Security Policy DSL - Shrimp

Magnus Carlsson, Joe Hurd, Peter White

Galois

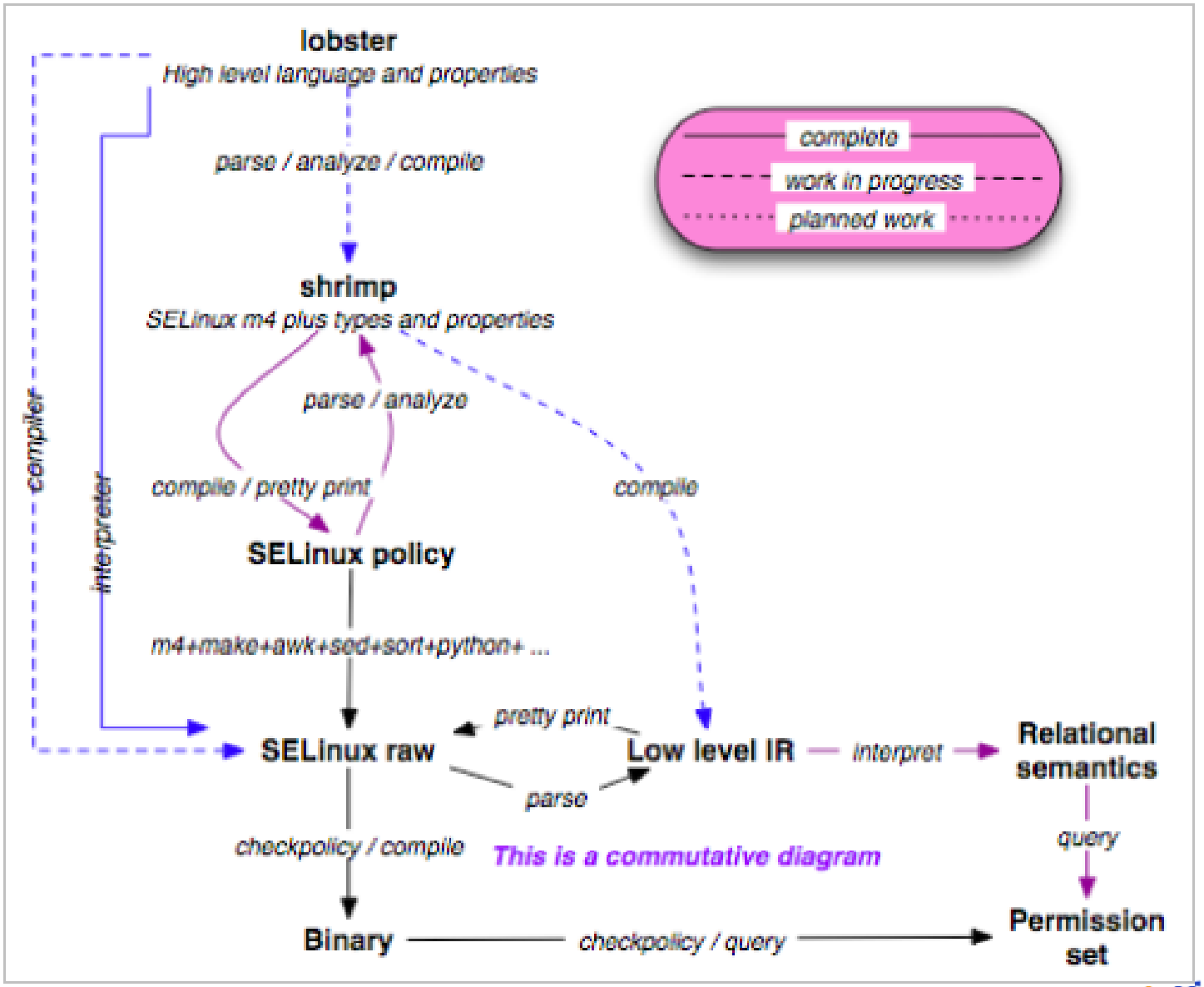
2008-03-04

|galois|

Outline

Shrimp and SCD
Background
Purposes and possibilities of Shrimp
Shrimp Anatomy
A kind system for Shrimp
Kind-inference results as HTML
“Lint” results from kind analysis
Next steps

