# Lopol: A Deductive Database Approach to Policy Analysis and Rewriting

Aleks Kissinger
*Center for Information Security*
*University of Tulsa*

John C. Hale
*Center for Information Security*
*University of Tulsa*

## Abstract

This paper presents a method for deductive database-driven analysis of Security-Enhanced Linux policies. First, it discusses how directives in an SELinux policy can be normalized and encoded as logical relations. Next, the paper describes how to use these relations in conjunction with inference rules to perform a number of simple analyses. The techniques used to detect security characteristics within a policy can then be applied to automated policy rewriting, where those characteristics (i.e. security goals) are actually projected onto the policy to generate a new one. The paper closes by discussing the use of a logical ruleset as a high-level language, allowing a policy writer to quickly tailor a large default policy (such as `strict`, `targeted`, or `refpolicy`) to the specific needs of a system or a class of systems.

## 1 Introduction

Mandatory access control (MAC) methods such as type enforcement (TE) and role-based access control (RBAC) can offer fine-grained control of security on very complex systems. However, an unfortunate side-effect of this control is that an administrator can quickly become distracted by the sheer size of the policy and lose sight of its overall aims. The goal of policy analysis is to aid the administrator or policy writer in understanding the policy and to call his attention to aspects that are important or in need of modification.

All policy analysis tools essentially have the same purpose: to evaluate a security policy's fulfillment of a set of goals and to suggest ways those goals might be better fulfilled. Current policy analysis tools fall into one of two categories: interactive and non-interactive. Interactive policy analyzers aim to provide rapid policy browsing and analysis feedback. A user can peruse the policy and get a feel for its inner workings in a much more natural and less intimidating manner than wading through a sea of directives. However, these tools often view analysis on a per-query basis, and provide little mechanism for building up a set of complex goals over time.

Non-interactive tools usually allow a user to specify some specific goal that a policy either will or will not fulfill, then run the analysis all at once. These kinds of tools often allow much more complex rulesets to be evaluated in greater depth, and can provide useful assessments of the policy as a whole, but they lack the organic analysis experience of interactive tools.

Logical policy analysis aims to bridge this gap. Lopol represents an SELinux policy as a collection logical relations and uses a logical language called Datalog[1] to analyze and manipulate the policy. Expressing policy analysis as a collection of Datalog inference rules allows for a brief, responsive, and moderately expressive way to explore a large, complicated policy. Since there is a one-to-one mapping from an inference rule back to a flat relation, rules can be converted back into SELinux policy.

## 2 Goals

There are three main goals for Lopol. The first is to provide a simple, flexible way to specify static characteristics of a policy. These characteristics may include dataflow, possible vectors of privilege escalation, and other things that can be determined before a policy is actually running on a system.

The second goal is to offer an interactive and responsive testbed for ruleset development. A developer can interactively build up a set of rules that works, then use those rules as primitives for higher-order rules. This allows a developer to start by constructing rules that provide useful information and come back later to support them with mathematical rigor.

The third goal is to shift the bulk of policy analysis development from compiler writers to policy writers and security professionals. In the past, the first step in analyzing security policy was to compile the policy into some

form that the analysis engine could understand. Only when this is done can one write the actual analysis code. By providing a set of security primitives and a mechanism for analyzing arbitrary relationships between those primitives, a larger set of custom-tooled analyses can be developed faster.

## 3 Representation

The core elements of Lopol analysis are relations and inference rules. Relations are sets of tuples with some predicate dictating how its members are related. They correspond to directives within the SELinux policy (e.g. "`allow ...;`"). Inference rules are simply Horn clauses (i.e. logical statements of the form $r_1 \wedge r_2 \wedge \ldots \wedge r_n \Rightarrow R$) consisting of relations and other inference rules. Inference rules are used to describe goals and characteristics within the security policy. All statements in the policy yield one or more base relations within Lopol. Simple policy directives usually yield a single corresponding tuple:

