Computer-aided cryptographic proofs

Gilles Barthe
MPI-SP, Germany
IMDEA Software Institute, Spain
Computer-aided cryptography

Develop tool-assisted methodologies for helping the design, analysis, and implementation of cryptographic constructions (primitives and protocols)

Building on formal methods
  ▶ program analysis and verification/program synthesis
  ▶ compilation (certifying compilation/verified compilation)
  ▶ logic
  ▶ etc
Potential benefits

Formal methods for cryptography

- higher assurance
- smaller gap between provable security and crypto engineering
- new proof techniques

Cryptography for formal methods

- Challenging and non-standard examples
- New theories and applications
Provable security

For every adversary $A$ executing in time $t_A$ there exists an adversary executing in time $t_B$ such that

$$Pr[A \text{ breaks } C] \leq Pr[B \text{ breaks } P] + \epsilon$$

and

$$t_A \leq t_B + M$$
EasyCrypt\(^1\)

Domain-specific proof assistant
- proof goals tailored to reductionist proofs
- proof tools support common proof techniques (bridging steps, failure events, hybrid arguments, eager sampling...)

Control and automation from state-of-art verification
- interactive proof engine and mathematical libraries (a la Coq/ssreflect)
- back-end to SMT solvers

Many case studies:
- Encryption, signatures, key exchange, zero-knowledge, multi-party and verifiable computation, SHA3, voting, KMS

\(^1\)Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, Santiago Zanella Béguelin: Computer-Aided Security Proofs for the Working Cryptographer. CRYPTO 2011
Probabilistic Relational Hoare logic\(^2\)

- **Code-based approach**

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

- **Game-playing technique:** \(\models \{\phi\} c_1 \sim c_2 \{\psi\}\) where \(\phi\) and \(\psi\) are relations on states

- **Validity:**

\[(m_1, m_2) \in \phi \implies \downarrow_{\psi} \langle [c_1]_{m_1} \& [c_2]_{m_2} \rangle\]

\(^2\)Gilles Barthe, Benjamin Grégoire, Santiago Zanella Béguelin: Formal certification of code-based cryptographic proofs. POPL 2009
Probabilistic couplings

Let \( \mu_1, \mu_2 \in \text{Dist}(A) \) and \( R \subseteq A \times A \). Let \( \mu \in \text{Dist}(A \times A) \).

- \( \mu \) is a coupling for \( (\mu_1, \mu_2) \) iff \( \pi_1(\mu) = \mu_1 \) and \( \pi_2(\mu) = \mu_2 \).
- \( \mu \) is a \( R \)-coupling for \( (\mu_1, \mu_2) \) if moreover \( \Pr_{y \leftarrow \mu} [y \notin R] = 0 \).

Let \( \mu \) be a \( R \)-coupling for \( (\mu_1, \mu_2) \).

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

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

- Reduction: If \( x R y \) iff \( F(x) \Rightarrow G(y) \), then
  \[
  \Pr_{x \leftarrow \mu_2}[G] \leq \Pr_{y \leftarrow \mu_1}[F]
  \]
A program logic for probabilistic couplings

Rules for sequence and conditionals:

\[
\begin{align*}
\models \{\psi\} \ c_1 &\sim c_2 \ \{\Theta\} \quad \models \{\Theta\} \ c_1' &\sim c_2' \ \{\phi\} \\
\models \{\psi\} \ c_1; c_1' &\sim c_2; c_2' \ \{\phi\}
\end{align*}
\]

\[
\begin{align*}
\models \{\psi \land b_1\} \ c_1 &\sim c_2 \ \{\phi\} \\
\models \{\psi \land \neg b_1\} \ c_1' &\sim c_2' \ \{\phi\} \\
\models \{\psi\} \ if \ b_1 \ then \ c_1 \ else \ c_1' &\sim \ if \ b_2 \ then \ c_2 \ else \ c_2' \ \{\phi\}
\end{align*}
\]

Rules for random sampling

\[
\begin{align*}
\vec{\varphi}\langle\llbracket \mu_1 \rrbracket \ &\land \llbracket \mu_2 \rrbracket \rangle \quad \varphi \triangleq \forall v_1 : T_1, v_2 : T_2, \varphi \implies \psi[v_1/x_1][v_2/x_2] \\
\models \{\varphi\} \ x_1 &\leftarrow \mu_1 \sim x_2 \leftarrow \mu_2 \ \{\psi\} \\
\models \{\forall v_1 \in \text{supp}(d_1), \psi[v_1/x_1]\} \ x_1 &\leftarrow d_1 \sim \text{skip} \ \{\psi\}
\end{align*}
\]