Shrimp and SCD



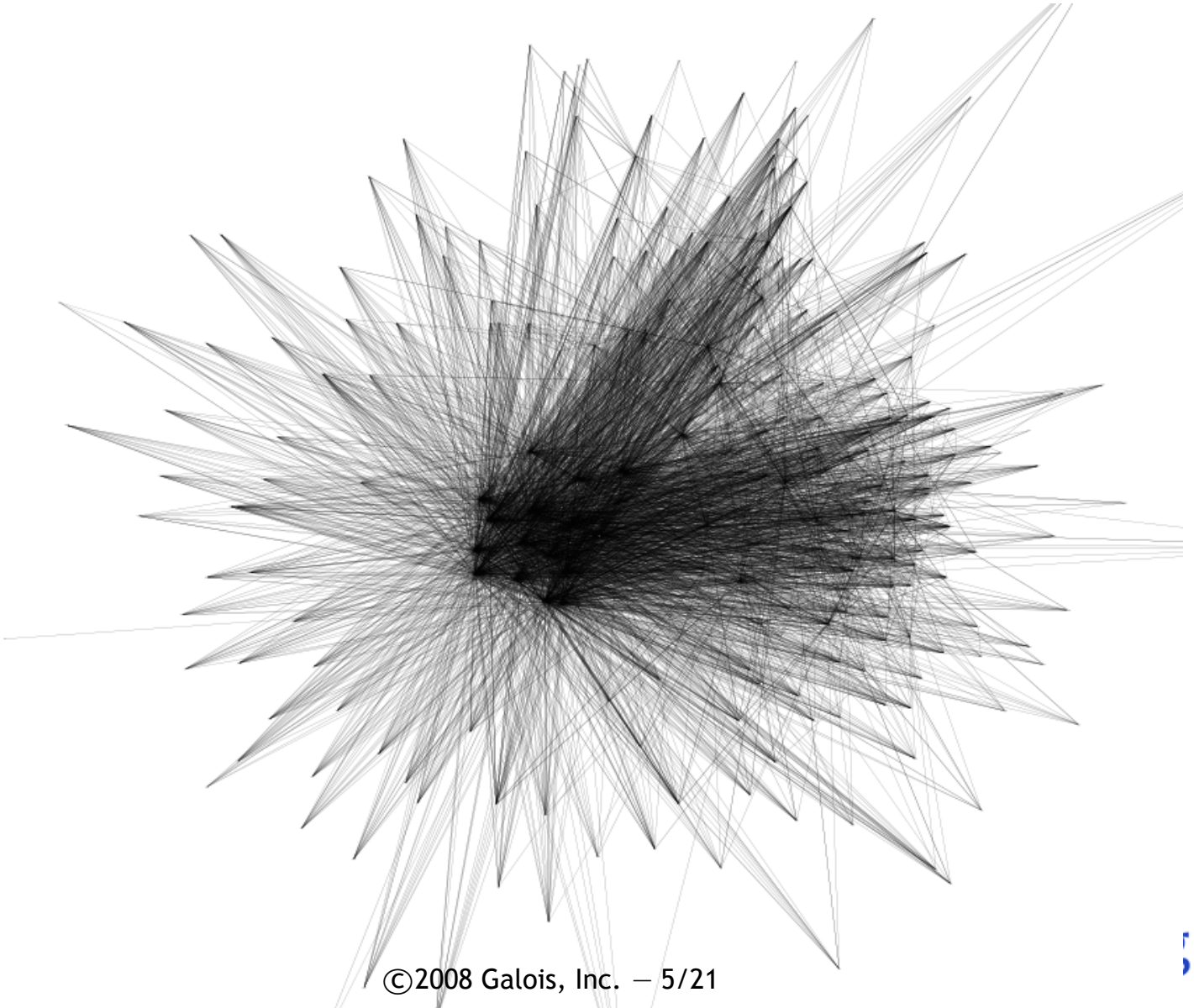
galois

Background

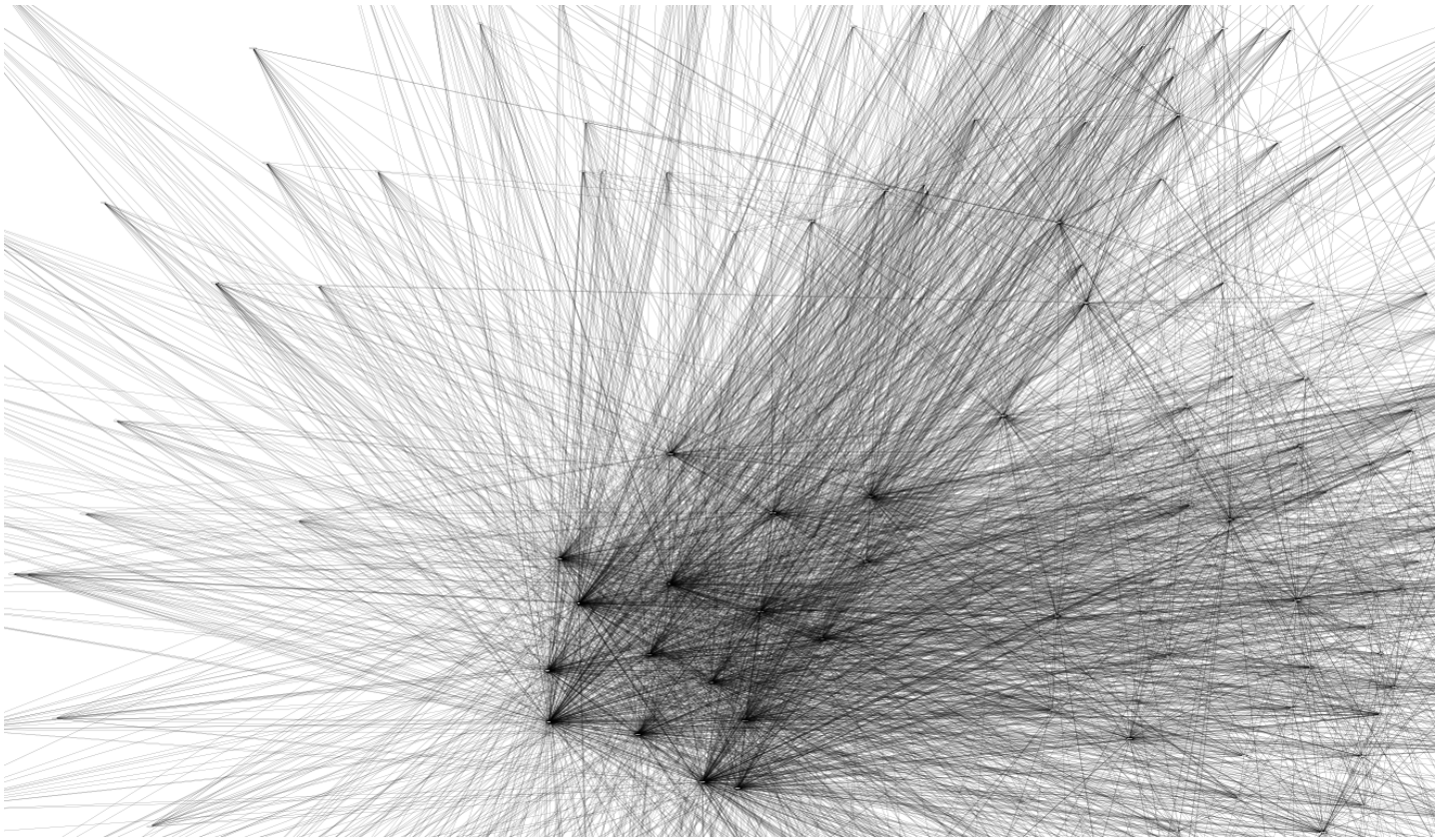
Reference Policy is a great improvement over Native Policy in terms of modularity, size, reuse and level of abstraction. It is a success in applying software-engineering principles to get an improved configuration language. However:

- Due to use of M4 macro preprocessor in the current prototype tools, it is difficult to analyze.
- It is still large and complex (more than 100 Kloc, 250 modules).

