Fable: Enforcing User-defined Software Security Policies

Mike Hicks

with Nikhil Swamy

and Brian Corcoran
Security Policies

• Many applications are concerned with security
  - Medical information systems, on-line stores, web portals, voting systems, collaborative planning systems (blog/wiki), ...

• Many styles of security policy to choose from
  - Access control (role-based, history-based, ...), information flow control, separation of duty, ...

• Many enforcement mechanisms to choose from
  - Access control lists, capabilities, stack inspection, static analysis, security automata, ...
Example: Information Sharing Application

• The military uses a blog/wiki-style application for sharing information across security levels
  - Goal: maximize information sharing for better decision making
• Protect confidentiality of high-security information
  - Should release as much as possible, for better analysis
  - Record information releases for post-hoc analysis
• Ascertain information integrity
  - Tamper-resistance
  - Provenance
• Protect actions, not just information
The Big Questions

Once we have chosen a set of security policies, how do we know if

1. the security policies protect our data as we intend?

2. the software enforces our security policies correctly?
Assured Enforcement via Program Analysis

• Typical security enforcement mechanisms regularly circumvented
  - access control bypasses, information leaks, etc.

• Solution: ensure proper enforcement through programming languages or analysis
  - Dataflow analysis for ensuring complete mediation
  - Tainting analysis for flagging injection vulnerabilities
  - Information flow type systems for preventing information leaks (e.g., Jif, FlowCaml)
  - ...

• Problem: these approaches specialized to particular styles of policy and/or enforcement mechanism
Toward General, Assured Enforcement

**Goal:** A language for expressing security policies and their enforcement such that

1) The language ensures that policy enforcement cannot be circumvented
2) The policy and its enforcement mechanism can be shown to satisfy high-level security objectives
3) A variety of security policies can be applied

Ultimately we want to apply this language to web applications
Key Result #1: Fable

- Core formalism for user-defined policies
- Shown that Fable is powerful enough to encode a variety of security policies
  - Information flow control
  - Access control via capabilities and AC lists
  - Provenance
  - Stack inspection
  - Security automata
- Proved useful properties about these policies
Key Result #2: Security-enhanced Links

• Links is a web programming language developed at Edinburgh by Wadler’s group
  - An entire web application can be written as a single program which is split into tiers by the compiler

• We have created a “security-enhanced” Links (aka SELinks) by extending it with Fable
  - Labels can be stored with data in the database to express that data’s policy in a customized way
Key Result #3: Application Experience

• SEWiki: A blog/wiki case study
  - Roughly 3500 lines of SELinks code
  - Uses a combination of access control and data provenance policies
    • Username: demo
    • Password: demopw

• SEWinestore: an on-line store
  - Implements a policy according to a security automaton
Demo 1 Overview

This page contains the complete overview of this document.

1. Demo 1 Overview
2. Demo 2 Background
   1. Demo 2a Links
   2. Demo 2b SELinks
   3. Demo 2c SEWiki
3. Demo 3 Wiki Features
   1. Demo 3a Text Editing
   2. Demo 3b Text Formatting
   3. Demo 3c URLs
   4. Demo 3d Other Features
   5. Demo 3e Copy Paste
4. Demo 4 Labels
   1. Demo 4a Creating a Page Revisited
   2. Demo 4b Modifying Content Revisited
   3. Demo 4c Modifying Access Control
   4. Demo 4d Copy Paste Revisited
   5. Demo 4e Access Control in Action
5. Demo 5 Conclusion

Let’s begin with Demo 2 Background.
Demo 4 Labels

The SEWiki currently supports two types of labels:

- access control (ACL)
- data provenance (PROV)

In the next few sections, we detail the format of these labels, and describe how to view, modify, and interact with them.

Access Control Labels

Access control labels are used to enforce who and how data can be viewed and modified. In the case of the SEWiki, data refers to particular blocks on a page. Currently, access control is group-based, where a group is a list of users. Access control labels begin with the string `ACL`, and contain a list of groups with `read` access, and as list of groups with `write` access. A sample access control label representation is:

ACL[READ[readers, writers], WRITE[writers]]

This label mentions two groups, `readers` and `writers`. It grants `READ` access to both the `readers` and `writers` groups, and `WRITE` access just to the `writers` group.

