Lobster: A Domain Specific Language for SELinux Policies

Joe Hurd, Magnus Carlsson, Brett Letner and Peter White
Galois, Inc.

{joe,magnus,bletner,peter}@galois.com

16 December 2008

Abstract

This paper defines the syntax and semantics of the Lobster domain specific language for describing SELinux security policies.

1 Introduction

A security policy designer might view an SELinux [1, 2] system in terms of information flows between security domains. For example, the application in Figure 1 receives information from a web server and sends information to a mail server: the arrows in the diagram indicate the dominant flow of information.

Information flow can also represent privileges or capabilities, in the following way: if subject A has the privilege to act on object B, then this can be represented by an information flow from A to B. The dominant flow of a write privilege would be from A to B, and conversely the dominant flow of a read privilege would be from B to A. For another example, the `init` process in Figure 1 has the privilege to start up the application, and this is represented by an information flow between the two security domains.

Representing privilege with information flow is a different way of viewing a system, and this shift of viewpoint emphasizes different aspects of the system. Viewing a system in terms of privileges makes it important to distinguish subjects and objects, but from an information flow perspective there

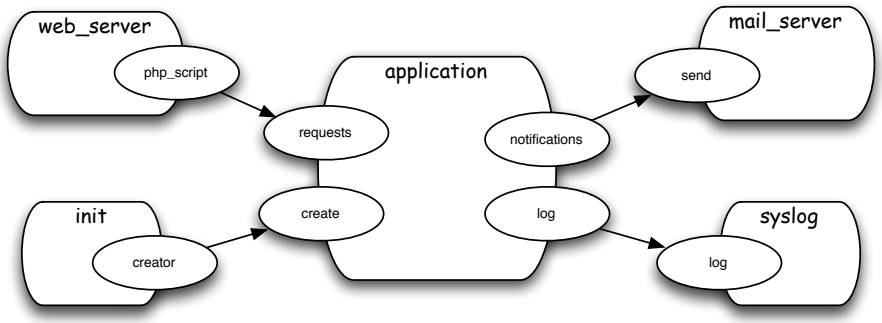


Figure 1: Modelling an application in an SELinux system.

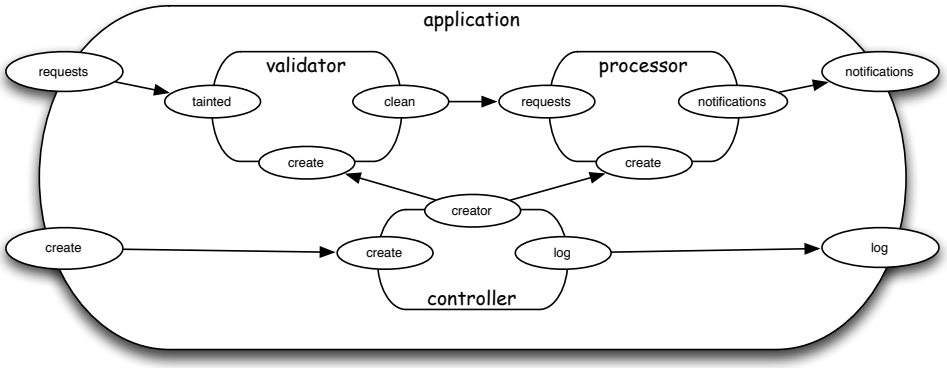


Figure 2: Inside the SELinux application.

is no difference between A having the power to write to B and B having the power to read from A.[1] Emphasizing flow makes it easier to see how information flows through a system. Returning again to Figure 1, the policy representation makes it easy to see that there is a flow of information from the web server to the mail server moderated by the application.

SELinux applications are not restricted to run within a single security domain: separate process and resources can exist in their own security domains, and the SELinux system is used to ensure that all interaction between

[1]In fact, the distinction between subjects and objects can be encoded in an information flow view, and in Section 4.1 such an encoding is used to compile an information flow policy to SELinux.

```
allow application_t lib_t:dir { getattr search };
allow application_t ld_so_t:file { getattr read execute ioctl };
allow application_t usr_t:dir { getattr search read lock ioctl };
allow application_t syslogd_t:unix_dgram_socket sendto;
allow application_t syslogd_t:unix_stream_socket connectto;
```

Figure 3: Example SELinux native policy statements.

them are consistent with the security policy in force. Figure 2 shows how the example application might be internally broken down into separate security domains, with explicit information flows permitted between them.