Rules for adversary (simplified)

\[
\begin{align*}
\models \{e_1 = e_2 \land \varphi\} \ x_1 &\leftarrow \mathcal{A}(e_1) \sim x_2 \leftarrow \mathcal{A}(e_2) \ \{x_1 = x_2 \land \varphi\}
\end{align*}
\]
From algorithms to implementations

- We need cryptographic libraries that we can trust
- Correct implementation frequently remain vulnerable to attacks, including implementation bug attacks and side channel attacks
- Efficiency considerations often force developers to carry out very aggressive optimizations

Can we carry guarantees to implementations?
Question: which guarantees?

- Fast
- Correct (functional correctness)
- Side-channel resistant (constant-time)
- Provably secure
Question: which implementations

**Source**
- Portable
- Convenient software-engineering abstractions
- Readable, maintainable

**Assembly**
- Efficient
- Controlled (instruction selection and scheduling)
- Predictable
A gap between source and assembly languages

- **Assembly** is not programmer/verifier friendly
  - Harder to understand
  - More error prone
  - Harder to prove / analyze

- **Source** is not security/efficiency friendly
  - Trust compiler
  - Certified compilers are less efficient
  - Optimizing compilers can break side-channel resistance
Fast and formally verified assembly code

- Source language: assembly in the head with formal semantics
  → programmer & verification friendly

- Compiler: predictable & formally verified (in Coq)
  → programmer has control and no compiler security bug

- Verification toolchain (based on EasyCrypt):
  - security
  - safety
  - side channel resistance (constant-time)
  - functional correctness

Case studies
ChaCha20, Poly1305, Curve25519, SHA3

---


Jasmin language

Zero-cost abstractions:
- Variable names
- Arrays
- Conditional and loops
- Inline functions

Control:
- *For* loops are unrolled. *While* loops are not unrolled
- Spilling and instruction scheduling is controlled manually
- Instruction-set architectures are available (often in user-friendly form)
Initialization of Chacha20 state

```c
inline fn init(reg u64 key nonce, reg u32 counter) → stack u32[16]
{
    inline int i;
    stack u32[16] st;
    reg u32[8] k;
    reg u32[3] n;

    st[0] = 0x61707865;
    st[1] = 0x3320646e;
    st[2] = 0x79622d32;
    st[3] = 0x6b206574;

    for i=0 to 8 {
        k[i] = (u32)[key + 4*i];
        st[4+i] = k[i];
    }

    st[12] = counter;

    for i=0 to 3 {
        n[i] = (u32)[nonce + 4*i];
        st[13+i] = n[i];
    }

    return st;
}
```
Compiler: efficiency, predictability and verification-preserving

**Preservation of functional correctness (proved in Coq)**

\[ \forall p \, p' \, \text{compile}(p) = \text{ok}(p') \Rightarrow \]
\[ \forall f \in \text{exports}(p) \Rightarrow \]
\[ \forall m \, \text{enough-stack-space}(f, p', m) \Rightarrow \]
\[ \forall v_a \, v_r \, m'. \, p : f, v_a, m \Downarrow v_r, m' \Rightarrow p' : f, v_a, m \Downarrow v_r, m' \]

**Preservation of side-channel resistance (proved on paper)**

\[ \forall p \, p' \cdot \text{compile}(p) = \text{ok}(p') \Rightarrow \]
\[ \forall f \in \text{exports}(p) \Rightarrow \]
\[ \forall m_1 \, m_2 \, \land_{i=1,2} \text{enough-stack-space}(f, p', m_i) \land m_1 \sim m_2 \Rightarrow \]
\[ \forall v_a. \, p : f, v_a, m_1 \leadsto \ell_1 \land p : f, v_a, m_2 \leadsto \ell_2 \Rightarrow \ell_1 = \ell_2 \Rightarrow \]
\[ \forall v_a. \, p' : f, v_a, m_1 \leadsto \ell_1 \land p' : f, v_a, m_2 \leadsto \ell_2 \Rightarrow \ell_1 = \ell_2 \]
Constant-time programming

Software-based countermeasure against cache-based timing attacks:
- control-flow should not depend on secret data
- memory accesses should not depend on secret data

Rationale: crypto implementations without this property are vulnerable

Secure compilation
- Can we reason about constant-time at the source level?
- Do compilers preserve constant-time?
Preservation of constant-time

