

A Certified Code Generator for Security Protocols

Master Thesis Introduction Talk

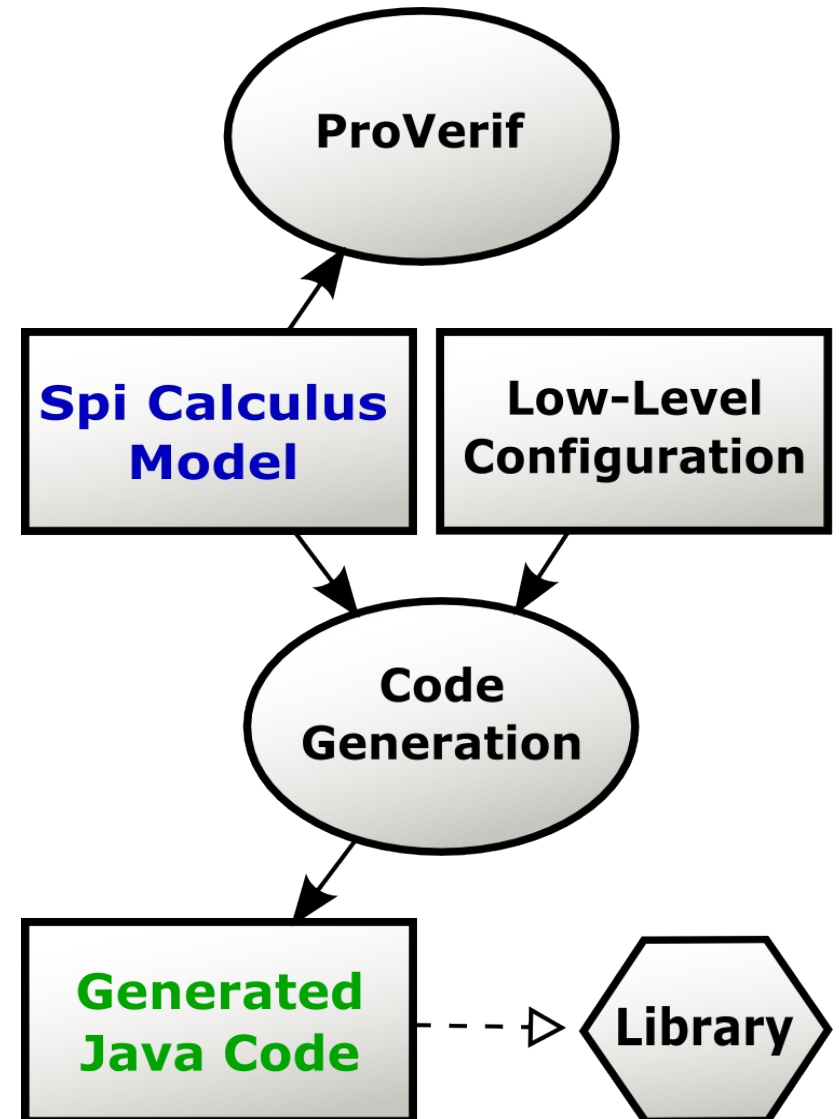
Alex Busenius

Supervisor: Prof. Dr. Michael Backes

Adviser: Cătălin Hrițcu, M.Sc.

Expi2Java

- A code generator for security protocols
- Highly customizable and extensible
 - ▶ User-defined types, cryptographic primitives and low-level configuration
 - ▶ User-provided implementation classes
 - ▶ Code generation using code templates



Expi2Java

- Input language: A variant of Spi calculus
- Simple type system (with type inference)
 - ▶ Prevents incorrect use of cryptographic primitives and configurations
- Produces interoperable Java code
 - ▶ Common cryptographic primitives and data types supported out of the box
 - ▶ Full control over the low-level data format
- Mature project
 - ▶ 6 major releases since December 2008

Case Study: TLS Protocol

- Fully-functional TLS v1.0 client and server
 - ▶ Started in Bachelor's thesis (client only)
 - ▶ Secure channel and HTTPS server
- Verified the TLS handshake with ProVerif
 - ▶ About 2x faster than fs2pv, 4x smaller model

```
(** Protocol start: Receive ClientHello **)
let server =
  new s_time      : $Timestamp;
  new session_id : $TLSSession;
  new server_rnd  : $TLSSRandom;
  let server_nonce = (s_time, server_rnd) in
    (* read certificate and secret key *)
    in(chFileSKey, server_rsa_key);
    in(chFileCert, certificate);
    event BeginServer();
  ...
```

```
config TLS_AES extends AES(
  keylength   = "256",
  provider    = "SunJCE",
  type SymEnc(
    class      = "TLSSymEnc",
    mode       = "CBC",
    padding    = "TLSSv1Padding"),
  type SymKey(class = "SymKey"),
  type Int(
    class      = "BigNonce",
    size       = "16" (* IV *))).
```

