FINE + DCIL

End-to-end Verification of Security Enforcement
Security Policies

- Many languages/logics to specify security policies
  - XACML, DKAL, SecPAL, DCC, SD3, Binder, Ponder, CDatalog, RT, SPKI/SDSI, ...

- Policies address a variety of security concerns
  - Authentication, authorization, usage controls, information flow, security automata, ...

- But, disconnect between specification and implementation
• SecPAL policy for EHR, an e-health database

  P can read P’s record r
  D can read P’s record r if D is treating P and r.Subject <> HIV

• C# function to enforce this policy

```csharp
public Record GetRecord (string pat, string recId) {
    Record rec = m_data.GetRecord (pat, recId);
    if (rec != null &&
        GetAuthContext().query("CanGetRecord", new List {
            pat, recId,
            rec.Author,
            rec.Subject })
    {
        return result;
    }
    return null;
}
```
- SecPAL policy for EHR, an e-health database

  P can read P’s record r
  D can read P’s record r if D is treating P and r.Subject <> HIV

- C# function to enforce this policy

  ```csharp
  public Record GetRecord (string pat, string recId) {
    Record rec = m_data.GetRecord (pat, recId);
    if (rec != null &&
        GetAuthContext().query("CanGetRecord", new List { pat, recId, rec.Author, rec.Subject })) {
      return result;
    }
    return null;
  }
  ```

  Okay to read record before security check?

  Can credentials be forged?
• SecPAL policy for EHR, an e-health database

  P can read P’s record r
  D can read P’s record r if D is treating P and r.Subject <> HIV

• C# function to enforce this policy

```csharp
public Record GetRecord (string pat, string recId) {
  Record rec = m_data.GetRecord (pat, recId);
  if (rec != null && GetAuthContext().query("CanGetRecord", new List {
    pat, recId,
    rec.Author,
    rec.Subject
  })) {
    return result;
  }
  return null;
}
```
SecPAL policy for EHR, an e-health database

- P can read P’s record r
- D can read P’s record r if D is treating P and r.Subject <> HIV

C# function to enforce this policy

```csharp
public Record GetRecord (string pat, string recId) {
    Record rec = m_data.GetRecord (pat, recId);
    if (rec != null &&
        GetAuthContext().query("CanGetRecord", new List {
            pat, recId,
            rec.Author,
            rec.Subject
        })) {
        return result;
    }
    return null;
}
```

Unclear if policy is enforced correctly
Verifying Security Enforcement by Typing

- Security policies embedded in a program’s types
  - Security checking amounts to type checking
    - Volpano and Smith ’96

- Many proposals
  - FlowCaml, Jif, Fable, Aura, F7

- But, not yet widely applicable
Limitations of Security by Typing

- Cannot handle many constructs of real policies
  - Jif and FlowCaml: only information flow
  - Aura and Fable: only stateless policies
  - F7: targets stateless authentication

- Either not meant for source programming
  - Fable and Aura require explicit security proofs

- Or cannot be used to generate checkable binaries
  - Jif, FlowCaml, and F7 erase security types
Our Approach: **FINE + DCIL**

**FINE: A source-level type system for F#**
- Refinement types for authorization policies
- Dependent types for information flow
- Affine types for state-modifying policies

**Automated security proof construction**
- Type checked with assistance from Z3
- Security proofs synthesized from Z3 proofs

**Verifiable binaries by type preservation**
- DCIL: an extension of CIL’s type system
- Security proofs carried from source level
- Checked syntactically (without Z3)

**Z3**
- Type checker
- Expressive
- Easier to program
- Verifiable with small TCB

```
\Lambda \alpha::*.
  \lambda x: \tau_1.
  \lambda y: \{ \tau_2 \mid \varphi \}.
  \lambda z: ! \tau_3.
  ...
```
Outline

- Overview
- FINE, by example
- Proof terms
- Translation to DCIL
- Results and future work
EHR Application