Continuing with the example, suppose that a developer implements the application and also writes an SELinux policy for it. Some example policy statements are shown in Figure 3. How is it possible to check that the security policy designer's permitted information flows and the developer's SELinux policy statements are consistent?

One approach to solving this problem is to express the information flows in a high level language, and to compile it down to SELinux policies. This is the subject of this paper: Lobster, a domain specific language for expressing information flows between security domains.

In general, the SELinux policy written by the developer will contain much more detail than the information flow view of the security policy designer. This is the intended use of the Lobster DSL:

1. A security policy designer writes a Lobster information flow diagram for the application.

2. A developer writes a Lobster policy for the application.

3. An automatic tool verifies that the Lobster policy (2) is a refinement of the Lobster information flow diagram (1), in that *no extra information flows have been introduced*.

4. A compiler takes the Lobster policy (2) and generates SELinux policy statements.

The remainder of this paper defines the syntax and semantics of the Lobster policy language. Section 2 describes the structure of information flow graphs; Section 3 defines the Lobster language; Section 4 explains how

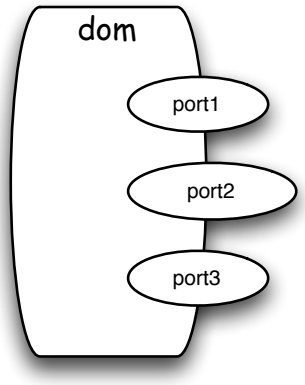


Figure 4: A Lobster domain with named ports.

Lobster policies are compiled to SELinux native policy statements; Section 5 looks briefly at the Lobster compiler tool; Section 6 summarizes and compares with related work; and finally Section 7 considers future directions.

2 Information Flow Graphs

Figures 1 and 2 from the Introduction presented information flows between nested security domains in a graphical form. Such diagrams are called *information flow graphs*, and these graphs are the semantic foundation of Lobster policies. A Lobster policy is compiled to an information flow graph, as an intermediate step on the way to a native SELinux policy.

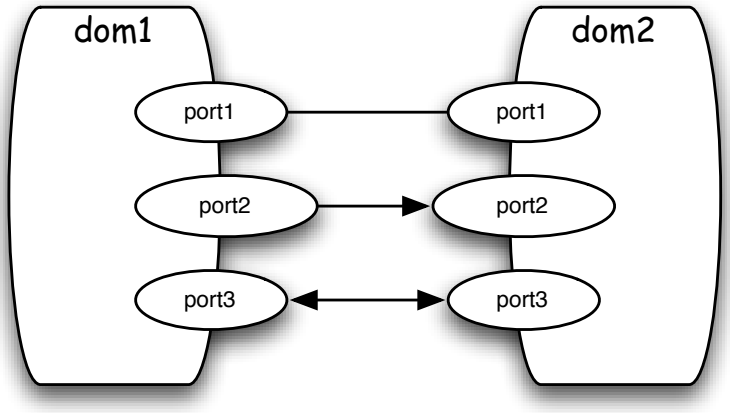
2.1 Domains

Figure 4 shows a Lobster domain with named ports. A domain is a security domain, and the inside can only communicate with the outside via its ports.

2.2 Ports

Ports can be declared with information flow properties:

```
port p1 : {direction = input, type = requests};
port p2 : {direction = bidirectional};
```




```
dom1.port1 -- dom2.port1;
dom1.port2 --> dom2.port2;
dom1.port3 <--> dom2.port3;
```

Figure 5: Example Lobster connections.

All information flow properties are optional: an unspecified property is deemed to be polymorphic in its value. When specified, the `direction` property is either `input`, `output` or `bidirectional`, and restricts the kind of connections that can be made to the port. The `type` value captures the type of information handled by the port—it has nothing to do with SELinux types.