Provenance Labels

Data provenance labels record all actions to a block, to record when, how, and by whom
Page Labels View

To demonstrate labels in action, we need to create another new page. In your second window, click Main menu, then Create page. Name your new page My Labeled Page and enter some text. Click Save page.

Now, click View all labels. The Page Labels View displays all the labels for your page in the red, left-hand box. Both ACL and PROV labels are included.

The page has an ACL label saying that the page can be viewed by anyone in the readers group, and modified by anyone in the writers group. This is the default access for a new page; the demo user is a member of both of these groups.

Furthermore, there is a PROV CREATEPAGE label detailing that the page was created by user demo, the time it was created, and the original title of the page.
Demo Restricted

This page is divided into three sections. This first section is writable by writers, the second section is readable by readers, but writable only by administrators.

*** Access denied ***
This page is divided into three sections. This first section is writable by writers, readers, and administrators.

SECURITY
.PROV.ACTIONS
...ACTION_20070925222539.MODIFYPAGE
.....DESC.Path:-->1
.....TIME.15:27:03
.....DATE.2007-08-17
.....USER.Brian
...ACTION_20070925222539.CREATEPAGE
.....DESC.Path:-->1
.....TIME.15:25:05
.....DATE.2007-08-17
.....USER.Brian
...ACTION_20070925222539.MODIFYPAGE
.....DESC.ACL{WRITE[administrators];READ[readers]}
.....TIME.15:27:46
.....DATE.2007-08-17
.....USER.Brian
...ACTION_20070925222539.MODIFYPAGE
.....DESC.Path:-->2-->1
.....TIME.20:09:57
.....DATE.2007-09-18
.....USER.Cody
.ACL
...WRITE.administrators
....READ.readers

SECURITY
.PROV.ACTIONS.ACTION_20070925222539.LABEL
.....DESC.ACL{WRITE[readers];READ[readers]}
.....TIME.19:31:16
.....DATE.2007-09-18
.....USER.demo
.ACL
...WRITE.readers
....READ.readers

SECURITY
.PROV.ACTIONS.ACTION_20070925222539.LABEL
.....DESC.ACL{WRITE[administrators];READ[administrators]}
.....TIME.15:27:29
.....DATE.2007-08-17
.....USER.Brian
.ACL
...WRITE.administrators
....READ.administrators

** Access denied **
The first step: Fable

**Fable** is a core formalism for a programming language supporting custom security policies

Two key elements

1) Security policies associated with program data and actions via customizable *security labels*

2) The semantics of labels is specified within a separate part of the program called the *enforcement policy*
   - May call a separate, high-level policy engine, in the style of trust management
Association via Dependent Types

• Security labels are data with type \textit{lab}
  - \textit{lab} \( l = \text{HIGH} \);

• Protected data refers to its label in its type
  - \textit{int} \( w = \ldots \) // unprotected data
  - \textit{int}\{\textit{l}\} \( x = \ldots \) // protected by label \( l \)
  - \textit{bool}\{\textit{ACL(mwh,nswamy)}\} \( z = \ldots \)
    • // protected by label \textit{ACL(mwh,nswamy)}
  - In general, protected types have form \textit{t\{e\}} where \( e \) has type \textit{lab}
Enforcement Policy and Label Semantics

- Normal send function
  \[
  \text{send} : \text{socket} \to \text{bytes} \to \text{unit}
  \]

- Cannot call this function with labeled data
  - data of type \text{socket\{e\}} indicates a protected socket
  - data of type \text{bytes\{e\}} indicates a sensitive buffer

- Must therefore use a policy-enforcing wrapper
  \[
  \text{letpol sock\_send(l:lab, s:socket\{l\}, m:lab, msg:bytes\{m\}) =}
  \]

  \[
  \text{if policy\_allows l m then}
  \]

  \[
  \text{send {} s {}msg}
  \]

  \[
  \text{else ()}
  \]

  **Relabeling can only occur in policy code**
Fable: Simplified Syntax

\[ P ::= \pi; e \]

\[ \pi ::= \text{letpol } x_1 = v_1 \ldots \text{letpol } x_n = v_n \]

\[ e ::= \ldots \mid C(e_1, \ldots, e_n) \mid \{e\}e \mid \{\}e \]

\[ t ::= \ldots \mid \text{lab} \mid \text{lab} \sim e \mid t\{e\} \mid x: t_1 \rightarrow t_2 \]

