Certified software security with dependent types

Nikhil Swamy
F* is joint work with several people at MSR Redmond, MSR Cambridge, and MSR-INRIA; and interns from UCSD, Brown, Wisconsin, MIT, Princeton, Cornell, Berkeley.
No shortage of security vulnerabilities in low-level code

And these are just the ones detected by \GS etc.
Fix: Move to higher level languages ...

Higher abstractions

→ Improved productivity and fewer bugs
Secure?

- Your crypto protocol is implemented correctly
  - TLS renegotiation ➔ man in the middle
- Your web app framework has no backdoors
  - ASP.NET improper handling of session state
- Your web site blocks all code injection attacks
  - Samy JS.SpaceHero worm
- Your browser extension doesn’t steal your history
  - 66% of Chrome extensions can send your browsing history to a 3rd party
- Your cloud service is in compliance with (inter)national data privacy and retention laws
- ...
Recipe for certified secure software:

• Design it with security in mind, formally specify the properties you care about
• Implement it with high-level abstractions in a safe programming language
• Verify and compile your program with certified tools

Cost of software verification is still very high

Of course, attacks will remain:
- Physical (in)security
- Hardware attacks
- Incomplete/incorrect specs
...
A program verification tool

For a higher-order programming language
So, freely use high-level functional abstractions

With a specification language based on higher-order logic
Allows you to specify almost any program property

With a verifier built using a state-of-the-art theorem prover (Z3)
Aims to provide good automation

With a verifier that has itself been verified
Self-certified!
This talk: A brief overview of F*

• F*. What is it and what is it good for?
  – One example domain: Browser extension security

• A brief introduction to dependently typed lambda calculi

• Formally certifying the F* verifier
  – A “self-certifying” verifier
F* Source: core-ML with dependent refinement types

\[ x: \text{int} \rightarrow \{ y: \text{int} \mid x > y \} \]

Preserve types in .NET
class C<'a::int => *>

Interop with C#, VB.NET, F#, ...

Certified core type-checker!
Preserve types in .NET
class C<‘a::int => *>

Interop with
C#, VB.NET, F#,...

Certified core
type-checker!
WEB BROWSER EXTENSION SECURITY
1. $\frac{1}{3}$rd of Firefox users run extensions (~34 million users)
2. Popular Chrome extensions have thousands of users
mailto:joe@cs.brown.edu

Change mailto:

https://mail.google.com/mail/?view=cm&tf=1&to=joe@cs.brown.edu&cc=&su=&body=&fs=1

Change links to evil.com?
I might point out that in addition to having reliability at two sites, Matthias/Matthew/Steve have it at their apartment too. In addition, while I was a postdoc, I stayed in the apartment I stayed in also routinely had connections, which was a convenient situation for my research.

I was surprised to find that chaise is a word that exists. A chaise is a very pleasant and comfortable chair that can be reclined into a forward-facing position, with a low back and open top and two wheels. Some people use it as a form of transport for one or two people, typically one with an open top and two wheels. It is not a word that I use often, but it is certainly a useful one.
Access Control in Chrome

"permissions": [
  "tabs",
  "http://www.twitter.com/*",
  "http://api.bit.ly/",
  ...
]

1. Sensitive APIs
2. Extension runs on these URLs
Policy analysis:
Accessible URLs

Access to all data on all websites

Access to all data on one website

2—86 websites

1,137 extension policies
Policy analysis:
Access to history

why?

1,137 extension policies
desired, least-privilege security policy is *inexpressible*

Sends selected word to Google from *any website*
Access to all websites

Full History Access

Access to all data on all websites
IBEX: Rethinking Browser Extension Security
Extension in Fine Type-safe high-level language
Developers
• Write extension and policy in F*
• Use tools to ensure extension conforms to policy

Gallery
• Uses tools to ensure extension conforms to policy
• Uses visualizer to help understand policy

