Policy Language: Theoretical Design Document

October 2018

This document gives a formal definition to a portion of the Draper/Dover Policy Language (DPL). Section 1 gives the complete syntax for this portion and describes its meaning informally. Section 2 gives an operational semantics and provides some background information on design decisions that went into the language.

1 Syntax

Metavariables: C ranges over constructor names, x and y range over variable names, which include opgroups and pattern variables.

1.1 Tags

In DPL, metadata is always a set of *tags*. Tags are primitive units declared by each micropolicy. For example, the RWX policy has `Rd`, `Wr` and `Ex` tags. The metadata associated with a memory word or register is a set of these tags. For example, stack temporary memory typically has the metadata `{Rd,Wr}`, and executable memory typically has the metadata `{Ex}`. We use the words "metadata" and "tag set" interchangeably, and take care to avoid using the word "tag" where "metadata" is meant.

In informal policy language documentation, we use the word "values" to refer to what we call "tags" in this document. We haven't made that switch here in order to reserve the word "value" for use by the operational semantics. At the moment, however, the operational semantics don't need a notion of values, so it might be better to unify the language.

tdecl	::=		tag declarations
	|	$C\ x_1, .., x_k$	
tag, *t*	::=		tags
	|	$C\ tf_1, .., tf_k$	
tf	::=		tag field
	|	_	
	|	x	
	|	t	
	|	new	

1.2 Policies

The "policies" section of a DPL file contains a series of policy declarations, assigning a name to a policy expression. Policy expressions are built out of one primitive "rule" form and three composition operators.

The $pexp_1 \mid pexp_2$ and $pexp_1 \uparrow pexp_2$ forms have the same operational semantics but are intended to have different static semantics to help rule out common errors (see Section 2.4) below. In both cases, the left policy is tried first. If it has a rule that "matches" the current instruction's metadata, that rule's answer is used. otherwise, the policy on the right is used.

The $pexp_1$ & $pexp_2$ form is used when combining two completely distinct policies to be run in parallel. This form is only allowed if $pexp_1$ and $pexp_2$ use different tags. When evaluating $pexp_1$ & $pexp_2$, each policies sees metadata containing only its tags, as if it were running in isolation. An instruction is allowed only if both subpolicies allow it.

pdecl	::=		policy declaration
	|	$x = pexp$	
pexp, *p*	::=		policy expression
	|	x	
	|	*rule*	
	|	$pexp_1 \mid pexp_2$	exclusive
	|	$pexp_1 \uparrow pexp_2$	priority
	|	$pexp_1$ & $pexp_2$	disjoint modules

Rules comprise an opgroup C, a list of patterns, and result. Opgroups specify the subset of RISC-V instructions for which this rule applies and define what memory and register metadata will be available in the rest of the rule (for more details, see Section 2.1.1).

A rule only "matches" the current instruction if that instruction is in opgroup *and* all its patterns match. Otherwise, the rule "implicitly fails" and (typically) the next rule is tried. If the rule does match, its result either indicates an "explicit" policy failure, or provides instructions on how to update the metadata (indicating that the instruction is accepted). More on implicit and explicit failure in Section 2.1.2.

rule	::=		rule
	|	$C(tspats \rightarrow result)$	
result	::=		rule results
	|	fail	
	|	*tsexps*	

Pattern constraints have the form $x = tspat$. Here, x is a name defined by the opgroup to refer to a particular memory address or register used by this category of instructions. *tspat* expresses a constraint on the corresponding tag set.

The simplest forms of patterns are $\{t_1, .., t_k\}$ and $[tr_1, .., tr_k]$. The former matches only if the corresponding tag set is exactly the set shown (order does not matter). The latter matches if each of the individual "tag requirements" is true. The tag requirement $+t$ indicates that a certain tag must be present, while $-t$ indicates that a certain tag must be absent. A tag t by itself in this context is shorthand for $+t$.

The tag set may be bound to a name x for use in the rule's results, using the pattern x which matches every tag set, or $x@tspat$ which matches the same sets as *tspat*. The pattern _ binds no names and matches every tag set.

tspats	::=		tag set pattern list
	|	$x_1 = tspat_1, .., x_j = tspat_j$	
tspat	::=		tag set pattern
	|	_	
	|	x	

	$x@tspat$	
	$\{t_1, .., t_k\}$	exact
	$[tr_1, .., tr_k]$	includes

tr	::=		tag requirement
	|	t	
	|	$+t$	
	|	$-t$	

When a rule accepts an instruction, its conclusion indicates how to update the relevant metadata with the form $x = tsexp$. Here, x is a name defined by the opgroup to refer to a particular memory address or register that is updated by this category of instructions. *tsexp* describes the new tag set for this location.

