# FABLE: A Language for Enforcing User-defined Security Policies

Nikhil Swamy Brian J. Corcoran Michael Hicks
University of Maryland, College Park
{nswamy, bjc, mwh}@cs.umd.edu

## Abstract

*This paper presents* FABLE, *a core formalism for a programming language in which programmers may specify security policies and reason that these policies are properly enforced. In* FABLE, *security policies can be expressed by associating* security labels *with the data or actions they protect. Programmers define the semantics of labels in a separate part of the program called the* enforcement policy. FABLE *prevents a policy from being circumvented by allowing labeled terms to be manipulated only within the enforcement policy; application code must treat labeled values abstractly. Together, these features facilitate straightforward proofs that programs implementing a particular policy achieve their high-level security goals.* FABLE *is flexible enough to implement a wide variety of security policies, including access control, information flow, provenance, and security automata. We have implemented* FABLE *as part of the* LINKS *web programming language; we call the resulting language* SELINKS. *We report on our experience using* SELINKS *to build two substantial applications, a wiki and an on-line store, equipped with a combination of access control and provenance policies. To our knowledge, no existing framework enables the enforcement of such a wide variety of security policies with an equally high level of assurance.*

## 1 Introduction

For 35 years or more, computer security researchers have explored techniques for ensuring that a software system correctly enforces its security policy, and that, as a result, the software exhibits a desirable security property [22]. A notable success toward this goal has been work on defining programming language-based techniques for enforcing *information flow* security policies [32]. A common form of information flow policy defines a set of security levels that can be ordered as a lattice, where sensitive data within a program is assigned a label derived from this lattice.Correct enforcement of this policy implies that a program exhibits some flavor of *noninterference*, which states that no information visible at level $h$ can be leaked onto a channel visible to level $l < h$. By including the notion of a security label in a programming language's types, one can show that a correctly typed program is certain to enforce its security policy [41]. This approach has been implemented successfully in the Jif [10] and FlowCaml [30] languages.

While information flow policies are useful and important, there are many other styles of policy that are in common use, including access control, type enforcement [4] (as in SELinux [26]), tainting [35, 37] (e.g., via Perl's *taint mode* [29]), provenance tracking [7], stack inspection [17, 14], and forms of security automata [17, 42]. One approach to verifying the correct enforcement of these policies is to encode them as information flow policies for programs written in Jif of FlowCaml. While this will work in some cases (e.g., access control, type enforcement, and tainting could be encoded in conjunction with Jif's declassification operators [33]) it is not likely to scale. For example, Jif's use of noninterference as a baseline property and its attendant restrictions on *implicit flows* via the program's control flow, can be cumbersome to work with. Moreover, Jif and FlowCaml fix the format of security labels, which complicates the means to interface with external infrastructure, such as policy management systems, databases, etc.

What we want is a programming language that can enforce a wide range of policies—including, but not limited to, information flow—while providing the same assurance as Jif or FlowCaml that programs enforce their policies correctly. As a step toward this goal, this paper presents FABLE, a core language for writing programs that enforce a variety of security policies. A key observation is that many security policies work by associating labels with data, where the label expresses the security policy for that data. What varies among policies is the *specification* and *interpretation*

Prepared through collaborative participation in the Communications and Networks Consortium sponsored by the U. S. Army Research Laboratory under the Collaborative Technology Alliance Program, Cooperative Agreement DAAD19-01-2-0011. The U. S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon. This work was also supported in part by NSF grant CCF-0524036.

of labels, in terms of the actions that are permitted or denied.

This observation is embodied in FABLE in two respects. First, programmers can define custom security labels and associate them with the data they protect using dependent types. For example, a programmer could define a label *LOW*, and an integer value protected by this label would have type int$\{LOW\}$. As another example, the programmer could define a label *ACL*(*Alice*, *Bob*) where an integer with type int$\{ACL(Alice, Bob)\}$ is meant to be accessed by only *Alice* or *Bob*. Second, programmers define the interpretation of labels in special *enforcement policy* functions separated from the rest of the program. For example, the semantics of our access control label could be implemented by the following enforcement policy function:

```
policy access_simple (acl:lab, x:int{acl}) =
    if (member user acl) then {∘}x else −1
```

This function takes a label like *ACL*(*Alice*, *Bob*) as its first argument, and an integer protected by that ACL as its second argument. If the current user (represented by variable *user*) is a member of *x*'s ACL (according to some function *member*, not shown), then $x$ is returned with its label removed, expressed by the syntax $\{\circ\}x$, so that it can be accessed by the main program. If the membership test fails, it returns $-1$ and *x*'s value is not released.

FABLE does not, in and of itself, guarantee that a security policy is correctly implemented, but FABLE's design greatly simplifies proof of this fact. In particular, FABLE's type system ensures that labeled data (that is, data with a type $t\{l\}$) is treated *abstractly* by the main program, since terms with a labeled type can only be constructed, examined, or changed within enforcement policy code. Moreover, FABLE's type system ensures that the main program cannot sever or forge the association between a label and the data it protects. In effect, FABLE ensures *complete mediation* of the user's label policy in that no data can be accessed without consulting the correct security policy.

To demonstrate FABLE's flexibility we have used it to encode a range of policies, including access control, static [32] and dynamic information flow [46] with forms of declassification [20], provenance tracking [7] and policies based on security automata [42]. In our experience, the soundness of FABLE makes proofs of security properties no more difficult—and arguably simpler—than proofs of similar properties in specialized languages [30, 40, 41]. To demonstrate this fact we present proofs of correctness for our access control, provenance, and static information flow policies. FABLE opens the possibility of partially automating such proofs, along the lines of user-defined type systems [8], though we leave exploration of this issue to future work. To our knowledge, no existing framework enables the enforcement of such a wide variety of security policies with an equally high level of assurance.

To evaluate FABLE's practicality we have implemented FABLE as an extension to the LINKS web programming language [12]. We call the resulting language SELINKS (for *Security-Enhanced* LINKS). We have built two substantial applications using SELINKS: SEWIKI, a 3500-line secure blog/wiki inspired by *Intellipedia* [31] that implements a combined access control and provenance policy, and SEWINESTORE, a 1000-line e-commerce application distributed with LINKS extended with an access control policy. In general, we have found that FABLE's label-based security policies are neither lacking nor burdensome, and the modular separation of the enforcement policy permitted some reuse of policy code between the two applications.

In the remainder of the paper we present FABLE, our core language for defining and enforcing custom, label-based security policies (Section 2). We show how FABLE can be used to define a range of security policies and that FABLE's design simplifies proofs that these policies are implemented correctly (Section 3). In Section 4 we discuss our SELINKS implementation of FABLE for building web applications and our experience building SEWIKI and SEWINESTORE. Section 5 discusses related work, and Section 6 sketches future work and concludes.

## 2 FABLE: System F with Labels

This section presents the syntax, static semantics, and operational semantics of FABLE. The next section illustrates FABLE's flexibility by presenting example policies along with proofs of their attendant security properties.

### 2.1. Syntax

Figure 1 defines FABLE's syntax. Throughout, we use the notation $\vec{a}$ to stand for a list of elements of the form $a_1, \ldots, a_n$; where the context is clear, we will also treat $\vec{a}$ as the set of elements $\{a_1, \ldots, a_n\}$.

Expressions $e$ extend a standard polymorphic $\lambda$-calculus, System F [16]. Standard forms include integer values $n$, variables $x$, abstractions $\lambda x{:}t.e$, term application $e_1\, e_2$, the fixpoint combinator $\mathsf{fix}\, x{:}t.v$, type abstraction $\Lambda\alpha.e$ and type application $e\,[t]$. We exclude mutable references from the language to simplify the presentation. Our technical report [38] extends the language with references and considers their effect on various policies, e.g., information flows through side effects.

The syntactic constructs specific to FABLE are distinguished in Figure 1. The expression $C(\vec{e})$ is a label, where $C$ represents an arbitrary constructor and each $e_i \in \vec{e}$ must itself be a label; e.g., in *ACL*(*Alice*, *Bob*), *ACL* is 2-ary label constructor and *Alice* and *Bob* are 0-ary label constructors. Labels can be examined by pattern matching. For example,

$$\begin{array}{rcll}
\text{Expressions} & e & ::= & n \mid x \mid \lambda x{:}t.e \mid e_1\, e_2 \mid \mathsf{fix}\ x{:}t.v \mid \Lambda\alpha.e \mid e[t] \\
\text{(Fable-specific)} & & \mid & C(\vec{e}) \mid \mathsf{match}\ e\ \mathsf{with}\ p_i \Rightarrow e_i \mid \llparenthesis e \rrparenthesis \mid \{\circ\}e \mid \{e'\}e \\
\text{Types} & t & ::= & \mathsf{int} \mid \alpha \mid \forall\alpha.t_2 \mid (x{:}t_1) \rightarrow t_2 \\
\text{(Fable-specific)} & & \mid & \mathsf{lab} \mid \mathsf{lab}{\sim}e \mid t\{e\}
\end{array}
\qquad
\begin{array}{rcll}
\text{Patterns} & p & ::= & x \mid C(\vec{p}) \\
\text{Pre-values} & u & ::= & n \mid C(\vec{u}) \mid \lambda x{:}t.e \mid \Lambda\alpha.e \\
\text{App. values} & v_{app} & ::= & u \mid \llparenthesis \{e\}v_{pol} \rrparenthesis \\
\text{Policy values} & v_{pol} & ::= & u \mid \{e\}v_{pol}
\end{array}$$

**Figure 1. Syntax of** FABLE

the expression match $z$ with $ACL(x,y) \Rightarrow x$ would evaluate to *Alice* if $z$'s run-time value were $ACL(Alice, Bob)$.

As explained earlier, FABLE introduces the notion of an *enforcement policy* that is a separate part of the program authorized to manipulate the labels on a type. Following Grossman et al. [18] we use *bracketed expressions* $\llparenthesis e \rrparenthesis$ to delimit policy code $e$ from the main program. In practice, one could use code signing as in Java [17] to ensure that untrusted policy code cannot be injected into a program.As illustrated earlier, the expression $\{\circ\}e$ removes a label from $e$'s type, while $\{e'\}e$ adds one. Labeling and unlabeling operations may only occur within policy code; we discuss these operations in detail below.