We will refer to the *opposite* of a direction property: the opposite of an `input` direction is an `output` direction; the opposite of an `output` direction is an `input` direction; and the opposite of a `bidirectional` direction is a `bidirectional` direction.

Although `direction` and `type` are the only guaranteed information flow properties, the Lobster language is designed to be extensible in this regard. For example, to compile Lobster policies to SELinux native policy we will define and use a `position` information flow property that encodes the subject/object relationship that is central to SELinux type enforcement.

2.3 Connections

Two domain ports with compatible information flow properties may be connected. Two sets of information flow properties are compatible if:

- they have opposite `direction` properties;
- and they have the same `type` properties.

Recall that unspecified properties are regarded as polymorphic, and are able to take any value to be compatible.

Figure 5 shows three example connections between domain ports. The different kinds of connection direction symbols add requirements to the information flow directions of the ports:

- `dom1.port1 -- dom2.port1;` adds no requirements;
- `dom1.port2 --> dom2.port2;` requires that `dom1.port2` have an `output` direction property, and `dom2.port2` have an `input` direction property.
- `dom1.port3 <--> dom2.port3;` requires that the `dom1.port3` have a `bidirectional` direction property, and `dom2.port3` have an `bidirectional` direction property.

As we will see in Section 4, Lobster connections are compiled into SELinux `type` and `allow` statements.

2.4 Nested Domains

One domain may be nested inside another; connections are permitted between ports in the containing domain and ports in the nested domain. Figure 6 shows an example of connections between nested domains: note that `port1` is indirectly connected via `port2` to both `port3a` and `port3b`.

A domain corresponds to an SELinux type, where the name of the type is determined by the name of the domain and the names of the containing domains. For example, the SELinux type corresponding to the domain in Figure 4 is simply `dom_t`, and the name of the *controller* domain in Figure 2 is `application_controller_t`.

A step in the compilation of information flow graphs to SELinux policies involves flattening the graph by eliminating domains that contain other domains. A containing domain d may be eliminated by following these steps:

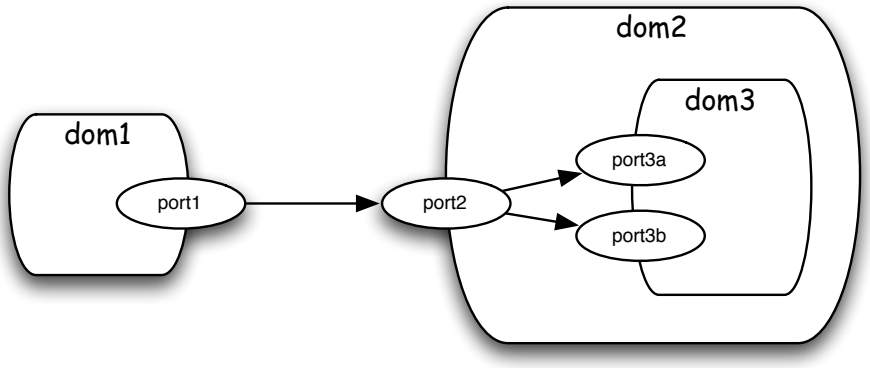


Figure 6: Lobster domains can be nested.

1. Prepend the name of d to each domain directly contained in d.

2. For each port p in the domain d, for each external connection c' from a port $d'.p'$ to p, for each internal connection c'' from p to a port $d''.p''$, join c' and c'' to create a direct connection from $d'.p'$ to $d''.p''$.

3. Finally, eliminate the domain d, all of its ports, and all connections to its ports.

2.5 Checking Connection Consistency

Section 2.3 listed the compatibility requirements on the information flow properties of ports to be able to make a connection between them. The merged information flow properties of the ports can be thought of as the information flow properties of the connection. If connections are declared with a direction symbol, then this also becomes part of their information flow properties.

Section 2.4 introduced nested domains, and required that connections be joined in Step 2 of the procedure to eliminate containing domains. When two connections are joined, their information flow properties are merged to form the information flow properties of the new connection. Note that this step might fail because the two sets of information flow properties might not be compatible.

