Type-Driven Repair for Information Flow Security

Supplementary Material

Well-Formedness $\boxed{\Gamma \vdash r}\ \boxed{\Gamma \vdash B}\ \boxed{\Gamma \vdash S}$

$$\text{WF-}\psi\ \frac{\Gamma \vdash \psi : \texttt{Bool}}{\Gamma \vdash \psi}$$

$$\text{WF-}\pi\ \frac{\Gamma \vdash \pi\ \bar{x} : \texttt{Bool} \qquad \Gamma(\pi) \neq T[\ominus]}{\Gamma \vdash \pi\ \bar{x}}$$

$$\text{WF-}\alpha\ \frac{\Gamma(\alpha) \neq \ominus}{\Gamma \vdash \alpha} \qquad \text{WF-SC}\ \frac{\Gamma \vdash B \qquad \Gamma; \nu : B \vdash r}{\Gamma \vdash \{B \mid r\}}$$

$$\text{WF-FUN}\ \frac{\Gamma^{-} \vdash T_x \qquad \Gamma; x : T_x \vdash T}{\Gamma \vdash T_x \to T}$$

$$\text{WF-D}\ \frac{\begin{array}{c}\Gamma(D) = \overline{\forall_{\circ}\alpha_i}.\overline{\forall_{\circ}\langle \pi_j : U_j \rangle}.T \\ |T_i| = |\alpha_i| \qquad \Gamma \vdash p_j : U_j\end{array}}{\Gamma \vdash D\ \overline{T_i}\ \langle \overline{p_j} \rangle}$$

$$\text{WF-FO}\ \frac{\Gamma^{-} \vdash \{B \mid r\} \qquad \Gamma; x : \{B \mid r\} \vdash T}{\Gamma \vdash x \colon \{B \mid r\} \to T}$$

$$\text{WF-HO}\ \frac{T_x \text{ non-scalar} \qquad \Gamma^{-} \vdash T_x \qquad \Gamma \vdash T}{\Gamma \vdash T_x \to T}$$

$$\text{WF-}\forall\alpha\ \frac{\Gamma; \alpha : \circ \vdash S}{\Gamma \vdash \forall_{\circ}\alpha.S} \qquad \text{WF-}\forall\pi\ \frac{\Gamma; \pi : T[\circ] \vdash S}{\Gamma \vdash \forall_{\circ}\langle \pi : T \rangle.S}$$

Figure 1. Well-formedness rules of $\mathcal{BL}$.

Subtyping $\boxed{\Gamma \vdash T <: T'}$

$$\text{<:-REFL}\ \frac{}{\Gamma \vdash B\ <:\ B}$$

$$\text{<:-SC}\ \frac{\Gamma \vdash B\ <:\ B' \qquad \mathsf{Valid}([\![\Gamma]\!] \wedge r \Rightarrow r')}{\Gamma \vdash \{B \mid r\}\ <:\ \{B' \mid r'\}}$$

$$\text{<:-FUN}\ \frac{\Gamma \vdash T_y\ <:\ T_x \qquad \Gamma; y : T_y \vdash [y/x]T\ <:\ T'}{\Gamma \vdash x \colon T_x \to T\ <:\ y \colon T_y \to T'}$$

$$\text{<:-D}\ \frac{\begin{array}{c}\Gamma(D) = \overline{\forall_{\circ_i}\alpha_i}.\overline{\forall_{\circ_j}\langle \pi_j \rangle}.T \\ \Gamma \vdash T_i \sim_{\circ_i} T_i' \qquad \Gamma \vdash p_j \sim_{\circ_j} p_j'\end{array}}{\Gamma \vdash D\ \overline{T_i}\ \langle \overline{p_j} \rangle\ <:\ D\ \overline{T_i'}\ \langle \overline{p_j'} \rangle}$$

$$\frac{\Gamma \vdash T <: T'}{\Gamma \vdash T \sim_{\oplus} T'} \qquad \frac{\Gamma \vdash T' <: T}{\Gamma \vdash T \sim_{\ominus} T'}$$

$$\frac{\Gamma \vdash T <: T' \qquad \Gamma \vdash T' <: T}{\Gamma \vdash T \sim_{\odot} T'}$$

$$\frac{\Gamma; x : T \vdash p \sim_{\circ} p'}{\Gamma \vdash \lambda x : T.p \sim_{\circ} \lambda x : T.p'}$$

$$\frac{\Gamma \vdash \{\texttt{()} \mid r\} \sim_{\circ} \{\texttt{()} \mid r'\}}{\Gamma \vdash r \sim_{\circ} r'}$$

Figure 2. Subtyping rules of $\mathcal{BL}$.

