Review of Existing Analysis Tools for SELinux Security Policies: Challenges and a Proposed Solution

Amir Eaman[1,2(✉)], Bahman Sistany[2], and Amy Felty[1]

[1] School of Electrical Engineering and Computer Science, University of Ottawa, Ottawa, Canada
{aeama028,afelty}@uottawa.ca
[2] Irdeto Canada Corporation, Ottawa, Canada
bahman.sistany@irdeto.com

Abstract. Access control policy management is an increasingly hard problem from both the security point of view and the verification point of view. SELinux is a Linux Security Module (LSM) implementing a mandatory access control mechanism. SELinux integrates user identity, roles, and type security attributes for stating rules in security policies. As SELinux policies are developed and maintained by security administrators, they often become quite complex, and it is important to carefully analyze them in order to have high assurance of their correctness. There are many existing analysis tools for modeling and analyzing SELinux policies with the goal of answering specific safety and functionality questions. In this paper, we identify and highlight current gaps in these existing tools for SELinux policy analysis, and propose new tools and technologies with the potential to lead to significant improvements. The proposed solution includes adopting a certified access control policy language such as ACCPL (A Certified Access Core Policy Language). ACCPL comes with formal proofs of important properties, and our proposed solution includes adopting it to facilitate various analyses and proof of reasonability properties. ACCPL is general, and our goal is to design a certified domain-specific policy language based on it, specialized to our task.

Keywords: SELinux · Access control · Security policies · Analysis tools

1 Introduction

On Linux based systems, many security exploits attempt to target system daemons that often run with elevated or even unlimited privileges (e.g. as root). Once the attacker gets access to a daemon, the whole system is compromised since the attacker obtains permanent root privileges on the system. The traditional Discretionary Access Control (DAC) mechanism that Unix/Linux systems use leaves important security decisions up to the discretion of the individual users

E. Aïmeur et al. (Eds.): MCETECH 2017, LNBIP 289, pp. 116–135, 2017.
DOI: 10.1007/978-3-319-59041-7_7

and administrators, resulting in an ad-hoc system where some applications or daemons are well configured whereas others have too many unnecessary permissions. SELinux [18] as an access control mechanism at the operating system level integrates DAC and Mandatory Access Control (MAC). MAC, and in particular SELinux, mandates a central policy-driven approach to access control and regulates DAC's access decisions. SELinux works based on the principle of least privilege, and every grant of access must have the corresponding allow rule in the security policy to permit that access. This means that when DAC allows access to a subject, the access request still needs to be checked by MAC as well. If an access request is denied by DAC, MAC will not get involved.

The policy language used to develop SELinux policies is a complex language encompassing an integration of Role-Based Access Control (RBAC), Type Enforcement (TE), and Multi-Level Security (MLS) [16]. Partly as a result of this fact and partly due to the way the language has been designed to specify fine-grained access control needs (among other reasons), the policies typically are comprised of thousands of policy statements; this makes policy development and analysis very difficult. The complexity of resulting SELinux policies means that for example, safety guarantees cannot be given, defeating the main purpose for SELinux in the first place. Even when a policy is considered both safe and functional, each addition, deletion or modification of the policy has the potential to break the baseline. The need for analysis tools for SELinux policies has been recognized from almost the very beginning with the expectation that such tools would be the silver bullet for SELinux security administrators.

This paper aims to identify and highlight current gaps in existing tools and technologies for SELinux policy analysis, which could potentially lead to improvements and new tools and technologies. A proposal toward closing the gaps identified in the technology and tools discussed in this paper is given. Section 2 describes basic concepts of access control. Section 3 presents the SELinux policy language structures that are used for expressing SELinux policies. In Sect. 4, existing tools developed by different research teams are analyzed, and important problems are identified. Section 5 describes identified challenges for SELinux and proposes a solution for overcoming these challenges by adopting a certified access control policy language. Finally, Sect. 6 concludes.

2 Preliminaries

2.1 Access Control

Access control can be described as a security service that guards protected resources against unauthorized access [4]. Access control, as an IT security service, deals with three primary entities in a system: *subjects* that require access to resources, *objects* or resources that are accessed by subjects, and *actions* that are performed by subjects on objects. Actions can range from being as simple as reading data to sharing or executing data [16]. The final protected system must satisfy information security measures consisting of confidentiality, integrity, and availability (known as the CIA triad).

2.2 Access Control Models

Access control models define the structure and language for describing system policies and relevant procedures for processing them. Three widely used models for different access control policy types were mentioned earlier: Discretionary Access Control (DAC), Mandatory Access Control (MAC), and Role-Based Access Control (RBAC) [23].

RBAC maps users to roles, which are sets of authorized permissions. RBAC lumps users together in bunches and assigns permissions to these groups; thus, a user can have a specific permission if the permission is assigned to a role that is associated with the user.

The traditional DAC model relies heavily on user identity, which can lead to a compromise of the whole system in the case when the attacker obtains root privileges on the system. The owner grants privileged access to objects. DAC defines an access control list for every object in the system. DAC-based systems are coarse-grained since, trivially, DAC cannot provide fine-grained controls using only user identity as the basis of decisions. Two user privileges that are possible in these systems are *admin* and *non-admin*.

MAC overcomes drawbacks of DAC to restrict access to objects solely on user identity by abstracting system resources into subjects and objects. MAC assigns security attributes to system resources and provides a foundation for security administrators to define access control policies for their environments, which requires more systems administration. Different MAC security models target the preservation of different security objectives in the system, provided by defining fixed security rules as their access control policies. Three important security models for MAC include the *Bell-LaPadula* (BLP) model preserving confidentiality, the *Biba* model preserving integrity, and the *Clark-Wilson* model preserving integrity [23]. We describe each in more detail below.

The Bell-LaPadula model ensures confidentiality of information by not allowing a subject to write objects of lower security level and not allowing a subject to read objects of higher security level. BLP security rules restrain the transfer of information from a higher security level subject to a lower security level object in a system.