An information flow graph is said to have *connection consistency* if every connection is between two ports with compatible information flow properties, and in addition all nested domains can be eliminated with no compatibility failures.

It is possible to statically check whether an information flow graph has connection consistency, without eliminating all the containing domains. The following algorithm performs this check for a domain d:

1. For each domain d' directly contained within d, recursively call this algorithm (which may alter the information flow properties of ports of d').

2. For all connections between domains directly contained within d, check that the information flow properties of the ports are compatible.

3. For each connection between a port of d and a domain directly contained within d, check that the information flow properties are compatible.

4. For each port p of d, take all the connections between p and ports of domains directly contained within d, and merge together all their information flow properties. Merge these with the information flow properties of d. If any information flow properties conflict, do *not* generate an error, but rather record the conflict. This conflict value will not necessarily break connection consistency, but can only merge with a polymorphic value (which results in another conflict value).

One feature of this algorithm for checking connection consistency is that all of the connections inside a domain can be summarized by information flow properties of its ports, by propagating properties from the subdomains. The ports and their properties are the interface of the domain.

The algorithm for checking connection consistency is illustrated on the example information flow graph in Figure 7. When the check is complete, the consistency requirements imposed by the connections within the `filesystem` domain has had an effect on the information flow properties of its port. This result is shown in Figure 8. Both the internal connections of the `filesystem.write` port had dominant flow from the port to ports of nested domains, so its direction property was set to input. The type property is more interesting. Because the two types in the `file1.write` and `file2.write`

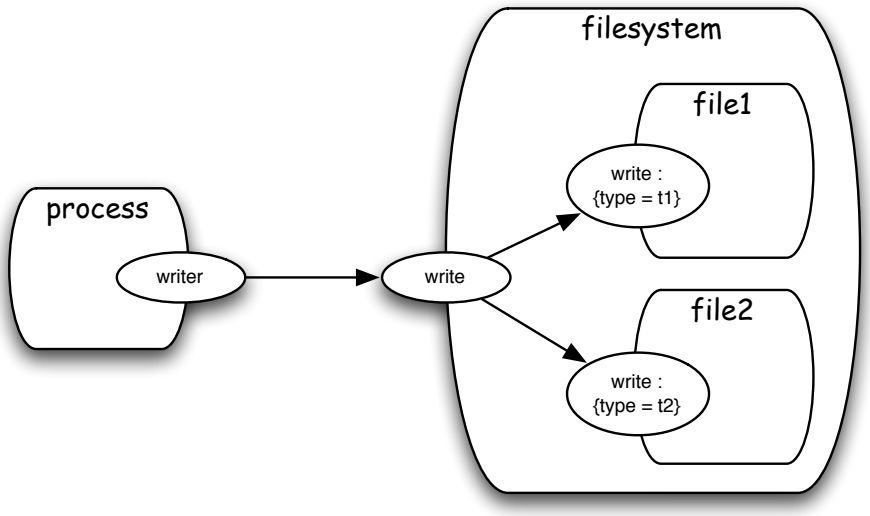


Figure 7: The information flow graph to be checked for connection consistency.

ports are different, the `filesystem.write` port records a conflict in the type information flow property. However, this conflict is consistent with the polymorphic type of `process.writer` port, so the consistency check passes.

Finally, Figure 9 shows the information flow graph where the `file1` and `file2` domains nested within the `filesystem` domains have been omitted, but the information that they contribute to the interface of the `filesystem` domain remains in the form of information flow properties on its ports. Any connections that are consistent with the annotated interface of the `filesystem` domain will be consistent with the implementation using the nested `file1` and `file2` domains.