Standard types $t$ include int, type variables $\alpha$, and universally quantified types $\forall\alpha.t$. Functions have dependent type $(x{:}t_1) \rightarrow t_2$ where $x$ names the argument and may be bound in $t_2$. We illustrate the usage of these types shortly. Labels can be given either type lab or the *singleton type* $\mathsf{lab}{\sim}e$, which describes label expressions equivalent to $e$. For example, the label constructor *High* can be given the type lab and the type $\mathsf{lab}{\sim}High$. Singleton types are useful for constraining the form of label arguments to enforcement policy functions. For example, we could write a specialized form of our previous *access_simple* function:

policy $access\_pub\ (acl{:}\mathsf{lab}{\sim}ACL(World),\ x{:}\mathsf{int}\{acl\}) = \{\circ\}x$

The FABLE type checker ensures this function is called only with expressions that evaluate to the label $ACL(World)$—i.e., the call $access\_pub(ACL(Alice,Bob),e)$ will be rejected. In effect, the type checker is performing access control at compile time according to the constraint embodied in the type. We will show in Section 3.3 that these constraints are powerful enough to encode an information flow policy that can be checked entirely at compile time.

The dependent type $t\{e\}$ describes a term of type $t$ that is associated with a label $e$. Such an association is made using the syntax $\{e\}e'$. For example, $\{High\}1$ is an expression of type $\mathsf{int}\{High\}$. Conversely, this association can be broken using the syntax $\{\circ\}e$. For example, $\{\circ\}(\{High\}1)$ has type int. Now we can illustrate how dependent function types $(x{:}t_1) \rightarrow t_2$ can be used. The function *access_simple* can be given the type $(acl{:}\mathsf{lab}) \rightarrow (x{:}\mathsf{int}\{acl\}) \rightarrow \mathsf{int}$ which indicates that the first argument $acl$ serves as the label for the second argument $x$. Instead of writing $(x{:}t_1) \rightarrow t_2$ when $x$ does not appear in $t_2$, we simply omit it. Thus *access_simple*'s type could be written $(acl{:}\mathsf{lab}) \rightarrow \mathsf{int}\{acl\} \rightarrow \mathsf{int}$.

The operational semantics of Section 2.3 must distinguish between application and policy values in order to ensure that policy code does not inadvertently grant undue privilege to application functions. Application values $v_{app}$ consist of either "pre-values" $u$—integers $n$, labels containing values, type and term abstractions—or labeled policy values wrapped with $\llparenthesis\cdot\rrparenthesis$ brackets. Values within policy code are pre-values preceded by zero or more relabeling operations.

**Encodings.** To make our examples more readable, we use the syntactic shorthands shown in Figure 2. The first three shorthands are mostly standard. We use the policy keyword to designate policy code instead of using brackets $\llparenthesis\cdot\rrparenthesis$. A dependent pair $(e, e')$ of type $x{:}t \times t'$ allows $x$, the name for the first element, to be bound in $t'$, the type of the second element. For example, the first two arguments to the *access_pub* function above could be packaged into a dependent pair of type $(acl{:}\mathsf{lab}{\sim}ACL(World) \times \mathsf{int}\{acl\})$ which is inhabited by terms such as $(ACL(World),\{ACL(World)\}1)$. Dependent pairs can be encoded using dependently typed functions. We extend the shorthand for function application, policy function definitions, type abbreviations, and tuples to multiple type and term arguments in the obvious way. We also write _ as a wildcard ("don't care") pattern variable.

**Phantom label variables.** We extend the notation for polymorphic functions in a way that permits quantification over the *expressions* that appear in a type. Consider the example below:

policy $add\langle l\rangle(x{:}\mathsf{int}\{l\},\ y{:}\mathsf{int}\{l\}) = \{l\}(\{\circ\}x + \{\circ\}y)$

This policy function takes two like-labeled integers $x$ and $y$ as arguments, unlabels them and adds them together, and finally relabels the result, having type $\mathsf{int}\{l\}$. This function is unusual because the label $l$ is not a normal term argument, but is being quantified—any label $l$ would do.

The reason this makes sense is that in FABLE, (un)labeling operations are merely hints to the type checker to (dis)associate a label term and a type. These operations, along with all types, can be erased at runtime without affecting the result of a computation. After erasing types, our example would become policy $add\ (x, y) = x + y$, which is clearly only a function of $x$ and $y$, with no mention of $l$. For this reason, we can treat *add* as *polymorphic in the labels* of

| | | | | |
|---|---|---|---|---|
| type abbreviation | $\mathsf{typename}\ N\ \alpha = t\ \mathsf{in}\ e_2$ | $\equiv$ | $(N\,t' \mapsto ((\alpha \mapsto t')t))e_2$ | |
| let binding | $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ | $\equiv$ | $(\lambda x{:}t.e_2)\ e_1$ | for some $t$ |
| polymorphic function def. | $\mathsf{let}\ f\langle\alpha\rangle(x{:}t) = e_1\ \mathsf{in}\ e_2$ | $\equiv$ | $\mathsf{let}\ f = \mathsf{fix}\ f{:}t'.\Lambda\alpha.\lambda x{:}t.e_1\ \mathsf{in}\ e_2$ | for some $t'$ |
| policy function def | $\mathsf{policy}\ f\langle\alpha\rangle(x{:}t) = e_1\ \mathsf{in}\ e_2$ | $\equiv$ | $\mathsf{let}\ f = \mathsf{fix}\ f{:}t'.\Lambda\alpha.\lambda x{:}t.(\!\|e_1\|\!)\ \mathsf{in}\ e_2$ | f or some $t'$ |
| dependent tuple type | $x{:}t \times t'$ | $\equiv$ | $\forall\alpha.((x{:}t) \to t' \to \alpha) \to \alpha$ | |
| dependent tuple introduction | $(e, e')$ | $\equiv$ | $\Lambda\alpha.\lambda f{:}((x{:}t) \to t' \to \alpha).f\ e\ e'$ | for some $t, t'$ |
| dependent tuple projection | $\mathsf{let}\ x{,}y = f\ \mathsf{in}\ e$ | $\equiv$ | $f\,[t_e](\lambda x{:}t.\lambda y{:}t'.e)$ | for some $t, t'$, and $t_e$ |

**Figure 2. Syntactic shorthands**

$x$ and $y$—it can be called with any pair of integers that have the same label, irrespective of what label that might be. We express this kind of polymorphism by writing the *phantom label variable l*, together with any other normal type variables like $\alpha, \beta, \ldots$, in a list that follows the function name. In the example above, the phantom variable of *add* are listed as $\langle l\rangle$. Of course, not all label arguments are phantom. For instance, in the *access_simple* function of Section 1, the *acl* is a label argument that *is* passed at runtime. For simplicity, we do not formalize phantom variable polymorphism. Our technical report [38] does model phantom variables and contains the associated proof of soundness.

**Example: Access control policy.** Figure 3 illustrates a simple, but complete, enforcement policy for access control. Protected data is given a label listing those users authorized to access the data. In particular, such data has type $t\{acl\}$ where *acl* encodes the ACL as a label.

The policy's *login* function calls an external function *checkpw* to authenticate a user by checking a password. If authentication succeeds (the first pattern), *checkpw* returns a label *USER*($k$) where $k$ is some unique identifier for the user. The *login* function returns a pair consisting of this label and a integer labeled with it; this pair serves as our runtime representation of a principal. The *access* function takes the two elements of this pair as its first two arguments. Since FABLE enforces that only policies can produce labeled values, we are assured that the term with type $\mathsf{int}\{USER(k)\}$ can only have been produced by *login*. The *access* function's last two arguments consist of the protected data's label, *acl*, and the data itself, *data*. The *access* function calls the *member* function to see whether the user token $u$ is present in the ACL. If successful, the label *TRUE* is returned, in which case *access* returns the data with its *acl* label removed.

## 2.2. Typing

Figure 4 defines the typing rules for FABLE. The main judgment $\Gamma \vdash_c e : t$ types expressions. The index $c$ indicates whether $e$ is part of the policy or the application. Only policy terms are permitted to use the unlabeling and relabeling operators. $\Gamma$ records three kinds of information; $x{:}t$ maps variables to types, $\alpha$ records a bound type variable, and $e \succ p$ records the assumption that $e$ matches pattern $p$, used when checking the branches of a pattern match.

The rules (T-INT), (T-VAR), (T-FIX), (T-TAB) and (T-TAP) are standard for polymorphic lambda calculi. (T-ABS) and (T-APP) are standard for a dependently typed language. (T-ABS) introduces a dependent function type of the form $(x{:}t_1) \to t_2$. (T-APP) types an application of a (dependently typed) function. As usual, we require the type $t_1$ of the argument to match the type of the formal parameter to the function. However, since $x$ may occur in the return type $t_2$, the type of the application must substitute the actual argument $e_2$ for $x$ in $t_2$. As an example, consider an application of the *access_simple* function, having type $(acl{:}\mathsf{lab}) \to \mathsf{int}\{acl\} \to \mathsf{int}$, to the term $ACL(Alice,Bob)$. According to (T-APP) the resulting expression is a function with type $\mathsf{int}\{ACL(Alice,Bob)\} \to \mathsf{int}$, which indicates that the function can be applied only to an integer labeled with precisely $ACL(Alice,Bob)$. This is the key feature of dependent typing—the type system ensures that associations between labels and the terms they protect cannot be forged or broken.

Rule (T-LAB) gives a label term $C(\vec{e})$ a singleton label type $\mathsf{lab}{\sim}C(\vec{e})$ as long as each component $e_i \in \vec{e}$ has type $\mathsf{lab}$. According to this rule $ACL(Alice,Bob)$ can be given the type $\mathsf{lab}{\sim}ACL(Alice,Bob)$. For that matter, the expression $((\lambda x{:}\mathsf{lab}.x)\ High)$ can be given the type $\mathsf{lab}{\sim}((\lambda x{:}\mathsf{lab}.x)\ High)$; there is no requirement that $e$ be a value. The rule (T-HIDE) allows a singleton label type like this one to be subsumed to the type of all labels, $\mathsf{lab}$. Rule (T-SHOW) does the converse, allowing the type of a label to be made more precise.