The Biba integrity model protects the integrity of information by enforcing a policy defined by particular security rules. These security rules include rules that do not allow a subject to read objects of a lower integrity level and do not allow a subject to write objects of a higher integrity level.

The Clark-Wilson model focuses on the integrity of information and uses four security categories as the language for defining access control policy rules [4]. Policy rules control the integrity of the system by ensuring the integrity of the security categories of the model. These categories are:

- Constraint Data Items (CDIs): objects that are integrity protected.
- Unconstrained Data Items (UDIs): objects that are not integrity protected.
- Integrity Verification Procedures (IVPs): verifiers to check CDI integrity.

– Transformation Procedures (TPs): certified procedures to transition CDIs or UDIs to other CDIs. TPs are supposed to be a filter to control information transfers from low or high integrity objects to high integrity objects.

3 SELinux Overview

SELinux is a Linux-based access control framework developed by the United States National Security Agency (NSA) [18]. SELinux is compiled into the kernel and supported through the Linux Security Module (LSM). LSM is a kernel-level security framework that provides the possibility of attaching various security mechanisms to the Linux kernel, such as SELinux, without directly depending on the kernel objects. SELinux implements the MAC model within Linux-based distributions and provides more granular control of security. SELinux primarily involves labeling that divides system resources into subjects, which are processes, and objects, such as files and sockets. Every system resource receives a label which is a combination of values of the user, role, and type attributes. These attribute-value pairs form the *security context* for system resources.

3.1 SELinux Architecture

The SELinux security module implements the Flask architecture in a Linux environment [14]. A feature of the Flask architecture is the separation of security policy logic from the enforcement mechanism. The Security Server is a kernel component responsible for making security decisions, and the Object Manager enforces these security decisions in the system. The Access Vector Cache (AVC), is another component of the SELinux architecture. AVC stores the security policy look-up results to improve the performance of the decision-making procedure. Searching the AVC is faster, so access requests that have been previously processed can be quickly answered without searching the entire security policy again. Figure 1 depicts the core decision-making architecture of SELinux. Attributes used to determine the decisions of the SELinux access control mechanism are described in the following section.

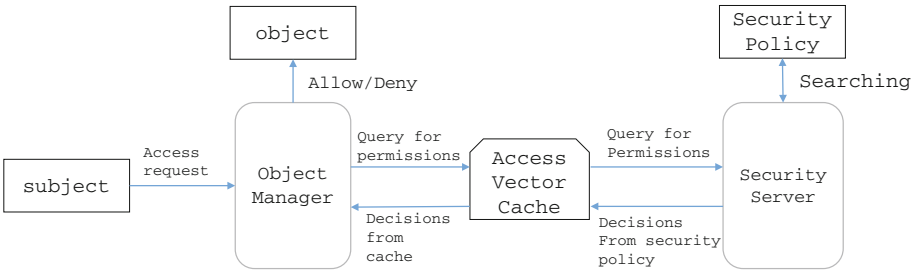

Fig. 1. Core decision-making architecture in SELinux

3.2 SELinux Access Control Criteria

The SELinux attributes include *user*, *role*, *type* or *domain*, and *level*, each described below.

SELinux introduces its own user attribute, and the Linux user attribute is mapped to the SELinux one [16]. The mapping of Linux users to SELinux users can be viewed using the Linux shell command "`semanage login l`."

The role attribute comes from RBAC. The SELinux security policy determines which users are authorized for each role. In particular, these roles are used for making role-based access control decisions. They specify which domains are authorized for which users and thus permit user entry into these domains. The shell command "`seinfo -r`" lists roles that are available in the system.

The SELinux type attribute is the most important attribute within a security context. Terminologically, to help distinguish subjects and objects, types and domains mean the same thing, but domains classify subjects, while types classify objects.

The SELinux level attribute is the final attribute in the security context. This attribute is used only in the MLS access control mechanism, which is not the main policy type of the SELinux access control framework. MLS policies use the level attribute for expressing rules that restrict access requests, which makes it a suitable access control scheme for military type environments. SELinux can be loaded into the Linux kernel without accommodating MLS [8].

Labeling is the main functionality of SELinux with the goal of labeling all system resources with a proper security context. SELinux primarily focuses on Type Enforcement (TE) related to the type/domain field of security contexts. TE allows creation of different domains in the system through assigning subjects to domains, and subsequently associating them with objects. All of these authorized associations are stated in a SELinux security policy by using TE rules. In addition to TE, SELinux allows the expression of restrictions on the other fields of the security context.

3.3 The SELinux Security Policy Language

A SELinux security policy is a collection of statements that defines the threshold for accepting an access request. SELinux denies interaction of subjects and objects by default; in particular, with an empty SELinux policy every access request will be denied. Figure 2 lists syntax of important SELinux security policy rules. Some main rules of the SELinux policy language related to TE, or to user and role components of security context are described.

Type Enforcement (TE) Rules of SELinux mainly include two kinds of rules [16]: Access Vector (AV), and Type Rules, which consist of Object Transition Rules and Domain Transition rules. Access Vector (AV) rules allow, audit, or deny interaction between two types. AV rules include `allow`, `dontaudit`, `auditallow`, and `neverallow` statements [14]. For example, consider the AV rule in Fig. 2 appearing on the first line. This rule allows the process with domain `SourceDType` to have actions `perm1` or `perm2` on the object of type `TargetType`

```
allow SourceDType TargetType : class1 {perm1 perm2};
type_transition SourceDomain TargetType: class1 new_type
type_transition SourceDomain TargetType: process new_type;
constrain classobject_list  permission_list B(t1,r1,u1,t2,r2,u2)
```

Fig. 2. Sample rules of a SELinux security policy

and object class of `class1`. An object class specifies a possible instance of all resources of a certain kind, such as files, sockets, and directories.

