Refinement Types for Secure Implementations

Jesper Bengtson, Uppsala University
Karthikeyan Bhargavan, Microsoft Research
Cédric Fournet, Microsoft Research
Andrew D. Gordon, Microsoft Research
Sergio Maffeis, Imperial College and UC Santa Cruz

http://research.microsoft.com/F7
Refinement types for secure implementations

\begin{verbatim}
 type payload = string
 \leadsto type payload = x: string \{ Send(a,x) \}
\end{verbatim}

1. Context, motivation
   - Verifying protocol implementations
   - Logics for authorization and access control
2. A refinement-typed concurrent lambda-calculus (theory)
3. Cryptography by kinding, subtyping, and sealing
4. Experimental results (F#)

F# demo available on request
VERIFYING IMPLEMENTATIONS
Verifying protocol implementations

• Cryptographic protocols specs, models, and implementations
  – Protocol specifications remain largely informal
  – Formal models are short, abstract, hand-written
  – Specs, models, and implementations drift apart...

• Our current approach is to verify reference implementations
  – Executable code has more details than models
  – Executable code has better tool support:
    types, compilers, testing, debuggers, libraries, verification
The F#|FS2PV|PV tool chain [CSF’06]
Even with some aggressive abstraction, our tools are hitting long and unpredictable run times
Can we do better with source-level security types?

A Reference InfoCard Implementation

Safety Results

<table>
<thead>
<tr>
<th>Name</th>
<th>LOC</th>
<th>Crypto Ops</th>
<th>Auth</th>
<th>Secrecry</th>
<th>Verif Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>SelfIssued-SOAP</td>
<td>1410 (80)</td>
<td>9,3</td>
<td>A1-A3</td>
<td>S1,S2</td>
<td>38s</td>
</tr>
<tr>
<td>UserPassword-TLS</td>
<td>1426(96)</td>
<td>0,5,17,6</td>
<td>A1-A3</td>
<td>S1,S2</td>
<td>24m40s</td>
</tr>
<tr>
<td>UserPassword-SOAP</td>
<td>1429(99)</td>
<td>9,11,17,6</td>
<td>A1-A3</td>
<td>S1,S2</td>
<td>20m53s</td>
</tr>
<tr>
<td>UserCertificate-SOAP</td>
<td>1429(99)</td>
<td>13,7,11,6</td>
<td>A1-A3</td>
<td>S1-S3</td>
<td>66m21s</td>
</tr>
<tr>
<td>UserCertificate-SOAP-\v</td>
<td>1429(99)</td>
<td>7,5,7,4</td>
<td>A3 Fails!</td>
<td>S1-S3</td>
<td>10s</td>
</tr>
</tbody>
</table>
Protocol verification vs program verification

• Cryptographic tools (ProVerif, CryptoVerif, AVISPA) are great but...
  – They use global analyses, not suitable for “giant” protocols
  – Security applications combine both crypto protocols and ordinary code

• General-purpose verification techniques are also making rapid progress
  – They can deal with much larger programs
  – They don’t directly support protocol verification with cryptographic primitive, active adversaries, etc

• Using dependent types, we can integrate cryptographic protocol verification as a part of program verification
Motivation (2/2)

AUTHORIZATION POLICIES & PARTIALLY-TRUSTED CODE
Authorization Logics

• Authorization policies are complex and changing
  – How to express policies? \( \rightarrow \) use some logic [Datalog, DCC, SecPAL]
  – How to **statically** check whether a system implements a policy?

• Authorization by typing [ESOP’05, CSF’07]:
  – Represent code in a process calculus
  – Represent policies as logical formulas within types
  – Generalize type and effect systems for formal cryptography

• This work:
  – Typecheck implementations (in F#) against policies (in FOL)
  – Simplify the theory: we now derive formal cryptography
Suppose there is a global set of formulas, the log

To evaluate assume $C$, add $C$ to the log.

To evaluate assert $C$,

- If $C$ logically follows from the logged formulas, the assertion succeeds; otherwise, the assertion fails.
- The log is only for specification purposes; it does not affect execution.

