t`, `postgresql_tmp_t`, and `postgresql_var_run_t`. We label flows from these types to `postgresql_t` as necessary, because cutting them would undermine the functionality of PostgreSQL, and the alternative option of labeling them as trusted filters accomplishes very little given the situation of a compromised `postgresql_t`. Furthermore, we label the flow `postgresql_port_t` → `postgresql_t` as a filtering interface given the assumption that the PostgreSQL daemon is capable of correctly filtering input from its designated port type. Finally, we run our application with these additional parameters and the result is the TCB with 118 types shown in Table 5.1.

Finally, to implement this TCB configuration, our tool provides a list of all SELinux rules that need to be modified, in this case 450 rule changes.

devicekit_t	proc_kcore_t
firstboot_t	proc_kmsg_t
flatpak_helper_t	pulseaudio_xproperty_t
fsadm_t	puppetagent_t
fwupd_t	realmd_consolehelper_t
games_xproperty_t	realmd_t
gpg_pinentry_xproperty_t	remote_xproperty_t
inetd_child_t	rolekit_t
inetd_t	root_input_xevent_t
initrc_t	root_xproperty_t
install_t	rpm_script_t
internet_packet_t	rpm_t
intranet_packet_t	rtas_errd_t
ipmievd_helper_t	sandbox_xserver_xproperty_t
kdumpctl_t	seclabel_xproperty_t
kernel_t	semanage_t
kmod_t	sepgsql_ranged_proc_exec_t
kpatch_t	sepgsql_ro_blob_t
sepgsql_secret_table_t	unpriv_sepgsql_proc_exec_t
sepgsql_seq_t	unpriv_sepgsql_seq_t
sepgsql_sysobj_t	unpriv_sepgsql_sysobj_t
sepgsql_trusted_proc_exec_t	unpriv_sepgsql_table_t
setfiles_mac_t	user_sepgsql_proc_exec_t
sge_job_t	user_sepgsql_sysobj_t
sge_shepherd_t	user_xproperty_t
sosreport_t	virtd_lxc_t
spc_t	virtd_t
ssh_xproperty_t	vmtools_helper_t
stratisd_t	vmtools_t
system_cronjob_t	vmware_host_t
systemd_coredump_t	vmware_xproperty_t
tcb	wine_t
tomcat_t	wireshark_xproperty_t
tuned_t	xdm_input_xevent_t
tvtime_xproperty_t	xdm_xproperty_t
udev_t	xserver_t
unpriv_sepgsql_blob_t	zabbix_script_t

Table 5.1: The final TCB of 118 types obtained after labeling the necessary and filtering flows.

6 Evaluation

In Section 3, we described the main objectives of this thesis. This section evaluates how well those objectives were satisfied.

O-1 Semi-automated analysis
We observed that contemporary analysis tools often require significant manual user intervention for achieving the desired result. We opted for a semi-automated approach, because we deemed full automation impossible given the information provided by the SELinux policy alone. The automated part of the process is the algorithms that we run. Instead of requiring a prespecified TCB, we propose a TCB and the corresponding SELinux rule modifications via graph traversal and minimum cut algorithms. However, we require manual user intervention for security analysis. For example, the user needs to have the ability to identify necessary flows that should not be cut, improving the results of the algorithm in the process.

We have successfully eliminated the need for a prespecified TCB, thus reducing manual work on behalf of the user. Because of this, this objective is satisfied.

O-2 Establish and preserve TCB integrity
To protect the integrity of an application, we form a TCB. If this TCB can be proven to be integrity-protected, we can also assume the same for the application. We achieve this by eliminating the information flows between a modeled adversary and the TCB for the application. To reason about the integrity of the TCB, we employ policy approximation, where we use an existing SELinux policy to approximate a CW-Lite policy. The CW-Lite policy is satisfied if each information flow allowed by the SELinux policy is also allowed by the CW-Lite policy. We then show that one can enforce these requirements with SELinux rules. In other words, we establish TCB integrity with CW-Lite and we preserve TCB integrity by enforcing SELinux rules that abide by CW-Lite constraints. As a result, this objective is satisfied.