Authentication

Reference Monitor

| Health Records |

Application Code
module Authentication

val login : u:prin -> str -> option<cred<u>>
let login u pw =
  if (* password ok? *)
    then Some (MkCred u)
  else None
module Authentication

type prin =
  | User : str -> prin
  | Admin : prin

val login : u:prin -> str -> option<cred<u>>
let login u pw =
  if (* password ok? *)
  then Some (MkCred u)
  else None
module Authentication

type prin =
  | User : str -> prin
  | Admin : prin

type option<\alpha> =
  | Some : \alpha -> option<\alpha>
  | None : option<\alpha>

val login : u:prin -> str -> option<cred<u>>
let login u pw =
  if (* password ok? *)
    then Some (MkCred u)
  else None

Polymorphic type constructor
  • Type parametrized by another type
module Authentication

type prin =
  | User : str -> prin
  | Admin : prin

type option<α> =
  | Some : α -> option<α>
  | None : option<α>

type cred<p:prin> =
  | MkCred : p:prin -> cred<p>

val login : u:prin -> str -> option<cred<u>>
let login u pw =
  if (* password ok? *)
    then Some (MkCred u)
  else None
module Authentication

type prin =
  | User : str -> prin
  | Admin : prin

type option<α> =
  | Some : α -> option<α>
  | None : option<α>

type cred<p:prin> =
  | MkCred : p:prin -> cred<p>

val login : u:prin -> str -> option<cred<u>>
let login u pw =
  if (* password ok? *)
    then Some (MkCred u)
  else None

Dependent function type
  • Formal parameter is bound in range type
  • Way to create values of a dependent type
module Authentication

type prin =
  | User : str -> prin
  | Admin : prin

type option<α> =
  | Some : α -> option<α>
  | None : option<α>

type cred<p:prin> =
  | MkCred : p:prin -> cred<p>

val login : u:prin -> str -> option<cred<u>>
let login u pw =
  if (* password ok? *)
    then Some (MkCred u)
    else None

Want login to be the only way to obtain a credential for a user
module Authentication

type prin =
  | User : str -> prin
  | Admin : prin

type option<α> =
  | Some : α -> option<α>
  | None : option<α>

private type cred<p:prin> =
  | MkCred : p:prin -> cred<p>

val login : u:prin -> str -> option<cred<u>>
let login u pw =
  if (* password ok? *)
  then Some (MkCred u)
  else None

Want login to be the only way to obtain a credential for a user
module EHR_ReferenceMonitor

private type record = { patient:prin; subject:str; data:str }

let read_data p' c r = r.data
module EHR_ReferenceMonitor

private type record = { patient: prin; subject: str; data: str }

read_data mediates access to patient records
  • The user must present a login credential

val read_data : p’: prin -> cred<p’> ->
  {x: record | hasPerm p’ (CanRead x)} -> str

let read_data p’ c r = r.data
module EHR_ReferenceMonitor

private type record = { patient: prin; subject: str; data: str }

read_data mediates access to patient records
- The user must present a login credential
- The record to be read must be a record \( x \) such that \( \text{hasPerm } p' \ (\text{CanRead } x) \) is true

Refinement type \( \{ x : \tau \mid \varphi \} \)
- \( \varphi \) is a FOL+equality formula over propositions

val read_data : p': prin -> cred<p'> ->
\[ \{ x : \text{record} \mid \text{hasPerm } p' \ (\text{CanRead } x) \} \] -> str

let read_data p' c r = r.data
module EHR_ReferenceMonitor

private type record = { patient: prin; subject: str; data: str }

val read_data : p’: prin -> cred<p’> ->
  { x: record | hasPerm p’ (CanRead x) } -> str

let read_data p’ c r = r.data
module EHR_ReferenceMonitor

private type record = { patient: prin; subject: str; data: str }

type perm =
  | CanWrite : record -> perm
  | CanRead : record -> perm

val read_data : p': prin -> cred<p'> ->
  {x: record | hasPerm p' (CanRead x)} -> str

let read_data p' c r = r.data
module EHR_ReferenceMonitor

private type record = { patient: prin; subject: str; data: str }

type perm =
  | CanWrite : record -> perm
  | CanRead : record -> perm

prop hasPerm :: prin -> perm -> *

val read_data : p': prin -> cred<p'> =>
  {x: record | hasPerm p' (CanRead x)} -> str

