From Crypto to Code

Greg Morrisett

Cornell CIS
COMPUTING AND INFORMATION SCIENCE

Languages over a career

- Pascal/Ada/C/SML/Ocaml/Haskell
- ACL2/Coq/Agda
- Latex
- Powerpoint
- Someone else's Powerpoint

Cryptographic techniques

- Already ubiquitous: e.g., SSL/TLS
- Offer great hope: e.g., homomorphic encryption
- Perhaps most importantly:
  - Offer a rigorous way to think about, model, and verify protocols for important security properties.
  - A true “science” basis for security?

Yet…

“In theory there is no difference between theory and practice; in practice there is.”

- Jan L.A. van de Snepscheut

For instance

The Heartbleed Bug

The Heartbleed Bug is a serious vulnerability in the popular OpenSSL cryptographic software library. This weakness allows stealing the information protected, under normal conditions, by the SSL/TLS encryption used to secure the Internet.

"Catastrophic" is the right word. On the scale of 1 to 10, this is an 11.

- Bruce Schneier

Auditing

Open source libraries, such as OpenSSL, power the internet.

But frankly, we cannot rely upon the open source community to do an adequate job of auditing security-critical code.

In addition to coding errors:

- Flawed schemes
  - Needham-Schroeder-(Lowe)
    - Assumed secure for 17 years before broken.
  - Dual-EC-DRBG
    - Flaw suspected for 6 years, ignored due to culture of trust.
  - Advanced Privacy Protection (APP) scheme
    - Proved secure, proof independently verified, still flawed.
- The situation is getting worse
  - Bellare & Rogaway (2006): *"Our field may be approaching a crisis of rigor."*
  - Halevi (2005): *"…we generate more proofs than we carefully verify."*

One Way Forward

- Mechanized proofs of security for cryptographic schemes.
  - Instead of humans, a machine takes care of auditing proofs.
  - Forces community to standardize definitions.
- Coupled with formal verification of code.
  - Connect the actual (source) code to the cryptographic algorithms.
- And fully abstract, verified compilers.
  - Verified compilation ensures machine code preserves behaviors.
  - Full abstraction ensures possible attacks at the target level are reflected in the source.

Mechanizing Cryptography

- Goal: no need to trust the proof of security
  - Still need to inspect definitions and assumptions.
  - (Even this is fraught with peril.)
- Two basic models:
  - Symbolic: functions/values are opaque, adversary capability is algebraic, proof by underlying logic.
  - Computational: considers probabilities, adversary is computationally bounded, proof by reduction.

Some very influential computational work:

- 2008: CertiCrypt (Barthe et al.)
  - First fully-general proof framework for crypto
  - Library in Coq; deep embedding
- 2011: EasyCrypt (Barthe et al.)
  - As expressive as CertiCrypt, but much easier to use
  - Standalone system using Why3 as backend

EasyCrypt case study (circa 2012)

- External team attempted to use EasyCrypt
  - MIT Lincoln Lab, US Naval R. Lab, U. of Maryland
  - Mix of PL and crypto experts
  - Goal: security of a private information retrieval protocol
- Outcome: partial success
  - Most lemmas could be proved.
  - Found minor flaws in scheme.
  - But unable to prove certain results.

The Foundational Cryptography Framework

1. A formal language, embedded in Coq, for specifying cryptographic protocols, games, and other specifications.

2. The language comes equipped with both a simple operational model, as well as a denotational one.

3. From the model, we *derive* a program logic that allows one to formally prove (probabilistic) correctness and security.

4. Set of libraries for common cryptographic constructions and a set of tactics that help automate some of the proofs.

Adam Petcher
(*POST 2015)*

Probabilistic Programs

We re-use Coq's functional language, Gallina and add a (discrete) probability monad:

$$[\![\texttt{ret}\ a]\!] = \texttt{promote a Gallina term}$$

$$[\![x \xleftarrow{\$} c; f\ x]\!] = \texttt{sequence}$$

$$[\![\{0,1\}^n]\!] = \texttt{uniformly random bit vector}$$

$$[\![\texttt{Repeat}\ c\ P]\!] = \texttt{repeated experiment}$$

So for example:

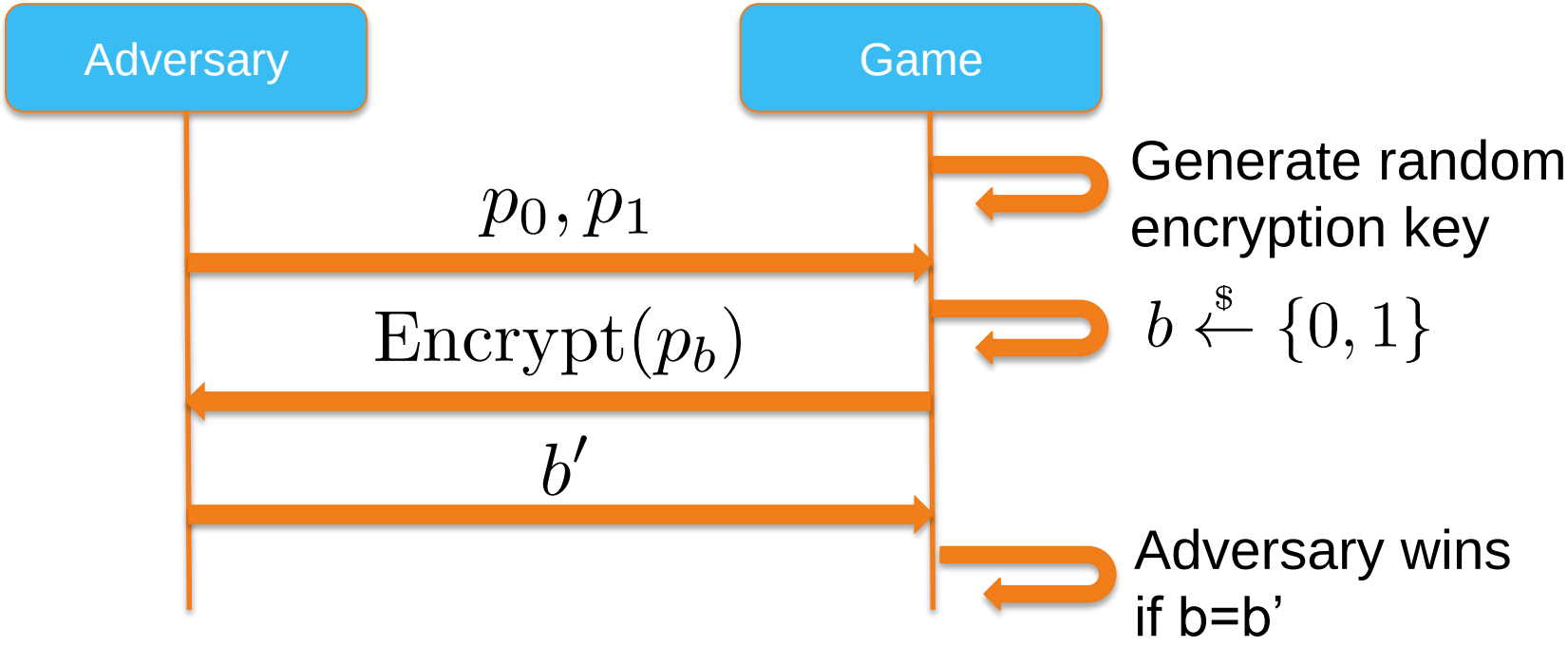
A one-time-pad encryption for a message of *n* bits.

```
Definition OTP(n:nat)(msg:Bvector n) :=
   p <-$ {0, 1}^n ;
   ret (p xor msg).
```

More Generally

- Security definitions given as “games”
- Adversary should not “win” the game
- Alternatively: two games that adversary cannot distinguish
- Advantage: the probability that the adversary wins the game
- Also used to describe (assumed) hard problems

Example: Encryption (IND-CPA)



Not shown: adversary can request ciphertext for any plaintext

Reasoning via Game Hopping

We often need to show that a given program has the same distribution as another program.

Or more generally, that the probability of some certain bad events is bounded when moving from one program to another.