O-3 Applying application domain knowledge
The information in the SELinux policy alone is not sufficient for optimal decision-making in the analyses. Because of this, we incorporate additional application domain knowledge as input from the user. First, we demand initial input from the user concerning the protected and compromised types. To identify these the user could, for example, perform a threat assessment procedure, where potential threats along with attack surfaces are identified. Second, the user might have knowledge of information flows identified as initial filtering interfaces, which can be excluded from the subsequent analyses. Third, when our minimum cut algorithm suggests cuts to be made, the user is expected to review these cuts and label the information flows as necessary or as additional filtering interfaces at their own discretion. N-TCB types that have necessary flows to the TCB are then added to the TCB, while filtering flows are ignored for the purposes of subsequent analyses.

We have successfully applied user application domain knowledge. However, there are several other types of input from the user that can be utilized to improve the

analyses. We discuss these in more detail in Section 8. Thus, we conclude that this objective is partially satisfied.

O-4 Usable by non-experts
We set our target demography as domain-savvy application developers, as opposed to security experts. We argue that our tool is usable by application developers with a basic understanding of information security. Because we do not expect our user to possess the ability to rewrite complex SELinux policies, we outsource this part of the workflow to security experts. However, the output provided by our application provides the means for a developer to communicate the necessary rule changes that have to be made in the policy to an expert. Therefore, we conclude that this objective is satisfied.

7 Related work

Most SELinux analysis tools share the common objective of improving the user's understanding of the SELinux policy. However, Eaman *et al.* suggest that many tools inadvertently add additional complexity by introducing intermediate languages. While these typically manage to reduce the complexity in some areas of the analysis, they run the risk of increasing the complexity in other areas with "equally complex semantics" (Eaman et al., 2017). Ultimately, humans have different preferences of breaking down complexity into more digestible parts. Some are visual learners, others have certain background knowledge that they can apply to improve their understanding of the policy. For this reason, a variety of SELinux policy analysis exists to specialize in certain types of analysis or to accommodate a certain user demographic. This chapter examines existing SELinux policy analysis tools.

Apol is a graphical SELinux policy analysis tool. It is included in SETools (SELinuxProject, 2021b), a collection of various analysis tools. Among other analyses, Apol supports information flow analysis, which can be used to detect potentially unintended flows of information between components allowed by a TE policy (SELinuxProject, 2021a).

By specifying queries the user can either find all information flow paths up to a specified length, or the shortest path (measured in the number of direct flows) from a specified domain to a specified type. Additionally, the user can perform a "flows out of" or "flows into" analysis to determine all direct information flows from or to a specified type, respectively.

Apol offers a flexible array of building blocks that can be used for various forms of policy analysis. However, using Apol for higher-level analyses, such as CW-Lite policy approximation, requires a significant amount of manual user effort. This is because realizing a higher-level goal with policy analysis requires processing and interpreting the results of multiple lower-level analyses. As an example, CW-Lite policy approximation requires identification of all information flows to the TCB. The user has to manually perform multiple lower-level analyses to determine the information flows into each TCB component.

SCIATool (Zhai et al., 2015) attempts to integrate multiple approaches to policy analysis. These include access control space analysis, colored Petri-nets, and information flow analysis. The main motivation is to combine these analysis methods to cover their weaknesses, thus providing a more complete analysis. For instance, (Zhai et al., 2015) argue that access control space analysis is most suited for validity analysis, where a policy is analyzed to see whether it implements the access control policy that the author intended. This refers to the capability of a policy configuration to fulfill desired security goals. Likewise, Petri-nets support validity analysis in addition to providing the means for finding other possible problems, such as information leaks. Finally, information flow analysis combined with access control spaces can be used to analyze TCB integrity.

Similarly to Apol, SCIATool operates with user defined queries. Identifying a CW-Lite policy with queries alone is a time-consuming task. Additionally, the multiple approaches to policy analysis present a challenge to inexperienced analysts,

who do not know which approach to apply to a given situation.

Policy Visualization Analysis (PVA) is an information flow analysis tool proposed by (Xu et al., 2013). It features a GUI that uses semantic substrates (Shneiderman and Aris, 2006) and adjacency matrices (Keller et al., 2006) to visualize information flows in the policies. Semantic substrates are user defined regions in which nodes are placed according to their attributes. For instance, in SELinux we can define users, roles, domains and types as separate semantic substrates. However, this approach is severely limited by the size of the graph due to poor scaling capabilities. As a result, PVA also offers a different visualization mechanism in the form of adjacency matrices. In contrast to other tools, the authors (Xu et al., 2013) argue that the visual elements in their tool enable users with less SELinux proficiency to obtain meaningful results.