Labels are uninterpreted constructors; think XML
Label types may reveal the label itself

\[
e_i : \text{lab} \\
C(e_1 \ldots e_n) : \text{lab}\sim C(e_1 \ldots e_n) \quad \text{labeling}
\]

\[
e : \text{lab}\sim e' \\
e : \text{lab} \quad \text{weakening}
\]

\[
e : \text{lab} \\
e : \text{lab}\sim e \quad \text{strengthening}
\]
Example typings

⇒ HIGH : lab~HIGH
⇒ HIGH : lab
⇒ USER(MWH) : lab
l:lab  ⇒ USER(l) : lab~USER(l)
l:lab  ⇒ USER(l) : lab

--------------------------
⇒ USER(7) : lab
--------------------------

m:t→t'  ⇒ USER(m) : lab
Labeling and Unlabeling

Policy code can add to or strip a label from the outermost level of a term

\[
\begin{align*}
l : \text{lab} & \quad e : \top \\
\{l\}e & : \top\{l\} \\
\{\}e & : \top
\end{align*}
\]

labeling

unlabeling
Example typings

$$\Rightarrow \{H\} 1 : \mathbf{int}\{H\}$$

$$m : \mathbf{int}\{L\} \Rightarrow \{\} m : \mathbf{int}$$

$$l : \mathbf{lab}, m : \mathbf{int} \Rightarrow \{\} \{l\} m : \mathbf{int}$$

$$f : t\{H\} \rightarrow t' \Rightarrow \{L\} f : (t\{H\} \rightarrow t')\{L\}$$

$$f : t \rightarrow t' \Rightarrow \{\} f : t \rightarrow t'$$
Access Control Policies

• Step 1: implement authentication using capabilities

```ocaml
letpol login(user:string, pw:string) =
  let token =
    match (checkpw user pw) with
    USER(k) -> USER(k)
    _ -> FAILED in
    (token, {token}1)
```

Success: UID in label
Failure: failure label

Capability: labeled term can only be constructed by policy
Access Control Policies

- Step 2: query the policy

\[
\text{letpol } \text{access}^{k,\alpha}(u:\text{lab} \sim \text{USER}(k), \text{cap} : \text{int}\{u\}, \\
\text{acl} : \text{lab}, \text{data} : \alpha\{\text{acl}\}) = \\
\text{match } (\text{member } u \text{ acl}) \text{ with} \\
\text{MEMBER} \rightarrow \{\} \text{data} \\
\text{else } \rightarrow \text{raise Failure} \\
\]

Success: remove label
Failure: throw exception

Phantom term: label term polymorphism
Type variable: type polymorphism

A list, e.g., CONS(mwh, CONS(nswamy, NIL))
Refined label types

• Pushes obligations of matching to the caller
  - E.g., in access, \texttt{u:lab~USER(k)} rather than \texttt{u:lab}

• For example:

\begin{verbatim}
let (u,c) = login name pw in
match u with
    USER(k) -> access u c dlab d
    FAILED  -> raise LoginFailure
\end{verbatim}

Types are \texttt{u:lab, c:int\{u\}}

In this expression, match refines type \texttt{u:lab~USER(k)}
Information Flow Policies

• Data labels are arranged according to a graph
  - Typically a lattice
  - E.g., labels LOW and HIGH, s.t. LOW ≤ HIGH

• Data with label l may flow to a context expecting data labeled m where l ≤ m
  - A context expecting HIGH data can view LOW data as well
  - But not the converse

• Stronger than access control
  - Concerned with information release and its propagation
Information Flow Policies

• Step 1: legal flows, according to policy, are witnessed by capabilities

\[
\text{letpol flow}(\text{src}:\text{lab}, \text{dst}:\text{lab}) = \\
\quad \text{let token} = \\
\qquad \text{if mayflow src dst then FLOW(src,dst)} \\
\qquad \text{else NOFLOW in} \\
\quad (\text{token}, \{\text{token}\}1)
\]

Similar to login function in access control policies
Information Flow Policies

• Step 2: use results of flow to relabel terms as needed

\[ \text{letpol relabel}\langle s,d,\alpha\rangle(\text{cap:}\text{int}\{\text{FLOW}(s,d)\}, x:\alpha\{s\}) = \{d\}x \]

• Step 3: wrap primitive operations on labeled terms

\[ \text{letpol add}\langle l,m\rangle(x:\text{int}\{l\}, y:\text{int}\{m\}) = \{\text{JOIN}(l,m)\}(x+y) \]

Similarly for other destructors like function application, if-then-else, record projection, ...