Rule (T-MATCH) checks pattern matching. The first premise confirms that expression $e$ being matched is a label. The second line of premises describes how to check each branch of the match. Our patterns differ from patterns in, say, ML in two respects. First, the second premise on the second line requires $\Gamma, \vec{x}_i : \mathsf{lab} \vdash_c p_i : lab$, indicating that patterns in FABLE are allowed to contain variables that are defined in the context $\Gamma$. Second, pattern variables may occur more than once in a pattern. Both of these features make it convenient to use pattern matching to check for term equality. For example, in the expression $\mathsf{let}\ y = Alice\ \mathsf{in\ match}\ x\ \mathsf{with}\ ACL(y{,}y) \Rightarrow e$, the branch

```
policy login(user:string, pw:string) =
  let token = match checkpw user pw with
    USER(k) ⇒ USER(k)
    _ ⇒ FAILED in
  (token, {token}0)
```

```
let member(u:lab, a:lab) =
  match a with
    ACL(u, i) ⇒ TRUE
    ACL(j, tl) ⇒ member u tl
    _ ⇒ FALSE
```

```
policy access⟨k,α⟩(u:lab ∼ USER(k),
        cap:int{u}, acl:lab, data:α{acl}) =
  match member u acl with
    TRUE ⇒ {∘}data
    _ ⇒ halt#access denied
```

**Figure 3. Enforcing a simple access control policy**

$e$ is evaluated only if the runtime value for the label variable $x$ is *ACL*(*Alice*, *Alice*).

A key feature of (T-MATCH) is the final premise on the second line, which states that the body of each branch expression $e_i$ should be checked in a context including the assumption $e \succ p_i$, which states that $e$ matches pattern $p_i$. This assumption can be used to refine type information during checking (similar to typecase [19]) using the rule (T-CONV), which we illustrate shortly. (T-MATCH) also requires that variables bound by patterns do not escape their scope by appearing in the final type of the match; this is ensured by the second premise, $\Gamma \vdash t$, which confirms $t$ is well formed in the top-level environment (i.e., one not including pattern-bound variables). For simplicity we require a default case in pattern-matching expressions: the third premise requires the last pattern to be a single variable $x$ that does not occur in $\Gamma$.

Rule (T-UNLAB) types an unlabeling operation. Given an expression $e$ with type $t\{e'\}$, the unlabeling of $e$ strips off the label on the type to produce an expression with type $t$. Conversely, (T-RELAB) adds a label $e'$ to the type of $e$. The *pol*-index on these rules indicates that both operations are only admissible in policy terms. This index is introduced by (T-POL) when checking the body of a bracketed term $(\!|e|\!)$. For example, given expression $e \equiv \lambda x{:}\mathsf{int}\{Public\}.(\!|\{\circ\}x|\!)$, we have $\cdot \vdash_{app} e : \mathsf{int}\{Public\} \to \mathsf{int}$ since $\{\circ\}x$ will be typed with index *pol* by (T-POL).

Rule (T-CONV) allows $e$ to be given type $t'$ assuming it can given type $t$ where $t$ and $t'$ are *convertible*, written $\Gamma \vdash t \cong t'$. Rules (TE-ID) and (TE-SYM) define convertibility to be reflexive and symmetric. Rule (TE-CTX) structurally extends convertibility using *type contexts* $T$. The syntax $T \cdot t$ denotes the application of context $T$ to a type $t$ which defines the type that results from replacing the occurrence of the hole $\bullet$ in $T$ with $t$. For example, if $T$ is the context $\bullet\{C\}$, then $T \cdot \mathsf{int}$ is the type $\mathsf{int}\{C\}$. (Of course, rule (TE-CTX) can be applied several times to relate larger types.)

The most interesting rules are (TE-REFINE) and (TE-REDUCE), which consider types that contain labels (constructed by applying context $L$ to an expression $e$). Rule (TE-REFINE) allows two structurally similar types to be considered equal if their embedded expressions $e$ and $p$ have been equated by pattern matching, recorded as the constraint $e \succ p$ by (T-MATCH). To see how this would be used, consider the following example:

```
let tok,cap = login "Joe" "xyz" in
  match tok with USER(k) ⇒ access tok cap
                 _ ⇒ halt
```

We give the *login* function the type $\mathsf{string} \to \mathsf{string} \to (l{:}\mathsf{lab} \times \mathsf{int}\{l\})$. The type of *access* (defined in Figure 3) is $(u{:}\mathsf{lab} \sim USER(k)) \to \mathsf{int}\{u\} \to t$. We type check *access tok* using rule (T-APP), which requires that the function's parameter and its formal argument have the same type $t$. However, here *tok* has type lab while *access* expects type $\mathsf{lab} \sim USER(k)$. Since the call to *access* occurs in the first branch of the match, the context includes the refinement $tok \succ USER(k)$ due to (T-MATCH). From (T-SHOW) we can give *tok* type $\mathsf{lab} \sim tok$, and by applying (TE-REFINE) we have $\mathsf{lab} \sim tok \cong \mathsf{lab} \sim USER(k)$ and so *tok* can be given type $\mathsf{lab} \sim USER(k)$ as required. Similarly, for *access tok cap*, we can check that the type $\mathsf{int}\{tok\}$ of *cap* is convertible with $\mathsf{int}\{USER(k)\}$ in the presence of the same assumption.

Rule (TE-REDUCE) allows FABLE types to be considered convertible if the expression component of one is reducible to the expression component of the other [2]; reduction $e \overset{c}{\rightsquigarrow} e'$ is defined shortly in Figure 4. For example, we have $\cdot \vdash int\{(\lambda x{:}\mathsf{lab}.x)\ Low\} \cong int\{Low\}$ since $(\lambda x{:}\mathsf{lab}.x)\ Low \overset{c}{\rightsquigarrow} Low$. One complication is that type-level expressions may contain free variables. For example, suppose we wanted to show $y : \mathsf{lab} \vdash int\{(\lambda x{:}\mathsf{lab}.x)\ y\} \cong int\{y\}$. It seems intuitive that these types should be considered convertible, but we do not have that $(\lambda x{:}\mathsf{lab}.x)\ y \overset{c}{\rightsquigarrow} y$ because $y$ is not a value. To handle this case, the rule permits two types to be convertible if, for every well-typed substitution $\sigma$ of the free variables of $e_1$, $\sigma(e_1) \overset{c}{\rightsquigarrow} \sigma(e_2)$. This captures the idea that the precise value of $y$ is immaterial—all reductions on well-typed substitutions of $y$ would reduce to the value that was substituted for $y$.

Satisfying this obligation by exhaustively considering all possible substitutions is obviously intractable. Additionally, we have no guarantee that an expression appearing in a type will converge to a value. Thus, type checking in FABLE, as presented here, is undecidable. This is not uncommon in a dependent type system; e.g., type checking in Cayenne is undecidable [3]. However, other dependently typed systems impose restrictions on the usage of recursion in type-level expressions to ensure that type-level terms always terminate [5]. Additionally, there are several possible decision

$\Gamma \vdash_c e : t$ Expression $e$ has type $t$ in environment $\Gamma$ under color $c$

$$\begin{array}{rcll} \text{Environments} & \Gamma & ::= & \cdot \mid x{:}t \mid \alpha \mid e \succ p \mid \Gamma_1, \Gamma_2 \\ \text{Substitutions} & \sigma & ::= & \cdot \mid (x \mapsto e) \mid (\alpha \mapsto t) \mid \sigma_1, \sigma_2 \\ \text{Colors} & c & ::= & pol \mid app \end{array} \qquad \Gamma \vdash_c n : \mathsf{int}\ \text{(T-INT)} \qquad \frac{x{:}t \in \Gamma}{\Gamma \vdash_c x : t}\ \text{(T-VAR)}$$

$$\frac{\Gamma \vdash t \qquad \Gamma, f{:}t \vdash_c v : t}{\Gamma \vdash_c \mathsf{fix}\ f{:}t.v : t}\ \text{(T-FIX)} \qquad \frac{\Gamma, \alpha \vdash_c e : t}{\Gamma \vdash_c \Lambda\alpha.e : \forall\alpha.t}\ \text{(T-TAB)} \qquad \frac{\Gamma \vdash t \qquad \Gamma \vdash_c e : \forall\alpha.t'}{\Gamma \vdash_c e[t] : (\alpha \mapsto t)t'}\ \text{(T-TAP)}$$

$$\frac{\Gamma \vdash t \qquad \Gamma, x{:}t \vdash_c e : t'}{\Gamma \vdash_c \lambda x{:}t.e : (x{:}t) \to t'}\ \text{(T-ABS)} \qquad \frac{\Gamma \vdash_c e_1 : (x{:}t_1) \to t_2 \qquad \Gamma \vdash_c e_2 : t_1}{\Gamma \vdash_c e_1\, e_2 : (x \mapsto e_2)t_2}\ \text{(T-APP)}$$

$$\frac{\Gamma \vdash_c e_i : \mathsf{lab}}{\Gamma \vdash_c C(\vec{e}) : \mathsf{lab} \sim C(\vec{e})}\ \text{(T-LAB)} \qquad \frac{\Gamma \vdash_c e : \mathsf{lab} \sim e'}{\Gamma \vdash_c e : \mathsf{lab}}\ \text{(T-HIDE)} \qquad \frac{\Gamma \vdash_c e : \mathsf{lab}}{\Gamma \vdash_c e : \mathsf{lab} \sim e}\ \text{(T-SHOW)}$$

$$\frac{\begin{array}{ccc} \Gamma \vdash_c e : \mathsf{lab} & \Gamma \vdash t & p_n = x \quad \text{where } x \notin \mathrm{dom}(\Gamma) \\ \vec{x}_i = FV(p_i) \setminus \mathrm{dom}(\Gamma) & \Gamma, \vec{x}_i{:}\mathsf{lab} \vdash_c p_i : \mathsf{lab} & \Gamma, \vec{x}_i{:}\mathsf{lab}, e \succ p_i \vdash_c e_i : t \end{array}}{\Gamma \vdash_c \mathsf{match}\ e\ \mathsf{with}\ p_1 \Rightarrow e_1 \ldots p_n \Rightarrow e_n : t}\ \text{(T-MATCH)} \qquad \frac{\Gamma \vdash_{pol} e : t\{e'\}}{\Gamma \vdash_{pol} \{\circ\}e : t}\ \text{(T-UNLAB)}$$

$$\frac{\Gamma \vdash_{pol} e : t \qquad \Gamma \vdash_{pol} e' : \mathsf{lab}}{\Gamma \vdash_{pol} \{e'\}e : t\{e'\}}\ \text{(T-RELAB)} \qquad \frac{\Gamma \vdash_{pol} e : t}{\Gamma \vdash_c (\!| e |\!) : t}\ \text{(T-POL)} \qquad \frac{\Gamma \vdash_c e : t \qquad \Gamma \vdash t \cong t'}{\Gamma \vdash_c e : t'}\ \text{(T-CONV)}$$