Object Transition Rules in SELinux can be used to specify the type of objects that will be created at runtime. For example, consider the object transition rule on the second line in Fig. 2. This type transition means objects of type `TargetType` that are newly created by a process with the domain of `SourceDomain` will take the default type `new_type` instead of `TargetType`. The object class `class1` specifies the object category of `SourceDomain` and `new_type`.

Domain Transition Rules change the domain of a subject to a new domain. For example, consider the domain transition rule on the third line in Fig. 2. This domain transition states that if a process of the domain `SourceDomain` executes a file with the type `TargetType`, the new domain of the process will be `new_type`.

SELinux policies also include Constraints. Software developers use constraints to introduce new criteria for granting access requests to objects. Constraints can refine an explicitly allowed access request through enforcing extra considerations for certain users, roles, and types in the decision-making process of the access, expressed as boolean conditions. For example, consider the fourth line in Fig. 2. `B(t1,r1,u1,t2,r2,u2)` is a boolean expression expressing constraints on the type, role, and user of the source entity security context `(t1,r1,u1)` and target entity security target `(t2,r2,u2)`. This constraint defines the requirements under which the operations in `permission_list` are allowed for the class objects in `classobject_list`. If these requirements are not met by an access request, the operations in `permission_list` will be denied.

The policy language that is used to develop SELinux policies is a complex language consisting of a combination of RBAC, TE, and optionally MLS rules. As mentioned, SELinux policies typically include thousands of policy statements, which makes development and analysis of SELinux policies quite difficult. SELinux policy language statements enable security administrators to configure the required permissions for accesses. Sample policy rules for an application (called App here) are shown in Fig. 3. These rules define a single domain entry to execute App through a domain transition.

3.4 SELinux Policy Analysis Tools

Many analysis tools have been proposed to help policy administrators analyze SELinux policies with respect to these properties. Among existing tools, some are developed while others are at a prototype stage. The typical structure of policy analysis tools is demonstrated in Fig. 4. The complexity of the SELinux

```
require {
attribute domain;
attribute file_type;
attribute exec_type;
type sysadm_t;
attribute sysadm_r;
class process transition;
role sysadm_r; }
```

Adding types and attributes that are required by the rules

```
type app_t;
typeattribute app_t domain;
type app_exec_t;
typeattribute app_exec_t file_type;
typeattribute app_exec_t exec_type;
```

Declaring new types and classify them by attributes

```
role sysadm_r types app_t;
```

Assigning roles to types

```
type_transition sysadm_t app_exec_t : process app_t;
allow sysadm_t app_exec_t : file {getatr execute};
allow app_t app_exec_t : file entrypoint;
allow sysadm_t app_t : process transition;
```

Defining default transition and its required access

Fig. 3. App program security policy rules in SELinux

policy language makes analyzing SELinux policies and even implementing policies very difficult. As a result, virtually all analysis tools provide some kind of other intermediate language for SELinux security administrators, as shown in Fig. 4.

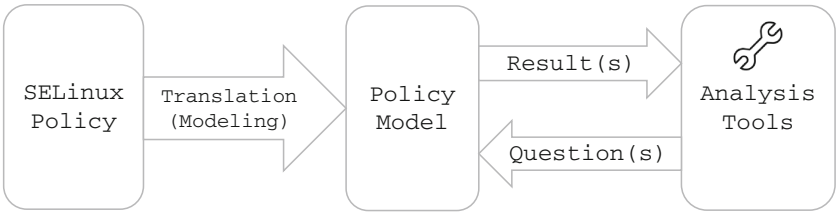

Fig. 4. Typical structure of SELinux analysis tools

3.5 Access Faults

In SELinux, access is a unique combination of (1) a source domain, (2) a destination type, (3) associated action(s), and (4) a destination object class. Almost all policy analysis tools try to detect different access faults that are implicitly leaked through security policy rules. Access faults are caused by implicitly assigning privileges using type attributes or default types available in the security policy [19,20] or generating rules from the SELinux audit log [28].

There are several kinds of access faults. Any access that doesn't meet the security goals of the system is called a *sneak access*. A *backdoor access* occurs

when the policy accepts a request that was not explicitly allowed in the specification, but was introduced manually by policy developers or automatically by some tools that audit SELinux logs. These backdoors are usually inserted in a policy in order to run a legitimate program that has some problems reaching its required resources because of SELinux access control. *Wrong actions* are any access in which proper actions are not allowed by the security policy. *Missing actions* are any access in which the intended actions are not allowed because of improper policy rules.

3.6 Answering Security Questions

SELinux analysis tools answer questions about the properties of a policy configuration using two fundamental methods, *Information Flow* [2,8] and *Access Control Spaces* [11].

Information flow is about the reachability of a resource from another resource where some information is transferred by performing a particular operation. For example, there is an information flow between a subject `S_1` to a subject `S_2` if `S_1` can perform a write operation on some objects on which `S_2` can perform a read operation [29].

A subject's access control space is composed of all realizable permissions of the subject. The access control space of a subject forms a set which can be classified as the following five subspaces [11].

- Specified Permissions: permissions that are currently assigned to the subject according to the current specification
- Permissible Permissions: permissions assigned to the subject that are authorized by the policy developers
- Prohibited Permissions: permissions whose assignment violates security goals
- Unknown Permissions: permissions that are neither permissible nor prohibited
- Obligated Permissions: permissions that the subject must have according to policy rules

Zanin and Manicini [31] replace the access control space concept by the concept of an *Accessibility Space*, which introduces additional specific sets over possible permissions of an entity. Accessibility spaces remove unknown permissions as SELinux is based on a closed world assumption [31]. Information flow uses abstract views over the possible permissions in the available configuration of a security policy; hence, it can be considered as using just two subspaces, the allowed subspace and the denied subspace, to check the properties of the policy.

