Computer-aided cryptography

Gilles Barthe
IMDEA Software Institute, Madrid, Spain

December 1, 2015
Introduction

Two models of cryptography:

- Computational: strong guarantees but complex proofs
- Symbolic: automated proofs but weak guarantees

Computational soundness:

- Symbolic security entails computational security
- Great success, but some limitations

Issues with cryptographic proofs

- In our opinion, many proofs in cryptography have become essentially unverifiable. Our field may be approaching a crisis of rigor. Bellare and Rogaway, 2004-2006
- Do we have a problem with cryptographic proofs? Yes, we do [...] We generate more proofs than we carefully verify (and as a consequence some of our published proofs are incorrect). Halevi, 2005
Motivation

- Programs
  - Code-based approach
- Specifications
  - Security definition
- Verification
  - Security proofs
- Challenges
  - Randomized programs + Non-standard properties
- Appeal
  - Small programs + Complex and multi-faceted proofs
Our work

Goal: machine-checked proofs in computational model
  - All proof steps should be justified
  - Proof building may be harder; proof checking is automatic

Main directions:
  - (2006-) Reduction proofs in the computational model
  - (2012-) Verified implementations
  - (2012-) Automated analysis and synthesis

Focus on primitives, some work on protocols and assumptions

http://www.easycrypt.info
Formal verification

Goals: improve program/system reliability using computer tools and formalized mathematics

Some recent success stories

- Verified C compiler and verified L4 microkernel
- Kepler’s conjecture and Feit-Thomson theorem

Many methods and tools. Even for program reliability, many dimensions of choice:

- property (safety vs. correctness)
- find bugs vs. build proof
- automation vs. precision
- etc.
Deductive verification

- program $c$ is annotated with “sufficient” annotations, including pre-condition $\Psi$ and post-condition $\Phi$
- judgment $\{\Psi\}c\{\Phi\}$ is valid iff value output by program $c$ satisfies $\Phi$, provided input satisfies $\Psi$
- logical formula (a.k.a. proof obligation) $\Theta$ extracted from annotated program and spec $\{\Psi\}c\{\Phi\}$
- validity of $\Theta$ proved automatically or interactively
Example: RSA signature

- Sign($m$) and Verif($m, x$) are programs:

  \[
  \begin{align*}
  \text{Sign}(m) : & \quad \text{Verif}(m, x) \\
  z & \leftarrow m^d \mod n \quad w \leftarrow x^e \mod n \\
  \text{return } z & \quad \text{return } y
  \end{align*}
  \]

- specification:

  \[\{ x = \text{Sign}(m) \} \text{Verif}\{ y = \text{true} \}\]

- proof obligation:

  \[x = m^d \mod n \Rightarrow m = x^e \mod n\]

- context: $p$ and $q$ are prime, $n = pq$, etc...

- discharging proof obligation uses some mathematics (Fermat’s little theorem and Chinese remainder theorem)
Program verification for cryptography

Two main challenges:
  ▶ Programs are probabilistic
  ▶ Properties are reductions: reason about two systems

Existing techniques:
  ▶ Verification of probabilistic programs
  ▶ Relational program verification
Deductive verification of probabilistic programs

- With probability $\geq p$, output of program $c$ satisfies $\Psi$
- Since the 70s
- Mostly theoretical
- Lack of automation and tool support
- Foundational challenges: probabilistic independence, expectation, concentration bounds...
- Practical challenges: reals, summations
Relational verification of programs

- Programs are equivalent
  \[ \{m\langle 1 \rangle = m\langle 2 \rangle \} \text{Sign} \sim \text{SignCRT}\{z\langle 1 \rangle = z\langle 2 \rangle \} \]

- Recent: \( \sim 10 \) years
- Dedicated tools, or via mapping to deductive verification
- Large examples
- Focus on deterministic programs
Key insight

Relational verification of probabilistic programs
- avoids issues with verification of probabilistic programs
- nicely builds on probabilistic couplings

Couplings: the idea
- Put two probabilistic systems in the same space.
- Coordinate samplings