$\Gamma \vdash t \cong t'$ Types $t$ and $t'$ are convertible

$$\begin{array}{rcll} \text{Type contexts} & T & ::= & \bullet \mid \bullet\{e\} \mid x{:}\bullet \to t \mid x{:}t \to \bullet \mid \forall\alpha.\bullet \\ \text{Term label contexts} & L & ::= & \mathsf{lab} \sim \bullet \mid t\{\bullet\} \end{array}$$

$$\Gamma \vdash t \cong t\ \text{(TE-ID)} \qquad \frac{\Gamma \vdash t \cong t'}{\Gamma \vdash t' \cong t}\ \text{(TE-SYM)} \qquad \frac{\Gamma \vdash t \cong t'}{\Gamma \vdash T \cdot t \cong T \cdot t'}\ \text{(TE-CTX)} \qquad \frac{e \succ p \in \Gamma}{\Gamma \vdash L \cdot e \cong L \cdot p}\ \text{(TE-REFINE)}$$

$$\frac{\forall\sigma.(\mathrm{dom}(\sigma) = FV(e_1)\ \wedge\ \Gamma \vdash \sigma(e_1) : \mathsf{lab})\ \Rightarrow\ \sigma(e_1) \overset{c}{\rightsquigarrow} \sigma(e_2)}{\Gamma \vdash L \cdot e_1 \cong L \cdot e_2}\ \text{(TE-REDUCE)}$$

$\Gamma \vdash t$ Type $t$ is well-formed in environment $\Gamma$

$$\Gamma \vdash \mathsf{int}\ \text{(K-INT)} \qquad \frac{\alpha \in \Gamma}{\Gamma \vdash \alpha}\ \text{(K-TVAR)} \qquad \Gamma \vdash \mathsf{lab}\ \text{(K-LAB)} \qquad \frac{\Gamma \vdash_{pol} e : \mathsf{lab}}{\Gamma \vdash \mathsf{lab} \sim e}\ \text{(K-SLAB)}$$

$$\frac{\Gamma \vdash t \qquad \Gamma \vdash_{pol} e : \mathsf{lab}}{\Gamma \vdash t\{e\}}\ \text{(K-LABT)} \qquad \frac{\Gamma \vdash t_1 \qquad \Gamma, x{:}t_1 \vdash t_2}{\Gamma \vdash (x{:}t_1) \to t_2}\ \text{(K-FUN)} \qquad \frac{\Gamma, \alpha \vdash t}{\Gamma \vdash \forall\alpha.t}\ \text{(K-ALL)}$$

**Figure 4. Static semantics of FABLE**

procedures that can be used to partially decide type convertibility. One simplification would be to attempt to show convertibility for closed types only—i.e. no free variables. In our implementation of FABLE, SELINKS, we use a combination of three techniques. First, we use type information. If $l$ is free in a type, and the declared type of $l$ is $\mathsf{lab} \sim e$, then we can use this information to substitute $e$ for $l$. Similarly, if the type context includes an assumption of the form $l \succ e$ (when checking the branch of a pattern), we can substitute $l$ with $e$. Finally, since type-level expressions typically manipulate labels by pattern matching, we use a simple heuristic to determine which branch to take when pattern matching expressions with free variables. These techniques suffice for all the examples in this paper and both our SEWIKI and SEWINESTORE applications. Our technical report [38] discusses these decision procedures in greater detail and proves them sound.

Finally, the judgment $\Gamma \vdash t$ states that $t$ is well-formed in $\Gamma$. Rules (K-INT), (K-TVAR), and (K-LAB) are standard, (K-FUN) defines the standard scoping rules for names in dependent function types, and (K-ALL) defines the standard scoping rule for universally quantified type variables. (K-SLAB) and (K-LABT) ensure that all expressions $e$ that appear in types can be given $\mathsf{lab}$-type. Notice that type-level expressions are typed in *pol*-context. Because FABLE enjoys a type-erasure property any (un)labeling operations appearing in types pose no security risk. We use this feature to good effect in Section 3.2 to protect sensitive information that may appear in labels.

### 2.3. Operational Semantics

Figure 5 defines FABLE's operational semantics. We define a pair of small-step reduction relations $e \overset{app}{\rightsquigarrow} e'$ and $e \overset{pol}{\rightsquigarrow} e'$ for application and policy expressions, respectively. Rules of the form $e \overset{c}{\rightsquigarrow} e'$ are *polychromatic*—they apply both to policy and application expressions. Since the values for each kind of expression are different, we also parameterize the evaluation contexts $E_c$ by the color of the expression; i.e., the context, either *app* or *pol*, in which the expression is to be reduced. Rule (E-CTX) uses these evaluation contexts $E_c$, similar to the type contexts used above, to enforce a left-to-right evaluation order for a call-by-value semantics. Policy expression reduction $e \overset{pol}{\rightsquigarrow} e'$ takes place within brackets according to (E-POL). The rules (E-APP), (E-TAP), and (E-FIX) define function application, type application, and fixed-point expansion, respectively, in terms of substitutions; these are all standard. Rule (E-MATCH) relies on a standard pattern-matching judgment $v \succ p : \sigma$, also defined in Figure 5, which is true when the label value matches the pattern such that $v = \sigma(p)$. (E-MATCH) determines the first pattern $p_j$ that matches the expression $v$ and reduces the match expression to the matched branch's body after applying the substitution. The (U-CON) rule in the pattern-matching judgment $v \succ p : \sigma$ is the only non-trivial rule. As explained in Section 2.2, since pattern variables may occur more than once in a pattern, (U-CON) must propagate the result of matching earlier sub-expressions when matching subsequent sub-expressions. For example, pattern matching should fail when attempting to match $ACL(Alice, Bob)$ with $ACL(x, x)$. This is achieved in (U-CON) because, after matching $(Alice \succ x : x \mapsto Alice)$ using (U-VAR), we must try to match $Bob$ with $(x \mapsto Alice)x$, which is impossible.

An applied policy function will eventually reduce to a bracketed policy value $v_{pol}$. When $v_{pol}$ has the form $(\!|u|\!)$, the brackets may be removed so that the value $u$ can be used by application code. (E-BLAB) and (E-BINT) handle label expressions $(\!|C(\vec{u})|\!)$ and integers $n$, respectively. To maintain the invariant that (un)labeling operators only appear in policy code, rules (E-BABS) and (E-TABS) extrude only the $\lambda$ and $\Lambda$ binders, respectively, from bracketed abstractions, allowing them to be reduced according to (E-APP) or (E-TAP). Brackets cannot be removed from labeled values $(\!|\{e\}u|\!)$ by application code, to preserve the labeling invariant. On the other hand, brackets can be removed from any expression by policy code, according to (E-NEST). This is useful when reducing expressions such as $(\!|\lambda x{:}t.x|\!)(\!|v|\!)$, which produces $(\!|(\!|v|\!)|\!)$ after two steps; (E-NEST) (in combination with (E-POL)) can then remove the inner brackets. Finally, (E-UNLAB) allows an unlabeling operation to annihilate the top-most relabeling operation. Notice that the expressions within a relabeling operation are never evaluated at runtime—relabelings only affect the types and are purely compile time entities. The types that appear elsewhere, such as (E-TAP), are also erasable, as is usual for System F.

### 2.4. Soundness

We state the standard type soundness theorems for FABLE here. In addition to ensuring that well-typed programs never go wrong or get stuck, we have put this soundness result to good use in proving that security policies encoded in FABLE satisfy desirable security properties. We discuss this further in the next section. Our technical report [38] contains proofs of the following theorems for an extension of FABLE that includes references and substructural types.

**Theorem 1** (Type soundness). *If* $\cdot \vdash_c e : t$; *then either* $\exists e'.e \overset{c}{\rightsquigarrow} e'$ *or* $\exists v_c.e = v_c$. *Furthermore, if* $e \overset{c}{\rightsquigarrow} e'$; *then,* $\cdot \vdash_c e' : t$.

## 3 Example Policies in FABLE

This section uses FABLE to encode several security policies. We prove that any well-typed program using one of these policies enjoys relevant security properties—i.e., the program is sure to enforce the policy correctly. Space constraints preclude presentation of all of the encodings we have explored, so we focus on three kinds of policies: access control, provenance, and static information flow. We conclude by discussing how FABLE's design eases the construction of proofs of policy correctness.

### 3.1. Access Control Policies

Access control policies govern how programs release information but, once the information is released, do not control how it is used. To prove that an access control policy is implemented correctly, we must show that programs not authorized to access some information cannot learn the information in any way, e.g., by bypassing a policy check (something not uncommon in production systems [34]) or by exploiting leaks due to control-flow or timing channels. We call this security condition *non-observability*.

