A Datalog Semantics for Paralocks

STM 2012

Niklas Broberg, David Sands, Bart van Delft
Information Flow Control

Enforcing policies on data inside a program
Information Flow Control

Not putting policies on files etc., but on:

- Variables
- Fields
- Method's arguments
- Method's return value
- Method's side effects
- ...
Information Flow Control

- Policy: “Missile Launch Codes are only to be read by the US Government”

send_To_Swedish_King ( Missile_Launch_Codes ) ;
Information Flow Control

• Policy: “The gender of the Queen's child is only to be announced on baptism day”

```java
if ( gender == "boy" )
    send_To_Newspaper ( "It's a boy!" ) ;
else
    send_To_Newspaper ( "It's a girl!" ) ;
```
Paragon

- **Statically (type) checks** policy violations

- Fields etc. need to be **annotated** with policies by the programmer

- Policies are **defined** using
Paralocks

A simple and expressive policy specification language

(simpler than DLM)
Paralocks Policies

- Policy: “Missile Launch Codes are only to be read by the US Government”

{ USGovernment : }

(Annotate the Missile Launch Codes with this policy)
Paralocks Policies

- Policy: “The newspaper can only know the gender on baptism day”

{ Newspaper : Today_Is_Baptism_Day }

Actors

Locks
Paralocks Policies

- Policy: “The newspaper can only know the gender on baptism day”

- Locks can be opened and closed

- Paragon keeps track of the lock state: the set of definitely opened locks

- Validity of an information flow depends on the current lock state
Policy: “Everyone can listen to this online music stream if they paid for it”

\[
\{ \ 'x : \text{Has\_Paid}(\ 'x) \ \} 
\]
{ 'x : Has_Paid('x) }

{ alice : }
\{'x : Has_Paid('x) \}

\{ alice : \}

Lock State
\{ 'x : Has\_Paid('x) \}

{ alice : }

Lock State
\{ 'x : Has_Paid('x) \} 

\textbf{open} Has_Paid(alice) ;

\textbf{Lock State}

Has_Paid(alice)

\{ alice : \}
Paralocks Policies

- Policy: “Everyone who is Alice's boss can read her data”

\[
\{ \text{alice} : ; \ 'x' : \text{BossOf('x, alice)} \}
\]

\[
\{ \text{BossOf('x, 'y)} : \text{BossOf('x, 'z)},
\text{BossOf('z, 'y)} \}
\]
\{ alice : ; 'x : BossOf('x, alice) \}
Policy Operations

```java
method m() {
    my_salary = 0;
    your_salary = 0;
}

your_data := my_salary

my_salary + your_salary

method m() {
    my_salary = 0;
    your_salary = 0;
}
```
In earlier work, Ad-hoc semantics for policies have been proposed. Algorithmic semantics for $\sqsubseteq, \sqcup, \sqcap$ have also been studied. Mentioning of Datalog provides a framework for expressive dynamic language. Paralocks, a language for building expressive and fine-grained information flow policies. The expressive power of Flow Locks (Broberg & Sands) allows policies involving runtime polynomial-time actions. We encode of Myers and Liskov’s Decentralized Label Model (DLM). Paralocks can be statically verified by providing a simple programming language incorporating Paralock policy specifications, and a static type system which soundly enforces information flow security according to the Paralock semantics.

Categories and Subject Descriptors D.3 [PROGRAMMING LANGUAGES]: F.3.1 [LOGICS AND MEANINGS OF PROGRAMS]: Specifying and Verifying Reasoning about Programs

General Terms Security, Languages, Verification

1. Introduction

Issues of software security can be cruelly categorized into three broad domains:

• Access control deals with security at the end points of a system, to verify that an entity is allowed to access the system, and to what extent.
Datalog

- A deductive database query language (70's)

I am having an unbirthday party!
Datalog

• Typical **rule**: 

\[
\text{atParty}(X) :\;\text{friendOf}(X, \text{madHatter}).
\]