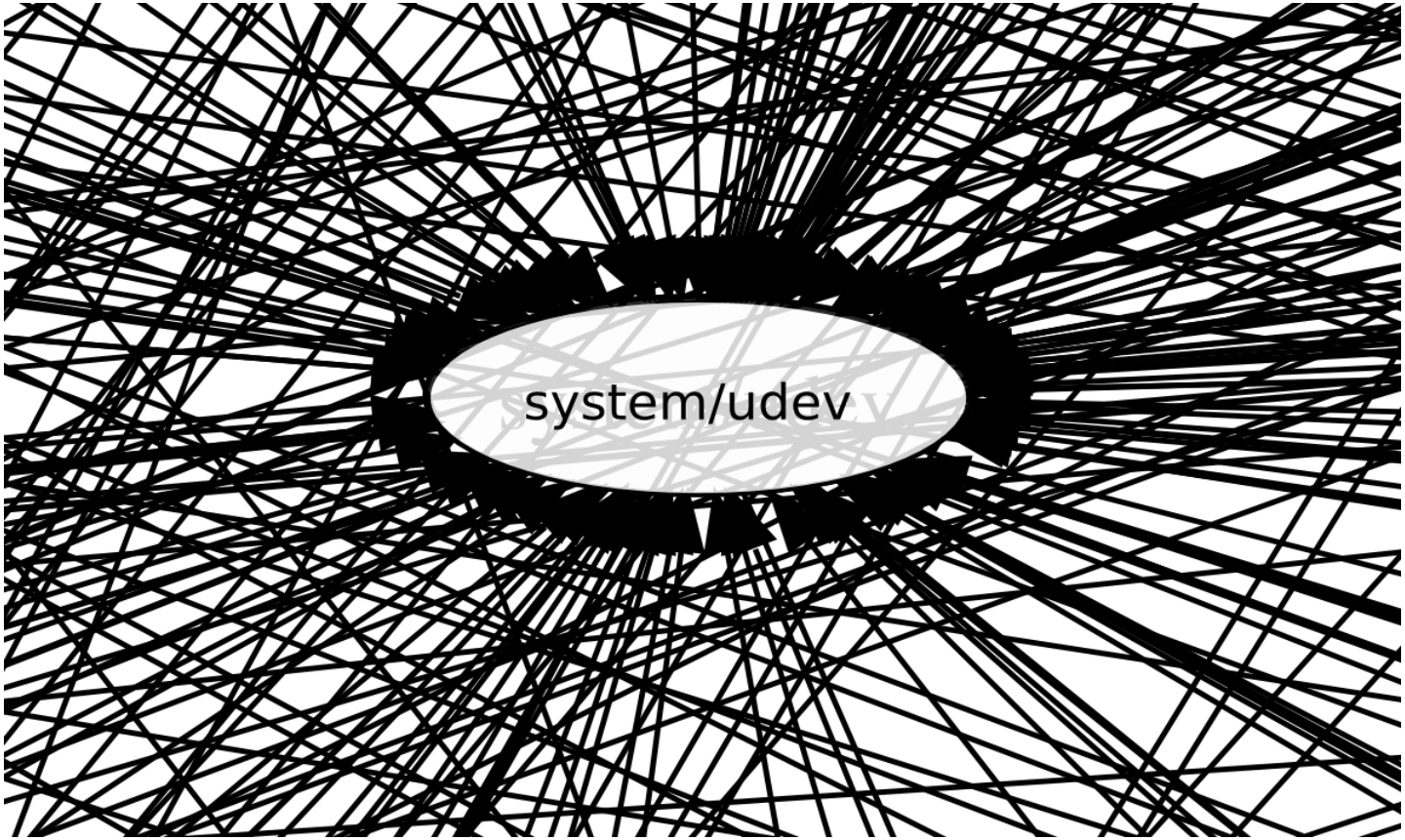
Module interdependencies of Reference Policy



Module interdependencies of Reference Policy



Module interdependencies of Reference Policy



Purposes and possibilities of Shrimp

- Support for analysis of Reference Policy on its own level - not in terms of Native Policy.
- “lint” tool for Reference Policy.
- HTML generation of documentation + analysis results for Reference Policy.
- Prototyping workbench for a new Reference Policy language “Shrimp”.
- Target for Lobster compilation (future).
- Conversion tool from Reference Policy to Shrimp (future).