Ensuring Correctness

- Implementation size (Java code):
 - ▶ ~8000 SLOC (type checker, translation, etc.)
 - ▶ ~4500 SLOC (concrete crypto library)
- We don't want to verify concrete crypto
 - ▶ Solution: Symbolic crypto library
 - Only ~2200 SLOC (90% automatically generated)
- The translation alone is still ~1000 lines of Java
 - ▶ Idea: Simplify as much as possible
- Still too large for proving by hand
 - ▶ Use a proof assistant, e.g. Coq

First step: Formalization

Formalization: Spi calculus

- Done in the last months
- Locally nameless representation of binders
 - ▶ Using Ott and LNgen
- Work in progress: Proving type preservation

```
proc, P, Q, R ::= 'proc_' ::=
| out( y , t ); P      ::      :: out
| in( y , x ); P      ::      :: in          (+ bind x in P +)
| !in( y , x ); P     ::      :: bangin      (+ bind x in P +)
| let x = g in P else Q ::      :: let          (+ bind x in P +)
| new a : T ; P       ::      :: new          (+ bind a in P +)
| P '|' Q             ::      :: fork
| 0                   ::      :: null
| ( P )               :: S    :: parenproc  {{ coq ([[P]]) }}
| P [ x := t ]       :: M    :: substproc
                        {{ coq (subst_t [[t]] [[x]] [[P]]) }}
```

Formalization: Java

- Full-fledged Java is too complex
 - ▶ Need to use as few features as possible
- Currently used features:
 - ▶ Some base types (boolean, String (constants only))
 - ▶ User-defined classes
 - Fields, methods (also static), constructors
 - Object creation (“new” operator)
 - Field update
 - ▶ Concurrency (thread creation, join, shared memory)
 - ▶ Exceptions (throw, try..catch..finally)
 - ▶ Generics (classes & methods)

Formalization: Java

- Existing formalized fragments:
 - ▶ Featherweight Java [Igarashi et al, by hand, 2001]
 - Generics: FGJ
 - ▶ Lightweight Java [Ott guys, Ott + Isabelle/HOL, 2008]
 - ▶ Jinja [Nipkow, Isabelle/HOL, 2006]
 - ▶ Jinja with threads [Lochbihler, Isabelle/HOL, 2008]
 - ▶ Bicolano MT [MOBIUS Project, Coq, 2008]
- Still, none of them has all the features we need
 - ▶ Combine them, adapt for use with Coq
 - ▶ Further simplify translation

Formalization: Translation

- Proving translation directly seems hard
- Known challenges:
 - ▶ Restrictions
 - $\text{new } a; P$
 - ▶ Structural equivalence
 - $(P \mid Q) \mid R \equiv P \mid (Q \mid R)$
 - $P \equiv P \mid 0$
 - ▶ Scope extrusion/intrusion
 - $a \notin \text{fn } Q \Rightarrow (\text{new } a; P) \mid Q \equiv \text{new } a; (P \mid Q)$
- Break the translation into several steps

Formalization: Translation

- Step 1: Replace restrictions with unique name generation
 - ▶ $\text{step}_1(\text{new } a; P) = \text{gen } x \text{ in } \text{step}_1(P\{x/a\})$
 - ▶ This makes the calculus closer to implementation
 - ▶ Fully-abstract translation
 - [Old Names for Nu, Lucian Wischik, 2004]
 - $P \approx_{\text{spi}} Q \Leftrightarrow \text{step}_1(P) \approx_{\text{spi}'} \text{step}_1(Q)$
- Not clear how many more steps we'll need

What does “correct” mean?

Defining “Correctness”

- Correct = “same behavior”
- Preservation of observational equivalence?
 - ▶ Source and target languages are very different
 - ▶ Our transformation is probably not fully abstract
 - Java contexts can distinguish more (differently)
- More interesting: Preservation of (robust) safety