Formal definition
- Let $\mu_1$ and $\mu_2$ be sub-distributions over $A$
- A sub-distribution $\mu$ over $A \times A$ is a coupling for $(\mu_1, \mu_2)$ iff $\pi_1(\mu) = \mu_1$ and $\pi_2(\mu) = \mu_2$
- Extends to interactive systems and distinct prob spaces
- Perfect simulation: existence of simulator + coupling
Lifting

Formal definition

- Let \( R \) be a binary relation on \( A \) and \( B \), i.e. \( R \subseteq A \times A \)
- Let \( \mu_1 \) and \( \mu_2 \) be sub-distributions over \( A \)
- \( \mu_1 R\# \mu_2 \) iff there exists a coupling \( \mu \) s.t. \( \Pr_{y\leftarrow\mu}[y \notin R] = 0 \)

Applications

- Bridging step: \( \mu_1 = \# \mu_2 \), then for every event \( X \),
  \[
  \Pr_{z\leftarrow\mu_1}[X] = \Pr_{z\leftarrow\mu_2}[X]
  \]

- Failure Event: If \( x R y \) iff \( F(x) \Rightarrow x = y \) and \( F(x) \Leftrightarrow F(y) \), then for every event \( X \),
  \[
  |\Pr_{z\leftarrow\mu_1}[X] - \Pr_{z\leftarrow\mu_2}[X]| \leq \max(\Pr_{z\leftarrow\mu_1}[\neg F], \Pr_{z\leftarrow\mu_2}[\neg F])
  \]

- Reduction: If \( x R y \) iff \( F(x) \Rightarrow G(y) \), then
  \[
  \Pr_{x\leftarrow\mu_2}[G] \leq \Pr_{y\leftarrow\mu_1}[F]
  \]
Code-based approach to probabilistic liftings

▶ Programs:

\[
\begin{align*}
C & ::= \text{skip} & \text{skip} \\
    & | \ V \leftarrow \mathcal{E} & \text{assignment} \\
    & | \ V \leftarrow \$ \mathcal{D} & \text{random sampling} \\
    & | C; C & \text{sequence} \\
    & | \text{if } \mathcal{E} \text{ then } C \text{ else } C & \text{conditional} \\
    & | \text{while } \mathcal{E} \text{ do } C & \text{while loop} \\
    & | \ V \leftarrow \mathcal{P}(\mathcal{E}, \ldots, \mathcal{E}) & \text{procedure (oracle/adv) call}
\end{align*}
\]

▶ Logic: \( \models \{ P \} \ c_1 \sim c_2 \ \{ Q \} \) iff for all memories \( m_1 \) and \( m_2 \), \( P(m_1, m_2) \) implies \( Q^\#(\llbracket c_1 \rrbracket m_1, \llbracket c_2 \rrbracket m_2) \)

▶ \( P \) and \( Q \) are relations on states (no probabilities)

\[\Rightarrow\text{ very similar to standard deductive verification}\]
EasyCrypt

- probabilistic Relational Hoare Logic
- libraries of common proof techniques (hybrid arguments, eager sampling, independent from adversary’s view, forking lemma...)
- probabilistic Hoare Logic for bounding probabilities
- full-fledged proof assistant, and backend to SMT solvers
- module system and theory mechanism

Case studies

- encryption, signatures, hash designs, key exchange protocols, zero knowledge protocols, garbled circuits...
- (computational) differential privacy
- mechanism design
What now?

Status
- Solid foundations
- Variety of emblematic examples
- Some theoretical challenges: automated complexity analysis, precise computation of probabilities, couplings (shift, modulo distance)

Perspectives
- Standards and deployed systems
- Implementations
- Automation
Provable security vs practical cryptography

- Proofs reason about algorithmic descriptions
- Standards constrain implementations
- Attackers target executable code and exploit side-channels

Existing solutions bring limited guarantees
- Leakage-resilient cryptography (mostly theoretical)
- Real-world cryptography (still in the comp. model)
- Constant-time implementations (pragmatic)

**Approach**

- Machine-checked reductionist proofs for executable code
- Separation of concerns:
  1. prove algorithm in computational model
  2. verify implementation in machine-level model
Outline of approach