Counter-examples

Before

```c
int cmove(int x, int y, bool b) {
    return x + (y - x) * b;
}
```

After

```c
int cmove(int x, int y, bool b) {
    if (b) {
        return y;
    } else {
        return x;
    }
}
```

Before

```c
long long llmul(long long x, long long y) {
    return x * y;
}
```

After

```c
long long llmul(long long x, long long y) {
    long a = High(x);
    long c = High(y);
    if (a | c) {
        ...
    } else {
        return Low(x) * Low(y);
    }
}
```

However, many compiler optimizations do preserve “constant-time”:

- (Adjusted) CompCert preserves constant-time (Coq proof)
- Jasmin preserves constant-time (paper proof)

**Constant-time implies system-level security**

Gilles Barthe, Gustavo Betarte, Juan Diego Campo, Carlos Daniel Luna, David Pichardie: System-level Non-interference for Constant-time Cryptography. CCS 2014
Using Hoare Logic: \{P\} c \{Q\}
Interpretation:

\[ P m \Rightarrow m \downarrow^c m' \Rightarrow Q m' \]

Using Relational Hoare Logic: \{P\} \sim c_1 c_2 \{Q\}
Interpretation:

\[ P m_1 m_2 \Rightarrow m_1 \downarrow^{c_1} m'_1 \Rightarrow m_2 \downarrow^{c_2} m'_2 \Rightarrow Q m'_1 m'_2 \]

Example:

\[ c_1 \] is the reference implementation (the specification)
\[ c_2 \] is the optimized implementation

\{\text{args}\langle 1 \rangle = \text{args}\langle 2 \rangle\} \sim c_1 c_2 \{\text{res}\langle 1 \rangle = \text{res}\langle 2 \rangle\}

EasyCrypt already provides Hoare Logic and Relational Hoare Logic
Functional correctness by game hopping

We have built an EasyCrypt model for Jasmin

Jasmin compiler is able to translate programs into the EasyCrypt syntax

We perform functional correctness proofs by game hopping:

\[ c_{\text{ref}} \sim c_1 \sim \ldots \sim c_{\text{opt}} \]
Game hopping: example Chacha20

Chacha20 is a stream cipher that iterate a \textit{body} on all block of the message

Reference

\begin{verbatim}
while (i < len) {
    chacha_body;
    i += 1;
}
\end{verbatim}

Loop tiling

\begin{verbatim}
while (i < len + 4) {
    chacha_body;
    chacha_body;
    chacha_body;
    chacha_body;
    i += 4;
}
chacha_end
\end{verbatim}

Scheduling

\begin{verbatim}
while (i < len + 4) {
    chacha_body4_swapped;
    i += 4;
}
chacha_end
\end{verbatim}

Vectorization

\begin{verbatim}
while (i < len + 4) {
    chacha_body4_vectorized;
    i += 4;
}
chacha_end
\end{verbatim}
Side channel resistance

Verify using Relational Hoare Logic on instrumented code:

\[
\text{if } t \text{ then } c_1 \text{ else } c_2 \quad \text{leaks} \leftarrow t :: \text{leaks}; \text{if } t \text{ then } c_1 \text{ else } c_2
\]

\(c\) is constant time if its model \(\overline{c}\) verifies the relational property:

\[
\models \{=_{\text{public inputs}}\} \overline{c} \sim c \{\text{leaks} \langle 1 \rangle = \text{leaks} \langle 2 \rangle\}
\]

Inspired and equivalent to method implemented in ctverif\(^7\)

---

\(^7\) Jose Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, Michael Emmi: Verifying Constant-Time Implementations. USENIX Security Symposium 2016
Verified implementations

- FOR EVERY adversary that breaks implementation,
- IF implementation is safe and constant-time,
- AND implementation is correct with respect to algorithm,
- THERE EXISTS an adversary that breaks the algorithm

Moreover implementation is fast
Other approaches

- HACL*: Verified functional correctness and constant-time of source code. Broad coverage. Portable, but compiler trade-off
- FiatCrypto: Verified functional correctness of assembly code. Focus on arithmetic routines.
Future work

- More architectures
- Machine-checked proof of preservation of constant time
- Automation of equivalence proofs
- More examples
Summary

Foundations and tools for high-assurance cryptography

- Provable security
- Practical cryptography
- Reducing the gap between security proofs and implementations

Broader vision

- Scalable program verification for mainstream languages
- Efficient verified compilers for mainstream languages
- How to protect cryptography against micro-architectural attacks?