Translation Preserves Safety

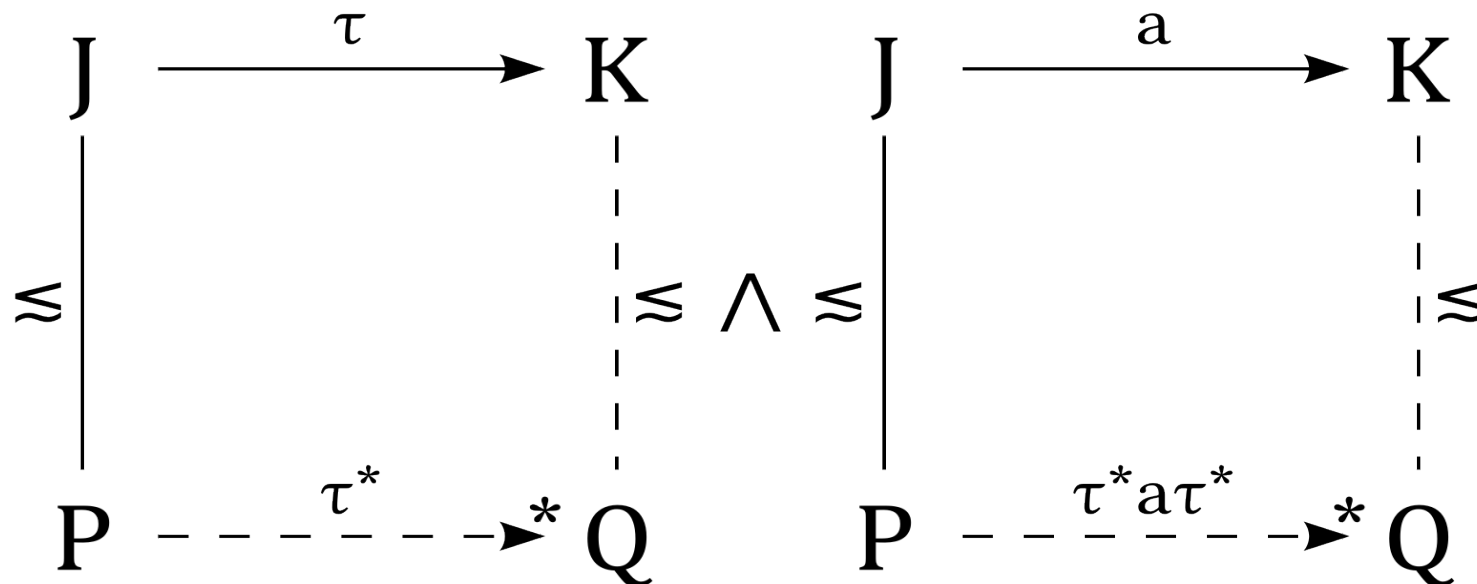
- Trace: Sequence of events ($t \in \text{Event}^*$)
- Trace property: Set of traces ($X \subseteq \text{Event}^*$)
- Safety property (prefix): $t.u \in X \Rightarrow t \in X$
 - ▶ Example: correspondence assertions (e.g. in ProVerif)
 - query $\text{ev:send}(A,B,M) \Rightarrow \text{ev:authenticate}(A,B,M)$
 - $X := \{ t \mid t = t_1.\text{send}(A,B,M).t_2 \Rightarrow \text{authenticate}(A,B,M) \in t_1 \}$
 - ▶ $\text{Traces}_{\text{spi}}(P) = \{ t \mid P \xrightarrow{t} P' \}$ and $\text{Traces}_{\text{java}}(J) = ?$
 - ▶ Model-checkers prove safety properties of processes
 - $\text{Traces}_{\text{spi}}(P) \subseteq X$
- We want $\text{Traces}_{\text{java}}(\text{expi2java}(P)) \subseteq \text{Traces}_{\text{spi}}(P)$

Translation Preserves Robust Safety

- Safety for all attackers
 - ▶ Attacker := a context C that doesn't raise events
- $\text{RobustTraces}_{\text{spi}}(P) = \bigcup_{C \in \text{SpiAttacker}} \text{Traces}_{\text{spi}}(C[P])$
- ProVerif can prove robust safety
 - ▶ $\text{RobustTraces}_{\text{spi}}(P) \subseteq X$
- We want
 - ▶ $\text{RobustTraces}_{\text{java}}(\text{expi2java}(P)) \subseteq \text{RobustTraces}_{\text{spi}}(P)$
 - ▶ This would require defining a decompiler (java2expi)

Proof Idea: Simulation

- Labeled-transition system
 - ▶ More events that capture attacker actions
 - $\text{in}(c,v)$, $\text{out}(c,v)$, τ
- Weak labeled (bi)simulation
 - ▶ Relating Java programs to processes



Proof Idea: Simulation

- We want to show:
 - ▶ $\text{expi2java}(P) \lesssim P$
- Simulation usually implies trace inclusion
 - ▶ We would get that translation preserves safety
 - $\text{Traces}_{\text{java}}(\text{expi2java}(P)) \subseteq \text{Traces}_{\text{spi}}(P)$
- If we additionally show:
 - ▶ $J \lesssim P \Rightarrow \forall C. \text{expi2java}(C)[J] \lesssim C[P]$
 - ▶ We would get preservation of robust safety with respect to spi-calculus contexts
 - $\bigcup_{C \in \text{SpiAttacker}} \text{Traces}_{\text{java}}(\text{expi2java}(C[P])) \subseteq \text{RobustTraces}_{\text{spi}}(P)$

Thesis Goals

- Formalize (in Coq):
 - ✓ Spi calculus
 - ▶ Java subset
 - ▶ Translation (in several steps)
- Prove that each step preserves trace properties
 - ▶ Define simulation relation
 - ▶ Prove that simulation implies
 - Trace inclusion: Safety
 - Trace inclusion for all Spi contexts: Robust safety
 - ▶ Optional: Prove the other direction of the trace inclusions using bisimulation