`type_transition t1 t2:c1 t3;`
$$\rightarrow (t_1,\ t_2,\ c_1,\ t_3)$$

However, statements consisting of sets[2] or complements[3] must be normalized into a relation consisting of flat (set-free) tuples. Policy directives are normalized in two steps. The first is to eliminate syntactic complements and replace them by the actual complemented set. The second is to take the cross product of all the arguments of the directive to yield a set of normalized tuples. Shown here is a sample normalization, where **T** is the set of all defined types and **C** is the set of all defined classes:

$$\begin{aligned}
&\mathbf{T} : \{t_1,\ t_2,\ t_3,\ t_4,\ t_5\};\ \mathbf{C} : \{c_1\}\\
&\texttt{type_transition \{t1 t2\} ~t3:c1 t4;}\\
\rightarrow\ &\{t_1,\ t_2\} \times \overline{\{t_3\}} \times \{c_1\} \times \{t_4\}\\
\rightarrow\ &\{t_1,\ t_2\} \times \{t_1,\ t_2,\ t_4,\ t_5\} \times \{c_1\} \times \{t_4\}\\
\rightarrow\ &\{(t_1,\ t_1,\ c_1,\ t_4),\\
&\ (t_1,\ t_2,\ c_1,\ t_4),\\
&\ (t_1,\ t_4,\ c_1,\ t_4),\\
&\ (t_1,\ t_5,\ c_1,\ t_4),\\
&\ (t_2,\ t_1,\ c_1,\ t_4),\\
&\ (t_2,\ t_2,\ c_1,\ t_4),\\
&\ (t_2,\ t_4,\ c_1,\ t_4),\\
&\ (t_2,\ t_5,\ c_1,\ t_4)\}
\end{aligned}$$

The final, normalized tuples form part of the new logical relation "*typetransition*". This normalization can cause a significant expansion of the data, but since it is handled by automated means, this growth is usually acceptable. By eliminating sets embedded in rules, the user can perform analysis directly on the set of relationships created by the statements within the policy, eliminating a layer of complexity. Note how the final product of this transformation is almost identical to the corresponding avtab entries produced by compiling with `checkpolicy`. The only major distinction is the preservation of names given in policy.conf, rather than their numeric security identifiers. All of the core SELinux directives are processed into relations like this. Below is a list of all the normalized relations represented by Lopol:

$$\begin{aligned}
\textit{userrole} &:= (u_1,\ r_1)\\
\textit{roletyple} &:= (r_1,\ t_1)\\
\textit{roledominance} &:= (r_1,\ r_2)\\
\textit{typealias} &:= (t_1,\ t_2)\\
\textit{typetransition} &:= (t_1,\ t_2,\ c_1,\ t_3)\\
\textit{avallow} &:= (t_1, t_2, c_1, p_1)\\
\textit{avauditallow} &:= (t_1, t_2, c_1, p_1)\\
\textit{avauditdeny} &:= (t_1, t_2, c_1, p_1)\\
\textit{avdontaudit} &:= (t_1, t_2, c_1, p_1)
\end{aligned}$$

Lopol only supports the core directives, but it is easily extensible to support additions such as MLS, MCS, conditional, and modular policy. Though the use of *avauditdeny* is deprecated, it is included simply for the sake of completeness.

In addition to the relations generated from the SELinux directives, Lopol also generates identity relations for each identifier type called "X-is". These relations are defined as: $\{(x_1, x_2) : x_1, x_2 \in \mathbf{X} \wedge x_1 = x_2\}$, where **X** is the set of all types, classes, SID's, users, attributes, permissions, or roles.

$$\begin{aligned}
\textit{typeis} &:= (t_1,\ t_2)\\
\textit{classis} &:= (c_1,\ c_2)\\
\textit{roleis} &:= (r_1,\ r_2)\\
\textit{sidis} &:= (s_1,\ s_2)\\
\textit{useris} &:= (u_1,\ u_2)\\
\textit{attributeis} &:= (a_1,\ a_2)\\
\textit{permissionis} &:= (p_1,\ p_2)
\end{aligned}$$

These relations are used primarily to serve the purpose of an equality operator within an inference rule. See section 5.1.2 for an example of this.

# 4 Implementation

Lopol is not a single program, but rather a series of utilities re-tooled for logical policy analysis. The most important component is a modified version of `checkpolicy`, which when passed the "-L" flag yields normalized tuple output in addition to the standard binary output. This output is in the form of two kinds of flat files: ".tuples" files and ".map" files. ".map" files contain an ordered list of strings corresponding to all of the identifiers matched by the policy of a particular kind (e.g. type, class, SID). ".tuples" files contain one tuple per line, each consisting of space-delimited integers referencing the ".map" file.