let read_data p' c r = r.data
module EHR_ReferenceMonitor

private type record = { patient:prin; subject:str; data:str }

type perm =
  | CanWrite : record -> perm
  | CanRead : record -> perm

prop hasPerm :: prin -> perm -> *

assume Ax1 :
  forall r:record, hasPerm Admin (CanRead r)

val read_data : p’:prin -> cred<p’> ->
  {x:record | hasPerm p’ (CanRead x)} -> str

let read_data p’ c r = r.data

Security assumptions
- establish ground facts and inference rules for policy
module EHR_ReferenceMonitor

Additional policy rule

assume A\text{x2} : \forall r:record. \text{hasPerm} r.\text{patient} (\text{CanRead} r)

Can write functions with post-conditions in return values

val check : a:prin \to b:perm \to \{z:bool \mid z=\text{true} \Rightarrow \text{hasPerm} a b\}
let check a b = match a, b with
| Admin, CanRead _ \to true
| _, CanRead r \to a = r.\text{patient}
| _ \to false
val check : a:prin -> b:perm -> {z:bool | z=true => hasPerm a b}
let check a b = match a, b with
  | Admin, CanRead _ -> true
  | _, CanRead r -> a = r.patient
  | _ -> false

Ax1 : forall r:record, hasPerm Admin (CanRead r)
Ax2 : forall r:record, hasPerm r.patient (CanRead r)
a : prin
b : perm
a = Admin
b = CanRead tmp

true : {z:bool | z=true => hasPerm a b}

not (true=true => hasPerm a b)
val check : a:prin -> b:perm -> {z:bool | z=true => hasPerm a b}
let check a b = match a, b with
| Admin, CanRead _  -> true
| _, CanRead r      -> a = r.patient
| _                  -> false

Ax1 : forall r:record, hasPerm Admin (CanRead r)
Ax2 : forall r:record, hasPerm r.patient (CanRead r)

true : {z:bool | z=true => hasPerm a b}
not (true=true => hasPerm a b)
val check : a:prin -> b:perm -> {z:bool | z=true => hasPerm a b}
let check a b = match a, b with
| Admin, CanRead _ -> true
| __, CanRead r         -> a = r.patient
| __                        -> false
Type Checking

val check : a:prin -> b:perm -> {z:bool | z=true => hasPerm a b}
let check a b = match a, b with
  | Admin, CanRead _  -> true
  | _, CanRead r     -> a = r.patient
  | _                 -> false

Typing context \( \Gamma \)

Well-typed?

(a = r.patient) : {z:bool | z=true => hasPerm a b}

Sat?

not ((a = r.patient)=true => hasPerm a b)

Z3

Unsat!
val check : a:prin -> b:perm -> {z:bool | z=true => hasPerm a b}
let check a b = match a, b with
    | Admin, CanRead _  -> true
    | _, CanRead r     -> a = r.patient
    | _                 -> false

Ax1 : forall r:record, hasPerm Admin (CanRead r)
Ax2 : forall r:record, hasPerm r.patient (CanRead r)

Well-typed?
false : {z:bool | z=true => hasPerm a b}

Sat?
not (false=true => hasPerm a b)

Z3
Unsat!
val check : a:prin -> b:perm -> {z:bool | z=true => hasPerm a b}
let check a b = match a, b with
  | Admin, CanRead _  -> true
  | _, CanRead r     -> a = r.patient
  | _                 -> false
Type Checking

- Is application code using API correctly?

```ocaml
val ehr_app : p:prin -> cred<p> -> record -> option<str>

let ehr_app : p c rec =
  if check p (CanRead rec) then Some (read_data p c rec)
  else None
```
Type Checking

- Is application code using API correctly?

```plaintext
val ehr_app : p:prin -> cred<p> -> record -> option<str>

let ehr_app : p c rec =
  if check p (CanRead rec) then Some (read_data p c rec)
  else None
```