The simplest form of expression are $\{t_1, .., t_k\}$ and $tsexp[tr_1, .., tr_k]$. The former provides an exact tag set. The latter provides a list of modifications for another tag set *tsexp* (typically a variable bound by the rule's patterns). These modifications have the form $+t$ or $-t$ indicating that a tag should be added if absent or removed if present.

$tsexps$	::=		tag set expression list
	|	$x_1 = tsexp_1, .., x_j = tsexp_j$	

$tsexp$	::=		tag set expression
	|	x	
	|	$\{t_1, .., t_k\}$	exact
	|	$tsexp[tr_1, .., tr_k]$	modify exp
	|	$tsexp_1 \cup tsexp_2$	union
	|	$tsexp_1 \cap tsexp_2$	intersection

2 Operational Semantics

2.1 Design Decisions

2.1.1 Opgroups

In the implementation, opgroups serve two purposes: one is to identify the instruction being executed, and the other is to map the operand names used in the ISA documentation to the intuitive field names written in our rules. This is desirable for two reason: First, there are some classes of instructions where the operands have

different names in the ISA spec, but the meaning of the instruction is similar enough that we want to write rules that cover the whole class. Second, it abstracts away from the ISA, potentially making it possible to write policies that apply for multiple ISAs.

In the language we describe here, we use opgroups only to identify the instruction, and ignore the question of hardware/ISA spec names. Put another way, before executing the semantics described here, it's necessary to first translate from the ISA spec operand names to the rule names, and translate back after. This simplifies the semantics.

A more complete formalization of the language would include the "opgroups" section from the DPL files and construct the mapping from hardware names to intuitive names accordingly.

2.1.2 Failure

At execution time, a policy can fail for two reasons:

- The policy specifies that the given set of tags is a violation, via the fail keyword. We call this *explicit* failure.
- The policy does not contain a rule that matches the given set of tags. We call this *implicit* failure.

Explicit and implicit failure have different meanings, especially when dealing with incomplete pieces of policies. A rule that fails explicitly indicates a real policy violation, while it may fail implicitly because it is intended to be composed with other rules to create a complete policy.

The operational semantics needs to be aware of this distinction. Consider the "pattern matching" form of policy composition, $p_1 \uparrow p_2$ The intended semantics here is that p_1 applies if it has a rule for the given input, and otherwise p_2 applies. If p_1 contains an explicit failure, then $p_1 \uparrow p_2$ should also fail. But an implicit failure of p_1 does not trigger the failure of $p_1 \uparrow p_2$.

2.2 Syntax

We let $\mathcal{T}$ range over sets of tags. Tag environments Δ are maps variables x to tag sets $\mathcal{T}$. They are used in the input and output of policy evaluation. Variables in the domain are either provided by the execution environment for the opgroup's memory and register metadata, or are created by the rule by binding tag sets to variables

with the @ form. We write $\mathsf{dom}\,(\Delta)$ for the domain of a tag environment (that is, the set of variables that are mapped to tag sets).

The empty tag environment (i.e., the tag environment with an empty domain) is written $\cdot$. We write $\Delta[x \mapsto \mathcal{T}]$ for the tag environment that maps x to $\mathcal{T}$ and otherwise has the same mappings as Δ. Finally, we write $\Delta(x)$ for the tag set mapped to x by Δ if $x \in \mathsf{dom}\,(\Delta)$. It is undefined otherwise.

We use R to denote the result of a policy evaluation, which is either implicit failure $\bot$, explicit failure fail, or a new tag environment Δ.

$\mathcal{T}$	::=		tag sets
	|	$\{t_1, .., t_k\}$	
Δ	::=		tag environments
	|	$\cdot$	
	|	$\Delta[x \mapsto \mathcal{T}]$	
$\mathcal{R}$	::=		policy results
	|	Δ	
	|	$\bot$	implicit failure
	|	fail	explicit failure

Two policies that refer to different sets of tags may be combined with the & operator, as in p_1 & p_2. We implement this operator via a union over policy results $\mathcal{R}_1 \sqcup \mathcal{R}_2$, which is defined as follows: If either input is fail, it returns fail. Otherwise, if either input is $\bot$, it returns $\bot$. Finally, n the $\Delta_1 \sqcup \Delta_2$ case, it is defined to map each field name $x \in \mathsf{dom}\,(\Delta_1) \cup \mathsf{dom}\,(\Delta_2)$ to $\Delta_1(x) \cup \Delta_2(x)$ if it appears in both domains, and to $\Delta_1(x)$ or $\Delta_2(x)$ if it appears in only one domain.