Users
• Trust curated extension gallery
• Install approved extensions
EXAMPLE:
ONLY READ TEXT IN <HEAD>
type elt

val getInnerText : elt -> string

val getTagName : elt -> string
type elt

val getInnerText : elt { CanRead e } -> string

val getTagName : elt
e:elt
-> s:string { EltTagName e s }
Secure DOM API

val getInnerText : 
  e:elt { CanRead e }  
-> string

val getTagName : 
  e:elt  
-> s:string { EltTagName e s }  

Policy

assume forall (e:elt) . EltTagName e "head" ➞ CanRead e
type elt

val getInnerText : e:elt { CanRead e } -> string

val getTagName : e:elt -> s:string { EltTagName e s }

assume forall (e:elt) . EltTagName e "head" ⇒ CanRead e

let read e = 
  if getTagName e = "head" then 
    getInnerText e 
  else 
    "not <head>"

F* checks pre- and post-conditions \textit{statically}

1. No manual code audit (only policy audit) 
2. No security exceptions (robust) and no runtime overhead (fast)
<table>
<thead>
<tr>
<th>Extension Name</th>
<th>Limited Extension Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gmail checker</td>
<td>Rewrites “mailto:” links to open Gmail compose page</td>
</tr>
</tbody>
</table>

Precise, fine-grained policies
Under the hood:

A brief review of typed lambda calculi
### Kinds $\kappa$

<table>
<thead>
<tr>
<th>Kinds $\kappa$</th>
<th>Types $t$</th>
<th>Values $v$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$*$</td>
<td>$* \Rightarrow *$</td>
<td>0, 1, 2, ...</td>
</tr>
<tr>
<td>$P$</td>
<td>$\text{int} \Rightarrow * \Rightarrow *$</td>
<td>true, false, ...</td>
</tr>
<tr>
<td></td>
<td>$\text{int} \Rightarrow \text{int} \Rightarrow P$</td>
<td></td>
</tr>
</tbody>
</table>
Stratified core language

Syntax: kinds, types, expressions

\[ \begin{align*}
\kappa & ::= P \mid \ast \mid E \mid A \mid x : t \Rightarrow \kappa \mid \alpha :: \kappa \Rightarrow \kappa' \\
\phi , t & ::= \alpha \mid T \mid x : t \rightarrow t' \mid \forall \alpha :: \kappa . t \mid t \, v \mid t \, t' \mid \lambda x : t . t' \mid x : t \{ \phi \} \\
v & ::= x \mid \lambda x : t . e \mid \Lambda \alpha :: \kappa . e \mid D \, t \, \bar{v} \mid \ldots \\
e & ::= v \mid e \, v \mid e \, t \mid \text{let } x = e \text{ in } e' \mid \text{match } v \text{ with } D \, \bar{a} \, \bar{x} \rightarrow e \text{ else } e' \\
& \quad \mid \text{assume } \phi \\
& \quad \mid \text{ref } v \mid v_1 ::= v_2 \mid ! v \mid \ldots 
\end{align*} \]

4 base kinds = 4 universes with controlled interactions
Theorem 1 (Type safety):
Well-typed F* programs do not go wrong.

Informally: Well-typed F* programs never have failing assertions
Formalizing the simply typed lambda calculus in F*
This talk: An overview of F*

• F*. What is it good for?
  – Example: Browser extension security

• Key elements of the language design

• Formally certifying the F* verifier
  – A “self-certifying” verifier
Many dependently typed languages

- Agda, Aura, ATS, Cayenne, Coq, DML, Epigram, F*, F7, Fine, Guru, PCML5, Ur, Trellys, ...

- Lots of nicely formalized theories
- Some stable, mature implementations
- But all implementations are *ad hoc*
Theorem?
If src.fst typechecks then it does not go wrong.
Theorem?
If src.fst typechecks then it does not go wrong.
Theorem:
If src.fst typechecks then it does not go wrong.
Bootstrapping a certified F* verifier
1. Type system of F* formalized in Coq

Reference.v (Coq)

Inductive Typing : environment -> expr -> typ -> Type :=
   | TVar : ...
   | Tapp : ...
   | ...

with Kinding : environment -> typ -> kind -> Type :=
   | KVar: ...
   | ...