• **Database Schema** *(including recursive rules)*: 

\[
\text{unbirthday\_party} = \{
\text{atParty}(\text{madHatter}).
\text{atParty}(X) :\;\text{friendOf}(X, \text{madHatter}).
\text{atParty}(X) :\;\text{friendOf}(X, Y), \text{atParty}(Y). \}
\]
Datalog

- **Query**: set of rules defining one predicate

```
who_drinks_tea =
    { drinksTea(X) :- atParty(X). }
```

- **Extensional Database**: set of **facts** not occurring as heads of rules

```
friendships = {
    friendOf(whiteRabbit, madHatter),
    friendOf(redQueen, madHatter),
    friendOf(marchHare, madHatter),
    friendOf(alice, whiteRabbit)
}
```
Datalog

- **Answer Set**: all facts derivable on the query, given a Database Schema and Extensional Database

```
who_drinks_tea(unbirthday_party, friendships) =
{ drinksTea(madHatter),
  drinksTea(whiteRabbit),
  drinksTea(redQueen),
  drinksTea(marchHare),
  drinksTea(alice)        }
```
Paralocks ~ Datalog

- Strong *correspondence* between the two:

<table>
<thead>
<tr>
<th>Locks</th>
<th>~</th>
<th>Facts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock properties</td>
<td>~</td>
<td>Database Schema</td>
</tr>
<tr>
<td>Policies</td>
<td>~</td>
<td>Queries</td>
</tr>
<tr>
<td>Lock state</td>
<td>~</td>
<td>Extensional Database</td>
</tr>
</tbody>
</table>
Paralocks ~ Datalog

- Strong **correspondence** between the two:

  - **Locks** ~ **Facts**
  - **Lock properties** ~ **Database Schema**
  - **Policies** ~ **Queries**
  - **Lock state** ~ **Extensional Database**

```
{ BossOf('x, 'y) :
  BossOf('x, 'z),
  BossOf('z, 'y) }
```

```
Lock State
BossOf(cat, alice)
BossOf(madHatter, cat)
```
Policies – Datalog Style

{ alice : ; 'x : BossOf('x, alice) }

alice_data_policy =
{ Flow(alice) :- . 
  Flow(X) :- BossOf(X, alice) . }
Policies – Datalog Style

- **Answer set** to a policy = the **actor set** to whom information may flow

```datalog
who_drinks_tea(unbirthday_party, friendships) =
{ drinksTea(madHatter), drinksTea(whiteRab) ... }
```

```datalog
alice_data_policy(lock_properties, lock_state) =
{ Flow(alice), Flow(cat), Flow(madHatter) }
```
Redefining semantics

\[
\text{my\_salary + your\_salary}
\]

\[
\text{my\_salary}
\]

\[
\text{my\_policy(lock\_prop, l\_state)} = P
\]

\[
\text{your\_salary}
\]

\[
\text{your\_policy(lock\_prop, l\_state)} = Q
\]

\[
\text{new\_policy(lock\_prop, l\_state)} = P \cap Q
\]
Redefining semantics

```java
method m() {
  my_salary = 0;
  your_salary = 0;
}
```

\[
\begin{align*}
\text{my\_salary} & = 0; \\
\text{my\_policy}(\text{lock\_prop}, \text{l\_state}) & = P \\
\text{your\_salary} & = 0; \\
\text{your\_policy}(\text{lock\_prop}, \text{l\_state}) & = Q \\
\text{new\_policy}(\text{lock\_prop}, \text{l\_state}) & = P \cup Q
\end{align*}
\]
Redefining semantics

Paralocks – Role-Based Information Flow Control and Beyond

Niklas Broberg
Department of Computer Science and Engineering,
University of Gothenburg
and Chalmers University of Technology, Sweden
nbro@chalmers.se

David Sands
Department of Computer Science and Engineering,
Chalmers University of Technology, Sweden
dave@chalmers.se

Algorithmic semantics for \(\sqcup, \sqcap\)
provides computation for Datalog semantics

Abstract
This paper presents Paralocks, a language for building expressive
but statically verifiable fine-grained information flow policies. Paralocks
combine the expressive power of Flow Locks (Broberg & Sands, ESOP’06)
with the ability to express policies involving run-time
principles, roles (in the style of role-based access control),
and relations such as “acts-for” in discretionary access control.
We illustrate the Paralocks policy language by giving a simple en-
coding of Myers and Liskov’s Decentralized Label Model (DLM).
Furthermore – and unlike the DLM – we provide an information
flow semantics for full Paralocks policies. Lastly we illustrate how
Paralocks can be statically verified by providing a simple program-
ning language incorporating Paralock policy specifications, and a
static type system which soundly enforces information flow se-
curity according to the Paralock semantics.

