Efficient Provably Secure Machine Code from High-Level Implementations
Towards Certified Computer-Aided Cryptography

François Dupressoir
IMDEA Software Institute, Madrid, Spain

Real-World Cryptography - January 14th 2014 - New York

Based on joint work (CCS’13) with J. C. Bacelar Almeida¹, Manuel Barbosa¹ and Gilles Barthe²
1. HASLab-INESC TEC & Universidade do Minho  2. IMDEA Software Institute
Mind the Gap(s)

- Cryptographers prove abstract schemes secure.
- Concrete schemes are standardized.
- Implementations are run.

**Goal**

We aim to bridge these gaps, and bring formal cryptographic guarantees to the level of executable code:

- Perform cryptographic proofs on concrete schemes.
- Certify compilation from schemes to executable code.
- Along the way, we capture some side-channel leakage.
Computer-aided cryptography

[...] many proofs in cryptography have become essentially unverifiable. Our field may be approaching a crisis of rigor. Bellare and Rogaway, 2004-2006

- Computer-aided tools developed to increase confidence in complex cryptographic proofs.
  - CryptoVerif, CertiCrypt, EasyCrypt
- Good automated tools to deal with additional, “real-world” details (refinement, adversary).
  - SMT solvers are decent with programming constructs;
  - tactics give some form of automation for higher-order logics.
Reduction-based proofs

**Diagram:**

- **Primitive**
  - **Generic construction**
  - **Scheme**

- **Attack**
  - **Black-box reduction**

**Notes:**

- Reduction-based proofs involve showing that an attack on the scheme implies an attack on the primitive.
- Generic construction is a method to create new schemes from existing ones.
- Black-box reduction is a technique to prove security by reducing a problem to a simpler one.

**Key Terms:**

- Reduction-based proofs
- Scheme
- Primitive
- Generic construction
- Attack
- Black-box reduction
Public-key encryption
Indistinguishability against chosen-ciphertext attacks

Algorithms \((\mathcal{K}, \mathcal{E}_{pk}, \mathcal{D}_{sk})\)

- \(\mathcal{E}\) probabilistic
- \(\mathcal{D}\) deterministic and partial

If \((sk, pk)\) is a valid key pair,

\[
\mathcal{D}_{sk}(\mathcal{E}_{pk}(m)) = m
\]

**Game** \(\text{IND}(\mathcal{A})\)

\[
(\mathcal{K}, \mathcal{E}_{pk}, \mathcal{D}_{sk}) \leftarrow \mathcal{K}();
(\mathcal{E}_{pk}, \mathcal{D}_{sk}) \leftarrow \mathcal{A}_{1}(pk);
\]

\[
b \leftarrow \{0, 1\};
\]

\[
c^* \leftarrow \mathcal{E}_{pk}(m_b);
\]

\[
b' \leftarrow \mathcal{A}_{2}(c^*);
\]

return \((b' = b)\)

- \(\mathcal{A}_1\) can call \(\mathcal{D}\).
- \(\mathcal{A}_2\) can call \(\mathcal{D}\) except on \(c^*\).
- \(\mathcal{A}\) wins if \(b = b'\).
One-way trapdoor permutations
set Partial-Domain One-Way

Algorithms \((\mathcal{K}, f_{pk}, f_{sk}^{-1})\)

- \(f_{pk}\) and \(f_{sk}^{-1}\) deterministic

If \((sk, pk)\) is a valid key pair,

\[ f_{sk}^{-1}(f_{pk}(m)) = m \]

\[
\text{Game } \text{sPDOW}(I) \\
(\text{sk, pk}) \leftarrow \mathcal{K}(); \\
s \leftarrow \{0, 1\}^{k_0}; \\
t \leftarrow \{0, 1\}^{k_1}; \\
x^* \leftarrow f_{pk}(s|t); \\
S \leftarrow I(pk, x^*); \\
\text{return } (s \in S)
\]

- \(I\) can compute \(f_{pk}\).
- \(I\) wins if \(s \in S\).

\[ \text{Pr}_{\text{sPDOW}(I)}[s \in S] \text{ is small} \]
Optimal Asymmetric Encryption Padding

Encryption $\mathcal{E}_{\text{OAEP}}(pk)(m)$:
- $r \leftarrow \{0, 1\}^{k_0}$;
- $s \leftarrow G(r) \oplus (m \parallel 0^{k_1})$;
- $t \leftarrow H(s) \oplus r$;
- return $f_{pk}(s \parallel t)$

Decryption $\mathcal{D}_{\text{OAEP}}(sk)(c)$:
- $(s, t) \leftarrow f^{-1}_{sk}(c)$;
- $r \leftarrow t \oplus H(s)$;
- if $([s \oplus G(r)]_{k_1} = 0^{k_1})$
  then $\{m \leftarrow [s \oplus G(r)]_k; \}$
  else $\{m \leftarrow \perp; \}$
- return $m$

$\oplus$ exclusive or  $\parallel$ concatenation  $[\cdot]$ projection  0 zero bitstring

Theorem (Fujisaki et al., 2004)
For every IND-CCA adversary $\mathcal{A}$ against $(\mathcal{K}, \mathcal{E}_{\text{OAEP}}, \mathcal{D}_{\text{OAEP}})$, there exists a set-PDOW adversary $\mathcal{I}$ against $(\mathcal{K}, f, f^{-1})$ s.t.

$$\left| \Pr_{\text{IND-CCA}(\mathcal{A})}[b' = b] - \frac{1}{2} \right| \leq \Pr_{\text{SPDOW}(\mathcal{I})}[s \in S] + \frac{2q_D q_G + q_D + q_G}{2^{k_0}} - \frac{2q_D}{2^{k_1}}$$
A Low-Level Model...