\text{JOIN}(l,m) \text{ designates the least upper bound of } l \text{ and } m \text{ according to the lattice}
Example: Unsuccessful relabeling

output : all m:lab. channel\{m\} -> int\{m\} -> int

\(s : channel\{LOW\}\)

\(h : int\{HIGH\}\)

\(l : int\{LOW\}\)

let \(x = \text{add } h \ l \text{ in}\)

let \((r, cap) = \text{flow JOIN}(HIGH,LOW) LOW \text{ in}\)

match \(r\) with

\(\text{NOFLOW} \rightarrow 0\)

\(\text{FLOW(...)} \rightarrow \ldots \text{ output } s \ x'\)

\(x : \text{int}\{\text{JOIN(HIGH,LOW)}\}\)

\(\text{cap: int}\{r\}\)

\(\text{cap: int}\{\text{NOFLOW}\}\)
Static Information Flow

• Information flow systems may also use purely static labels with a fixed security lattice
  - No possibility that a flow will be disallowed at runtime; it would be flagged by the type checker

• Fable can encode such policies by using $\beta\eta$-equivalence of types
  - Terms appearing in types makes type checking undecidable in general, but very useful when used with care
Static Information Flow

• Step 1: define security lattice

```plaintext
letpol lub(src:lab, dst:lab) =
  match src, dst with
  LOW, _ -> dst
  HIGH, _ -> src
```

• Step 2: define relabeling function

```plaintext
letpol relabel<\alpha>(x: \alpha\{l\}, m: lab) = \{lub l m\}x
```
Static Information Flow

• Step 3: wrap operations on labeled terms

\[
\text{letpol add}<l,m>(x:\text{int}\{l\}, y:\text{int}\{m\}) = \\
\{\text{lub } l \ m\}(x+y)
\]
Example: statically-successful relabeling

output : all m:lab. channel{m} -> int{m} -> int
            s : channel{HIGH}
            h : int{HIGH}
            l : int{LOW}

let x = add h l in
output s x

\[
\begin{align*}
e : \tau & \Rightarrow \tau' & e : \tau' \\
\frac{e \rightarrow^* \tau' \tau'}{\tau\{e\} \rightarrow^* \tau\{e'\}}
\end{align*}
\]
Reasoning about Policies

• We can prove that well-typed programs using a particular policy satisfy high-level security goals
  - We have proven that our static information flow policy satisfies noninterference
  - We have also proven a completeness theorem: that well-typed programs in a functional subset of the language FlowCaml are well-typed in our system
• Fable’s proof of soundness simplifies these proofs
Non-observability
Proof of correctness of access control

- Given an application $e$ with $x$ protected by an ACL
  \[ u : \text{UserCred}\{\text{High}\}, \, x : \text{bool}\{\text{acl}\} \mid \_\text{app} \, e : \top \]

- And the user $\text{uid}$ is not authorized to access $x$
  $\text{member} \, \text{uid} \, \text{acl} \rightarrow^* \text{false}$

- Then, executions with $x=true$ and $x=false$ are identical
  $\[ e[x \rightarrow true, \, u \rightarrow uid] \rightarrow e' \[x \rightarrow true, \, u \rightarrow uid \] \]
  $\iff$
  $\[ e[x \rightarrow false, \, u \rightarrow uid] \rightarrow e' \[x \rightarrow false, \, u \rightarrow uid \] \]$
Other Policies

• Provenance (done)
  - Labels indicate the source of information; useful for auditing

• Tagging (possible)
  - Labels can express arbitrary categorization
    • Think ITunes playlists or Google Mail
  - Queries identify data with matching labels and strip those labels for use by the application
    • I.e., labels don’t have to be security-related only
The Problem with State

• Fable relies on the type system to ensure policy code mediates all security-sensitive operations
  - The security-relevant effect of some operation, therefore, must be reflected in its type

• In information flow type systems, modifications of state may reveal information

• Policies may also be stateful
  - E.g., a security automaton
Violating information flow via the store

$x : \text{int ref}\{\text{LOW}\}$  

$y : \text{int}\{\text{HIGH}\}$

if $y$ then
  $x := 0$
else
  0

Memory that $x$ points to is LOW

Since an attacker may view the contents of $x$ he may infer that $y$ is nonnegative.
First cut for assignment policy