Well-typed? `rec : {x:record | hasPerm p (CanRead x)}`
Type Checking

- Is application code using API correctly?

```ocaml
val ehr_app : p:prin -> cred<p> -> record -> option<str>

let ehr_app : p c rec =
  if check p (CanRead rec) then Some (read_data p c rec)
  else None

Establishes the pre-condition

Well-typed? rec : {x:record | hasPerm p (CanRead x)}
```
Compiling with Explicit Security Proofs

- Proofs can be communicated between systems to attest authorization rights
- Proofs can be logged for auditing access rights
- Mobile code/plugins with explicit proofs can be checked before execution
- Target programs can be checked with a small TCB
  - Translation validation to catch compiler bugs
Types for Proofs

- Proof terms will have type pf<α>
- Types for logical connectives

  \[
  \text{type not}<\alpha> \\
  \text{type and}<\alpha,\beta> \\
  \text{type imp}<\alpha,\beta> \\
  \text{...}
  \]

- Universal quantifiers as dependent functions

  \[
  \text{forall } r: \text{record}, \ \text{hasPerm Admin } (\text{CanRead } r) \\
  \quad \&\& \ \text{hasPerm Admin } (\text{CanWrite } r)
  \]

  \[
  r: \text{record} \to pf < \text{and } \text{hasPerm Admin } (\text{CanRead } r), \\
  \text{hasPerm Admin } (\text{CanWrite } r)>>
  \]
Embedding Proof Terms in Source

- Before translation to target, “derefine” program
- Remove refinement types (not in DCIL)
- Each expression paired with proof of refinement

$$\{x : \tau \mid \varphi\}$$

$$(x : \tau \ast pf<\varphi>)$$

Dependent pair type
- $x$ is value of first component
- $x$ is bound in type of second component
Proof Combinators

- General proof constructors

  \[
  \begin{align*}
  \text{Tru} & : \quad \text{pf}<\text{True}> \\
  \text{Contra} & : \forall \alpha. \quad \text{pf}<\alpha> \implies \text{pf}<\neg\alpha> \implies \text{pf}<\text{False}> \\
  \text{UseFalse} & : \forall \alpha. \quad \text{pf}<\text{False}> \implies \text{pf}<\alpha> \\
  \text{Imp} & : \forall \alpha, \beta. \quad (\text{pf}<\alpha> \implies \text{pf}<\beta>) \implies \text{pf}<\text{imp}<\alpha,\beta>> \\
  \text{Lift} & : \forall \alpha. \quad \alpha \implies \text{pf}<\alpha> \\
  \ldots
  \end{align*}
  \]

- Specialized proof constructors for each prop

  \[
  \begin{align*}
  \text{Sub_1_hasPerm} & : \quad \text{p1:prin} \implies \text{p2:prin} \implies \text{q:perm} \implies \\
  & \quad \text{pf}<\text{eq_prin} \text{p1} \text{p2}> \implies \\
  & \quad \text{pf}<\text{hasPerm} \text{p1} \text{q}> \implies \\
  & \quad \text{pf}<\text{hasPerm} \text{p2} \text{q}>
  \end{align*}
  \]
Proof Term for check

val check : a:prin -> b:perm ->
  {z:bool | z=true => hasPerm a b}
val check : a:prin -> b:perm ->
(z:boolean * pf<imp<eq_boolean z true, hasPerm a b>>)

Derefined return type
Proof Term for check

val check : a:prin -> b:perm ->
(z:bool * pf<imp<eq_bool z true, hasPerm a b>>)

let check a b = match a, b with
| Admin, CanRead _ -> (true, pf₁)
| _, CanRead r       -> (a = r.patient, pf₂)
| _                   -> (false, pf₃)

Need to supply a proof for each branch
Proof Term for check

val check : a:prin -> b:perm ->
  (z:bool * pf<imp<eq_bool z true, hasPerm a b>>)

let check a b = match a, b with
  | Admin, CanRead _  -> (true, pf₁)
  | _, CanRead r     -> (a = r.patient, pf₂)
  | _                -> (false, pf₃)