Denotational Semantics: Probability Mass Fn.

$$[\![\texttt{ret}\ a]\!] = \mathbf{1}_{\{a\}}$$

$$[\![x \xleftarrow{\$} c; f\ x]\!] = \lambda x. \sum_{b \in \mathrm{supp}([\![c]\!])} ([\![f\ b]\!]\ x) * ([\![c]\!]\ b)$$

$$[\![\{0,1\}^n]\!] = \lambda x.\ 2^{-n}$$

$$[\![\texttt{Repeat}\ c\ P]\!] = \lambda x.(\mathbf{1}_P\ x) * ([\![c]\!]\ x) * \left(\sum_{b \in P} ([\![c]\!]\ b)\right)^{-1}$$

Lots of equational properties

Theorem (Associativity)**.**

$$[\![x \leftarrow (y \leftarrow c1; c2); c3]\!] = [\![y \leftarrow c1; x \leftarrow c2; c3]\!]$$

Theorem (Commutativity)**.** If x does not occur in c2 and y does not occur in c1, then

$$[\![x \leftarrow c1; y \leftarrow c2; c3]\!] = [\![y \leftarrow c2; x \leftarrow c1; c3]\!]$$

Theorem (Distribution Isomorphism)**.** For any f which is a bijection from $Sup([\![a2]\!])$ to $Sup([\![a1]\!])$,

$$\begin{aligned} &\forall x \in Sup([\![a2]\!]), [\![a1]\!](f\ x) = [\![a2]\!]x \\ \wedge\ &\forall x \in Sup([\![a2]\!]), [\![b1[(f\ x)/y]]\!]\ v1 = [\![b2[x/y]]\!]v2 \\ \Rightarrow\ &[\![y \leftarrow a1; b1]\!]\ v1 = [\![y \leftarrow a2; b2]\!]\ v2 \end{aligned}$$

Program Logic

Probabilistic relational post-condition logic (PRPL)

$$p \sim q\{\Phi\} \Leftrightarrow \exists\, d,$$
$$\forall (x, y) \in \mathrm{supp}([\![d]\!]), \Phi\ x\ y \wedge$$
$$[\![p]\!] = [\![z \xleftarrow{\$} d; \mathtt{ret}\ (\mathrm{fst}\ z)]\!] \wedge$$
$$[\![q]\!] = [\![z \xleftarrow{\$} d; \mathtt{ret}\ (\mathrm{snd}\ z)]\!]$$

Some Properties of the Logic

$$p \sim q\{=\} \Leftrightarrow [\![p]\!] = [\![q]\!]$$

$$p \sim q\{\lambda a.\, \lambda b.\, a = x \Leftrightarrow b = y\} \Leftrightarrow [\![p]\!]x = [\![q]\!]y$$

$$p \sim q\{\lambda a.\, \lambda b.\, a = x \Rightarrow b = y\} \Leftrightarrow [\![p]\!]x \leq [\![q]\!]y$$

$$n \sim o\{\Psi\} \Rightarrow \{\Psi\}p \sim q\{\Phi\} \Rightarrow$$

$$x \xleftarrow{\$} n; (p\ x) \sim y \xleftarrow{\$} o; (q\ y)\{\Phi\}$$

What's It Like to Use the Logic?

- Lots of standard constructions for encryption, authentication, etc.
  - See Petcher's 2015 Harvard PhD thesis
- Case Study: Searchable Symmetric Encryption
  - Based on work of Cash et al. (2013)
  - Petcher & Morrisett, Computer Security Foundations 2015
- Case Study: Security of OpenSSL HMAC code
  - Beringer et al., USENIX Security 2015

Searchable Symmetric Encryption

Two parties: client and server

Database: list of keyword, value pairs

Client

- Knows database and list of queries
- Creates encrypted database and queries to give to server

Server

- Executes queries and gives encrypted results to client
- Learns very little about database and queries

Key Component: Tuple Sets

Three procedures

- TSetSetup : Database -> (TSet * SecretKey)
- TSetGetTag : SecretKey -> Keyword -> Tag
- TSetRetrieve: TSet -> Tag -> list Value

Security: Adversary cannot distinguish T-Set and tags from those produced by simulator