Shrimp Anatomy

Shrimp can be seen as an extension to Reference Policy:

- *kind information* for interface parameters.
- *multiple-environment information* about interface bodies.
- Local and global *information-flow properties*.

A kind system for Shrimp

What follows is really a type system, but one of the types in Reference Policy is denoted `type`. To avoid confusion, we talk about the *kinds* of symbols instead. So a kind can be for example be a role, a type, an attribute, a class, or a permission.

A kind system for Shrimp

Statement judgments for Reference Policy statements are on the form: $\Gamma \vdash s :: R; O$, which reads "Given a symbol environment Γ, statement s demands that the symbols R are provided by the policy, and puts the symbols in O into the policy.

Example: $\Gamma \vdash \texttt{type}\ t :: \emptyset; t : \text{type}$, "`type` t puts the type t into the policy."

Composition of statements: the R and O demands enrich the symbol environment for later statements:

$$\frac{\Gamma \vdash s_1 :: R_1; O_1 \qquad \Gamma, R_1, O_1 \vdash s_2 :: R_2; O_2 \qquad O_1 \text{ and } O_2 \text{ disjoint}}{\Gamma \vdash s_1 ; s_2 :: R_1, R_2; O_1, O_2}$$

Requirement judgments

`require`-statements put demands on the policy environment through R, by means of *requirement judgments* $\vdash_r$:

$$\frac{R \vdash_r rs}{\Gamma \vdash \texttt{require}\ \{\ rs\ \} :: R; \emptyset}$$

Requirement statements are decomposed by $\dfrac{R_1 \vdash_r r_1 \qquad R_2 \vdash_r r_2}{R_1, R_2 \vdash_r r_1 ;\ \ r_2}$

Requirement axioms look like $\dfrac{}{r : \text{role} \vdash_r \texttt{role}\ r}$, which reads

"`role` r requires that the role r is provided by the policy."

The symbol environment

The symbol environment Γ is local to macro definitions and implementation modules, and it is consulted in e.g. access-rule statements:

$$\frac{}{\Gamma, s : \text{domain}, t : \text{type/attribute}, c : \text{class}, p : \text{permission} \vdash \texttt{allow}\ s\ t{:}c\ p :: \emptyset; \emptyset}$$

"If the symbol environment has that s is a domain, t is a type or an attribute, c is a class and p is a permission, then we can infer `allow` $s\ t{:}c\ p$, without any interaction with the policy environments."

Rules for macro definitions

When typing interfaces, we require that no symbols are put into the policy, and that the environment only contains formal parameters. This is captured in a *definition judgment* $\vdash_d$:

$$\frac{\$_1 : k_1, \ldots, \$_n : k_n \vdash ss :: R; \emptyset}{\$_1 : k_1, \ldots, \$_n : k_n \vdash_d \texttt{interface}(i, ss) :: \emptyset}$$

Template definitions are allowed to put symbols into the policy:

$$\frac{\$_1 : k_1, \ldots, \$_n : k_n \vdash ss :: R; O}{\$_1 : k_1, \ldots, \$_n : k_n \vdash_d \texttt{template}(i, ss) :: O}$$

Support definitions are also covered by definition judgments.

Calls to definitions

Macro calls may put symbols in the environment:

$$\frac{\$_1 : k_1, \ldots, \$_n : k_n \vdash_d d(i, ss) :: O}{\Gamma, a_1 : k_1, \ldots, a_n : k_n \vdash i(a_1, \ldots, a_n) :: \emptyset; O[a_1/\$_1, \ldots, a_n/\$_n]}$$

"A call to the macro *i* does not demand that the policy provide any symbols, but it puts the symbols of *i* (properly substituted) into the policy."