The "-L" flag also generates a Datalog header file called "lopol.datalog". This file contains all of the raw relation definitions and takes the tuple files as their input. The user can then write another Datalog file that includes "lopol.datalog" and defines a number of inference rules. A program called `bddshell` can then interpret this file and provide a means of interacting with the logical policy data via queries and other rule definitions.

`bddshell`[4] is a front-end to John Whaley's Binary Decision Diagram-Based Deductive DataBase, or `bddbddb` [7]. `bddbddb` has been primarily used for points-to analysis in Java bytecode, but its ability to understand transitive relationships and return Datalog queries quickly on huge knowledge bases has proven useful for policy analysis as well. Also, its solver can produce output from inference rule expansion either as binary decision diagrams (BDDs) or as flat tuples, which could be compiled back into SELinux policy.

# 5 Application

Inference rules can be constructed and queried to extract useful information from a policy, such as dataflow characteristics. Rules can also be constructed to behave functionally like primitive relations, but with additional constraints. These rules could be flattened into relations and compiled back into policy.

## 5.1 Inference Rule-driven Analysis

Inference rules are constructed on top of the relations generated by the policy. Many of these rules employ some manner of dataflow analysis. Implementing dataflow analysis in a logical system is fairly trivial. For low-level dataflow, this basically translates into computing the transitive closure of a "writes-to" relationship. Most of the dataflow analysis is a variation of this form:

```
### dataflow.datalog
# Define the writesTo relation.  Note
# how in Datalog, unlike PROLOG, atoms
# are enclosed in double quotes.

writesTo("a", "b").
writesTo("b", "c").
writesTo("b", "d").
writesTo("e", "f").

# The dataflow inference rule:
TraceFlow(x,y) :- writesTo(x,y).
TraceFlow(x,y) :- writesTo(x,z), \
                  TraceFlow(z, y).
```

Querying `TraceFlow` will yield the dataflow.

```
> TraceFlow("a", sink)?
    (sink="b")
    (sink="c")
    (sink="d")
```

Real world situations are more intricate than this, but this form acts as a basis for dataflow-driven analyses.

### 5.1.1 Example: Chasing Aliases

Often when faced with an unfamiliar security policy, the presence of type aliases can confound one's understanding of the policy's functionality. The following example defines what identifiers are aliases to a given type in the form of an inference rule, then provides a "flatted" type transition rule which accounts for the presence of aliases. This ruleset displays *all* the single-step transition rules, including those defined in the form of aliases to other types.

```
### alias.datalog

# look in '.bdd/' for Lopol data
.basedir ".bdd"

# include base SELinux relations
.include "lopol.datalog"

# Declare two rules, tell bddbddb to
# output them is *.bdd files.
TypeAl(t1:T, t2:T) output
TypeTr(t1:T, t2:T, c2:C, t3:T) output

# base case
TypeAl(t1, t2) :- typealias(t1,t2).

# reflexive case
TypeAl(t1, t2) :- typealias(t2,t1).

# transitive case
TypeAl(t1, t2) :- TypeAl(a, t1), \
                  TypeAl(a, t2).
```

```
# Define a version of typetransition that
# unrolls aliases.
TypeTr(t1, t2, c2, t3) :- \
       TypeAl(t1, a), \
       TypeAl(t2, b), \
       TypeAl(t3, c), \
       typetransition(a, b, c2, c).
```

This example shows how a few rules can quickly peel off a layer of complexity in the policy. It also shows how logical programs excel in recursively defining characteristics (e.g. A is an alias of B if A and B are both aliases of C). This is a broadening ruleset, which seeks to unroll some abstraction of the policy. As we shall soon see, there can also be narrowing rulesets, which may look for some particular characteristic.