3.7 Querying SELinux Security Policies

SELinux analysis tools help identify policy *conflicts* that are caused by policy violation against specifications that describe protection needs of the system. To identify conflicts and check that security goals are achieved, security

administrators can query policies about *safety*, *completeness*, *integrity*, *separation of duty* (SoD), as well as some other questions that identify conflicts by posing questions about access faults and other high level security goals.

A policy specification is safe if subjects have no sneak access to resources in the system and all security goals are satisfied by the specification. A policy is complete if all intended permissions are specified in the policy; in other words, all requests are explicitly allowed or prohibited in a complete policy [21]. Integrity and safety have almost the same security aims, but integrity uses security models such as Biba to identify conflicts rather than just focusing on the realization of security specifications to check security goals. Separation of duty can be considered as another kind of integrity checking that is defined with a simple security model. SoD means separating the domain of subjects into those that execute an executable file and those that create or modify the executable files [15].

4 Taxonomy for SELinux Policy Analysis Tools

Table 1 compares eighteen SELinux analysis tools. The comparison considers available features and techniques utilized in the tools. The table shows that different analysis tools have different capabilities in terms of providing safety, completeness, integrity, and SoD analyses. The other features that are compared in Table 1 are browsing a policy, rewriting a policy, and building customized queries. The analysis tools employ various forms of query language syntax to allow security administrators to make queries for checking specific properties of the security policy. Various techniques are utilized as methods of analysis; they model the security policy with well-known concepts such as mathematical sets [31], information visualization [15,30], and computer security models [1]. Some analysis methods expand all macros, while some perform on-demand expansion of macros [2] in the policies. SELint [19] goes further and replaces policy rules with proper macros of the policy rules, which provides the capability to suggest improvements. The last three tools in Table 1—SEAL, EASEAndroid, and SELint—are for analyzing Security Enhancements for Android (SEAndroid), which is an Android port of the SELinux MAC mechanism [20]. Because most of the tools in Table 1 are not available publicly, the information provided here is based on the studies conducted for the tools as presented by the authors. In Sect. 4.1, we briefly discuss some aspects of each tool. Other information can be read directly from the table. In Sect. 4.2 we discuss some general problems.

4.1 Tool Descriptions

APOL [25] is a member of the SETools suite [16]. A user loads a SELinux security policy file or a compiled binary policy file to APOL to begin the analysis procedure. By loading the policy file, the user can select attribute items from enabled lists, which are loaded according to the rules in the SELinux security policy file. Then, the user can use regular expressions to specify a search in several analysis modules for particular attributes. A great number of SELinux

Table 1. Analysis tools for SELinux security policies

Analysis tool	Safety analysis	Completeness analysis	Integrity analysis	SoD analysis	Information flow analysis	Method of analysis	Policy browsing	Rewriting the policy	Method of modeling	Query language	Macro expansion
APOL	✓		✓	✓	✓	Information flow	✓		Syntactic analysis	Selecting attributes from menus	✓
SLAT	✓	✓			✓	Information flow-Model checking			XSB logic	✓ (Regular expressions)	✓
XcelLog	✓	✓			✓	Information flow-Deductive spreadsheets			Sets of values	Writing set formulas	✓
GOKYO	✓	✓	✓			AC spaces-TCB			AC spaces-Graphcal AC model (Sets)	✓	✓
PAL	✓	✓	✓	✓	✓	Logic programming			XSB logic	✓	
SELAC	✓				✓	AC spaces-Mathematical set			25 Sets		✓
SPTrack	✓				✓	Data visualization			Graphs		✓
SEGrapher	✓			✓	✓	Data visualization - Clustering			Graphs-Clustering of nodes	Selecting types from menus	✓
SEAnalyzer	✓			✓	✓	Colored Petri nets			Diagrams & Sets	✓	✓
LOPOL	✓				✓	Deductive database		✓	Logical relations	Datalog query	✓
SEEdit	✓			✓	✓	Higher level language, SPDL		✓	Grouping permissions-Access log user decision		
PVA	✓	✓	✓	✓	✓	Information visualization techniques	✓	✓	Semantic substrates - Adjacency matrix	Graphical user interface	✓
GPA	✓		✓		✓	Information visualization techniques			Semantic substrates - Adjacency matrix	Graphical user interface	✓
Sepol2HRU	✓		✓			HRU security model			HRU Model simulation		✓
SCIATool	✓		✓		✓	Colored Petri nets, Information flow, AC space		✓	NA	Wizard-Style query	✓
SEAL	✓		✓		✓	Information flow			Syntactic analysis		✓
EASEAndroid						Semi-supervised learning		✓	Parsing the audit log		✓
SELint						Information flow			Syntactic analysis		✓

analysis tools (e.g., [20,29,30]) use APOL libraries for their development and often a comparison of the ease of use as compared with APOL is carried out.

Guttman and Herzog [8] describe a four-step procedure used in the SLAT tool for verifying security goals in SELinux configurations. These steps include modeling, expressing goals, enforcing goals of the model, and implementation. The language that encodes the security goals is based on information flow diagrams, and security goals are expressed using a language similar to regular expressions. Five different access control relations are defined to model SELinux configurations, which are based on key concepts of SELinux. The authorization relation uses access control relations to authorize class-permission pairs for a process against a resource. Finally, a model checker verifies establishment of security goals of the policy.

XcelLog [21] combines policy rules and deductive spreadsheets (DSS) for taking advantage of deductive reasoning. The transformation of policy rules to deductive spreadsheets is a semi-automated process. Cells of the deductive spreadsheets are capable of containing a set of values or recursive formulas.

GOKYO [12] tries to reveal various conflicts in the policy and find missing or incorrect constraints. GOKYO resolves these constraints, according to the concept of access control spaces, which can reduce the complexity of the policy. The process of resolving conflicts is based on removing the unknown subspace and performing a kind of balancing among different kinds of rules in the policy. The approach creates a near-minimal trusted computing base (TCB) in the SELinux policy model and verifies whether the TCB is integrity-protected.