Example of definition judgment

$\$_1$: domain_, $\$_2$: domain_, $\$_3$: domain
$\vdash_d$ `template(dbus_user_bus_client_template,`*ss*`)`
:: $\$_2$`_dbusd_`$\$_1$`_t` : type

"`dbus_user_bus_client_template` takes three parameters (two domain prefix parameters and one domain parameter), and puts the derived type $\$_2$`_dbusd_`$\$_1$`_t` into the policy."

Kind-inference results as HTML

template dbus_user_bus_client_template

Template for creating connections to a user DBUS.

index	name	kind	summary
$1	**user_prefix**	domain_	*The prefix of the domain (e.g., user is the prefix for user_t).*
$2	**domain_prefix**	domain_	*The prefix of the domain (e.g., user is the prefix for user_t).*
$3	**domain**	domain	*The type of the domain.*

	identifier	kind	origin
input	$3	domain	
origin	$2_dbusd_$1_t	type	
require	$1_dbusd_t	type	dbus_per_role_template (type) *userdom_restricted_xwindows_user_template (type)*
	dbus	class	
	send_msg	permission	

Extra information reveals inter-dependencies between interfaces.

`dbus_user_bus_client_template` depends on `dbus_per_role_template`.

Benefits of kind-inference system

- Allows us to precisely capture static semantics (scope and kind rules) for Reference Policy.
- Easily tweaked to model different scoping rules.
- Can inform design of implementation of Reference Policy compiler.
- Captures errors in Reference Policy.
- Possibly suggests an object-oriented structure through macro interdependencies. Making this structure more visible and enforcing it helps us avoiding subtle errors due to interaction between macro calls.

“Lint” results from kind analysis

```
Undefined identifiers: [
../Reference-Policy/refpolicy/policy/modules/kernel/kernel.if:1014:32:proc_t,[type/attribute]]
(100 errors like this.)
Mismatch between number of documented vs. referenced parameters:
## <param name="domain" />
## <param name="userdomain_prefix" />
## <param name="domain" />
[$1,[attribute_], $2,[type]]
(29 errors like this.)
Wrong number of arguments:
../Reference-Policy/refpolicy/policy/modules/apps/java.if:210:9:userdom_unpriv_usertype,
  [[attribute_], [type], [any]]
(19 errors like this.)
Call to undefined macro:
../Reference-Policy/refpolicy/policy/modules/system/userdomain.if:202:17:fs_read_nfs_named_sockets
(10 errors like this.)
Duplicate definition of macro:
../Reference-Policy/refpolicy/policy/support/obj_perm_sets.spt:334:9:all_nscd_perms
(5 errors like this.)
Illegal symbol declarations in interface: [
../Reference-Policy/refpolicy/policy/modules/kernel/selinux.if:514:14:$1]
Duplicate definition of (
../Reference-Policy/refpolicy/policy/modules/kernel/corenetwork.te:1533:25:netif_lo_t,type)
```

Types of errors

- 100 references to undefined identifiers. Highly sensitive to chosen scoping rules, fidelity might need to be increased.
- 29 errors are about mismatch between documented and actual parameters in macro definitions.
- A few syntactical errors needed correction, most notably a regular-expression bug. Patch excerpt:

```
-/dev/holter[0=9] -c gen_context(system_u:object_r:tty_device_t,s0)
+/dev/holter[0-9] -c gen_context(system_u:object_r:tty_device_t,s0)
```

  More analysis of file-context definitions has potential of catching subtle errors and inconsistencies.

Next steps

- Collaboration with Tresys for evaluation/improvement of Shrimp and tools.
- Add property constructs for e.g. information flow. Add information-flow inference to tools.
- Adapt for Lobster compilation target.
- Document static semantics: kind system, scope rules.
- Develop relational semantics (c.f. the semantics for Native policy).
- Translate Reference Policy into Shrimp.
- Long term: generalize this type of analysis for other configuration languages (TACLBox).