Correctness: Adversary cannot cause incorrect answers

SSE from Tuple Sets

T-Set is almost an SSE Scheme for single-keyword search

- Reveals results of query

Solution: Store ciphertexts in T-Set

- Encryption key is derived from keyword via PRF
- Proof requires secure **and correct** T-Set

Relatively simple proof

- ~ 1100 lines of Coq code
- 8 intermediate games

The Hard Part: Tuple-Sets

Cash et al. provide a T-Set scheme

Based on a fixed-size 2D table

- Row is determined by hash
- Location in row chosen at random

Complications

- **A row can become full** (setup restarts)
- Sampling without replacement
- Nested loops, loop manipulations

Hybrid Arguments

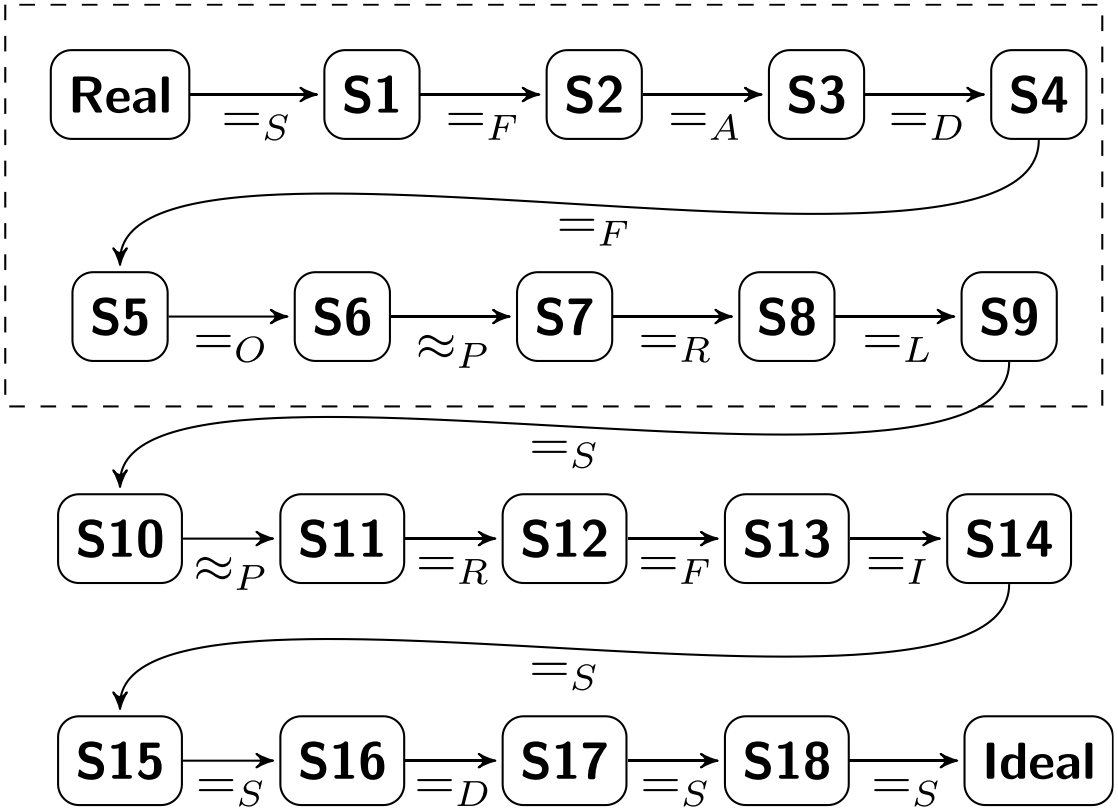
Security/correctness given one implies security/correctness given many