2.3 Pattern matching

The most basic form of policy is the *rule clause*.

$$C(\mathit{tspats} \rightarrow \mathit{result})$$

Let's examine the pieces of this rule (see Section 1 for a more complete description:

- C is an opgroup name. Opgroups are tags, so this expresses a constraint that this rule only applies when the tag C appears in the tag set for the current instruction.

- *tspats* is a list of pattern matching equations of the form $x = tspat$. Here, x is a *field name*: a field is an element of the system that can have an associated tag, including memory, registers, and the program counter. *tspat* is a pattern. This pattern serves two purposes: (a) it can put some constraints on x's tag set (for example, that it must include or not include a given tag), and (b) it can bind a name for x's tag set so that it can be mentioned in the conclusion of the rule.

- *result* is either fail (explicit failure), or it is a list of tag set definitions of the form $x = tsexp$. As in *tspats*, x identifies a tag set to be updated. The tag expression *tsexp* defines the updated tag set for x after the instruction executes.

Part of evaluating a rule clause against a tag environment Δ is checking whether *tspats* matches against Δ, and computing a binding for pattern variables if so. The computed map from pattern variables to tag sets is itself a tag environment Δ'. Below, we define three functions that are used in the dynamic semantics to compute this result:

- $\mathsf{contains}_{\mathcal{T}}(tr_1, \ldots, tr_k)$ checks whether $\mathcal{T}$ satisfies the requirement expressed by each tr_i, returning a boolean result.

- $\mathsf{match}_{\mathcal{T}}(\Delta, tspat)$ checks whether the pattern *tspat* matches against the tag set $\mathcal{T}$. If not, it returns $\bot$. If so, it returns the result of updating Δ with any corresponding new bindings.

- $\mathsf{matches}_{\Delta}(tspats)$ checks every pattern in *tspats* against the current tag environment Δ, and collects the results.

$\boxed{\mathsf{contains}_{\mathcal{T}}(tr_1, \ldots, tr_k) = b}$

$$\begin{aligned}
\mathsf{contains}_{\mathcal{T}}(\cdot) &= \mathsf{true} \\
\mathsf{contains}_{\mathcal{T}}(t, trs) &= t \in \mathcal{T} \wedge \mathsf{contains}_{\mathcal{T}}(trs) \\
\mathsf{contains}_{\mathcal{T}}(+t, trs) &= t \in \mathcal{T} \wedge \mathsf{contains}_{\mathcal{T}}(trs) \\
\mathsf{contains}_{\mathcal{T}}(-t, trs) &= t \notin \mathcal{T} \wedge \mathsf{contains}_{\mathcal{T}}(trs)
\end{aligned}$$

$\boxed{\mathsf{match}_{\mathcal{T}}(\Delta, tspat) = \mathcal{R}}$

$$\mathsf{match}_{\mathcal{T}}(\Delta, \_) = \Delta$$
$$\mathsf{match}_{\mathcal{T}}(\Delta, y) = \Delta[y \mapsto \mathcal{T}]$$
$$\mathsf{match}_{\mathcal{T}}(\Delta, y@tspat) = \begin{cases} \Delta'[y \mapsto \mathcal{T}] & \text{if } \mathsf{match}_{\mathcal{T}}(\Delta, tspat) = \Delta' \\ \bot & \text{if } \mathsf{match}_{\mathcal{T}}(\Delta, tspat) = \bot \end{cases}$$
$$\mathsf{match}_{\mathcal{T}}(\Delta, \{t_1, \ldots, t_k\}) = \begin{cases} \Delta & \text{if } \mathcal{T} = \{t_1, \ldots, t_k\} \\ \bot & \text{otherwise} \end{cases}$$
$$\mathsf{match}_{\mathcal{T}}(\Delta, [tr_1, \ldots, tr_k]) = \begin{cases} \Delta & \text{if } \mathsf{contains}_{\mathcal{T}}(tr_1, \ldots, tr_k) = \mathsf{true} \\ \bot & \text{if } \mathsf{contains}_{\mathcal{T}}(tr_1, \ldots, tr_k) = \mathsf{false} \end{cases}$$