Reference.v (Coq)
Bootstrapping a certified F* verifier

2. But, the same judgments can also be represented in F*

Reference.fst \((F^*)\)

type Typing :: environment => expr => typ => P =
  | TVar : ...
  | Tapp : ...
  | ...
and Kinding :: environment => typ => kind => P =
  | KVar: ...
  | ...

```
Theorem 2 (Consistency of P):

Terms in the P-universe can be embedded in CiC, where the embedding is a simulation.

Informally: For every well-typed F* program (e:t), where (t::P), there exists a Coq program \([e]\) that can be given the type \([t]\) and every step of reduction of e in F* is matched by one or more reduction steps of \([e]\) in Coq

More informally: The P fragment of F* corresponds to a fragment of Coq
Bootstrapping a certified F* verifier

3. A reference specification of the F* type system
Shared between CiC and F*, justified by Theorem 2

Reference.v
Inductive Typing := ...
with Kinding := ...

Reference.fst
Bootstrapping a certified F* verifier
4. Metatheory of F* formalized in Coq

Metatheory.v (Coq)

Theorem forall G A e t, Typing G e t
  -> ...
  -> steps (A, e) (A', e')
  -> Typing G e' t
Bootstrapping a certified F* verifier

4. A reference specification of the F* type system
Shared between CiC and F*, justified by Theorem 2

- Reference.v
  - Inductive Typing := ...
  - with Kinding := ...
- Reference.fst

- Metatheory.v (Coq)
  - Theorem ...

F*       Coq
Bootstrapping a certified F* verifier
5. Implementing a type checker for F* in F*

---

**Reference.v**
Inductive Typing := ...
with Kinding := ...

**Reference.fst**

tc is a partial function, with exceptions, state, Z3 calls etc

**Coretc.fst**
tc: G:env -> e:expr
   -> t:typ
   -> Typing G e t

**Metatheory.v (Coq)**
Theorem ...

---

F* | Coq
Bootstrapping a certified F* verifier
6. Run tc on itself extracting a typing derivation

Reference.v
Inductive Typing := ...
with Kinding := ...

Reference.fst

Certificate for tc

Coretc.fst

tc: G:env -> e:expr
-> t:typ
-> Typing G e t

Coq

Metatheory.v (Coq)
Theorem ...
Bootstrapping a certified F* verifier
7. Translate certificate to Coq and type check in Coq
Bootstrapping a certified F* verifier

8. Theorem: If \((tc \ (\text{embed } G) \ (\text{embed } e) \ (\text{embed } t) \Rightarrow ^* \ v)\)
then Typing \(G \ e \ t\)

Modulo correctness of F* parser etc. Modulo Z3, but can also extract and typecheck proof terms from Z3 (PLDI ’10)

Generate \(~8GB\) of Coq source files verified automatically by Coq in 24-machine days

 Metatheory.v (Coq)

Theorem ...

Coq

UNTRUSTED TRANSLATION

Reference.v

Reference.fst

Certificate for tc

Coretc.fst

\(tc: G:env \rightarrow e:expr\)
\(\rightarrow t:typ\)
\(\rightarrow\) Typing \(G \ e \ t\)
F* is emancipated from Coq

Theorem:
If src.fst typechecks then it does not go wrong.
If you want provably secure software:

• Design it with security in mind

• Implement it with high-level abstractions in a safe programming language

• Check and compile your program with certified tools (verified^2)

• PL theory is bringing this to practice ... but, still a LOT of effort needed to verify

But, of course, attacks will always remain

- Side channels not accounted for in the spec
- Physical (in)security
- Hardware attacks
  ...

Only as secure/correct as your specification
More about F*

• Verifying crypto protocols

• Verifying JavaScript programs

• DKAL, a domain specific message passing language embedded in F*

• Building Windows Azure cloud applications

http://research.microsoft.com/fstar
• Try out the F* tutorial

http://rise4fun.com/FStar/tutorial/guide
Read more about F* in many papers

http://research.microsoft.com/fstar

To date: Verified ~40,000 lines of code using F*

F* (ICFP ’11)
Self-certification (POPL ’12)

Fine and DCIL: (ESOP ‘10, PLDI ‘10)

Ibex/RePriv: Dependently typed web browser extensions (2 x Oakland ‘11)

F7: Refinement types for crypto protocols (CSF’08, POPL ‘10)

FX: Modular verification of stateful code (PLPV ‘11)