Intuitively, we can state non-observability as follows. If some program $P$ is not allowed to access a resource $v_1$ having a label $l$, then a program $P'$ that is identical to $P$ except that $v_1$ has been replaced with some other resource $v_2$

$e \overset{c}{\leadsto} e'$ Small-step chromatic reduction rules

$$\text{Evaluation contexts} \quad E_c \quad ::= \quad \bullet e \mid v_c \bullet \mid \bullet[t] \mid C(\vec{v}_c, \bullet, \vec{e}) \mid \textsf{match}\,\bullet\,\textsf{with}\; p_i \Rightarrow e_i \mid \{e\}\bullet \mid \{\circ\}\bullet$$

$$\frac{e \overset{c}{\leadsto} e'}{E_c \cdot e \overset{c}{\leadsto} E_c \cdot e'}\ \text{(E-CTX)} \qquad \frac{e \overset{pol}{\leadsto} e'}{(\!|e|\!) \overset{app}{\leadsto} (\!|e'|\!)}\ \text{(E-POL)}$$

$$(\lambda x{:}t.e)\; v_c \overset{c}{\leadsto} (x \mapsto v_c)e\ \text{(E-APP)} \qquad (\Lambda\alpha.e)\;[t] \overset{c}{\leadsto} (\alpha \mapsto t)e\ \text{(E-TAP)} \qquad \textsf{fix}\; f{:}t.v \overset{c}{\leadsto} (f \mapsto \textsf{fix}\; f{:}t.v)v\ \text{(E-FIX)}$$

$$\frac{\forall i < j.\, v_c \not\succ p_i : \sigma_i \qquad v_c \succ p_j : \sigma_j}{\textsf{match}\; v_c\; \textsf{with}\; p_1 \Rightarrow e_1 \ldots p_n \Rightarrow e_n \overset{c}{\leadsto} \sigma_j(e_j)}\ \text{(E-MATCH)} \qquad (\!|C(\vec{u})|\!) \overset{app}{\leadsto} C(\vec{u})\ \text{(E-BLAB)} \qquad (\!|n|\!) \overset{app}{\leadsto} n\ \text{(E-BINT)}$$

$$(\!|\lambda x{:}t.e|\!) \overset{app}{\leadsto} \lambda x{:}t.(\!|e|\!)\ \text{(E-BABS)} \qquad (\!|\Lambda\alpha.e|\!) \overset{app}{\leadsto} \Lambda\alpha.(\!|e|\!)\ \text{(E-BTAB)} \qquad (\!|e|\!) \overset{pol}{\leadsto} e\ \text{(E-NEST)} \qquad \{\circ\}\{e\}v_{pol} \overset{pol}{\leadsto} v_{pol}\ \text{(E-UNLAB)}$$

$e \succ p : \sigma$ Expression $e$ matches pattern $p$ under substitution $\sigma$

$$p \succ p : \cdot\ \text{(U-PATID)} \qquad v \succ x : x \mapsto v\ \text{(U-VAR)} \qquad \frac{\forall i.\sigma_i^* = (\sigma_0, \ldots, \sigma_{i-1}) \qquad e_i \succ \sigma_i^* p_i : \sigma_i}{C(\vec{e}) \succ C(\vec{p}) : \vec{\sigma}}\ \text{(U-CON)}$$

**Figure 5. Dynamic semantics of** FABLE

(having the same type and label as $v_1$) should evaluate in the same way as $P$—it should produce the same result and take the same steps along the way toward producing that result. If this were not true then, assuming $P$'s reduction is deterministic, $P$ must be inferring information about the protected resource.

To make this intuition formal, we will show that the evaluations of programs $P$ and $P'$ are *bisimilar*, where the only difference between them is the value of the protected resource. To express this, first we define an equivalence relation called *similarity up to l* (analogous to definitions of low equivalence [32, 7]) which holds for two terms $e$ and $e'$ if they only differ in sub-terms that are labeled with $l$, with the intention that $l$ is the label of restricted resources.

**Definition 2** (Similarity up to $l$)**.** *Expressions e and $e'$, identified up to $\alpha$-renaming, are similar up to label l according to the following relation:*

$$e \sim_l e \qquad \{l\}e \sim_l \{l\}e' \qquad \frac{e \sim_l e' \quad l' \neq l}{\{l'\}e \sim_l \{l'\}e'}$$

$$\frac{e \sim_l e'}{\lambda x{:}t.e \sim_l \lambda x{:}t.e'} \qquad \frac{e_1 \sim_l e_1' \quad e_2 \sim_l e_2'}{e_1\; e_2 \sim_l e_1'\; e_2'} \qquad \ldots$$

The second rule is the most important. It states that arbitrary expressions $e$ and $e'$ are considered similar at label $l$ when both are labeled with $l$. Other parts of the program must be structurally identical, as stated by the remaining congruence rules (not all are shown; the full relation can be found in our technical report [38]). We extend similarity to a bisimulation as follows: two similar terms are bisimilar if they always reduce to similar subterms, and do so indefinitely or until no further reduction is possible. This notion of bisimulation is the basis of our access control security theorem; it is both *timing and termination sensitive*.

**Definition 3** (Bisimulation)**.** *Expressions $e_1$ and $e_2$ are* bisimilar at label $l$, *written $e_1 \approx_l e_2$, if and only if $e_1 \sim_l e_2$ and there exists $e_1', e_2'$ such that $e_1 \overset{c}{\leadsto} e_1' \Leftrightarrow e_2 \overset{c}{\leadsto} e_2'$ and $e_1' \approx_l e_2'$.*

**Theorem 1** (Non-observability)**.** *Given a $(\!|\cdot|\!)$-free expression e such that $(a{:}t_a, m{:}t_m, cap{:}\mathsf{int}\{user\}, x{:}t\{acl\} \vdash_{app} e : t_e)$ where acl and user are label constants, and given a substitution $\sigma = (a \mapsto access, m \mapsto member, cap \mapsto (\!|\{user\}0|\!))$. Then, for type-respecting substitutions $\sigma_i = \sigma, x \mapsto v_i$ where $\cdot \vdash_{app} v_i : t\{acl\}$ for i=1,2, we have $(member\; user\; acl \overset{c*}{\leadsto} False) \Rightarrow \sigma_1(e) \approx_{acl} \sigma_2(e)$.*

This theorem is concerned with a program $e$ that contains no policy-bracketed terms (it is just application code) but, via the substitution $\sigma$, may refer to our access control functions *access* and *member* through the free variables $a$ and $m$. Additionally, the program is granted a single user capability $(\!|\{user\}0|\!)$ through the free variable *cap* which gives the program the authority of user *user*. The program may also refer to some protected resource $x$ whose label is *acl*, but the authority of *user* is insufficient to access $x$ according to the access control policy because $(member\; user\; acl \overset{c*}{\leadsto} False)$. Under these conditions, we can show that for any two (well-typed) $v_i$ we substitute for $x$ according to substitution $\sigma_i$, the resulting programs are bisimilar—their reduction is independent of the choice of $v_i$.

Notice that this theorem is indifferent to the actual implementation of the *acl* label and the *member* function. Thus, while our example policy is fairly simplistic, a far more sophisticated model could be used. For instance, we could have chosen labels to stand for RBAC- or RT-style roles [23] and *member* could invoke a decision procedure for determining role membership. Likewise, the theorem is not concerned with the origin of the *user* capability—a function more sophisticated than *login* (e.g., that relied on cryptography) could have been used. The important point is that FABLE ensures the second component of the user credential ($l$:lab$\sim$ *USER*($k$) $\times$ int$\{l\}$) is unforgeable by application code. Finally, it would be straightforward to extend our theorem to speak to policies that provide access to more than one resource with a single membership test, as in the following code

$$\begin{array}{l}\textsf{policy}\ \mathit{access\_cap}\langle k\rangle(u{:}\textsf{lab}\sim \mathit{USER}(k),\ \mathit{cred}{:}\textsf{int}\{u\},\ \mathit{acl}{:}\textsf{lab}) = \\ \quad \textsf{match}\ \mathit{member}\ u\ \mathit{acl}\ \textsf{with}\ \mathit{True} \Rightarrow \Lambda\alpha.\lambda x{:}\alpha\{\mathit{acl}\}.\{\circ\}x \\ \qquad\qquad\qquad\qquad\qquad\qquad \_ \Rightarrow \#\mathit{fail}\end{array}$$

Here the caller presents a user credential and an access control label *acl* (but no resource labeled with that label). If the membership check succeeds, a function with type $\forall\alpha.\alpha\{acl\} \rightarrow \alpha$ is returned. This function can be used to immediately unlabel any resource with the authorized label. This is useful when policy queries are expensive. It is also useful for encoding a form of delegation; rather than releasing his user credential, a user could release a function that uses that credential to a limited effect. Of course, this may be undesirable if the policy is known to change frequently, but even this could be accommodated. Variations that combine static and dynamic checks are also possible.

## 3.2. Dynamic Provenance Tracking

*Provenance* is "information recording the source, derivation, or history of some information" [7]. Provenance is relevant to computer security for at least two reasons. First, provenance is useful for auditing, e.g., to discover whether some data was inappropriately released or modified. Second, provenance can be used to establish data integrity, e.g., by carefully accounting for a document's sources. This section describes a label-based provenance tracking policy we constructed in FABLE. To prove that this policy is implemented correctly we show that all programs that use it will accurately capture the dependences (in the sense of information flow) on a value produced by a computation.

Figure 6 presents the provenance policy. We define the type *Prov* $\alpha$ to describe a pair in which the first component is a label $l$ that records the provenance of the second component. The policy is agnostic to the actual form of $l$. Provenance labels could represent things like authorship, ownership, the channel on which information was received, etc. An interesting aspect of *Prov* $\alpha$ is that the provenance label is itself labeled with the 0-ary label constant *Auditors*. This represents the fact that provenance information is subject to security concerns like confidentiality and integrity. Intuitively, one can think of data labeled with the *Auditors* label as only accessible to members of a group called *Auditors* (of course, a more complex policy could be used). Finally, note that because the provenance label $l$ is itself labeled (having type lab$\{Auditors\}$) it would be incorrect to write $\alpha\{l\}$ as the second component of the type since this requires that $l$ have type lab. Therefore we unlabel $l$ when it appears in the type of the second component. As explained in Section 2.2, unlabeling operations in types pose no security risk since the types are erased at runtime.

The policy function *apply* is a wrapper for tracking dependences through function applications. In an idealized language like FABLE it is sufficient to limit our attention to function application, but a policy for a full language would define wrappers for other constructs as well. The first argument of *apply* is a provenance-labeled function *lf* to be called on the second argument *mx*. The body of *apply* first decomposes the pair *lf* into its label $l$ and the function $f$ itself and does likewise for the argument *mx*. Then it applies the function, stripping the label from both it and its argument first. The provenance of the result is a combination of the provenance of the function and its argument. We write this as the label pair $Union(l,m)$ which is then associated with the final result. Notice that we strip the *Auditors* labels from $l$ and $m$ before combining them, and then add the label to the label of the final result.