PAL [2] (Policy Analysis using Logic-Programming) is implemented using the XSB logic programming language. A XSB program translates a policy to a set of facts and builds queries that are answered from these facts. This technique is macro-preserving, which means that the macros in the policy get expanded on demand. As stated in [2], PAL's use of macros that are not fully expanded is efficient and unique in contrast to other tools such as SLAT, APOL, and GOKYO.

SELAC [31] (SELinux Access Control) models each language construct in the security policy language as a mathematical set. A collection of sets is constructed in an incremental way from the specification. As stated in the paper, SELAC has removed the redundant space, unknown space, and general subspaces that are used in GOKYO.

SPTrack [6] represents SELinux security policies as interaction graphs. The nodes in an interaction graph are security contexts made up of subjects or objects. The edges in this graph are possible interactions among nodes, all of which are included according to rules in the policy. The edges of the graph are colored based on the criticality levels of paths between nodes.

SEGrapher [15] begins its analysis with data visualization of the SELinux policy and then generates optimized graphs using the concept of clustering. Cluster-based graphs represent policy analysis results, which have been simplified by the use of clusters. To model a SELinux policy, the tool focuses on the access vector rules within it. These rules are represented as edges in a directed graph.

The building block for clustering the nodes is a focus-graph, based on an object-type set. An object-type set is the set of all types that an object can access [15].

SEAnalyzer [5] utilizes Colored Petri Net (CPN) diagrams for representing SELinux security policies and security goals. A rather complex query language for expressing security goals has been developed for SEAnalyzer with a smaller character count in comparison to PAL and SLAT.

Lopol [13] takes advantage of deductive database analysis and Datalog queries. Lopol policy analysis includes analyzing a collection of logical relations and inference rules. Lopol is capable of rewriting the policy. Rewriting a policy is performed through goal-projection, which involves reverse compilation of the inference rules to the policy.

SEEdit [17] uses the concept of integrated permissions to reduce the number of configuration elements. Integrated permissions group related permissions into a single unit, which causes the removal of the macro entities from the policy. SEEdit creates security policies using a higher-level language called SPDL. SPDL tools consist of two sections including an allow generator and a template generator. The former reads the access log to generate an SPDL based specification for permitting access. The latter uses the user's knowledge to generate an SPDL configuration to make a program that is problematic due to access control restrictions run correctly. Finally, an SPDL converter generates the policy file.

PVA [29] and GPA [30] tools use a visualized-based framework for analyzing, expressing policy queries, and identifying policy violations of a SELinux policy. The concepts and proposed framework in GPA have been slightly enhanced in PVA. The framework begins by representing the policy layout using two visual mechanisms: Semantic Substrates and Adjacency Matrices. The framework provides a visual query formulation that helps system administrators specify precise queries on the policy. Subsequently, the framework generates a policy violation graph to represent the violations that are identified by the integrity model. The integrity model is based on Biba and the concepts of Trusted Computing Base (TCB) and Transaction Procedure in the Clark-Wilson security model. The framework introduces some approaches, such as filtering and ignoring, to modify the policy graph in order to remove any policy violations. GPA proposes identifying and protecting the TCB of a system using the Information Domain.

Sepol2HRU [1] establishes an isomorphic mapping between a SELinux access control system and a HRU security model as defined in [9]. Transforming the SELinux security policy to a HRU model allows the application of the analysis tools available for the HRU model to SELinux security policies. Transforming a policy to a HRU model is a three-step procedure. (1) The elements in an SELinux access control system such as rules and types are mapped to heterogeneous mathematic standard concepts like sets, matrices, and functions. (2) These elements are rewritten to a single composed matrix. (3) The authorization scheme is inferred. Sepol2HRU outputs the SELinux security policy as an HRU model description in a single file in a XML-based format.

SCIATool [32] integrates three policy analysis methods including access control spaces, information flows, and colored Petri-nets. The architectural design

of the SCIAtool is based on the modularity principle. SCIATool's approach to integrity analysis is the use of a TCB which means that integrity analysis verifies that subjects inside the TCB are prohibited from reading incorrect information from non-trusted objects.

SEAL [20] is a tool for SEAndroid policy analysis. Finding problematic patterns in SEAndroid policies is the main purpose of the study in [20]. The identified patterns consist of overuse of default types, overuse of predefined domains, forgotten or seemingly useless rules, and potentially dangerous rules.

EASEAndroid [28] proposes a semi-supervised learning approach to refining SEAndroid security policies. SEAndroid security policies require continuous refinements due to continuous updates to Android and to emerging new attacks. A policy is refined based on analyzing the audit log and information in one access event. The tool parses information available on access events, which provides information for building access patterns. These access patterns act as a knowledge base for the learning process of the approach.

SELint [19] helps Original Equipment Manufacturers (OEMs) to produce better SEAndroid policies by optimizing the security policy. SELint has several plugins, including simple macro expansion, parameterized macro expansion, risky rules, unnecessary rules, and user *neverallow* rules. Plugins that operate on macros try to replace certain kinds of rules with macros. In contrast, other analysis tools seek to remove macros because the semantics of the m4-based language, i.e. macro language, is uncertain [10].

4.2 SELinux Security Policy Problems

Analysis tools can help administrators to check system security goals. However, most analysis tools provide some other intermediate language for SELinux security administrators. Although these extra facilities can help with the analysis of policies, at the same time, they often add more complexity to the whole access control process because they require equally complex semantics. On the other hand, developing policies in SELinux leads to quite complex policies, and developing a policy is a cumbersome and error-prone process [10]. Moreover, analysis tools only provide low-level queries, which fail to cover the very large potential query space of SELinux policies [11]. The following is a user's concern about SELinux policies from the Fedora SELinux support mailing list [24]:

> What directories and files does Guix [a package program] need to touch? ... What kinds of labels do I need to introduce to my system? What kinds of tools do I need to use to integrate a Guix policy to the prebuilt policies? ... After all, most software developers ignore SELinux and won't bother publishing a complete access requirement specification.