3 The Lobster Language

In an information flow graph modelling a realistic SELinux system, there is likely to be common patterns of nested domains connected together, and it would be tedious to explicitly specify this repetition in the policy language. This motivates the design of Lobster, where a policy is represented as a program that, when executed, generates the information flow graph. As we

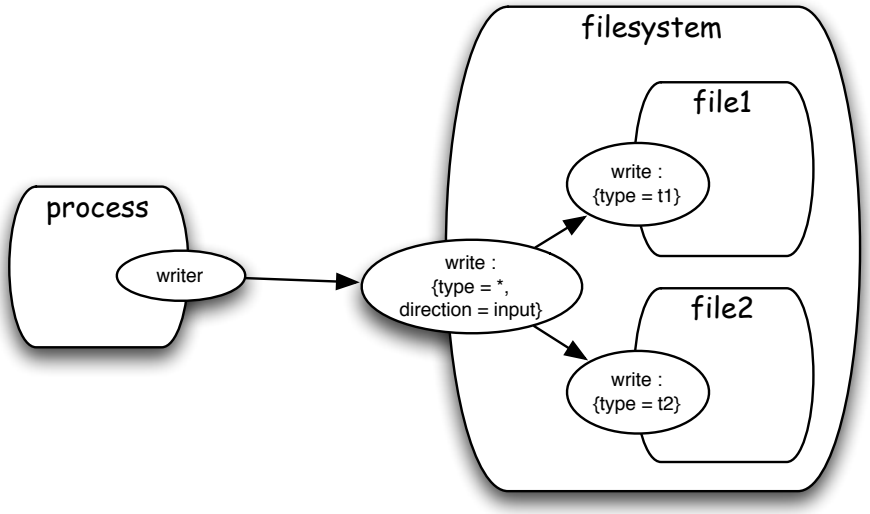


Figure 8: The result of checking connection consistency.

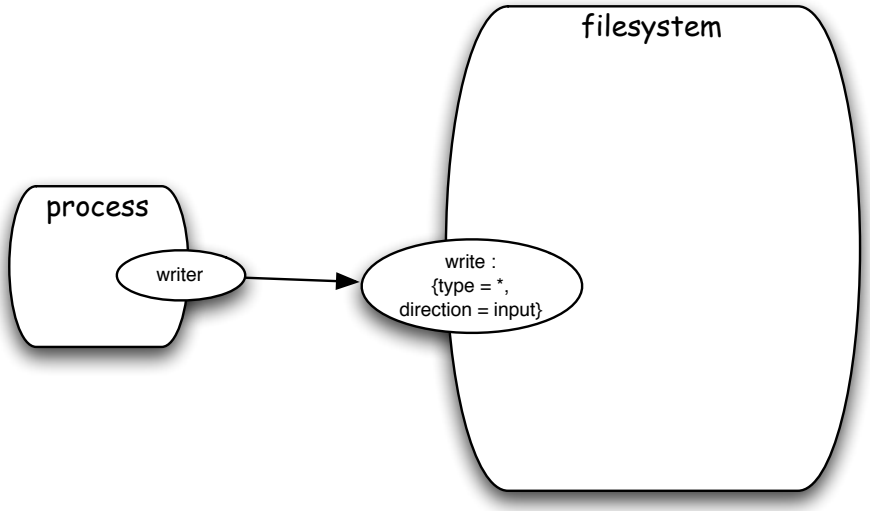


Figure 9: Deriving an interface from the results of checking connection consistency.

will see in the following subsections, Lobster policies are written using an object oriented syntax: classes are instantiated to domains; and connections are like method calls.

3.1 Classes

Here is a simple Lobster class with one type and one port:

```
class C() {
  type t;
  port p : {type = t};
}
```

The `domain` statement demonstrates how to instantiate the class C into a new domain:

```
domain c = C();
```

When a class is instantiated, all the statements inside the class definition are executed, like a class constructor function in an object oriented programming language. If any of these statements create new domains, they become nested domains.

Classes can take arguments, which are passed in when instantiating the class.

```
class D(t) { port p : {type = t}; }
domain d = D(requests);
```

3.2 Lobster Programs

Lobster programs are a sequence of class definitions and top level `domain` and connection statements. An example Lobster program can be found in Section 5. The full details of the lexical structure are described in Appendix A, and the full details of the syntactic structure are described in Appendix B.