The policy also defines a function *flatten* to convert a value of type *Prov* (*Prov* $\alpha$) to one of type *Prov* $\alpha$ by extracting the nested labels (the first two lines) and then collapsing them into a *Union* (third line) that is associated with the inner pair's labeled component (fourth line).

An example client program that uses this provenance policy is the following:

$$\begin{array}{l}\textsf{let}\ \mathit{client}\langle\alpha,\beta,\gamma\rangle\ (f : \mathit{Prov}(\alpha\rightarrow\beta\rightarrow\gamma),\ x : \mathit{Prov}\ \alpha,\ y : \mathit{Prov}\ \beta) = \\ \qquad \mathit{apply}\ [\beta][\gamma]\ (\mathit{apply}\ [\alpha][\beta\rightarrow\gamma]\ f\ x)\ y\end{array}$$

This function takes a labeled two-argument function $f$ as its argument and the two arguments $x$ and $y$. It calls *apply* twice to get a result of type *Prov* $\gamma$. This will be a tuple in which the first component is a labeled provenance label of the form *Union*(*Union*(*lf*,*lx*), *ly*) and the second component is a value labeled with that provenance label. In the label, we will have that *lf* is the provenance label of the function argument $f$ and *lx* and *ly* are the provenance of the arguments $x$ and $y$, respectively. Note that a caller of *client* can instantiate the type variable $\gamma$ to be a type like *Prov* int. In this case, the type of the returned value will be *Prov* (*Prov* int), which can be flattened if necessary.

We can prove that provenance information is tracked correctly following Cheney et al. [7]. The intention is that if a value $x$ of type *Prov* $\alpha$ influences the computation of some other value $y$, then $y$ must have type *Prov* $\beta$ (for some $\beta$) and its provenance label must mention the provenance label of $x$. If provenance is tracked correctly, a change to $x$ will only affect values like $y$; other values in the program will be unchanged. We can express this using a similarity relation $v_1 \sim_l v_2$ like the one defined in Section 3.1 which relates

```
typename Prov α= (l:lab{Auditors} × α{{∘}l})
policy flatten⟨α⟩ (x:Prov (Prov α)) =
    let l,inner = x in
    let m,a = inner in
    let lm = Union({∘}l, {∘}m) in
      ({Auditors}lm, {lm}a)
```

```
policy apply⟨α,β⟩ (lf:Prov (α → β), mx:Prov α) =
    let l,f = lf in
    let m,x = mx in
    let y = ({∘}f) ({∘}x) in
    let lm = Union({∘}l, {∘}m) in
      ({Auditors}lm, {lm}y)
```

**Figure 6. Enforcing a dynamic provenance-tracking policy**

two values if they differ only on sub-terms of type *Prov α* whose provenance label mentions *l*. Thus, an application program *e* that is compiled with the policy of Figure 6 and is executed in contexts that differ only in the choice of a tracked value of label *l* will compute results that differ only in sub-terms that are also colored using *l*.

**Theorem 2** (Dependency correctness)**.** *Given a* $(\!|\cdot|\!)$*-free expression e such that* $a : t_a, f : t_f, x : Prov\ t \vdash_{app} e : t'$*, and given a substitution* $\sigma = (a \mapsto apply, f \mapsto flatten)$*. Then, for type-respecting substitutions* $\sigma_i = \sigma, x \mapsto v_i$ *where* $\vdash_{app} v_i : Prov\ t$ *for i=1,2 it is the case that* $v_1 \sim_l v_2$ *implies* $(\sigma_1(e) \overset{app*}{\rightsquigarrow} v'_1 \ \wedge\ \sigma_2(e) \overset{app*}{\rightsquigarrow} v'_2) \Rightarrow v'_1 \sim_l v'_2$

## 3.3. Static Information Flow

Both policies discussed so far rely on runtime checks. This section illustrates how FABLE can be used to encode *static* lattice-based information flow policies that require no runtime checks. In a static information flow type system (as found in FlowCaml [32]) labels *l* have no run-time witness; they only appear in types $t\{l\}$. Labels are ordered by a relation $\sqsubseteq$ that typically forms a lattice. This ordering is lifted to a subtyping relation on labeled types such that $l_1 \sqsubseteq l_2 \Rightarrow t\{l_1\} <: t\{l_2\}$. Assuming the lattice ordering is fixed during execution, well-typed programs can be proven to adhere to the policy defined by the initial label assignment appearing in the types.

Figure 7 illustrates the policy functions, along with a small sample program. In our encoding we define a two-point security lattice with atomic labels *HIGH* and *LOW* and protected expressions will have labeled types like $t\{HIGH\}$. The ordering $LOW \sqsubset HIGH$ is exemplified by the *lub* (least upper bound) operation for the lattice. The *join* function (similar to the *flatten* function from Figure 6) combines multiple labels on a type into a single label. The interesting thing here is the label attached to *x* is a label expression *lub l m*, rather than an label value like *HIGH*. The type rule (T-CONV) presented in Figure 4 can be used to show that a term with type int{*lub HIGH LOW*} can be given type int{*HIGH*} (since $lub\ HIGH\ LOW \overset{c}{\rightsquigarrow} HIGH$). This is critical to being able to type programs that use this policy.

The policy includes a subsumption function *sub*, which takes as arguments a term *x* with type $\alpha\{l\}$ and a label

```
policy lub(x:lab, y:lab) = match x,y with
    _, HIGH | HIGH, _ ⇒ HIGH | _, _ ⇒ LOW
policy join⟨α,l,m⟩ (x:α{l}{m}) = ({lub l m}{∘}{∘}x)
policy sub⟨α,l⟩ (x:α{l}, m:lab) = ({lub l m}{∘}x)
policy apply⟨α,β,l,m⟩ (f:(α → β){l}, x:α) = {l}(({∘}f) x)
```

**Figure 7. Enforcing an information flow policy**

*m* and allows *x* to be used at the type $\alpha\{lub\ l\ m\}$. This is a restatement of the subsumption rule above, as $l \sqsubseteq m$ implies $l \sqcup m = m$. (Once types are erased, *join* and *sub* are both essentially the identity function and could be optimized away.) Finally, the policy function *apply* unlabels the function *f* in order to call it, and then adds *f*'s label on the computed result.

Consider the following client program as an example usage of the static information flow policy.

```
let client (f:(int{HIGH} → int{HIGH}){LOW}, x:int{LOW}) =
  let x = (sub [int] x HIGH) in
    join [int] (apply [int{HIGH}][int{HIGH}] f x)
```

The function *client* here calls function *f* with *x*, where *f* expects a parameter of type int{*HIGH*} while *x* has type int{*LOW*}. For the call to type check, the program uses *sub* to coerce *x*'s type to int{*lub LOW HIGH*} which is convertible to int{*HIGH*}. The call to *apply* returns a value of type int{*HIGH*}{*LOW*}. The call to *join* collapses the pair of labels so that *client*'s return type is int{*lub HIGH LOW*}, which converts to int{*HIGH*}.

We have proved that FABLE programs using this policy enjoy the standard noninterference property. We have also shown that a FABLE static information flow policy is at least as permissive as the information flow policy implemented by the functional subset of Core-ML, the formal language of FlowCaml [30]. Finally, we show how the dynamic provenance tracking and static information flow policies can be combined to enforce dynamic information flow. All the aforementioned proofs may be found in our technical report [38].

## 3.4. Proofs of Security Properties in FABLE

As mentioned in the introduction, FABLE does not, in and of itself, guarantee that well-typed programs implement a particular security policy's semantics correctly. That said, FABLE has been designed to facilitate proof of such theorems. To illustrate how, we chose to use three very different techniques for each of the correctness results reported here. We conclude from our experience that the metatheory of FABLE provides a useful repository of lemmas that can naturally be applied in showing the correctness of various policy encodings. As such, we believe the task of constructing a correctness proof for a FABLE policy to be no more onerous, and possibly considerably simpler, than the corresponding task for a special-purpose calculus that "bakes in" the enforcement of a single security policy. In the remainder of this section, we report on our experience with each of the three proofs and discuss preliminary progress towards reasoning about proofs involving multiple policies.

In all our proofs, two key features of FABLE play a central role. First, dependent typing in FABLE allows a policy analyst to assume that all policy checks are performed correctly. For instance, when calling the *access* function to access a value $v$ of type $t\{acl\}$, the label expressing $v$'s security policy must be *acl*, and no other. The type system ensures that the application program cannot construct a label, say *ACL*(*Public*), and trick the policy into believing that this label, and not *acl*, protects $v$; i.e., dependent typing rules out *confused deputies* [6]. Second, the restriction that application code cannot directly inspect labeled resources ensures that a policy function must mediate every access of a protected resource. Assuring complete mediation is not unique to FABLE— Zhang et al. [45] used CQual to check that SELinux operations on sensitive objects are always preceded by policy checks and Fraser [15] did the same for Minix. However, the analysis in both these instances only ensures that *some* policy check has taken place, not necessarily the correct one. As such, these other techniques are vulnerable to flaws due to confused deputies.

When combined with these two insights, our proof of non-observability for the access control policy (in our technical report [38]) is particularly simple. In essence, the FABLE system ensures that a value with labeled type must be treated abstractly by the application program. With this observation, the proof proceeds in a manner very similar to a proof of value abstraction [18]. This is a general semantic property for languages like FABLE that support parametric polymorphism or abstract types. Indeed, the policy as presented in Figure 3 could have been implemented in a language like ML, which also has these features. For instance, an integer labeled with an access control list could be represented in ML as a pair consisting of an access control list and an integer with type (string list $\times$ int). A policy module could export this pair as an abstract type, preventing application code from ever inspecting the value directly, and provide a function to expose the concrete type only after a successful policy check. While such an encoding would suffice for the simple policy of Figure 3, it would not work for other idioms like the function *access_cap* of Section 3.1, which reveals some of the structure of a label to avoid the need for additional checks. Abstract types on their own are also insufficient to support static checking of policies, as in the case of information flow.