5 SELinux Challenges and Proposed Solution

The SELinux policy language doesn't have formal semantics. Its semantics is given in terms of a natural language description. Expressing the semantics of

an access control policy language in a natural language (e.g. English) results in ambiguity in the specification of behavior of policy statements. For this reason, along with reasons mentioned earlier, i.e. the complexity of the language, and the fact that policies are expressed at a very fine-grained level, both the development and the analysis of policies are difficult. Consider the last three lines of Fig. 3. They are included to protect the `entrypoint access` [16] of the `app_t` domain. Removing any one of these rules will break the intended protection because in order for a domain transition to occur, all three rules are required. The first rule provides execution permission for the domain `sysadm_t` on the file with the type `app_exec_t`; the second rule provides an entrypoint for the domain `app_t`; the third rule provides a type transition to the new type `app_t` from the current type `sysadm_t`. The fact that SELinux rules are so fine-grained adds to the complexity of SELinux. Both writing and analyzing policies are difficult tasks. It is hard for administrators to express the desired protection using such a low-level language.

5.1 Existing SELinux Analysis Tools

As mentioned, the complexity of the SELinux policy language itself complicates both the implementation of policies as well as the ability to analyze them. As a result, many tools are complex and it is difficult to establish the correctness of the analyses they perform. One problem with these tools is that they do not use the same criteria in support of each other; moreover as mentioned, analysis tools try to provide some other intermediate language for SELinux security administrators. Although these extra facilities can help with writing various queries, they require equally complex semantics. Furthermore, existing SELinux analysis tools barely scratch the surface and only offer the possibility of doing simple queries.

5.2 SELinux Policy Language Challenges

As a result of our study described in the previous section, we can summarize the many gaps between the SELinux policy language and current existing analysis tools:

- SELinux as an access control framework requires third-party analysis tools to help security administrators write policies and check various properties.
- The inherent complexity of the SELinux policy language has caused a lot of tools to try to establish intermediate language structures to overcome this complexity; however, they require equally complex semantics and syntax.
- Software developers continually add new rules to SELinux security policies, while fine-tuning the policy to handle access problems of newly installed applications, using the system audit log file. The practice of making every deny access found in the SELinux audit log into new rules in the policy is extremely error prone and can lead to compromising the safety of the system, again due to the complexity of the SELinux access control policy language.

- There is no proof for the correctness of policy analysis tools or formal semantics to make sure their results are reliable. There are informal justifications for results, but no formal justification of results.
- Overall SELinux lacks clarity as an access control language. The clarity of an access control policy language can provide better decision making for incremental policy writing, ease of analysis, and ease of reasoning.

The goal of our proposed solution is to make the process of developing and analyzing policies simpler by adopting a security policy language that is more coarse-grained, in which administrators can more directly express their security goals at a higher level, and providing tools to translate such policies to the more fine-grained level of SELinux

5.3 Ease of Reasoning About SELinux Policies

A particular set of properties which may be used as a basis for formally comparing and contrasting access control policy languages include *safety*, *independent composition*, and *monotonicity* [26]. An access control policy language that is safe, independently composable and monotonic is said to be most amenable to reasoning as compared to one that does not have any of these properties. In addition, these properties and others mentioned in [22] can be used to classify different access control policy languages along the reasonability spectrum. Being able to reason about policies written in an access control policy language directly leads to another property that is desirable in a policy language. Such a policy language has the property that formal analysis and verification of specific policy statements can determine whether or not the policy meets the high-level goals of the system.

Using the definition of an access control policy language as presented in [26], the SELinux access control policy language can be considered as a tuple $L = (P, Q, G, N, \ll . \gg)$ where P is a set of SELinux policies, Q is a set of requests or queries, G is the granting decisions, and N is the non-granting decisions, with the constraint $G \cap N = \emptyset$. Let D denote the set of decisions $G \cup N$. The last element of L, $\ll . \gg$, is a function taking a policy $p \in P$ to a relation between Q and D. Given a policy $p \in P$, a query $q \in Q$ is assigned a decision of $d \in D$. L also defines a partial order on decisions such that $d \leq d'$ if either $d, d' \in N$ or $d, d' \in G$ or $d \in N$ and $d' \in G$; in other words, non-granting decisions are all the same, granting decisions are all the same, and all granting decisions are considered greater that all non-granting decisions. Note that for SELinux, $D = \{Granted, Denied\}$, $G = \{Granted\}$ and $N = \{Denied\}$. Let DC, TC, CLS, and PRM be the set of all domains, types, object classes, and permissions, respectively, available in a system. Queries are of the form (dc, tc, cls, prm, m) where $dc \in DC$ is the domain type of the subject, $tc \in TC$ is the type of the resource, $cls \in CLS$ is the class of the resource, $prm \in PRM$ is the permission or permissions, and m expresses properties of the query that are not about the Type Enforcement mechanism of SELinux. Two queries $q = (dc, tc, cls, prm, m)$ and $q' = (dc, tc, cls, prm, m')$ have relation $q \trianglelefteq q'$ if $m \implies m'$. In the rest

of this section, we assess SELinux with regard to its Type Enforcement (TE) mechanism to determine if it satisfies the three properties mentioned earlier. The TE mechanism is based on TE rules available in SELinux policies. SELinux policies are organized into modules, which allows on-the-fly dynamic loading as needed. Each policy module has its own set of rules.

An access control policy language is considered *safe* if a request with less information will lead to a decision that is less than the decision reached for a request with more information, according to the defined partial order on decisions [27]. For example, requests with incomplete information should only result in a grant of access if a request with more complete information results in a grant of access. Based on this definition, safety can be defined as the following formula:

$$\forall(p \in P), (q, q' \in Q), (d, d' \in D),$$
$$q \trianglelefteq q' \;\; \& \;\; q \ll p \gg d \;\; \& \;\; q' \ll p \gg d' \implies d \leq d'.$$

Theorem 1. *The SELinux access control policy language is not safe with respect to* $\trianglelefteq$.