Categories and Subject Descriptors D.3 [PROGRAMMING LANG-
UAGES]: F.3.1 [LOGICS AND MEANINGS OF PROGRAMS]: Specifying and Verifying and Reasoning about Programs

General Terms Security, Languages, Verification

1. Introduction
Issues of software security can be crudely categorized into three
broad domains:

- **Access control** deals with security at the end points of a system,
to verify that an entity is allowed to access the system, and to
what extent.
Policy Comparison

- In Datalog known as the **containment problem**
- Given a schema DS, $P$ is contained in $Q$ iff $P(DS, EDB) \subseteq Q(DS, EDB)$ for all $EDB$.
- Shown to be **undecidable** (see Shmueli 1987)
Policy Comparison

your_data := my_salary

- In Datalog known as the containment problem
- Given a schema DS, P is contained in Q iff

  \[ \text{for all } EDB \quad . \quad P(DS, EDB) \subseteq Q(DS, EDB) \]

- Shown to be undecidable
  (see Shmueli 1987)
Policy Comparison

- In Datalog known as the containment problem
- Given a schema DS, $P$ is contained in $Q$ iff

\[
\text{for all } EDB . \quad P(DS, EDB) \subseteq Q(DS, EDB)
\]

Lock state $\sim$ Extensional Database
Policy Comparison

- In Datalog known as the \textit{uniform containment problem}
- Given a schema DS, P is contained in Q iff

\[ \forall DB . \quad P(DS, DB) \subseteq Q(DS, DB) \]

Lock state \sim \textit{Extensional Database}
Uniform Containment

- Shown to be *decidable* (see Sagiv 1987)
- Intended as *approximation* to real containment
- Use existing Datalog algorithms instead of own
Extensions

- .bmp
- .txt
- .exe
- .v
- .so
- .cdata
- .dll
- .flv
- .rc
- .odp
- .xcf
- .flac
Negation

- Adding negation to policies, we can say e.g. (Brewer and Nash, '89)
Negation

- Adding negation to policies, we can say e.g.
  \[
  \{ \ 'x : \text{AppleEmpl}(x), \sim \text{MSEmpl}(x) \} \\
  \{ \ 'x : \sim \text{AppleEmpl}(x), \text{MSEmpl}(x) \} \\
  \]

(Brewer and Nash, '89)
Negation

- Poses limitations on lock properties (safe negation and stratification)

- Larger problem with paragon implementation:

  ```
  Lock State
  .. some facts ..
  call_Method();
  Lock State
  ?
  ```
Negation

- Before: Keep a *lower bound* on the open locks

- “This method *definitely opens* lock L and *might close* lock K”
Negation

- Now: Keep a lower and an upper bound on the open locks

- “This method
  definitely opens lock L,
  might close lock K,
  definitely closes lock M and
  might open lock N”

- Complicated signatures

- Two bounds → very (too?) conservative compiler
Negation

- For each lock L we create a lock nL

  \[
  \text{open } L; \quad \text{open } L; \quad \text{open } L; \\
  \text{close } nL; \quad \text{close } nL;
  \]

- For each actor a need to open nL locks
- Therefore only unary locks
Constraints

Two different managers have to allow flow X

(Li and Mitchell, '03)
Constraints

\{ 'x : may\_Read('x,'y) ,
    may\_Read('x,'z) ,
    'y != 'z \}
Constraints

- Effect on policy ordering check
  How to guarantee ordering holds for all possible databases?

- Solution from CQC – maybe not relevant if solution/ordering algorithm is not described?
Conclusions