Similarly to Apol and SCIATool, PVA can be used to query information flows between components in the policy. Whereas Apol and SCIATool do not have a concept for a group of components, such as the TCB, PVA does. As a result, the user can specify a single higher-level query with PVA as opposed to multiple lower-level queries, thus saving time in the process. Nevertheless, we believe our approach, which does not employ queries, is more user-friendly in terms of analysis time and knowledge required from the user.

Gokyo (Jaeger et al., 2003) is a tool that is used to establish the integrity of a TCB. Gokyo works by identifying conflicts between the SELinux policy and a proposed CW-Lite model, suggesting modifications to the CW-Lite model parameters (such as adding or removing entities from the TCB, or identifying filtering interfaces), or the SELinux policy (such as removing an unnecessary permission). We examine Gokyo more closely as it is the tool that is functionally most similar to what we require.

They propose to include the following SELinux subjects types in an initial TCB:

- Initialization subjects: `kernel_t`, `init_t`, `getty_t` `initrc_t`, `mount_t`, `inetd_t`.
- Administration subjects: `sysadm_t`, `setfile_t`, `bootloader_t`, `load_policy_t`.
- Authentication subjects: `local_login_t`, `sshd_t`.

The user then specifies the subjects that they seek to protect. These are added to the initial TCB. If this TCB specification has conflicts with the underlying SELinux policy, the conflicts must be resolved. A conflict is identified if a TCB subject attempts to read a file that has been written to by a N-TCB subject.

Gokyo classifies these conflicts based on their properties and then suggests a resolution strategy. Table 7.1 lists Gokyo's conflict resolution strategies.

Gokyo's conflict-based approach requires significant user input as each TCB component has to be manually specified. This causes the required user input to scale with the number of components in the TCB. We argue that our approach alleviates this problem by automatically deriving the TCB.

PolyScope (Lee et al., 2021) is a tool to analyze the security of systems such as Android that depend on a combination of Unix DAC and SELinux policies that

Class	Resolution Strategy
TCB or Candidate	Add the low-integrity subject to the TCB.
Exclude Type	Exclude the type from the analysis.
Sanitize	Sanitization of low-integrity data through a TP.
Denial	Deny the permission that causes the conflict.
Modify Policy	Modify the policy to circumvent the conflict.

Table 7.1: Classification of TCB integrity conflicts by Gokyo (Jaeger et al., 2003).

form the access control in Android systems. For this reason it is referred to as a multi-policy access control analysis tool. While similar multi-policy tools (Chen et al., 2017; Hernandez et al., 2020) have been realized in the past, PolyScope's novelty is the concept of `permission expansion`. This concept refers to adversaries expanding their permissions by exploiting discretionary elements of the access control system. This is important for Android systems, which map Android permissions into Unix DAC permissions, which can be modified by the owner of a resource in a potentially-exploitable way.

PolyScope's focus is on systems that depend on both Unix DAC permissions and SELinux. PolyScope is able to identify problematic DAC permissions that can lead to attacks executed by an adversary. This, however, does not align with our goals of policy approximation, where the objective is to identify permissions in the SELinux policy that conflict with the CW-Lite policy.

Most SELinux analysis tools attempt to answer the question: "How does the policy behave?". SELint (Reshetova et al., 2017) has a different goal and answers the question: "How well the policy written?". As a result, SELint does not provide the typical information flow analysis tools used to identify integrity violations. Instead, SELint provides varied functionality through plugins. Functionality provided by the included plugins includes: replacement of rules with equivalent macros, identification of potentially risky rules, identification of ineffective or unnecessary rules, and verification of the properties of the policy through rules that must not be allowed (`neverallow` rules).

SELint provides unique analyses, but does not support information flow analysis, which is essential for the implementation of an information flow model, such as Biba or CW-Lite. Compared to our approach, SELint does not address the correctness of the policy configuration, opting to focus on the policy clarity instead.

8 Discussion

In this chapter, we discuss the challenges associated with our conflict-free approach. We conclude by presenting ideas for future research.

8.1 Comparison of the conflict-free and the conflict-based approach

Compared to the conflict-based approach, we argue that the conflict-free approach reduces manual user intervention as well as the probability of human mistakes by minimizing the number of decisions that have to be made. These improvements can be attributed to two key differences.