• Step 1: labels now have the form $C(l, \text{EFF}(m))$ where $l$ and $m$ are labels as before (e.g., LOW, HIGH)
  - $l$ is the actual label of the term
  - $m$ is the label capturing the effect of the term’s computation
  - Default label $C(LOW, \text{EFF}(HIGH))$
    - I.e., as if only wrote to high-security locations when constructing this value, so not possible to leak information through them

• Step 2: define update wrapper

  \[
  \text{letpol update} < l, \alpha > (x : \alpha \ ref \{ l \}, y : \alpha) =
  \{ C(LOW, \text{EFF}(l)) \} (\{\} x := y)
  \]
Wrapping conditionals

- Step 3: the effect of each branch affects the label of the final result

```plaintext
letpol cond<l,m,α>(g:int{l},
   t:unit→α{m},
   f:unit→α{m},
   c:int{FLOW(l,m)}) =
   if {}g then t () else f ()
```
The problem: “forgetting effects”

if y then
  x := 0
else 0

let res = (update x 0) in (high 0)

Thus the LOW effect of the update is forgotten, and the term type checks!

h : int{C(HIGH,EFF(HIGH))}
x : int ref{C(LOW, __)}
y : bool{h}
z : int{FLOW(h, h)}
Relevant types

- We can prevent a programmer from “forgetting” information in a policy-provided label by making it *relevant*
  - Must be “used” at least once
    - either returned from a function (passing the buck) or passed into a policy function which “consumes” it

- In prior example, this would require `res` to be returned, and this would prevent the program from type checking
Substructural Types

• Relevant types are a kind of substructural type, which restricts certain usage
  - Relevant: used at least once
  - Linear: used exactly once
  - Affine: used at most once

• Affine types are also useful in our setting, as they can be used to encode state-based policies
  - Stack inspection, security automata (on-line store!)
Next Steps

• Fable’s label model is simple, but it’s use of type dependency could be more expressive
  – We have explored how a more expressive language can more easily express security automata

• Possible goal: rewriting policies
  – Allow programmers to write programs in a more typical language, and then rewrite them to insert calls to the enforcement code
  – Benefit: if the rewritten code is well-typed then it will satisfy the security property ensured by the policy
  – Not really necessary for access control, data-flow policies
Summary

• Fable is a core language formalism for specifying and enforcing custom, label-based security policies
  - We intend to use it as the core of a web application programming language
• Fable can express a variety of policies
  - Access control, information flow, stack inspection, security automata, provenance -- may be combined
• Fable’s structure and type system ensure that a policy cannot be circumvented
• Policies are clearly separated from programs, facilitating modular reasoning and reuse
Security-enhanced Links

• Links is a web programming language developed at Edinburgh by Wadler’s group
  - An entire web application can be written as a single program which is split into tiers by the compiler

• We have integrated Fable to create a “security-enhanced” Links (aka SELinks)
  - Labels are stored with data in the database to express that data’s policy
SEWiki: A Secure Document Management System

• 3500 lines of SELinks code

Enforces a fine-grained composite policy on documents

1. A group-based access control policy on fragments of a document
   • Similar to the example shown earlier

2. A provenance policy that tracks various operations on documents

3. Guards against XSS attacks using BEEP
Documents are n-ary trees with words at the leaves.

```
type Doc = [|
  Leaf : String -> Doc
  Node : [Doc] -> Doc
  Protected : (l:Label) -> Doc{l} -> Doc
| ...
|]
```

Some subtrees can be protected by a security label.

First component a label that protects the subtree in the second component.

A dependently-typed pair.
Representing Security Labels

typename Label =
  (Acl : (read : [Group], write : [Group]),
   Provenance : [ProvAction])

typename Group =
  [Admin | Audit | Guest | ... ]

typename ProvAction =
  (action : [Create | Modify | Relabel | ... ],
   group : Group)

1st component for access control
2nd component for provenance

Group-based access control

Provenance actions represent operations on documents
Next steps for SELinks

• Better integration with the database
  - Reference to enforcement policy functions in queries forces all data to be brought into the server, which is REALLY SLOW
  - Solution: store policy functions on the database

• Add support for affine/relevant types for enforcing automaton policies
http://www.cs.umd.edu/projects/PL/SELinks