Proof. Consider the policy module p_a below along with requests q_a and q_b:

p_a : `allow sAtype_t mytype_t : file read`
`role sCrole_r type sAtype_t`
$q_a =$ `(sAtype,mytype_t, file, read, {})`
$q_b =$ `(sAtype,mytype_t, file, read, {role ∈ sDrole_r})`

For request q_a, p_a produces *Granted*, while for q_b, it produces *Denied*. Note that $q_a \trianglelefteq q_b$, $q_a \ll p_a \gg$ *Granted*, $q_b \ll p_a \gg$ *Denied*, but *Granted* $\not\trianglelefteq$ *Denied*, which contradicts safety.

An access control policy language has the *independent composition* property if taking into account all policy modules and rendering a decision gives the same result as combining the decisions obtained from each primitive policy in isolation. As a result, independent composition can be defined as the following formula, in which $\square$ is the decision composition operator for combining policy decisions and $\oplus$ is the composition operator defined in the language for combining policies. Some policy languages, such as FOL [26], allow more than one interpretation of the operator that combines policies, thus preventing them from having the independent composition property.

$$\forall(p_1, \ldots p_n \in P), (q \in Q), (d_1, \ldots d_n, d^* \in D),$$
$$q \ll p_1 \gg d_1 \;\; \& \cdots \& \;\; q \ll p_n \gg d_n \;\; \& \;\; q \ll \oplus(p_1, \ldots, p_n) \gg d^* \implies$$
$$\square(d_1, \ldots d_n) = d^*.$$

Composing policies in SELinux simply means adding them together to form one big policy. A request is denied if any *one* of the individual policies produces `denied`. Trivially, SELinux access control always reaches a single decision when combining all policy modules or decisions.

Theorem 2. *The SELinux access control policy language has the independent composition property.*

Proof Sketch. By definition, $\square(d_1, \ldots d_n)$ is *Denied* if any of $d_1, \ldots, d_n$ are *Denied*. In this case, the combined policy decision d^* will also be *Denied* by the definition of SELinux policy combination. Otherwise, $d_1, \ldots, d_n$ are all *Granted*, and in this case, both $\square(d_1, \ldots d_n)$ and d^* will be *Granted*, again by definition.

An access control policy language is *monotonic* if adding another primitive policy does not change the combined decision from granting to non-granting.

Theorem 3. *The SELinux access control policy language is not monotonic.*

Proof. Consider policy modules p_c and p_d below along with request q_c :

p_c : `allow Dtype1_t type2_t : file open`
p_d : `constraint process transition`
`(u1=user1 and t1=type1 and t2=type2_t)`
q_c = `(Dtype1_t, type2_t, file, open, {user ∈ user2})`

The policy p_c will result in *Granted* for the request q_c and adding policy module p_d will result in *Denied*, which changes the decision from *Granted* to *Denied*.

5.4 Using a Certified Policy Language to Express SELinux

A small and certifiably correct policy language can be a good candidate for SELinux style access control. ACCPL (A Certified Core Policy Language) [22] is a certified policy language that can be used to represent general access control rules and policies. ACCPL has formal semantics, which include a precise definition of a function that takes a query and returns an allow or deny decision. The Coq Proof Assistant [3,7] has been used to develop proofs for theorems about the expected behavior of ACCPL when evaluating a request according to the given policy and to machine-check the proofs ensuring correctness guarantees are provided. The compactness and verifiability of ACCPL as an access control policy language provides for ease of analysis and reasoning, in comparison to the SELinux policy language. These capabilities are guaranteed because ACCPL satisfies the ease of reasoning properties of [27]. This fact helps system administrators to easily manage and check the intended security level of the system.

ACCPL can be used to encode and implement other policy-based access control languages such as SELinux policy language, taking advantage of its characteristics. We propose to develop a certified domain-specific policy language that appropriately accommodates specialized features of the SELinux Type-Enforcement mechanism. Once we do, we will be able to analyze policies formally using the proof environment for ACCPL implemented in Coq.

6 Conclusion

SELinux is a MAC based access control framework in Linux distributions. The inherent complexity of SELinux and its approach requires additional manual interaction of security administrators to develop or analyze policies. The complexity of the SELinux policy language itself complicates both the implementation of policies as well as the ability to analyze them. Many research projects have included the design and implementation of analysis tools to overcome this problem. Because of the lack of clarity and complexity of the policy language, the implemented tools often utilize languages that are different from but equivalent to SELinux, with equally complex semantics, or they simplify the languages so that they do not implement the full SELinux policy language. Thus, these tools cannot cover the identified SELinux challenges and it is difficult to establish the correctness of the analyses they perform as well. A certified access control policy language can provide ease of use and analysis; moreover, it provides an environment for verification of its properties. ACCPL (A Certified Core Policy Language) is a general certified access control policy language that is more amenable to analysis and reasoning. We plan to design a certified domain-specific policy language based on it for our task. Developing a certified analysis tools base on the certified Type-Enforcement policy language that simplifies and fosters policy analysis will be another direction of future work.

Acknowledgements. Financial support from the Network of Centres of Excellence (MITACS) and Irdeto Canada is gratefully acknowledged.