To show dependency correctness (in our technical report [38]) we followed a proof technique used by Tse and Zdancewic [40]. This technique involves defining a logical relation [28] that relates terms whose set of provenance labels include the same label $l$. Recall that our goal in this theorem is to show that given $x{:}Prov\ t \vdash_c e : t$ that $\sigma_1(e)$ is related to $\sigma_2(e)$, where $\sigma_i$ substitutes a provenance labeled value $v_i$ for $x$ in $e$. The crux of this proof involves showing that the logical relation is preserved under substitution— i.e., a form of substitution lemma for the logical relation. While constructing the infrastructure to define the logical relation requires some work, strategic applications of standard substitution lemma for FABLE can be used to discharge the proof without much difficulty.

While it would be possible to reuse our infrastructure for the dependency correctness proof to show the noninterference result for the static information flow policy (as in Tse and Zdancewic), we choose instead to use another technique, due to Pottier and Simonet [30] (in our technical report [38]). This technique involves representing a pair of executions of a FABLE program within the syntax of a single program and showing a subject reduction property holds true. As with the logical relations proof, once we had constructed the infrastructure to use this technique, the proof was an easy consequence of FABLE's preservation theorem.

All our correctness theorems impose the condition that an application program be "$(\!|\,|\!)$-free". That is, these theorems apply only to situations where a single policy is in effect within a program. However, in practice, multiple policies may be used in conjunction and we would like to reason that interactions between the policies do not result in violations of the intended security properties. To characterize the conditions under which a policy can definitely be composed with another, we defined a simple type-based criterion, which when satisfied by two (or more) policies $\pi_P$ and $\pi_Q$, implies that neither policy will interfere with the functioning of the other policy when applied in tandem to the same program.

Intuitively, a policy can be made composable by enclosing all its labels within a unique top-level label constructor that can be treated as a *namespace*. A policy that only manipulates labels and labeled terms that belong to its own namespace can be safely composed with another policy.

The main benefit of compositionality is modularity; when multiple composable policies are applied to a program, one can reason about the security of the entire system by considering each policy in isolation. Policy designers that are able to encapsulate their policies within a namespace can package their policies as libraries to be reused along with other policy libraries.

Our notion of composition is a noninterference-like property—a policy is deemed composable if it can be shown not to depend on, or influence the functioning of another policy. As with noninterference properties in other contexts, this condition is often too restrictive for many realistic examples in which policies, by design, must interact with each other. We find that policies that do not compose according to this definition perform a kind of declassification (or endorsement) by allowing labeled terms to exit (or unlabeled terms to enter) the policy's namespace. We conjecture that the vast body of research into declassification [33] can be brought to bear here in order to recover a degree of modularity for interacting policies. Our technical report [38] contains the formal statement and proof of the policy noninterference theorem and further discussion of the applicability of this condition.

Finally, although we have focused on bisimulation properties in this paper, we believe that our approach is also likely to be useful in proving other kinds of security properties. For instance, we have recently begun investigating the enforcement of information release protocols by adding affine types to FABLE [39]. We formulate these protocols in terms of security automata used as a kind of typestate [36]. We have been able to prove that type-correct programs produce execution traces that contain event sequences in compliance with specific information release protocols. We have also found other forms of substructural types to be useful. Our technical report [38] sketches the use of relevant types to track side-effects in programs that manipulate references to mutable state.

## 4 SELINKS: FABLE for Web Programming

We have implemented FABLE as an extension to the LINKS functional web-programming language [12]; we call our extension *Security-Enhanced* LINKS, or SELINKS. This section briefly describes our SELINKS implementation and presents our experience using it to build two applications, a wiki SEWIKI and an on-line store SEWINESTORE.

### 4.1. SELINKS

LINKS is a new programming language with which a programmer can write an entire multi-tier web application as a single program. The compiler splits that program into components to run on the client (as JavaScript), server (as a local fragment of LINKS code), and database (as SQL). By extending LINKS with FABLE's label-based security policies, we can build applications that police data within and across tiers, up to the level of trust we have in those tiers. In our test applications we assume the server and database are trusted but the client is not.

LINKS is a functional programming language equipped with standard features such as recursive variant types, pattern matching, parametric polymorphism, and higher-order functions. As such, the FABLE policies we have presented so far transliterate naturally into SELINKS. One difference is that rather than define a special type lab as in FABLE, in SELINKS we allow arbitrary expressions to be treated as labels. The examples in this paper can be represented in SELINKS using expressions with a variant type as a label. Our applications make use of variants, strings, integers, lists and records to more easily construct and inspect labels.

SELINKS also provides native support for the syntactic shorthands shown in Figure 2. Type abbreviations in LINKS have been extended in SELINKS to support abbreviations of dependent types. Policy functions are designated by the qualifier policy, as in the examples of this paper. We also provide native support for dependent tuples in terms of existential packages rather than requiring the programmer to encode them with higher-order functions. While this makes dependent tuples easier to use, existential packages in SELINKS must still be carefully manipulated using explicit *pack* and *unpack* operations.

Finally, although LINKS makes heavy use of type inference, in SELINKS we rely on annotations to check code that manipulates security labels and labeled types. However, we provide limited but convenient forms of inference to simplify programming and cut down on annotations. For instance, instantiations of phantom label variables are always inferred and, in many common cases, pack and unpack operations can also be inferred. Additionally, code that does not use our type extensions can still benefit from standard LINKS type inference.

### 4.2. SEWIKI and SEWINESTORE

SEWIKI is an on-line document management system inspired by *Intellipedia*, a set of web applications designed to promote information sharing throughout the United States intelligence community [31]. SEWIKI consists of approximately 3500 lines of SELINKS code. It enforces a fine-grained combination of a group-based access control policy and a provenance policy on documents. A document is represented as a *n*-ary tree according to the following type definition:

$$\text{typename } Doc = Node \text{ of } [Doc] \mid Leaf \text{ of } String \\ \mid Labeled \text{ of } (l{:}DocLabel \times Doc\{l\})$$

Here, $[t]$ is SELinks notation for a list of $t$-typed values. The *Labeled* constructor allows nodes to have a security label according to the dependent pair $(l{:}DocLabel \times Doc\{l\})$. The type *DocLabel* is the type of security labels for documents.

typename $Group = Authors \mid Auditors \mid Administrators$
typename $Acl = (\ read{:}[Group],\ write{:}[Group]\ )$
typename $DocLabel = (\ acl{:}\ Acl,\ prov{:}\ Prov\ )$

*DocLabel* is a record with the fields *acl* and *prov* for storing access control and provenance labels respectively. The type *Acl* is itself a record containing two fields, *read* and *write*, that maintain the list of groups authorized to read and modify a document node, respectively. At the moment, we have three groups: *Authors*, in which all document authors are members; *Auditors*, the group of users that are trusted to audit a document; and *Administrators*, which include only the system administrators. We implement authentication credentials as terms of the type *Cred* (not shown). This type is similar to the type of credentials produced by *login* in the Fable access control policy (Figure 3) except that *Cred* includes additional useful information such as the user's name and unique identifier.

Possible document modifications are mediated by the *write_access* policy function, which has the following type:

$$write\_access{:}\forall\ \alpha,\beta.\ Cred \rightarrow (f{:}\alpha \rightarrow \beta) \rightarrow (l{:}Acl) \rightarrow \alpha\{l\} \rightarrow \beta\{l\}$$

This function allows a caller to pass in a user's credential and a function $f$ that is intended to modify a resource $\alpha$ labeled with an access control label $l$. If *write_access* determines that the user is in the writer's group of the *Acl* $l$, the function $f$ is applied and the policy relabels the modified resource with the (access control) label of the original.

SEWiki also includes a revision history tracking policy, similar in spirit to the provenance tracking policy of Section 3.2. We track provenance through all operations that alter a document while still enforcing the access control policy. We represent provenance information using the *Prov* type shown below and store this information in the *prov* field of a *DocLabel*.

typename $Op = Create \mid Edit \mid Delete \mid Restore \mid Copy \mid Relabel$
typename $Prov = [(oper{:}Op,\ user{:}User)]$

The provenance label of a document node consists of a list of operations performed on that node together with the identity of the user that authorized that operation. Tracked operations are of type *Op* and include document creation, modification, deletion and restoration (documents are never completely deleted in SEWiki), copy-pasting from other documents, and document relabeling. For the last, authorized users are presented with an interface to alter the access control labels that protect a document.

Policy functions that enforce this composite policy are fairly modular. For instance, a policy function that mediates modification of a document first projects out the *acl* component of the label and calls *write_access* to ensure that the modification is authorized. It then records the *Edit* operation in the *prov* field of the edited document's label.

In addition to building SEWiki, we extended the "wine store" e-commerce application that comes with Links by creating labels to represent users and associating such labels with orders in a shopping cart and order history. This helps ensure that an order is only accessed by the customer who created it. As in the SEWiki, the user's credential is represented using *Cred*; order information is defined below. The policy functions to view and add items to an order are implemented as simple wrappers around the same read and write access policies used in the SEWiki.

typename $Order = (l{:}Acl \times List(CartItem)\{l\})$

Our experience using SELinks to write these applications has been quite positive. The access control policies were easy to define and to use, with policy code consisting of roughly 200 lines of code total (including helper functions). The access control and login policy code was modular enough to be shared in its entirety by the two applications. The provenance policy consists of about 100 lines of code, and was also straightforward to use. Unlike the provenance policy from Section 3.2, SEWiki provenance labels essentially track only direct data flows to and from other documents. This makes them much easier to program with since far fewer program operations need to be mediated by the policy. To support richer policies while easing the programming burden we are investigating an approach related to *weaving* in aspect-oriented programming [21] that, given a policy specification, automatically transforms a program to insert the appropriate label manipulations. We also plan to include limited support for type inference to better integrate the use of Fable-style dependent types with standard Links types in SELinks.