$\boxed{\mathsf{matches}\,(\Delta, tspats) = \mathcal{R}}$

$$\frac{}{\mathsf{matches}\,(\Delta, \cdot) = \cdot} \quad \textsc{MNil}$$

$$\frac{y \notin \mathsf{dom}\,(\Delta)}{\mathsf{matches}\,(\Delta, (y = tspat, tspats)) = \bot} \quad \textsc{MDomFail}$$

$$\frac{\mathsf{matches}\,(\Delta, tspats) = \bot}{\mathsf{matches}\,(\Delta, (y = tspat, tspats)) = \bot} \quad \textsc{MTailFail}$$

$$\frac{\begin{array}{l} y \in \mathsf{dom}\,(\Delta) \\ \mathsf{matches}\,(\Delta, tspats) = \Delta' \\ \mathsf{match}_{\Delta(y)}(\Delta', tspat) = \mathcal{R} \end{array}}{\mathsf{matches}\,(\Delta, (y = tspat, tspats)) = \mathcal{R}} \quad \textsc{MCons}$$

Finally, we also describe how to evaluate a collection of tag set expressions, formalized as a judgement $\Delta_1 \vdash tsexps \Rightarrow \Delta_2$. This definition makes use of a tag environment Δ_1 that provides values for any tag set variables in the expressions and is typically the result of the match function defined above. It relies on a secondary judgement, $\Delta_1 \vdash tsexp \Rightarrow \mathcal{T}$, which evaluates an individual tag set expression. There is also a helper function, $\mathsf{update}_{\mathcal{T}}(tr_1, \ldots, tr_k)$, which applies the modification denoted by each tr_i to the tag set $\mathcal{T}$

$\boxed{\mathsf{update}_{\mathcal{T}}(tr_1, \ldots, tr_k) = \mathcal{T}'}$

$$\begin{aligned}
\mathsf{update}_{\mathcal{T}}(\cdot) &= \mathcal{T} \\
\mathsf{update}_{\mathcal{T}}(t, trs) &= \mathsf{update}_{\mathcal{T}}(trs) \cup \{t\} \\
\mathsf{update}_{\mathcal{T}}(+t, trs) &= \mathsf{update}_{\mathcal{T}}(trs) \cup \{t\} \\
\mathsf{update}_{\mathcal{T}}(-t, trs) &= \mathsf{update}_{\mathcal{T}}(trs) \setminus \{t\}
\end{aligned}$$

$\boxed{\Delta \vdash tsexp \Rightarrow \mathcal{T}}$

$$\frac{y \in \mathsf{dom}\,(\Delta)}{\Delta \vdash y \Rightarrow \Delta(y)} \quad \textsc{ETSEVar}$$

$$\frac{}{\Delta \vdash \{t_1, .., t_k\} \Rightarrow \{t_1, .., t_k\}} \quad \textsc{ETSEExact}$$

$$\frac{\Delta \vdash tsexp \Rightarrow \mathcal{T}}{\Delta \vdash tsexp[t_1, .., t_k] \Rightarrow \mathsf{update}_{\mathcal{T}}(t_1, .., t_k)} \quad \textsc{ETSEUpdate}$$

$$\frac{\Delta \vdash tsexp_1 \Rightarrow \mathcal{T}_1 \quad \Delta \vdash tsexp_2 \Rightarrow \mathcal{T}_2}{\Delta \vdash tsexp_1 \cup tsexp_2 \Rightarrow \mathcal{T}_1 \cup \mathcal{T}_2} \quad \textsc{ETSEUnion}$$

$$\frac{\Delta \vdash tsexp_1 \Rightarrow \mathcal{T}_1 \quad \Delta \vdash tsexp_2 \Rightarrow \mathcal{T}_2}{\Delta \vdash tsexp_1 \cap tsexp_2 \Rightarrow \mathcal{T}_1 \cap \mathcal{T}_2} \quad \textsc{ETSEIntersection}$$

$\boxed{\Delta_1 \vdash tsexps \Rightarrow \Delta_2}$

$$\frac{}{\Delta \vdash \cdot \Rightarrow \cdot} \quad \textsc{ETSENil}$$

$$\frac{\begin{array}{c}\Delta_1 \vdash tsexp \Rightarrow \mathcal{T} \\ \Delta_1 \vdash tsexps \Rightarrow \Delta_2\end{array}}{\Delta_1 \vdash x = tsexp, tsexps \Rightarrow \Delta_2[x \mapsto \mathcal{T}]} \quad \textsc{ETSECons}$$