Unreasonable to write these manually

pf₁ =
  (Imp <eq_bool true true, hasPerm a b>
   (λfoo: pf<eq_bool true true>.
    (Sub_1_hasPerm Admin a b
     (Refl_prin a : (pf<eq_prin Admin a>))
     (Lift <hasPerm Admin b>
      (Ax1 tmp)))))))
Z3v2 produces a proof for an unsat formula
let proof_fun (neg#gensym@0_185:Prims.pf<Prims.l_not<Prims.pf<InfoFlow.Eq_tii005(gensym@29_1#gensym@29_1)(InfoFlow.Low)>>) =
let p79_f76 = neg#gensym@0_185 in
let p81_f82 = InfoFlow.Iff_tii005<;gensym@29_1#gensym@29_1;InfoFlow.Low> in
let p80_f84 = Prims.Mono_not<InfoFlow.Eq_tii005(gensym@29_1#gensym@29_1)(InfoFlow.Low),
                InfoFlow.Eq_tii005(InfoFlow.Low)(gensym@29_1#gensym@29_1);p81> in
let p78_f85 = Prims.Mp_iff<Prims.l_not<Prims.pf<InfoFlow.Eq_tii005(gensym@29_1#gensym@29_1)(InfoFlow.Low)>>,
              Prims.l_not<Prims.pf<InfoFlow.Eq_tii005(InfoFlow.Low)(gensym@29_1#gensym@29_1)>>;p79;p80> in
let p88_f74 = (InfoFlow.Refl_tii005<(gensym@30_3#gensym@30_3)) in
let p89_f90 = InfoFlow.Iff_tii005<;gensym@30_3#gensym@30_3;InfoFlow.Low> in
let p87_f91 = Prims.Mp_iff<InfoFlow.Eq_tii005(gensym@30_3#gensym@30_3)(InfoFlow.Low),
                InfoFlow.Eq_tii005(InfoFlow.Low)(gensym@30_3#gensym@30_3);p88;p89> in
let p92_f73 = (InfoFlow.Refl_tii005<(gensym@30_3#gensym@30_3)) in
let p86_f83 = InfoFlow.Trans_tii005<;InfoFlow.Low;gensym@30_3#gensym@30_3;gensym@29_1#gensym@29_1;p87;p92> in
let p77_f93 = Prims.Contra_2<InfoFlow.Eq_tii005(InfoFlow.Low)(gensym@29_1#gensym@29_1);p78;p86> in
p77_f93
let proof = Proof_by_contra <InfoFlow.Eq_tii005(gensym@29_1#gensym@29_1)(InfoFlow.Low);proof_fun>
## Automatic Proof Construction

<table>
<thead>
<tr>
<th>application</th>
<th>LOC</th>
<th># of proof obligations</th>
<th># of Z3 proof rule applications</th>
<th># of FINE proof rule applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>mini ehr</td>
<td>32</td>
<td>5</td>
<td>131</td>
<td>432</td>
</tr>
<tr>
<td>mini iflow</td>
<td>111</td>
<td>20</td>
<td>1047</td>
<td>2912</td>
</tr>
<tr>
<td>mini lookout</td>
<td>190</td>
<td>7</td>
<td>494</td>
<td>3942</td>
</tr>
</tbody>
</table>

- Z3 proof terms map to several FINE proof terms
  - n-ary logical connectives converted to binary
  - Our quantifier intro proofs uses many subterms
  - Our monotonicity proofs reason about all subterms

- **Opportunities for improvement**
  - Simplifying Z3 proofs before translation
  - Handling tautologies and rewrites more generally
DCIL

- Extends the .NET Common Intermediate Language

```csharp
class C<T1::**, T2::A, p:prin, λx:T1.T>
```

Type parameters
DCIL

- Extends the .NET Common Intermediate Language

```csharp
class C <T1::* , T2::A, p:prin, λx:T1.T>
```

- Type parameters
- Affine type parameters
DCIL

- Extends the .NET Common Intermediate Language

class C <T1::*, T2::A, {p: prin, \lambda x: T1.T}>
DCIL

- Extends the .NET Common Intermediate Language

class C <T1::**, T2::A, p:prin, $\lambda x:T1.T$>
Translating FINE to DCIL

- Type and data constructors

  type prin
  type cred =
  MkCred :
  p:prin -> cred p

  abstract class prin {}
  abstract class Cred<p:prin>{
    class MkCred<p:prin>
      : Cred<p> { prin p; }
  }

- Dependent functions

  login :
  p:prin -> cred p

  abstract class
    DepArr<arg::*:*, ret::*:arg => *>
  {
    (ret x) App (x:arg) {}
  }
  class login
    : DepArr<prin, λx:prin.cred x>
    {
      (cred p) App (p:prin) { ... }
    }
Conclusions

- **FINE+DCIL** tries to bridge the gap between policy specification and enforcement.