1. $\mathcal{BL}$ Static Semantics

Figures 1,2, and 3 give the full version of the static semantics of $\mathcal{BL}$.

1.1 Proving Non-Interference Using $\texttt{Tagged}^2$

We now prove that executions involving the `Tagged` monad preserve *contextual noninterference*: if a sensitive value v may not flow to a given viewer, then any pair of executions involving different assignments to v should yield equivalent outputs.

Reasoning directly about noninterference is inconvenient because it requires talking about two executions. We simplify our noninterference proof using a technique similar to that of Pottier and Simonet [1]: we introduce auxiliary constructs that allow us to reason about two executions in one. Being able to encode security labels as a library makes the formalization particularly nice: the only auxiliary construct we need for the proof is an alternative definition of the `Tagged` monad. We introduce the $\texttt{Tagged}^2$ monad with new implementations of the four primitive operations, yielding the property that if a program type-checks with $\texttt{Tagged}^2$, then it preserves contextual noninterference with `Tagged`.

The $\texttt{Tagged}^2$ ***monad.*** To simplify formalization of noninterference, we parameterize the semantics of $\mathcal{BL}$ by the context, *i.e.* the principal who is observing the execution and the world at the time of output. More concretely, we assume that

Type Checking $\boxed{\Gamma \vdash e :: S}$

$$\textsc{Subt}\ \frac{\Gamma \vdash e :: T' \qquad \Gamma \vdash T' <: T}{\Gamma \vdash e :: T}$$

$$\textsc{Var-Sc}\ \frac{\Gamma(x) = \{B \mid r\}}{\Gamma \vdash x :: \{B \mid \nu = x\}}$$

$$\textsc{Var}\ \frac{\Gamma(x) = S \qquad S \text{ non-scalar}}{\Gamma \vdash x :: S}$$

$$\textsc{Abs}\ \frac{\Gamma \vdash T_x \qquad \Gamma; x\colon T_x \vdash e :: T}{\Gamma \vdash \lambda x : T_x.e :: (x\colon T_x \to T)}$$

$$\textsc{Let}\ \frac{\begin{array}{c}\Gamma \vdash v_1 :: (y\colon T_y \to T') \qquad \Gamma \vdash v_2 :: T_2 \\ \Gamma \vdash T_2 <: T_y \qquad \Gamma; x : [v/y]T' \vdash e :: T\end{array}}{\Gamma \vdash \mathtt{let}\ x = v_1\ v_2\ \mathtt{in}\ e :: T}$$

$$\textsc{Match}\ \frac{\begin{array}{c}\Gamma \vdash x :: \{D\ \bar{T_x}\ \langle \bar{p_x} \rangle \mid r_x\} \\ \Gamma(D) = \forall_\circ \bar{\alpha}.\forall_\circ \langle \bar{\pi} \rangle. \\ T_1 \to \ldots \to T_n \to \{D\ \bar{\alpha}\ \langle \bar{\pi} \rangle \mid r\} \\ \Gamma; y_i : [\bar{T_x}/\bar{\alpha}][\bar{p_x} \rhd \bar{\pi}]T_i; \\ x : \{D\ \bar{T_x}\ \langle \bar{p_x} \rangle \mid r_x \wedge r\} \vdash e :: T\end{array}}{\Gamma \vdash \mathtt{match}\ x\ \mathtt{with}\ D\ \bar{x} \to e :: T}$$

$$\textsc{If}\ \frac{\begin{array}{c}\Gamma \vdash x :: \{\mathtt{Bool} \mid r\} \\ \Gamma; [\top/\nu]r \vdash e_1 :: T \qquad \Gamma; [\bot/\nu]r \vdash e_2 :: T\end{array}}{\Gamma \vdash \mathtt{if}\ x\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 :: T}$$

$$\textsc{T-Gen}\ \frac{\Gamma; \alpha : \circ \vdash e :: S}{\Gamma \vdash e :: \forall_\circ \alpha.S}$$

$$\textsc{T-Inst}\ \frac{\Gamma \vdash e :: \forall_\circ \alpha.S \qquad \Gamma \vdash \{B \mid r\}}{\Gamma \vdash e :: [\{B \mid r\}/\alpha]S}$$

$$\textsc{P-Gen}\ \frac{\Gamma; \pi : T[\circ] \vdash e :: S \qquad \Gamma \vdash T}{\Gamma \vdash e :: \forall_\circ \langle \pi : T \rangle.S}$$

$$\textsc{P-Inst}\ \frac{\Gamma \vdash e :: \forall_\circ \langle \pi : T \rangle.S \qquad \Gamma \vdash p : T}{\Gamma \vdash e :: [p \rhd \pi]S}$$

Figure 3. Type-checking rules of $\mathcal{BL}$.

the environment always contains two variables `cw` : W and `cu` : U; when a program executes, it executes with all possible values of `cw` and `cu` "in parallel", but in each of these parallel threads, **`print`** only performs the output when its arguments match `cw` and `cu`, so this parametric semantics has no effect on the output.