Our use of first-order logic predicates generalizes conventional assertions (like assert $i>0$ in eg Spec#)

- Such predicates can also represent security-related concepts like roles, permissions, events, compromises, access rights,...
Example: access control for files

- **Untrusted** code may call a **trusted** library
- Trusted code expresses security policy with assumes and asserts

- Each policy violation causes an assertion failure
- We **statically** prevent any assertion failures by typing

```plaintext
type facts = CanRead of string | CanWrite of string

let read file = assert(CanRead(file)); ...
let delete file = assert(CanWrite(file)); ...

let pwd = "C:/etc/password"
let tmp = "C:/temp/tempfile"

assume CanWrite(tmp)
assume ∀x. CanWrite(x) → CanRead(x)

let untrusted() =
  let v1 = read tmp in // ok, by policy
  let v2 = read pwd in // assertion fails
```

Typechecking failed at acls.fs(39,9)–(39,12)
Error: Cannot establish formula CanRead(pwd)
Logging dynamic events

- Security policies often stated in terms of dynamic events such as role activations or data checks

- We mark such events by adding formulas to the log with `assume`

```plaintext
type facts = ... | PublicFile of string
let read file = assert(CanRead(file)); ...
let readme = "C:/public/README"

// Dynamic validation:
let publicfile f =
  if f = "C:/public/README" || ...
  then assume (PublicFile(f))
  else failwith "not a public file"

assume \forall x. PublicFile(x) → CanRead(x)

let untrusted() =
  let v2 = read readme in // assertion fails
  publicfile readme; // validate the filename
  let v3 = read readme in () // now, ok
```
Access control with refinement types

\[
\begin{align*}
\textbf{val} & \text{ read: file:string}\{\text{CanRead(file)}\} \rightarrow \text{string} \\
\textbf{val} & \text{ delete: file:string}\{\text{CanDelete(file)}\} \rightarrow \text{unit} \\
\textbf{val} & \text{ publicfile: file:string} \rightarrow \text{unit}\{\text{PublicFile(file)}\}
\end{align*}
\]

- Preconditions express access control requirements
- Postconditions express results of validation
- We typecheck partially trusted code to guarantee that all preconditions (and hence all asserts) hold at runtime

- Related work: eg types for stack inspection (Pottier, Skalka, Smith), Aura (Zdancevic et al)
a concurrent call-by-value lambda-calculus with refinement types

“RCF”
Syntax for values and expressions

• A concurrent call-by-value lambda-calculus

• A formal core for F#

• Specifications expressed by **assume** and **assert** over logic formulas
We use a standard small-step reduction semantics; runtime configurations are expressions of the form

\[ S ::= (\forall a_1) \ldots (\forall a_\ell) \left( \prod_{i \in 1..m} \text{assume } C_i \right) \stackrel{\hat{\cdot}}{\rightarrow} \left( \prod_{j \in 1..n} c_j M_j \right) \stackrel{\hat{\cdot}}{\rightarrow} \left( \prod_{k \in 1..o} L_k \{e_k\} \right) \]

active assumptions pending messages running threads

An expression is \textbf{safe} when, for all runs of A, \textbf{all assertions succeed}
Refinement types

- An assembly of standard components
  \[ H, T, U ::= \text{type} \]
  \[ \alpha \quad \text{type variable} \]
  \[ \text{unit} \quad \text{unit type} \]
  \[ \Pi x : T . U \quad \text{dependent function type (scope of } x \text{ is } U) \]
  \[ \Sigma x : T . U \quad \text{dependent pair type (scope of } x \text{ is } U) \]
  \[ T + U \quad \text{disjoint sum type} \]
  \[ \mu \alpha . T \quad \text{iso-recursive type (scope of } \alpha \text{ is } T) \]
  \[ \{ x : T \mid C \} \quad \text{refinement type (scope of } x \text{ is } C) \]

- For example, \texttt{type} filename = x:string\{ CanRead(x) \} declares a type of strings for filename with the read access right
Safety by typing

\[ E \vdash \Diamond \quad \text{E is syntactically well-formed} \]
\[ E \vdash T \quad \text{in } E, \text{ type } T \text{ is syntactically well-formed} \]
\[ E \vdash C \quad \text{formula } C \text{ is derivable from } E \]
\[ E \vdash T :: \nu \quad \text{in } E, \text{ type } T \text{ has kind } \nu \in \{\text{pub, tnt}\} \]
\[ E \vdash T <: U \quad \text{in } E, \text{ type } T \text{ is a subtype of type } U \]
\[ E \vdash A : T \quad \text{in } E, \text{ expression } A \text{ has type } T \]

\[ \mu ::= \]
\[ \alpha :: \nu \quad \text{kinding} \]
\[ \alpha <: \alpha' \quad \text{subtyping} \]
\[ \alpha : T \uparrow \quad \text{name (of channel type)} \]
\[ x : T \quad \text{variable (of any type)} \]
\[ E ::= \mu_1, \ldots, \mu_n \quad \text{environment} \]

An expression \( A \) is \textit{safe} if and only if, in all evaluations of \( A \), all assertions succeed.