#### 5.1.2 Some Helper Rules

Before we delve into a dataflow example, we define some helper rules. The first two rules are `WritePerms` and `ReadPerms`, which only match the types of permissions that can cause a dataflow. The third rule is `TypeAl`, which is taken from the previous example, to flatten aliases.

```
### lopolhelper.datalog

# include base SELinux relations
.include "lopol.datalog"

# Define a permission set for writes-to
# relationships.  Note how permissions
# are interpreted relative to class c.
WritePerms(p, c) :- \
  classis(c, "file"),
  permissionis(p, "write").
WritePerms(p, c) :- \
  classis(c, "file"),
  permissionis(p, "append").
# ...many more

# Same as about, but with reads-from.
ReadPerms(p, c) :- \
  classis(c, "file"),
  permissionis(p, "read").
ReadPerms(p, c) :- \
  classis(c, "file"),
  permissionis(p, "getattr").
# ...many more

TypeAl(t1, t2) :- typealias(t1,t2).
TypeAl(t1, t2) :- typealias(t2,t1).
TypeAl(t1, t2) :- TypeAl(a, t1), \
                  TypeAl(a, t2).
```

The use of these rules will vastly decrease the number of cases that need to be written for higher-order policy analysis rules. This will become apparent in the next example.

#### 5.1.3 Example: Dataflow

This example demonstrates the application of the dataflow pattern described in section 5.1. It also makes use of the helper rules defined in the previous section.

```
### dataflow2.datalog

.basedir ".bdd"
.include "lopolhelper.datalog"

FlowExists(t1, t2) :- \
  WritePerms(p, c), \
  TypeAl(t1, ta1), TypeAl(t2, ta2), \
  avallow(ta1, ta2, c, p).

FlowExists(t1, t2) :- \
  ReadPerms(p, c), \
  TypeAl(t1, ta1), TypeAl(t2, ta2), \
  avallow(ta2, ta1, c, p).

# A simple transitive closure:
TraceFlow(t1,t2) :- FlowExists(t1,t2).
TraceFlow(t1,t2) :- FlowExists(t1,x), \
                    TraceFlow(x, t2).
```

Now, querying `TraceFlow` will yield the dataflow to or from a type:

```
> TraceFlow(t, "system_t")?
    (t="system_t")
    (t="sysadm_t")
    (t="crond_t")
    (t="init_t")
    ...
```

In addition, `TraceFlow` can be used as the building block of an even more specific query. If the system administrator wanted to know all of the users on the system that have a dataflow path to "`sysadm_t`":

```
### myquery.datalog

# Include for the TraceFlow rule and the
# SELinux primitives.
.include "dataflow2.datalog"

MyQuery(u, r, tf) :- userrole(u, r), \
                     roletype(r, t), \
                     TraceFlow(t, tf).
```

And the query:

```
> MyQuery(u, r, "sysadm_t")?
    (u="root", r="sysadm_r")
    (u="root", r="system_r")
    (u="jjohnson", r="sysadm_r")
    (u="crookedjoe", r="sneaky_r")
```

Assuming the system administrator would not want the user "crookedjoe" to have any live dataflows to "sysadm_t", he could either remove his access to "sneaky_r" or conduct subsequent queries to find which type is responsible for the unwanted flow.

This is still somewhat of a toy query, as security problems are rarely this obvious, and mappings to the user level are often not the concern of a type-enforcement policy writer. In a real world situation, TraceFlow could be used in conjunction with rules defining trusted, "mediator" types to display situations where potentially dangerous and unmediated flows exist.

## 5.2 Automated Policy Rewriting

Lopol already has the ability to *describe* certain goals a policy must meet. However, since rules in Lopol can be transported back into SELinux policy, Lopol also has the potential to *impose* certain rules upon an existing policy via policy rewriting.

The core idea behind automated policy rewriting is goal-projection. It becomes useful to consider the concept of a security policy as existing in two separate domains: the model domain and the goal domain. The model domain, in this case, is the set of raw relationships described in an SELinux policy. The goal domain is represented by the set of security goals defined as inference rules. Projecting the data within in the model domain into the goal domain yields the final implementation of a policy guaranteed to uphold those goals.

Policy rewriting, given a ruleset, consists of three steps. First, inference rules designated for policy output are flatted into relations. This is possible because Datalog will only work with stratifiable rules (i.e. rules that always have a finite, flat representation). Next, relations sharing members are collapsed to eliminate redundancy. Finally, these new relations are compiled back into a new SELinux policy within the goal domain.