3.3 Building the Information Flow Graph

The following algorithm is used to interpret Lobster programs and build the information flow graph:

1. The top level class definitions are read, and the statements within them stored.

2. The top level `domain` statements are used to instantiate some classes and create the top level domains. In this step the statements in the class definition are interpreted to build up nested structure within the domain.

3. Finally, the top level connection statements are interpreted to connect up the top level domains.

4 Compiling to SELinux Native Policies

This is how a Lobster policy is compiled to SELinux:

1. Interpret the Lobster program to generate the information flow graph.

2. Check that the information flow graph satisfies connection consistency.

3. Flatten the information flow graph by eliminating domains that contain other domains nested within them. This step results in a flat information flow graph consisting only of primitive domains and connections between their ports.

4. Finally, generate the SELinux type enforcements and file contexts.

4.1 Primitive SELinux Classes

For every SELinux class declared in the reference policy, there must be a Lobster class of the same name. The SELinux permissions are its ports. For example, here is an excerpt of the Lobster class corresponding to the SELinux `file` class:

```
class File(regexp) {
  port read : {direction = output};
  port write : {direction = input};
  ...
}
```

In addition to the ports resulting from their permissions, *active* classes must have a special `active` port with a `position` information flow property indicating its subject status:

```
class Process {
  port active : {position = subject}
  ...
}
```

These classes would be part of a Lobster version of the SELinux policy in force, allowing Lobster application policies to be checked in the right context.

4.2 Generating SELinux Native Policy

The connections between primitive domains define the SELinux type enforcement rules. For example, consider the following Lobster program:

```
domain p = Process();
domain f = File("/tmp/file");
p.active -- f.read;
```

The connection would generate the SELinux type enforcement:

```
allow p_t f_t : file read;
```

It is the special `subject` information flow property of the `active` port of the `Process` class that tells the policy generator that the `p_t` type should be in the subject position and the `f_t` type should be in the object position of the `allow` rule. Exactly one of the connected ports should have the `subject` information flow property.

In addition, the special regexp argument in the instantiation of the File class would generate an SELinux file context:

```
/tmp/file  --  gen_context(system_u:domain_r:f_t,s0)
```

5 The Lobster Tool

There is a command `lobster` that compiles a lobster policy to the SELinux native policy language. Its usage is simple:

```
lobster [-I include.lsr] input.lsr ...
  -I include.lsr  --include=include.lsr  Include a lobster file
```

The lobster command takes as input a collection of lobster policy files and compiles them into SELinux type enforcement and file context native policy statements. Some of the input files can be tagged as *include* files using the `-I` option. In the current version of the lobster compiler these files are treated exactly the same as the other input files—the option is present to support a future version of the lobster compiler which outputs an SELinux reference policy module. [*It is intended that include files will be used for checking the consistency of the input lobster policy, but will not contribute to the resulting SELinux policy module.*]

For example, consider the simple Lobster policy file `example1.lsr` containing the following class definitions and `domain` and connection statements:

```
class Process() {
  port active : {position = subject};
}

class File(filenameRegex) {
  port read : {direction = output, position = object};
  port write : {direction = input, position = object};
}

class ExampleApp(dataFilenameRegexp) {
  domain app = Process();
  domain data = File(dataFilenameRegexp);
  app.active <-- data.read;
  app.active --> data.write;
}

domain example = ExampleApp("/tmp/example.*");
```

Running the command

```
lobster example1.lsr
```

results in the creation of the `example1.te` output file containing the type enforcement statements

```
policy_module(example1,1.0)
type example_app_t;
type example_data_t;
allow example_app_t example_data_t:file read;
allow example_app_t example_data_t:file write;
```

and the `example1.fc` output file containing the file contexts

```
example_data_t "/tmp/example.*"
```

[*Again, the prototype nature of the tool is apparent here: the file context statements do not have the correct syntax.*]

6 Summary

This paper has introduced the Lobster Domain Specific Language for information flow security policies. The key contribution is a way of refining information flow policies to SELinux native policies.

6.1 Related Work

The Tresys tool CDSFramework tool also uses hierarchical domains to design SELinux policies. There are minor differences, such as the distinction between resources and domains, and unlimited access within domains. The main difference is CDSFramework's focus on providing a higher level of abstraction on top of the reference policy. This provides an easier path to compiling down to working policies, but complicates the analysis of the higher level policy language.