- Provides a high-level source programming model, while retaining a small TCB for verification of proof-carrying target code.

- But, lots left to do …
  - Build more applications and integrate with .NET
  - Optimize proof extraction
Thanks!
Policy for Patient Records

- Recall the example SecPAL policy:
  - P can read P’s record r
  - D can read P’s record r if D is treating P and r.Subject <> HIV

- In FINE:
  - assume forall r:record. hasPerm r.patient (CanRead r)
Policy for Patient Records

- Recall the example SecPAL policy:

  P can read P’s record r
  
  D can read P’s record r if D is treating P and r.Subject <> HIV

- In FINE:

  assume forall r:record. hasPerm r.patient (CanRead r)

  prop isTreating :: prin -> prin -> *

  assume forall d:prin, r:record.
  (isTreating d r.patient && r.subject <> “HIV”) =>
  hasPerm d (CanRead r)
Stateful Policy for Patient Records

- `isTreating d p` can change over time
- In FINE, model state by passing around a store

```
assume forall d:prin, r:record, st:state.
  (isTreating d r.patient st && r.subject <> "HIV") =>
  hasPerm d (CanRead r)
```

Policy can quantify over states
Stateful Policy for Patient Records

- \texttt{isTreatiing d p} can change over time
- In \texttt{FINE}, model state by passing around a store

\begin{verbatim}
assume forall d:prin, r:record, st:state.
   (isTreatiing d r.patient st && r.subject <> "HIV") =>
   hasPerm d (CanRead r)
\end{verbatim}

Policy can quantify over states

Propositions can refer to current state
Stateful Policy for Patient Records

- **isTreating d p** can change over time
- **In FINE**, model state by passing around a store

Assume for all d:prin, r:record, st:state.

\[ \text{isTreating } d \text{ r.patient } st \land \text{r.subject } \neq \text{“HIV”} \Rightarrow \text{hasPerm } d \text{ (CanRead r)} \]

Propositions can refer to current state

Program operations can change state of world

\[
\text{val visit_doctor:}
\text{p:prin } \rightarrow \text{cred<p> } \rightarrow \text{doc:prin } \rightarrow \text{(st:state) } \rightarrow \text{({st’:state } | \text{isTreating doc p st’})}
\]
Stateful Policy for Patient Records

- `isTreating d p` can change over time.
- In FINE, model state by passing around a store.

```plaintext
assume forall d:prin, r:record, st:state.
  (isTreating d r.patient st && r.subject <> "HIV") =>
  hasPerm d (CanRead r)
```

**Policy can quantify over states**

**Propositions can refer to current state**

- Program operations can change state of world

```plaintext
val visit_doctor: p:prin -> cred<p> -> doc:prin ->
  (st:state * !stateToken<st>) =>
  ({st’:state | isTreating doc p st’} * !stateToken<st’>)
```

**Affine types to ensure stale states are not re-used**
Equality Propositions

- Need a proposition for equality of terms

  \[ \text{prop eq} :: \forall \alpha. \alpha \rightarrow \alpha \rightarrow * \]

  - Requires second-order quantification over types
  - Would require bigger change to CIL

- Instead, first-order treatment of equality

- Specialized proof constructors for each type \( t \)

  \[
  \text{type eq}_t :: t \rightarrow t \rightarrow * =
  \]

  \[
  \mid \text{Refl}_t : a : t \rightarrow \text{pf<eq}_t a a > \]

  \[
  \ldots
  \]

% ./fine.exe test/<file>.f9
   --skip_translation
   --extract_proofs
   --proof_stats