It is conceivable that a system developer could take a canned policy like `strict` and define the security requirements of some business-specific software as inference rules. The result of projecting the default policy onto this rule set would be a new policy as close to `strict` as possible but fulfilling the needs of the business-specific software.

A few issues arise when one looks at automatically generating policy, especially when using machine-rewritten policy in conjunction with manually modified policy. As a policy is rewritten, it becomes more verbose and less human-readable. However, rule-collapsing and certain organization standards may help to mitigate this problem.

# 6 Related Work

Tresys technology offers a suite of tools designed to aid in policy analysis and design. Among those tools is `apol`. `Apol` has two distinct parts: an underlying library for conducting dataflow analysis and a GUI front-end which allows a user to browse the policy and drive the back-end analysis engine [6].

`slat`, developed by MITRE corporation, uses model checking in combination with many-sorted first order logic to validate a policy against a collection of information flow assertions [2]. In various other papers leading up to `slat`, Guttman and Herzog describe a technique called "Eager Formal Methods", which consists of a four-step process of approaching a security problem formally (define model, define goals, enforce goals upon model, build implementation) [3, 1]. Policy rewriting in Lopol also seeks to abide by this methodology.

# 7 Conclusion

The policy language that SELinux employs is extremely powerful and expressive. However, the expense of this power is readability. Though they may be practical for policies written by hand, aliases, typesets, and other syntactic sugar can often confound the meaning of a policy when it contains thousands of directives. Policy analysis tools can help with understanding a policy, but it is often difficult for a policy writer to effectively transfer information provided by an automated engine to manual policy modification.

For a raw representation, Lopol strips away syntactic complexity and allows users to construct their own abstractions and build off of those. Lopol provides the unique ability to use a single language and methodology to both express policy characteristics and impose them upon an existing policy. That language is also designed such that low-level rule sets can be used as the basis for successively higher-order analyses. Lopol forms a backbone of a new kind of policy analysis, but it is only the first step. This paper presented a few rudimentary rulesets, but it has become clear that the bulk of the work left to be done is in development of rules that are genuinely useful to system administrators and policy writers. Lopol will reach its full potential as such rules are developed and formalized.

## 8 Acknowledgments

This research was supported by MPO Contract MDA 904-02-R-0039. Special thanks to Scott Fujan for his support and editing.

## 9 Availability

`bddshell`, `bddbddb`, and related tools are all released under the GNU Lesser/Library General Public License (LGPL) and available on SourceForge. To download `bddshell`, which contains everything needed for working with ".tuples" and ".map" files, go to:

`http://bddshell.sourceforge.net`

The modifications to `checkpolicy` are pending release, and will probably be in the form of a source code patch.

## References

[1] GUTTMAN, J. D., HERZOG, A. L., AND RAMSDELL, J. D. Information flow in operating systems: Eager formal methods.

[2] GUTTMAN, J. D., HERZOG, A. L., AND RAMSDELL, J. D. Slat: Inforation flow analysis in security enhanced linux.

[3] HERZOG, A. L., AND GUTTMAN, J. D. Eager formal methods for security management.

[4] KISSINGER, A. bddshell project. http://bddshell.sourceforge.net.

[5] LOSCOCCO, P., AND SMALLEY, S. Integrating flexible support for security policies into the linux operating system. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference* (Berkeley, CA, USA, 2001), USENIX Association, pp. 29–42.

[6] TRESYS TECHNOLOGY. setools project. http://tresys.com/selinux/selinux_policy_tools.shtml.

[7] WHALEY, J. bddbddb project. http://bddbddb.sourceforge.net.

## Notes

[1] Datalog is a functional subset of PROLOG. It contains relations (predicates) and inference rules (Horn clauses). It is limited to logical problems that can be stratified, or separated into distinct, strongly connected groups called strata. However, this enables any rules specified in the language to be completely flatted into raw relations.

[2] In SELinux policy syntax, sets are of the form: {`t1 t2 ... tn`}. Note the lack of commas. These should not be confused with regular, set-theoretic notation used to describe the logical model: $\{t_1, t_2, \ldots, t_n\}$.

[3] In the raw policy, complements are of the form: `~t1`. Their Lopol equivalent (set-complement) is represented as $\overline{\{t_1\}}$