Reductionist proof:

- FOR ALL adversary that breaks assembly code,
- IF assembly code does not leak,
- AND assembly code and C code semantically equivalent,
- THERE EXISTS an adversary that breaks the C code

Components:

- proofs in EasyCrypt,
- equivalence checking of EasyCrypt vs C,
- verified compilation using CompCert,
- leakage analysis of assembly
Security models: the case of constant-time

Language-level security
- sequence of program counters and memory accesses. Defined from instrumented semantics.
- security definitions use leaky oracles

System-level security
- active adversary controls scheduler and (partially) cache
- security games include adversarially-controlled oracles
- prove language-level security implies system-level security

Warning
Models are constructed!
Verification of constant-time

Two possible approaches:
▶ Static program analysis
▶ Program transformation and deductive verification

Comparison:
▶ Analysis is fast but conservative
▶ Transformation is fast and precise

Implementation
▶ Relatively easy for analysis
▶ Requires existing infrastructure for transformation

Instances:
▶ Standalone analysis for x86
▶ Transformation + Smack for LLVM
Constant-time verification by product programs

Judgment:
\[ c \leadsto c^\times \]

Example rules

\[
\begin{align*}
    x &\leftarrow e \leadsto x \leftarrow e; x' \leftarrow e' \\
    c_1 &\leadsto c_1^\times \quad c_2 \leadsto c_2^\times \\
    \text{if } b \text{ then } c_1 \text{ else } c_2 &\leadsto \text{assert } b = b'; \text{if } b \text{ then } c_1^\times \text{ else } c_2^\times
\end{align*}
\]

Correctness and precision

\( c \) is constant-time iff \( c^\times \) does not assert-fail, where \( c \rightarrow c^\times \)

Applications: NaCl, PKCS, MEE-CBC…
Provably secure implementations: challenges

- Refined models of execution platforms and compilers
- Formal models of leakage
  (how to model acoustic emanations?)
- Better implementation-level adversary models and connections with real-world cryptography
- Manage complexity of proofs
Automated analysis and synthesis

Goals:
▶ Capture the essence of cryptographic proofs
▶ Minimize time and expertise for verification
▶ Explore design space of schemes

Approach:
▶ Isolate high-level proof principles
▶ Automate proofs
▶ Synthesize and analyze candidate schemes

Warning: trade-off (some) generality for automation
Automated analysis

Ingredients

▶ Develop automated procedures for algebraic reasoning
▶ Core proof system (specialized proof principles)
▶ Adapt symbolic methods for reasoning about computational notions (reduction and entropy)
▶ Develop efficient heuristics
Synthesis

The next 700 cryptosystems

Do the cryptosystems reflect [...] the situations that are being catered for? Or are they accidents of history and personal background that may be obscuring fruitful developments? [...] We must systematize their design so that a new cryptosystem is a point chosen from a well-mapped space, rather than a laboriously devised construction. (Adapted from Landin, 1966. The next 700 programming languages)

Synthesis has many potential applications to cryptography

- Discover new and interesting constructions
- Prove optimality results
- Optimize existing constructions
- Find countermeasures

Methodology:

Smart generation + Attack finding + Automated proofs
Applications

- Assumptions in multilinear generic group model
- Pairing-based constructions in standard model
- Padding-based encryption
  - Analyzed over 1,000,000 schemes
  - Discovered ZAEP
- Structure-preserving signatures
  - Optimality result to minimize search space
  - Analyzed 1,000s of schemes
  - Discovered optimal scheme w.r.t. online/offline pairings

Tweakable blockciphers (Hoang, Katz, Malozemoff)

- Analyzed 1,000s of schemes
- Discovered several schemes competitive with OCB
Summary

Foundations and tools for high-assurance crypto
  ▶ Provable security
  ▶ Practical cryptography
  ▶ Reducing the gap between the two

Automated proofs and synthesis
  ▶ “Essence” of cryptographic proofs and “global” view
  ▶ New and interesting schemes

Perspectives
  ▶ Verified standards and cryptographic systems
  ▶ Improve usability of tools
  ▶ Teaching reductionist proofs