References

1. Amthor, P., Kühnhauser, W.E., Pölck, A.: Model-based safety analysis of SELinux security policies. In: 5th International Conference on Network and System Security (NSS), pp. 208–215 (2011)
2. Archer, M., Leonard, E.I., Pradella, M.: Modeling security-enhanced Linux policy specifications for analysis. In: 3rd DARPA Information Survivability Conference and Exposition (DISCEX-III), pp. 164–169 (2003)
3. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Springer, Heidelberg (2004)
4. Bishop, M.A.: The Art and Science of Computer Security. Addison-Wesley Longman Publishing Co. Inc., Boston (2002)
5. Chen, Y.-M., Kao, Y.-W.: Information flow query and verification for security policy of Security-Enhanced Linux. In: Yoshiura, H., Sakurai, K., Rannenberg, K., Murayama, Y., Kawamura, S. (eds.) IWSEC 2006. LNCS, vol. 4266, pp. 389–404. Springer, Heidelberg (2006). doi:10.1007/11908739_28
6. Clemente, P., Kaba, B., Rouzaud-Cornabas, J., Alexandre, M., Aujay, G.: SPTrack: visual analysis of information flows within SELinux policies and attack logs. In: Huang, R., Ghorbani, A.A., Pasi, G., Yamaguchi, T., Yen, N.Y., Jin, B. (eds.) AMT 2012. LNCS, vol. 7669, pp. 596–605. Springer, Heidelberg (2012). doi:10.1007/978-3-642-35236-2_60
7. Coq Development Team: The Coq Proof Assistant Reference Manual (Version 8.6) (2016). https://coq.inria.fr/distrib/current/files/Reference-Manual.pdf

8. Guttman, J.D., Herzog, A.L., Ramsdell, J.D., Skorupka, C.W.: Verifying information flow goals in Security-Enhanced Linux. J. Comput. Secur. **13**(1), 115–134 (2005)
9. Harrison, M.A., Ruzzo, W.L., Ullman, J.D.: Protection in operating systems. Commun. ACM **19**(8), 461–471 (1976)
10. Hurd, J., Carlsson, M., Finne, S., Letner, B., Stanley, J., White, P.: Policy DSL: high-level specifications of information flows for security policies. In: High Confidence Software and Systems (HCSS) (2009)
11. Jaeger, T., Edwards, A., Zhang, X.: Managing access control policies using access control spaces. In: 7th ACM Symposium on Access Control Models and Technologies (SACMAT), pp. 3–12. ACM Press (2002)
12. Jaeger, T., Sailer, R., Zhang, X.: Analyzing integrity protection in the SELinux example policy. In: 12th USENIX Security Symposium (2003)
13. Kissinger, A., Hale, J.C.: Lopol: a deductive database approach to policy analysis and rewriting. In: Security-Enhanced Linux Symposium, pp. 388–393 (2006)
14. Loscocco, P., Smalley, S.D.: Meeting critical security objectives with Security-Enhanced Linux. In: Ottawa Linux Symposium, pp. 115–134 (2001)
15. Marouf, S., Shehab, M.: SEGrapher: visualization-based SELinux policy analysis. In: 4th Symposium on Configuration Analytics and Automation (SAFECONFIG), pp. 1–8 (2011)
16. Mayer, F., Caplan, D., MacMillan, K.: SELinux by Example: Using Security Enhance Linux. Prentice Hall, Upper Saddle River (2006)
17. Nakamura, Y., Sameshima, Y., Tabata, T.: SEEdit: SELinux security policy configuration system with higher level language. In: 23rd Large Installation System Administration Conference, pp. 107–117 (2009)
18. National Security Agency: Security-Enhanced Linux (2016). https://www.nsa.gov/what-we-do/research/selinux/
19. Reshetova, E., Bonazzi, F., Asokan, N.: SELint: an SEAndroid policy analysis tool. CoRR abs/1608.02339 (2016)
20. Reshetova, E., Bonazzi, F., Nyman, T., Borgaonkar, R., Asokan, N.: Characterizing SEAndroid policies in the wild. CoRR abs/1510.05497 (2015)
21. Singh, A., Ramakrishnan, C.R., Ramakrishnan, I.V., Stoller, S.D., Warren, D.S.: Security policy analysis using deductive spreadsheets. In: ACM Workshop on Formal Methods in Security Engineering (FMSE), pp. 42–50 (2007)
22. Sistany, B.: A certified core policy language. Ph.D. thesis, University of Ottawa (2016). https://www.ruor.uottawa.ca/handle/10393/34865
23. Stallings, W., Brown, L.: Computer Security, Principles and Practices. Pearson Education, New York (2008)
24. The Fedora-SELinux Support List: Fedora SELinux Support. https://lists.fedoraproject.org/admin/lists/selinux.lists.fedoraproject.org/
25. Tresys Technology: APOL (2016). https://github.com/TresysTechnology/setools3
26. Tschantz, M.C.: The clarity of languages for access-control policies. Ph.D. thesis, Brown University (2005)
27. Tschantz, M.C., Krishnamurthi, S.: Towards reasonability properties for access-control policy languages. In: 11th ACM Symposium on Access Control Models and Technologies (SACMAT), pp. 160–169 (2006)
28. Wang, R., Enck, W., Reeves, D.S., Zhang, X., Ning, P., Xu, D., Zhou, W., Azab, A.M.: EASEAndroid: automatic policy analysis and refinement for Security-Enhanced Android via large-scale semi-supervised learning. In: 24th USENIX Security Symposium, pp. 351–366 (2015)

29. Xu, W., Shehab, M., Ahn, G.: Visualization-based policy analysis for SELinux: framework and user study. Int. J. Inf. Secur. **12**(3), 155–171 (2013)
30. Xu, W., Zhang, X., Ahn, G.: Towards system integrity protection with graph-based policy analysis. In: 23rd Annual International Federation for Information Processing (IFIP), Data and Applications Security XXIII, pp. 65–80 (2009)
31. Zanin, G., Mancini, L.V.: Towards a formal model for security policies specification and validation in the SELinux system. In: 9th ACM Symposium on Access Control Models and Technologies (SACMAT), pp. 136–145. ACM Press (2004)
32. Zhai, G., Guo, T., Huang, J.: SCIATool: a tool for analyzing SELinux policies based on access control spaces, information flows and CPNs. In: Yung, M., Zhu, L., Yang, Y. (eds.) INTRUST 2014. LNCS, vol. 9473, pp. 294–309. Springer, Cham (2015). doi:10.1007/978-3-319-27998-5_19