**Theorem 1 (Safety by Typing)** If \( \emptyset \vdash A : T \) then \( A \) is safe.
Rules for refinements

We can refine any type with any formula that follows from $E$

\[ \begin{align*}
  E \vdash T <: T' \\
  E \vdash \{x : T \mid C\} <: T'
\end{align*} \]

\[ \begin{align*}
  E \vdash M : T & \quad E \vdash C[M/x] \\
  \frac{}{E \vdash M : \{x : T \mid C\}}
\end{align*} \]

Rules for assume and assert

\[ \begin{align*}
  E \vdash \diamond \quad fnfv(C) \subseteq \text{dom}(E) & \quad E \vdash C \\
  \frac{}{E \vdash \text{assume } C : \{- : \text{unit} \mid C\}} & \quad \frac{}{E \vdash \text{assert } C : \text{unit}}
\end{align*} \]

We can assume any formula

We can assert any formula that follows from $E$
Robust safety

• Active opponents can intercept all communications, rewrite messages, inject new messages, but not break cryptography [Needham-Shroeder’76, Dolev-Yao’83]

• We represent opponents as top-level programs, with access to selected libraries and functions without asserts, but not necessarily well-typed
Robust safety by typing

An expression $A$ is *robustly safe* iff the application $O A$ is safe for all opponents $O$.

Let a type $T$ be *public* if and only if $T <: \text{Un}$. Let a type $T$ be *tainted* if and only if $\text{Un} <: T$.

**Theorem 2 (Robust Safety by Typing)**

*If $\emptyset \vdash A : \text{Un} \text{ then } A$ is robustly safe.*

**Lemma 1 (Universal Type)**

*There is a type $\text{Un}$ such that $E \vdash \emptyset$ implies $E \vdash \text{Un} <: T$ where $T$ ranges over $\text{unit}$, $\Pi x : \text{Un}$. $\Sigma x : \text{Un}$. $\text{Un} + \text{Un}$, $\mu \alpha. \text{Un}$, and $\text{ChanTypeUn}$.***

**Lemma 2 (Opponent Typability)**

*If $O$ is an opponent and $E \vdash u : \text{Un}$ for each $u \in \text{fnfv}(O)$, then $E \vdash O : \text{Un}$.***
Typed functional encoding of cryptography

- In prior (pi calculus) work, we included a selection of cryptographic primitives and typing rules
- We now derive typed cryptographic functions from `seals` [Morris’73]

```
type α Seal = (α → Un) * (Un → α)
val mkSeal: unit → α Seal

  - We rely on functions and fresh names
  - We obtain a symbolic model, similar to oracles
  - We then code typed symbolic implementations for standard primitives

type α SealChan = ((α * Un) list) Pi.chan
let seal: α SealChan → α → Un = fun s M →
  let state = recv s in match first (left M) state with
  | Some(a) → send s state; a
  | None →
    let a: Un = Pi.name "a" in
    send s ((M,a)::state); a

let unseal: α SealChan → Un → α = fun s a →
  let state = recv s in match first (right a) state with
  | Some(M) → send s state; M
  | None → failwith "not a sealed value"

let mkSeal () : α Seal =
  let s:α SealChan = chan "seal" in
  send s []; (seal s, unseal s)
```
Example: symbolic implementation for MACs

Message authentication codes (MAC) provide integrity using joint hashes of a shared key and authenticated message

1. We declare an abstract interface for MACs, eg HMACSHA1:

   ```ocaml
   type αhkey
   type hmac
   val mkHKey: unit → αhkey
   val hmacsha1: α hkey → α pickled → hmac
   val hmacsha1Verify: α hkey → Un → hmac → α pickled
   ```

2. We implement (and typecheck) HMACSHA1 symbolically using seals

   ```ocaml
   type α hkey = HK of (α pickled) Seal
   type hmac = HMAC of Un
   let mkHKey (): α hkey = mkSeal()
   let hmacsha1 (HK(seal,unseal) text = HMAC(seal text)
   let hmacsha1Verify (HK(seal,unseal)) text (HMAC h) =
      if unseal h = text then text else failwith "hmac verify failed"
   ```
EXPERIMENTAL RESULTS
Implementation for F#

- We use extended interfaces
  - We typecheck implementations
  - We kindcheck interfaces (all values must be public)
  - We generate .fsi interfaces by erasure from .fs7

- We support a large subset of F#
  - ADTs, records, patterns, refs
  - Value- and type-polymorphism

- We do some type inference
  - Plain F# types as usual
  - Refinements require annotations