1. *Method for TCB generation.* The conflict-based approach requires a prespecified TCB as input. This TCB is obtained through means unrelated to the primary analysis in the approach. The process of manually forming a TCB is a task that requires significant security expertise and effort. One could try to mitigate this issue by using a TCB created by someone else, such as the initial TCB proposed by (Jaeger et al., 2003). However, we argue that doing this fails to consider the uniqueness of each application, resulting in non-optimized TCBs.

   The conflict-free approach generates the TCB during the analysis using the graph traversal algorithm. If necessary, this TCB is further optimized using the minimum cut algorithm and user domain knowledge.

2. *Number of decisions.* The conflict-based approach yields all conflicts between the SELinux policy and the CW-Lite policy with regards to a TCB configuration. These conflicts are then evaluated and patterns in the conflict may be used as heuristics suggesting a particular resolution strategy.

   The conflict-free approach employs the minimum cut algorithm to determine the minimum number of flows that have to be cut to separate the TCB from an adversary. Evaluating these flows is similar to that of conflict-resolution. However, because the number of these flows is the smallest possible, the result is that the number of decisions is at most as large as with the conflict-based approach.

Our implementation of the conflict-free approach relies on the minimum cut algorithm. We recognize two challenges that this presents.

1. *Attack surface knowledge* The minimum cut algorithm requires modeling of an adversary as the source node. Subsequently, attack surface knowledge is required as the user is expected to provide SELinux types that may be compromised by the given adversary.

2. *Perspective on the TCB.* The conflict-based approach focuses on the interior of the TCB, because the entire process of TCB generation starts from the most important resources while expanding outwards to include other necessary types

in the TCB. As a result, the information flows to the TCB that cause conflicts are likely close in proximity to the TCB. An application developer is more likely to be familiar with information flows close to their application, resulting in more well informed decisions regarding conflict resolution.

The conflict-free approach focuses on the boundary of the TCB. Cuts suggested by the minimum cut algorithm are likely close in proximity to the modeled adversary. This is because cutting the flows as early as possible results in the minimum number of cuts. Compared to the conflict-based approach, information flows that have to be examined are likely further away in proximity from the application. An assumption can be made that developers may be less familiar with these flows.

8.2 Future work

In addition to the user input that we have already incorporated, we observe other types of input that could potentially improve our analysis.

8.2.1 Software installation data

The set of components present in SELinux reference policies are often a superset of the set of components present in an actual system. In other words, SELinux reference policies often contain types for components that are not present in the system. Thus, if an information flow involves a component that cannot be present in the system, the information flow can be excluded.

8.2.2 Code complexity data

Code complexity analysis is a heuristic that is often employed to identify software components that are most likely to be defective. The hypothesis that more complex code directly correlates with more software defects has been supported by several studies (Antinyan et al., 2017; Zhang et al., 2007; Zhang, 2009). Furthermore, since security bugs can be categorized as a type of software defect, we can extend the hypothesis to correlate more complex code with less secure software. As a result, code complexity metrics can be useful as an additional heuristic when determining filtering interfaces.

An easily acquirable metric is Source Lines of Code (SLOC) or simply Lines of Code (LOC). Measuring LOC is achieved by counting the number of lines in a program's source code. Thus, LOC is primarily a metric for measuring program size. However, for the purposes of code complexity, it is assumed that an increase in code size leads to an increase in complexity, and therefore vulnerability. More sophisticated metrics include McCabe's Cyclomatic Complexity (CC) (McCabe, 1976) and Halstead's complexity measures (Halstead, 1977).

We plan to incorporate code complexity measurement in our application. Ideally, we would display code complexity data associated with each SELinux type that is part of a suggested cut. Types with higher levels of code complexity than others

would suggest higher levels of risk if an information flow involving said type would be assigned as a filtering interface.

8.2.3 Alternative optimization goal

We recognize that minimizing the number of decisions on information flows does not necessarily reflect the workload imposed on the user. This is because the workflow also includes modifying the policy. Therefore, instead of minimizing the number of cuts, an alternative approach would be to minimize the number of modifications on the policy. Ultimately, testing the application with real-world applications is necessary to determine the optimal course of action.

8.2.4 Alternative LSMs

Another question to consider is whether SELinux is the correct tool for the task. While it is reasonable to assume that SELinux enforces all of the rules described in the policy, there is a possibility that other LSMs might be better suited for our purposes. SELinux was our initial choice due to a large number of previous research conducted on SELinux policy analysis.