2.4 Static semantics

Our intent is to add a type system that statically rules out common classes of errors. For example:

- The $p_1 \mid p_2$ operator is intended to be used only in situations where p_1 and p_2 don't "overlap". That is, for any Δ, at least one of p_1 or p_2 should result in implicit failure.
- The p_1 & p_2 operator is intended to be used only when p_1 and p_2 are defined in distinct modules that refer to distinct sets of tags. That is, its use is in composing disjoint policies, not pieces of one policy.

We don't currently have a type system or notion of modules in this semantics, though, so these and other statically-detectable errors are not caught at compile time and may result in buggy policies.

2.5 Policy Evaluation

We write $\Delta \vdash p \Rightarrow \mathcal{R}$ to indicate that the policy p has result $\mathcal{R}$ when evaluated in tag environment Δ. If the policy accepts this tag environment, the output will be a set of updated tags. In the case of an explicit failure, the output will be fail. In the case of implicit failure, the output will be $\bot$. See Section 2.1.2 for more on this distinction.

This judgement is defined by the "big-step" operational semantics that follows. One point of interest is the evaluation rules for | and ↑. The | operator is evaluated as if it were ↑. This makes sense, because if the non-overlapping check for | has succeeded, then it has the same behavior ↑. The rules for ↑ enforce the intent that the right policy applies only if the left doesn't. In general, the operational semantics assume that static checks will rule out obvious errors (even though we haven't implemented those checks yet).

Note that the rules make explicit the implementation detail that opgroups are just tags in the current instruction's metadata. The definition of the $\mathsf{matches}$ judgement allows for multiple patterns matches on the same tag set name. It would be good to find another way to express this, or to make opgroups more explicit in general.

$\boxed{\Delta \vdash p \Rightarrow \mathcal{R}}$

$$\frac{\mathsf{matches}\,(\Delta, (\mathsf{ci} = [+C\,], \mathit{tspats})) = \bot}{\Delta_1 \vdash C(\mathit{tspats} \rightarrow \mathit{result}) \Rightarrow \bot} \quad \textsc{EPRuleImpFail}$$

$$\frac{\mathsf{matches}\,(\Delta, (\mathsf{ci} = [+C\,], \mathit{tspats})) = \Delta'}{\Delta \vdash C(\mathit{tspats} \to \mathsf{fail}) \Rightarrow \mathsf{fail}} \quad \textsc{EPRuleExpFail}$$

$$\frac{\mathsf{matches}\,(\Delta_1, (\mathsf{ci} = [+C\,], \mathit{tspats})) = \Delta_2 \quad \Delta_2 \vdash \mathit{tsexps} \Rightarrow \Delta_3}{\Delta_1 \vdash C(\mathit{tspats} \to \mathit{tsexps}) \Rightarrow \Delta_3} \quad \textsc{EPRuleSuccess}$$

$$\frac{\Delta \vdash p_1 \Rightarrow \Delta'}{\Delta \vdash p_1 \uparrow p_2 \Rightarrow \Delta'} \quad \textsc{EPCompMatchLSuccess}$$

$$\frac{\Delta \vdash p_1 \Rightarrow \mathsf{fail}}{\Delta \vdash p_1 \uparrow p_2 \Rightarrow \mathsf{fail}} \quad \textsc{EPCompMatchLFail}$$

$$\frac{\Delta \vdash p_1 \Rightarrow \bot \quad \Delta \vdash p_2 \Rightarrow \mathcal{R}}{\Delta \vdash p_1 \uparrow p_2 \Rightarrow \mathcal{R}} \quad \textsc{EPCompMatchR}$$

$$\frac{\Delta \vdash p_1 \uparrow p_2 \Rightarrow \mathcal{R}}{\Delta \vdash p_1 \mid p_2 \Rightarrow \mathcal{R}} \quad \textsc{EPCompExcl}$$

$$\frac{\Delta \vdash p_1 \Rightarrow \mathcal{R}_1 \quad \Delta \vdash p_2 \Rightarrow \mathcal{R}_2}{\Delta \vdash p_1 \,\&\, p_2 \Rightarrow \mathcal{R}_1 \sqcup \mathcal{R}_2} \quad \textsc{EPCompModule}$$