- We call Z3, an SMT prover, on each non-trivial proof obligation
Libraries

- We annotate (and retype) some libraries
- We also provide concrete implementations for some system libraries (without extended typing)

```fsharp
open System.Security.Cryptography

type α hkey = bytes

type hmac = bytes

let mkHKey () = mkNonce()

let hmacsha1 (k:α hkey) (x:bytes) =
    (new HMACSHA1 (k)).ComputeHash x

let hmacsha1Verify (k:α hkey) (x:bytes) (h:bytes) =
    let hh = (new HMACSHA1 (k)).ComputeHash x
    if h = hh then x else failwith "hmac verify failed"
```
<table>
<thead>
<tr>
<th>Sample</th>
<th>.fs</th>
<th>.fs7</th>
<th>time (S)</th>
<th>Z3 proofs</th>
<th>time (S) / proof</th>
<th>proofs / loc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logs and queries</td>
<td>37</td>
<td>16</td>
<td>2.80</td>
<td>6</td>
<td>0.47</td>
<td>0.16</td>
</tr>
<tr>
<td>MAC protocols</td>
<td>40</td>
<td>12</td>
<td>2.50</td>
<td>3</td>
<td>0.83</td>
<td>0.08</td>
</tr>
<tr>
<td>Principals &amp; partial Compromise</td>
<td>48</td>
<td>26</td>
<td>3.10</td>
<td>12</td>
<td>0.26</td>
<td>0.25</td>
</tr>
<tr>
<td>Certificate chains</td>
<td>61</td>
<td>21</td>
<td>3.65</td>
<td>19</td>
<td>0.19</td>
<td>0.31</td>
</tr>
<tr>
<td>File access control</td>
<td>104</td>
<td>34</td>
<td>8.30</td>
<td>16</td>
<td>0.52</td>
<td>0.15</td>
</tr>
<tr>
<td>Flexible signatures</td>
<td>167</td>
<td>52</td>
<td>14.60</td>
<td>28</td>
<td>0.52</td>
<td>0.17</td>
</tr>
<tr>
<td>Typed libraries</td>
<td>440</td>
<td>146</td>
<td>12.10</td>
<td>12</td>
<td>1.01</td>
<td>0.03</td>
</tr>
</tbody>
</table>

- We verify non-trivial code and properties
- Ask for a demo!
Sample: flexible digital signatures

• Signing keys are often shared among several protocols, e.g. XML digital signatures provide extreme, programmable flexibility

• How to avoid ambiguous signatures?
  – When signing, we must ensure that
    (1) the signed content authenticates the protocol message; and
    (2) the signed content cannot be interpreted otherwise

• We type signed contents with a refinement that specifies the different possible interpretations of the signature
  – For signing either XML requests or XML responses, we use

\[
\text{type verified} = x: \text{siginfo}\{
(\forall \text{id, } b. (\text{Mem(IdHdr}(\text{id}), x) \land \text{Mem(RequestBody}(b), x))
\Rightarrow \text{Request}(\text{id}, b) )
\land (\forall \text{id, req, b. (Mem(IdHdr}(\text{id}), x) \land \text{Mem(ResponseBody}(b), x)
\land \text{Mem(InReplyTo}(\text{req}), x)) \Rightarrow \text{Response}(\text{id}, \text{req, b}) ) \}
\]
Context, discussion

• RCF is an assembly of standard parts, generalizing ad hoc constructions in language-based security
  – **FPC** (Plotkin 1985, Gunter 1992) – core of ML and Haskell
  – Concurrency in style of the **pi-calculus** (Milner, Parrow, Walker 1989) but for a lambda-calculus (like 80s languages PFL, Poly/ML, CML)
  – Formal crypto derived by coding up **seals** (Morris 1973)
  – Security specs via logical **assume/assert** (Floyd, Hoare, Dijkstra 1970s), generalizing eg correspondences (Woo and Lam 1992)
  – Typing with dependent functions, pairs, subtyping (Cardelli 1988), and **refinement types** (Pfenning 1992, ...) aka **predicate subtyping**
  – **Public/tainted kinds** to track data that may flow to or from the opponent, as in Cryptyc (Gordon, Jeffrey 2002)

• Our experimental approach is to target existing languages & tools
  – Checker for existing language (F#) with codebase of security-critical code
  – Refinement types based on FOL, not bespoke authorization logic or type- and-effect system, to benefit from general-purpose verification tools
Summary

• We use formulas as computational effects to integrate program logics and type systems, with applications to security.
  – We embed formulas using refinement types
  – We represent active opponents as Un-typed contexts using subtyping
  – We encode symbolic cryptography using typed seals

• We obtain a first tool for verifying implementations of security policies and protocols by typing source code
  – More scalable and flexible than our prior translation to ProVerif

• We are experimenting with larger applications and examples
  – Concrete language design
  – Type inference, or type compilation

• http://research.microsoft.com/F7