7 Future Work

7.1 Targeting the SELinux Reference Policy

The next immediate step for Lobster is to target the SELinux reference policy language, rather than native SELinux policies. This would result in Lobster policies being compiled to policy modules, and critically to be able to refer to existing SELinux policy modules. However, the current focus remains the type enforcement aspect of SELinux policies, leaving the role-based access control and multi-level security aspects for future work.

The existing Lobster `domain` statements can set up arbitrary internal information flow structure, and are expressive enough to model SELinux reference policy `template` procedures. The existing Lobster connection statements are expressive enough to handle a single SELinux `allow` rule, but

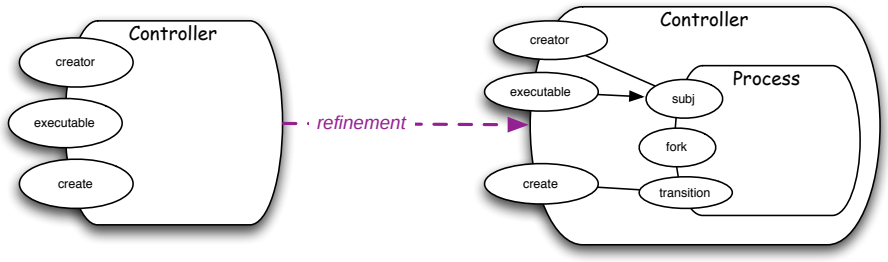


Figure 10: An information flow refinement.

SELinux reference policy `interface` procedures can declare an arbitrary number of `allow` rules, so Lobster needs to be extended to handle these.

One idea is to allow domains to have method functions which correspond to `interface` procedures. Method functions would not be allowed to contain `domain` statements, since these correspond to creating new SELinux types, but could contain connection statements and calls to other method functions. Thus, a single call to a method function would be able to set up multiple connections, and the type enforcement aspect of `interface` procedures could be modelled with method functions.

7.2 Refining Lobster Policies

Another unexplored area is to pin down exactly what it means for one Lobster policy to be an *information flow refinement* of another, as illustrated by Figure 10. When this is done, an automatic tool can be written to check that the Developer's Lobster policy is a refinement of the Security Policy Designer's Lobster specification.

7.3 Information Flow Assertions

For large policies, it is useful to capture global constraints on information flow. To this end, Lobster can be extended with information flow assertions, such as *"all information flow between domains of class A and B must go through domains of class C"*, and analysis tools can be written to statically check the assertions.

7.4 High Level Verification

A Lobster policy with information flow assertions could generate information flow specifications for application programs, as well as an SELinux policy. The system information flow properties would rely on the SELinux enforcement mechanism, but also that the programs satisfied their information flow specification. A well-designed system would have the property that only a few key programs were trusted in this way, and that most application programs could safely transfer any of their inputs into any of their output without breaking any of the system information flow properties.

Also, it would be interesting to add trust annotations to domains and connections, in support of analyzing trust relationships in systems.

References

[1] Frank Mayer, Karl MacMillan, and David Caplan. *SELinux by Example*. Open Source Software Development Series. Prentice Hall, 2007.

[2] Bill McCarty. *SELinux: NSA's Open Source Security Enhanced Linux*. O'Reilly, 2005.

A The Lexical Structure of Lobster

A.1 Literals

Integer literals ⟨*Int*⟩ are nonempty sequences of digits.

String literals ⟨*String*⟩ have the form "x", where x is any sequence of any characters except " unless preceded by \.

LIdent literals are recognized by the regular expression ⟨*lower*⟩(⟨*letter*⟩ | ⟨*digit*⟩ | '_')∗

UIdent literals are recognized by the regular expression ⟨*upper*⟩(⟨*letter*⟩ | ⟨*digit*⟩ | '_')∗

A.2 Reserved words and symbols

The set of reserved words is the set of terminals appearing in the grammar. Those reserved words that consist of non-letter characters are called symbols, and they are treated in a different way from those that are similar to

identifiers. The lexer follows rules familiar from languages like Haskell, C, and Java, including longest match and spacing conventions.