We first construct a *phantom encoding*: a new information flow monad, `Tagged`^{2}, that explicitly relates pairs of program executions. The intuition behind `Tagged`^{2} is as follows: it represents two versions of a sensitive value from two different executions of the program as seen by the current context. Mirroring what we want for our noninterference property, the two versions are only allowed to differ for those sensitive values that are *not visible* in the context. The `Tagged`^{2} constructor accepts two α values, `l` and `r`, which we call *projections*. Its third argument `prop` serves as a proof of the property p `cw` `cu` $\Rightarrow l = r$, that is, if the policy holds of the current context, the two projections must be equal.

```
module Tagged² where

private cw: W, cu: U -- Current context

-- | Tagged data constructor
private Tagged²: ∀α . ∀⊖<p:  W → U → Bool> .
  l: α → r: α → prop: ({() | p cw cu} → {() | l = r})
  → Tagged α <p>

return²: ∀α . ∀⊖<p:  W → U → Bool> . α → Tagged α <p>
return² = λx . Tagged² x x (λz . ())

bind²: ∀α β . ∀⊖<p:  W → U → Bool> . ∀<f: α → β →
Bool> .
  x:  Tagged α <p> → (y:  α → Tagged {β | f y ν} <p>)
  → Tagged {β | f (l x) ν} <p>
bind² = λx . λg .
  match x with Tagged² xl xr xp →
  match g xl with Tagged² yl _ yp →
    match g xr with Tagged² _ yr _ →
      Tagged² yl yr (λz  .  join (xp z) (yp z))

print²: ∀α . ∀⊖<p:  W → U → Bool> .
  w: W → u: Tagged² {U | p w ν}<p>
  → x: Tagged² α <p> → W
print² = λw . λu . λx .
  match u with Tagged² ul ur up →
    if w ≠ cw ∨ (ul ≠ cu ∧ ur ≠ cu) then w
      else if ul ≠ ur then fail (up ())
        else match x with Tagged² xl xr xp →
          if xl ≠ xr
            then fail (xp ())
            else doPrint w xl

downgrade²: ∀⊖<p:  W → U → Bool> . ∀<c:  Bool> .
  x:  Tagged² {Bool | ν ⇒ c} <λw u . p w u ∧ c>
  → Tagged² {Bool | ν ⇒ c} <p>
downgrade² = λx .
  match x with
    Tagged² xl xr xp →
      Tagged² xl xr (λz . if xl ∨ xr then xp z else ())
```

Figure 4. The `Tagged`^{2} monad, which keeps track of two projections.

A `Tagged`^{2} value with different projections corresponds to Pottier and Simonet's "bracket value" in [1], and the `prop` requirement corresponds to their rule that all bracket values are assigned high security labels. The main conceptual difference of our treatment is that the division between high and low security, as well as the notion of a leak, is context-specific.

We show the implementation of the `Tagged`^{2} in Fig. 4. The phantom encoding provides alternative implementations of the four primitive operations. The function **`return`**^{2} gives the same value for both projections, while **`bind`**^{2} applies the function projection-wise. The $\mathcal{BL}$ type checker can easily show both implementations type-safe.

The function **`print`**^{2} is designed to *fail* when it detects interference: namely, whenever the target of the output is different in the two executions ($\texttt{ul} \neq \texttt{ur}$) or because it outputs two different values ($\texttt{xl} \neq \texttt{xr}$). We assume that **`fail`** has the type `{() | False}`$\rightarrow$`a`, so the only way to type-check **`print`**^{2} is to prove that both failing branches are unreachable, which the $\mathcal{BL}$ type checker successfully accomplishes. To understand why the first failing branch is unreachable, recall that from the type of `u` we know that $\texttt{p w ul} \land \texttt{p w ur}$; we also know that $\texttt{w} = \texttt{cw}$ and $\texttt{ul} = \texttt{cu} \lor \texttt{ur} = \texttt{cu}$ from the path condition, thus `p cw cu` holds, which gives $\texttt{ul} = \texttt{ur}$ guaranteed by the `Tagged`^{2} constructor.