9 Conclusion

Much of the previous work on SELinux policy analysis in regards to TCB integrity has focused on resolving conflicts between an integrity policy and the SELinux policy. In this thesis, we have proposed and implemented an alternative approach, the conflict-free method, using a minimum cut algorithm to identify the minimum number of information flows that have to be cut in order to separate a modeled adversary from the protected resources in the TCB. The user is expected to manually examine only these information flows and apply their domain knowledge to classify the flows as either necessary (not cut) or as filtering interfaces (assumed to be safe). The minimum cut algorithm can then be executed again until a satisfactory result has been obtained. The resulting output includes a proposed TCB and the corresponding SELinux rule changes necessary to protect the integrity of this TCB.

We believe the greatest contribution of our approach is that the user is spared the need to manually construct a TCB. Instead, the user can focus their efforts on evaluating the TCB that we propose semi-automatically through our analyses. Partially automating this part of the process simplifies the task for the user, requiring less SELinux expertise from the user as well as saving time and effort.

References

G. J. Ahn, W. Xu, and X. Zhang. 2008. Systematic Policy Analysis for High-Assurance Services in SELinux. In *Proceedings of the 2008 IEEE Workshop on Policies for Distributed Systems and Networks*. IEEE, 3–10. https://doi.org/10.1109/POLICY.2008.18

R. Anderson. 2020. *Security Engineering: A Guide to Building Dependable Distributed Systems* (3rd ed.). Newark: John Wiley & Sons.

V. Antinyan, M. Staron, and A. Sandberg. 2017. Evaluating Code Complexity Triggers, Use of Complexity Measures and the Influence of Code Complexity on Maintenance Time. *Empirical Software Engineering* 22 (2017), 3057–3087. https://doi.org/10.1007/s10664-017-9508-2

D. E. Bell and L. J. LaPadula. 1975. Computer Security Model: Unified Exposition and Multics Interpretation. *Technical Report, MITRE Corp.* (1975).

K. J. Biba. 1974. Integrity Considerations for Secure Computer Systems. *Technical Report, MITRE Corp.* (1974).

H. Chen, N. Li, W. Enck, Y. Aafer, and X. Zhang. 2017. Analysis of SEAndroid Policies: Combining MAC and DAC in Android. In *Proceedings of the 33rd Annual Computer Security Applications Conference*. Association for Computing Machinery, 553—-565. https://doi.org/10.1145/3134600.3134638

D. D. Clark and D. R. Wilson. 1987. A Comparison of Commercial and Military Computer Security Policies. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*. IEEE, 184–184. https://doi.org/10.1109/SP.1987.10001

A. Eaman, B. Sistany, and A. Felty. 2017. Review of Existing Analysis Tools for SELinux Security Policies: Challenges and a Proposed Solution. In *Proceedings of the 2017 E-Technologies: Embracing the Internet of Things*. Springer International Publishing, 116–135. https://doi.org/10.1007/978-3-319-59041-7_7

J. D. Guttman, A. L. Herzog, and J. D. Ramsdell. 2003. Information Flow in Operating Systems: Eager Formal Methods. In *Proceedings of the 2003 Workshop on Issues in the Theory of Security*. MITRE Corp. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.473.8427&rep=rep1&type=pdf

M. H. Halstead. 1977. *Elements of Software Science*. North-Holland. https://cds.cern.ch/record/281413

I. Herman, G. Melancon, and M.S. Marshall. 2000. Graph visualization and navigation in information visualization: A survey. *IEEE Transactions on Visualization and Computer Graphics* 6, 1 (2000), 24–43. https://doi.org/10.1109/2945.841119

G. Hernandez, D. J. Tian, A. S. Yadav, B. J. Williams, and K. R. B. Butler. 2020. BigMAC: Fine-Grained Policy Analysis of Android Firmware. In *Proceedings of the 29th USENIX Security Symposium*. USENIX Association, 271–287. https://www.usenix.org/conference/usenixsecurity20/presentation/hernandez

T. Jaeger, R. Sailer, and X. Zhang. 2003. Analyzing Integrity Protection in the SELinux Example Policy. In *Proceedings of the 12th Conference on USENIX Security Symposium*, Vol. 12. USENIX Association, 5. https://www.usenix.org/legacy/events/sec03/tech/full_papers/jaeger/jaeger.pdf

M. Jahoda, R. Krátký, and B. Ančincová. 2021, accessed: October 2021. Red Hat Enterprise Linux 6 SELinux Documentation. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/security-enhanced_linux/index