## 5 Related Work

Dependently-typed languages have found use in a wide variety of applications [44, 43, 3, 5]. In the context of security, Zheng and Myers [46] formalize support for *dynamic security labels* that can be associated with data to express information flow policies. The technical machinery for associating labels to terms in their system is similar to ours. There are two main differences. First, the security policy—an information flow policy with a particular label model—is expressed directly in the type system whereas in Fable both the security policy and the label model are customizable. As discussed in Section 3.3, dynamic labels for information flow policies can be encoded in Fable as a combination of our dynamic provenance and static information flow policies. Second, Fable allows non-values to appear in types, e.g., *lub l m* in Figure 7. This permits a combination of static and dynamic policy checking, but at the cost of potentially undecidable type checking. Our SELINKS implementation uses heuristics to ensure that type-checking never diverges.

Walker's "type system for expressive security policies" [42] is also dependently typed. Labels in Walker's language are uninterpreted predicates rather than arbitrary expressions. Walker's system can enforce policies expressed as security automata, which can capture any *safety property*. This kind of policy is also enforceable in FABLE when extended with substructural types. However, in Walker's system, the policy is always enforced by means of a runtime check. In order to recover some amount of static checking, Walker suggests that a user might add additional rules to the type system, though he is not specific about how this would be done. These additional rules would have to be proved correct with respect to a desired security property.

It has been observed that dependent types can be used to express a kind of customized type system [43], and FABLE's policy functions fit this description. For example, the *sub* function in the policy of Figure 7 effectively introduces a subsumption rule into the type system. Researchers have explored how user-defined type systems can be supported directly via customizable *type qualifiers*. Shankar et al. [35] have used lattice-based type qualifiers in CQual [13] to track dataflow properties like taintedness [35], and Zhang et al. [45] and Fraser et al. [15] have used qualifiers to check complete mediation in access control systems. Millstein et al [8, 1] have developed an approach in which programmers can indicate data invariants that custom type qualifiers are intended to signify. In some cases, they are able to automatically verify that these invariants are correctly enforced by the custom type rules. While their invariants are relatively simple, we ultimately would like to develop a framework in a similar vein, in which correctness properties for FABLE's enforcement policies can be at least partially automated. Marino et al. [27] have proposed using proof assistants for this purpose, and we plan to explore this idea in the context of FABLE policies.

Li and Zdancewic show how to encode information flow policies in Haskell [25]. They define a meta-language that makes the control-flow structure of a program available for inspection within the program itself. Their enforcement mechanism relies on the lazy evaluation strategy of Haskell that allows the control flow graph to be inspected for information leaks prior to evaluation. While their encoding permits the use of custom label models, they only show an encoding of an information flow policy. It is not clear their system could be used to encode the range of policies discussed here.

In other work, Li and Zdancewic [24] have proposed labeling types with functions that describe conditions under which a type is allowed to be relabeled. Their goal is to control what information is declassified by a program, whereas we aim to enforce a variety of policies.

Our technique of separating the enforcement policy from the rest of the program is based on Grossman et al's *colored brackets* [18]. They use these brackets to model type abstraction, whereas we use them to ensure that the privilege of unlabeling and relabeling terms is not mistakenly granted to application code. As a result, we do not need to specially designate application code that may arise within policy terms, keeping things a bit simpler. We plan to investigate the use of different colored brackets to distinguish different enforcement policies, following Grossman et al.'s support for multiple agents.

Finally, inasmuch as we have targeted the LINKS web-programming language [12] as the platform on which to build FABLE, our work is related to Swift [9] and SIF [11], two Jif-based projects that aim to secure web applications. The former is a technique that permits a web application to be split according to a policy into JavaScript code that runs on the client and Java code on the server, while the latter is a framework in which to build secure servlets. As discussed in Section 4, LINKS provides similar functionality, except it additionally integrates database access code into the framework. With our new security checking features in SELINKS, as in Swift, practical, verified, end-to-end security for multi-tiered applications is within reach.

## 6 Conclusions

This paper has presented FABLE, a core formalism for a programming language in which programmers may specify security policies and reason that these policies are properly enforced. We have shown that FABLE is flexible enough to implement a wide variety of security policies, including access control, provenance, and static information flow, among other policies. We have also argued that FABLE's design simplifies proofs that programs using these policies do so correctly. We have implemented FABLE as part of the LINKS web programming language, and we have used the resulting language, which we call SELINKS, to build two substantial applications, a secure wiki and a secure online store. While more work remains to make SELINKS a fully satisfactory platform, to our knowledge, no existing framework enables the enforcement of such a wide variety of security policies with an equally high level of assurance.

**Acknowledgements:** We thank Jeff Foster, Boniface Hicks, Polyvios Pratikakis, Peter Sewell and the anonymous reviewers for helpful comments on a draft of this paper. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U. S. Government.

## References

[1] C. Andreae, J. Noble, S. Markstrum, and T. Millstein. A framework for implementing pluggable type systems. In *OOPSLA '06*. ACM Press, 2006.

[2] D. Aspinall and M. Hoffmann. *Advanced Topics in Types and Programming Languages*, chapter Dependent Types. MIT Press, 2004.

[3] L. Augustsson. Cayenne–a language with dependent types. In *ICFP '98*, New York, NY, USA, 1998. ACM Press.

[4] L. Badger, D. F. Sterne, D. L. Sherman, and K. M. Walker. A domain and type enforcement UNIX prototype. *Computing Systems*, 9(1):47–83, 1996.

[5] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer Verlag, 2004.

[6] M. Bishop. *Computer Security: Art and Science*. Addison Wesley, 2003.

[7] J. Cheney, A. Ahmed, and U. Acar. Provenance as dependency analysis. *Database Programming Languages*, 2007.

[8] B. Chin, S. Markstrum, and T. Millstein. Semantic type qualifiers. In *PLDI '05*. ACM Press, 2005.

[9] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web application via automatic partitioning. In *SOSP '07*. ACM Press, 2007.

[10] S. Chong, A. C. Myers, N. Nystrom, L. Zheng, and S. Zdancewic. Jif: Java + information flow. Software release. Located at `http://www.cs.cornell.edu/jif`, July 2006.

[11] S. Chong, K. Vikram, and A. C. Myers. Sif: Enforcing confidentiality and integrity in web applications. In *USENIX Security '07*, 2007.

[12] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. In *FMCO '06*, 2006.

[13] J. S. Foster, T. Terauchi, and A. Aiken. Flow sensitive type qualifiers. In *PLDI '02*. ACM Press, 2002.

[14] C. Fournet and A. D. Gordon. Stack inspection: theory and variants. In *POPL '02*. ACM Press, 2002.

[15] T. Fraser, J. Nick L. Petroni, and W. A. Arbaugh. Applying flow-sensitive CQUAL to verify MINIX authorization check placement. In *PLAS*. ACM Press, 2006.

[16] J.-Y. Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VI I, 1972.

[17] L. Gong. *Inside Java 2 platform security architecture, API design, and implementation*. Addison-Wesley, 1999.

[18] D. Grossman, G. Morrisett, and S. Zdancewic. Syntactic type abstraction. *ACM TOPLAS*, 22(6):1037–1080, 2000.

[19] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *POPL '95*. ACM Press, 1995.

[20] B. Hicks, D. King, P. McDaniel, and M. Hicks. Trusted declassification: high-level policy for a security-typed language. In *PLAS '06*. ACM Press, 2006.

[21] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP '07*. Springer-Verlag, 1997.

[22] C. E. Landwehr. The best available technologies for computer security. *IEEE Computer*, 16(7):89–100, July 1983.

[23] N. Li, J. C. Mitchell, and W. H. Winsborough. Design of a Role-Based Trust-Management Framework. In *S&P '02*. IEEE Computer Society Press, 2002.

[24] P. Li and S. Zdancewic. Downgrading policies and relaxed noninterference. In *POPL '05*. ACM Press, 2005.

[25] P. Li and S. Zdancewic. Encoding information flow in Haskell. In *CSFW '06*. IEEE Computer Society Press, 2006.

[26] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the Linux operating system. In *FREENIX 2001*. USENIX Association, 2001.

[27] D. Marino, B. Chin, T. Millstein, G. Tan, R. J. Simmons, and D. Walker. Mechanized metatheory for user-defined type extensions. In *WMM '06*, 2006.

[28] J. C. Mitchell. *Foundations of Programming Languages*. MIT Press, 1996.

[29] Perl 5.8.8 documentation - perlsec. `http://perldoc.perl.org/perlsec.html`.

[30] F. Pottier and V. Simonet. Information flow inference for ML. *ACM TOPLAS*, 25(1), Jan. 2003.

[31] Reuters, October 2006. U.S. Intelligence Unveils Spy Version of Wikipedia.

[32] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *JSAC*, 21(1):5–19, Jan. 2003.

[33] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *CSFW '05*. IEEE Computer Society, 2005.

[34] SecurityFocus: Access control bypass vulnerabilities. `http://search.securityfocus.com/swsearch?metaname=alldoc&query=access+control+bypass`.

[35] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting Format String Vulnerabilities with Type Qualifiers. In *USENIX Security '01*. USENIX Association, 2001.

[36] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, 1986.

[37] Z. Su and G. Wassermann. The essence of command injection attacks in web applications. In *POPL '06*. ACM Press.

[38] N. Swamy, B. J. Corcoran, and M. Hicks. Fable: A language for enforcing user-defined security policies. Technical Report CS-TR-4895, Depart. Comp. Sci., U. Maryland, 2007.

[39] N. Swamy and M. Hicks. Verified enforcement of automaton-based information release policies. Technical Report CS-TR-4906, Dept. Comp. Sci., U. Maryland, 2008.

[40] S. Tse and S. Zdancewic. Run-time Principals in Information-flow Type Systems. In *S&P '04*. IEEE Computer Society Press, 2004.

[41] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

[42] D. Walker. A type system for expressive security policies. In *POPL '00*. ACM Press, 2000.

[43] H. Xi. Applied Type System (extended abstract). In *TYPES 2003*. Springer-Verlag LNCS 3085, 2004.

[44] H. Xi and F. Pfenning. Dependent types in practical programming. In *POPL '99*. ACM Press, 1999.

[45] X. Zhang, A. Edwards, and T. Jaeger. Using CQUAL for static analysis of authorization hook placement. In *USENIX Security*, 2002.

[46] L. Zheng and A. C. Myers. Dynamic security labels and noninterference. In *FAST '04*. Springer, 2004.