The function **`downgrade`**^{2} simply reconstructs its argument, but provides a proof that $\texttt{xl} = \texttt{xr}$ under a weaker assumption. The proof can be understood as follows: if $\texttt{xl} \lor \texttt{xr}$, then `c` must hold, so we can invoke the proof `xp` of $\texttt{xl} = \texttt{xr}$ that we obtained form the argument; otherwise $\texttt{xl} = \texttt{xr} = \texttt{False}$.

Contextual noninterference. We now show that type-checking with `Tagged`^{2} implies contextual noninterference with `Tagged`. Because the `Tagged`^{2} functions type-check and because the type system of $\mathcal{BL}$ is sound [2], we know that no type-correct program that manipulates `Tagged`^{2} values can go wrong, *i.e.* attempts to print the results of two executions that are different. Now we only have to formally connect computations with `Tagged` values and those with `Tagged`^{2} values, and show how type safety of the latter implies noninterference for the former.

We first show that replacing a `Tagged`^{2} value with its projection in `Tagged` at the beginning of an execution yields the same result as projecting at the end of an execution. A *projection* of an expression e (written $\lfloor e \rfloor_j$, for $j = \{l, r\}$) is an expression where every occurrence of `Tagged`^{2} $x_l\ x_r\ \_$ in e is replaced by `Tagged` x_j.

Lemma 1 (Projection)**.** If $e \rightarrow^* e'$ then $\lfloor e \rfloor_j \rightarrow^* \lfloor e' \rfloor_j$, for $j = \{l, r\}$.

Proof outline. The only steps that are different in the evaluation of e and its projections are those resulting from the bodies of **`bind`** and **`print`**. By inspection of **`bind`**^{2} it is easy to see that it applies the function projection-wise, and thus preserves the property of the lemma. In case of **`print`**^{2}, since it does not fail, either it does not do any output, or the two projections are the same; in both cases, projections of its body will have the same behavior. □

Theorem (Contextual Noninterference)**.** Let

$$\Gamma; x : \texttt{Tagged}\ \alpha\ \langle p \rangle \vdash e :: W$$

and $\neg(p\ \texttt{cw}\ \texttt{cu})$. Let for $j \in \{l, r\}$, $\Gamma \vdash v_j :: \alpha$ and $[(\texttt{Tagged}\ v_j)/x]e \rightarrow^* w_j$. Then $w_l = w_r$.

Proof outline. Since $\neg(p\ \texttt{cw}\ \texttt{cu})$, we know

$$\Gamma \vdash \texttt{Tagged}^2\ v_l\ v_r\ \texttt{id} :: \texttt{Tagged}\ \alpha\ \langle p \rangle$$

for any v_l, v_r. Let e^2 be $[(\texttt{Tagged}^2\ v_l\ v_r\ \texttt{id})/x]e$; note that $\lfloor e^2 \rfloor_j = [(\texttt{Tagged}\ v_j)/x]e$. By inspection of typing rules of $\mathcal{BL}$, substitution of a subterm with the same type does not change the type of the term, so $\Gamma \vdash e^2 :: W$. By soundness of the type system, e^2 either diverges or reduces to a value w of type W. Note that the execution of e^2 differs from the executions of either $[(\texttt{Tagged}\ v_j)/x]e$ only in the bodies of **`bind`** and **`print`** functions; since none of them introduces divergence, e^2 cannot diverge either. By Lemma 1, $\lfloor e^2 \rfloor_j \rightarrow^* \lfloor w \rfloor_j$, that is $w_j = \lfloor w \rfloor_j$, but $\lfloor w \rfloor_l = \lfloor w \rfloor_r$ since w is a value of type W, which is different from `Tagged`. □

A note on the proof technique. Being able to express tagged values as a data type with a phantom predicate parameter is not only simpler, but also allows us to prove non-interference over pairs of traces simply by grounding phantom predicates. In the information flow monad `Tagged`, policies are phantom predicates that do not appear in the arguments of data constructors. In `Tagged`^{2}, the predicates are no longer phantom, but appear negatively in the type of `prop`, consistent with its variance annotation. Using these predicates for explicitly relating multiple program executions helps simplify the formalization and proof of non-interference.

References

[1] F. Pottier and V. Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems*, 25(1), Jan. 2003.

[2] N. Vazou, P. M. Rondon, and R. Jhala. Abstract refinement types. In *ESOP*, 2013.