Encryption/PRF with many keys/oracles

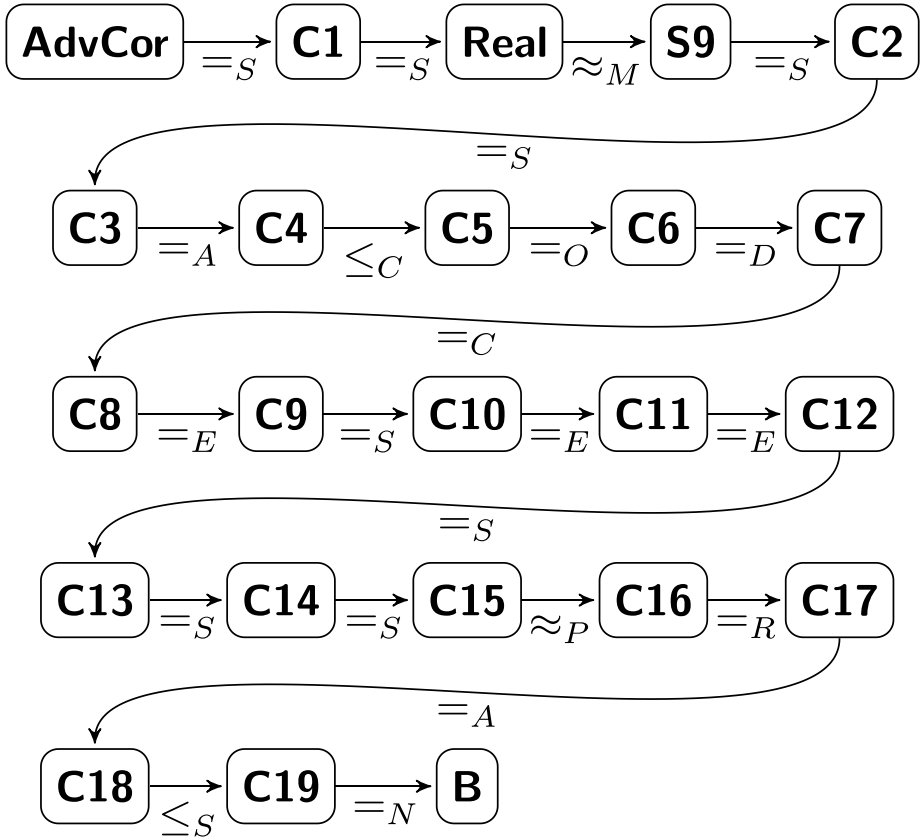
Simplify T-Set proofs

- Consider simplified “single-trial” T-Set scheme
- Conclude facts about full T-Set scheme

Single-Trial T-Set Security Proof



Single-Trial T-Set Correctness Proof



SSE Proof Size

Among largest mechanized crypto proofs to date

- 58 intermediate games in 9 reductions
- Over 14,000 lines of Coq code
- 1,300 lines of definition and intermediate games
- Unlike in traditional crypto, these intermediate games (S1-S18, C1-C19) do *not* have to be inspected.

Compilation

- It's possible to automatically extract an OCaml implementation from the Coq definitions.
  - But we have to trust extraction & OCaml compiler
- CertiCoq: a verified compiler for Coq
  - Joint project between Princeton, Cornell, Inria
  - Very much a work in progress

Reasoning about existing code

FCF can be combined with other Coq libraries.

Verified Software Toolchain (VST) by Appel:

- Allows to prove correctness of C code.
- Leroy's CompCert compiler produces assembly-level refinement.

Secure HMAC Code

Correct implementation of HMAC in C

- Developed by Appel and Beringer
- Equivalent to functional specification

HMAC is a PRF

- Developed by Petcher
- Assuming hash function has certain properties

Functional spec equivalent to crypto model

- Developed by Ye

What next?

- For FCF, complexity arguments are done by hand.
- Need much more proof automation.
- The work on HMAC did not consider side channels.
    - But see recent F* work out of INRIA on ECs.
- There's a serious issue around getting cryptographers to read and understand the definitions to show we are proving the right things.

Much Related Work

- CertiCrypt, EasyCrypt (Barthe et al.)
- CryptoVerif (Blanchet)
- F* (Fournet et al.)
- Nowak
- Verypto (Berg)
- Crypto-agda
- Probabilistic Protocol Composition Logic (Datta et al.)
- Backes
- …

To Summarize

Mechanizing crypto proofs is a way to support open source development without needing the same (misplaced) trust that we have today.

The tools are rapidly coming together to reason about computational security of real code executing on real systems.

Thanks!