The reserved words used in Lobster are the following:

```
bidirectional  class  direction
domain         input  object
output         port   position
subject        type
```

The symbols used in Lobster are the following:

```
(       )      {
}       ;      =
:       .      *
<-->    -->    <--
--      ,
```

A.3 Comments

Single-line comments begin with `//`.
Multiple-line comments are enclosed with `/*` and `*/`.

B The Syntactic Structure of Lobster

Non-terminals are enclosed between ⟨ and ⟩. The symbols ::= (production), | (union) and ϵ (empty rule) belong to the BNF notation. All other symbols are terminals.

⟨*Policy*⟩ ::= ⟨*ListStatement*⟩

⟨*Statement*⟩ ::= `class` ⟨*ClassId*⟩ `(` ⟨*ListIdentifier*⟩ `)` `{` ⟨*ListStatement*⟩ `}`
| `type` ⟨*TypeId*⟩ `;`
| `port` ⟨*PortId*⟩ ⟨*PortDeclarationType*⟩ ⟨*PortDeclarationConnection*⟩ `;`
| `domain` ⟨*Identifier*⟩ `=` ⟨*ClassId*⟩ `(` ⟨*ListExpression*⟩ `)` `;`
| ⟨*Identifier*⟩ `=` ⟨*Expression*⟩ `;`
| ⟨*ListExpression*⟩ ⟨*Connection*⟩ ⟨*ListExpression*⟩ `;`

⟨*PortDeclarationType*⟩ ::= ϵ
| `:` `{` ⟨*ListPortTypeConstraint*⟩ `}`

⟨*PortDeclarationConnection*⟩ ::= ϵ
| ⟨*Connection*⟩ ⟨*ListExpression*⟩

⟨*Expression*⟩ ::= ⟨*QualName*⟩
| ⟨*Integer*⟩
| ⟨*String*⟩
| `(` ⟨*Expression*⟩ `)`

⟨*QualName*⟩ ::= ⟨*Name*⟩
| ⟨*Name*⟩ `.` ⟨*Name*⟩

⟨*Name*⟩ ::= ⟨*TypeId*⟩
| ⟨*Identifier*⟩

⟨*PortTypeConstraint*⟩ ::= `type` = ⟨*NoneExpression*⟩
| `input` = ⟨*NoneExpression*⟩
| `output` = ⟨*NoneExpression*⟩
| `position` = ⟨*NonePosition*⟩
| `direction` = ⟨*NoneDirection*⟩

⟨*NoneExpression*⟩ ::= `*`
| ⟨*Expression*⟩

⟨*NonePosition*⟩ ::= `*`
| ⟨*Position*⟩

⟨*Position*⟩ ::= `subject`
| `object`

⟨*NoneDirection*⟩ ::= `*`
| ⟨*Direction*⟩

⟨*Direction*⟩ ::= `bidirectional`
| `input`
| `output`

⟨*Connection*⟩ ::= <−−>
| −−>
| <−−
| −−

⟨*PortId*⟩ ::= ⟨*LIdent*⟩

⟨*Identifier*⟩ ::= ⟨*LIdent*⟩

⟨*TypeId*⟩ ::= ⟨*UIdent*⟩

⟨*ClassId*⟩ ::= ⟨*UIdent*⟩

⟨*ListIdentifier*⟩ ::= ϵ
| ⟨*Identifier*⟩
| ⟨*Identifier*⟩ , ⟨*ListIdentifier*⟩

⟨*ListExpression*⟩ ::= ϵ
| ⟨*Expression*⟩
| ⟨*Expression*⟩ , ⟨*ListExpression*⟩

⟨*ListStatement*⟩ ::= ϵ
| ⟨*Statement*⟩ ⟨*ListStatement*⟩

⟨*ListPortTypeConstraint*⟩ ::= ϵ
| ⟨*PortTypeConstraint*⟩
| ⟨*PortTypeConstraint*⟩ , ⟨*ListPortTypeConstraint*⟩