R. Keller, C. M. Eckert, and P. J. Clarkson. 2006. Matrices or Node-Link Diagrams: Which Visual Representation is Better for Visualising Connectivity Models? *Information Visualization* 5, 1 (2006), 62–76. https://doi.org/10.1057/palgrave.ivs.9500116

Y. T. Lee, W. Enck, H. Chen, H. Vijayakumar, N. Li, Z. Qian, D. Wang, G. Petracca, and T. Jaeger. 2021. PolyScope: Multi-Policy Access Control Analysis to Compute Authorized Attack Operations in Android Systems. In *Proceedings of the 30th USENIX Security Symposium*. USENIX Association. https://www.usenix.org/conference/usenixsecurity21/presentation/lee-yu-tsung

F. Mayer, K. MacMillan, and D. Caplan. 2006. *SELinux by Example: Using Security Enhanced Linux*. Prentice Hall.

T. J. McCabe. 1976. A Complexity Measure. *IEEE Transactions on Software Engineering* SE-2, 4 (1976), 308–320. https://doi.org/10.1109/TSE.1976.233837

E. Reshetova, F. Bonazzi, and N. Asokan. 2017. SELint: An SEAndroid Policy Analysis Tool. *Proceedings of the 3rd International Conference on Information Systems Security and Privacy* (2017). https://doi.org/10.5220/0006126600470058

M. Said and S. Mohamed. 2011. SEGrapher: Visualization-based SELinux Policy Analysis. In *Proceedings of the 4th Symposium on Configuration Analytics and Automation*. IEEE, 1–8. https://doi.org/10.1109/SafeConfig.2011.6111675

P. Samarati and S. C. de Vimercati. 2001. Access Control: Policies, Models, and Mechanisms. In *Proceedings of the 2000 Foundations of Security Analysis and Design*. Springer Berlin Heidelberg, 137–196. https://doi.org/10.1007/3-540-45608-2_3

R. Sassu. 2019, accessed: October 2021. Infoflow LSM. Linux Security Summit North America. https://www.youtube.com/watch?v=pQHE-GYoXr8

SELinuxPolicy. 2014, accessed: October 2021. Reference Policy Documentation. https://github.com/TresysTechnology/refpolicy/wiki

SELinuxProject. 2014, accessed: October 2021b. SETools. https://github.com/SELinuxProject/setools

SELinuxProject. 2016, accessed: October 2021a. Apol Documentation. https://github.com/SELinuxProject/setools/blob/master/qhc/infoflow.html

U. Shankar, T. Jaeger, and R. Sailer. 2006. Toward Automated Information-Flow Integrity Verification for Security-Critical Applications. In *Proceedings of the 2006 ISOC Networked and Distributed Systems Security Symposium*. The Internet Society. https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.115.630&rep=rep1&type=pdf

B. Shneiderman and A. Aris. 2006. Network Visualization by Semantic Substrates. *IEEE Transactions on Visualization and Computer Graphics* 12, 5 (2006), 733–740. https://doi.org/10.1109/TVCG.2006.166

C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman. 2002. Linux Security Modules: General Security Support for the Linux Kernel. In *Proceesings of the 11th USENIX Security Symposium*. USENIX Association. https://www.usenix.org/conference/11th-usenix-security-symposium/linux-security-modules-general-security-support-linux

W. Xu, S. Mohamed, and G. J. Ahn. 2013. Visualization-based Policy Analysis for SELinux: Framework and User Study. *International Journal of Information Security* 12, 3 (2013), 155–171. https://doi.org/10.1007/s10207-012-0180-7

G. Zhai, T. Guo, and J. Huang. 2015. SCIATool: A Tool for Analyzing SELinux Policies Based on Access Control Spaces, Information Flows and CPNs. In *Proceedings of the 2015 Trusted Systems*. Springer International Publishing, 294–309. https://doi.org/10.1007/978-3-319-27998-5_19

H. Zhang. 2009. An Investigation of the Relationships Between Lines of Code and Defects. In *Proceedings of the 2009 IEEE International Conference on Software Maintenance*. IEEE, 274–283. https://doi.org/10.1109/ICSM.2009.5306304

H. Zhang, X. Zhang, and M. Gu. 2007. Predicting Defective Software Components from Code Complexity Measures. In *Proceedings of the 13th Pacific Rim International Symposium on Dependable Computing*. IEEE, 93–96. https://doi.org/10.1109/PRDC.2007.28