Decryption $D_{\text{OAEP}}(sk)(c)$:
\[
(s, t) \leftarrow f^{-1}_{sk}(c); \\
r \leftarrow t \oplus H(s); \\
\text{if } ([s \oplus G(r)]_{k_1} = 0^{k_1}) \\
\text{then } \{ m \leftarrow [s \oplus G(r)]^k; \} \\
\text{else } \{ m \leftarrow \bot; \} \\
\text{return } m
\]
Decryption $D_{\text{OAEP}}(sk)(c)$:

$(s, t) \leftarrow f^{-1}_{sk}(c)$;
$r \leftarrow t \oplus H(s)$;
if $([s \oplus G(r)]_{k_1} = 0^{k_1})$
then \{ $m \leftarrow [s \oplus G(r)]^{k}$; \}
else \{ $m \leftarrow \perp$; \}
return $m$

Decryption $D_{\text{PKCS-C}}(sk)(res, c)$:

if $(c \in \text{MsgSpace}(sk))$
\{ $(b_0, s, t) \leftarrow f^{-1}_{sk}(c)$;
$h \leftarrow \text{MGF}(s, hL); i \leftarrow 0$;
while $(i < hLen + 1)$
\{ $s[i] \leftarrow t[i] \oplus h[i]; i \leftarrow i + 1$; \}
g $\leftarrow$ \text{MGF}(r, dbL); $i \leftarrow 0$;
while $(i < dbLen)$
\{ $p[i] \leftarrow s[i] \oplus g[i]; i \leftarrow i + 1$; \}
l $\leftarrow$ \text{payload_length}(p);
if $(b_0 = 0^8 \land [p]^{hLen} = 0..01 \land$
\[ [p]^{hLen} = LHash \])
then
\{ $rc \leftarrow \text{Success};$
memcopy(res, 0, p, dbLen - l, l); \}
else \{ $rc \leftarrow \text{DecryptionError};$ \}
else \{ $rc \leftarrow \text{CiphertextTooLong};$ \}
return $rc$;
We consider Program Counter Security.

The adversary is given the list of program points traversed while executing the oracle.

Leakage due to the computation of the permutation is kept abstract but given.

Security assumption (sPDOW) is slightly adapted to deal with abstract leakage.
Game IND₁(𝒜)
{(sk, pk) ← K();
  (m₀, m₁) ← 𝒜₁(pk);
  b ← {0, 1};
  c*, τ ← Eₚk(m_b);
  b' ← 𝒜₂(c*, τ);
  return (b' = b)

- We don’t consider leakage (or even security) of the key generation algorithm.
- Oracles return a PC trace τ along with their results.
- Traces are passed on to the adversary (or directly returned if the adversary called the oracle).
Decryption $D_{PKCS-C}(sk)(res, c) :$

\begin{align*}
\tau & \leftarrow \epsilon; \\
& \text{if } (c \in \text{MsgSpace}(sk)) \\
& \quad \{ \tau \leftarrow L :: \tau; (b_0, s, t) \leftarrow f^{-1}_{sk}(c); \\
& \quad \quad h \leftarrow \text{MGF}(s, hL); i \leftarrow 0; \\
& \quad \quad \text{while } (i < hLen + 1) \\
& \quad \quad \quad \{ \tau \leftarrow L :: \tau; s[i] \leftarrow t[i] \oplus h[i]; i \leftarrow i + 1; \} \\
& \quad \tau \leftarrow R :: \tau; g \leftarrow \text{MGF}(r, dbL); i \leftarrow 0; \\
& \quad \text{while } (i < dbLen) \\
& \quad \quad \{ \tau \leftarrow L :: \tau; p[i] \leftarrow s[i] \oplus g[i]; i \leftarrow i + 1; \} \\
& \quad \tau \leftarrow R :: \tau; l \leftarrow \text{payload\_length}(p); \\
& \quad \text{if } (b_0 = 0^8 \land [p]_{hLen} = 0..01 \land [p]_{hLen} = LHash) \\
& \quad \quad \{ \tau \leftarrow L :: \tau; rc \leftarrow \text{Success}; \\
& \quad \quad \quad \text{memcpy}(res, 0, p, dbLen - l, l); \} \\
& \quad \text{else} \{ \tau \leftarrow R :: \tau; rc \leftarrow \text{DecryptionError}; \} \\
& \quad \text{else} \{ \tau \leftarrow R :: \tau; rc \leftarrow \text{CiphertextTooLong}; \} \\
& \text{return } rc, \tau;
\end{align*}
Modelling Leakage in sPDOW

**Game** $\text{sPDOW}_1(\mathcal{I})$

$(sk, pk) \leftarrow \mathcal{K}();$

$sk_l \leftarrow \text{leak}(sk);$

$s \leftarrow \{0, 1\}^{k_0};$

$t \leftarrow \{0, 1\}^{k_1};$

$x^*, \tau \leftarrow f_{pk}(s || t);$

$S \leftarrow \mathcal{I}(pk, x^*, sk_l, \tau);$

return ($s \in S$)

- PC traces produced when computing $f_{\text{sk}}^{-1}(c)$ and $f_{\text{leak}(sk)}^{-1}(c)$ are the same for any $c$.
- The leak function is “fixed” by the implementation of $f^{-1}$.
- Easily applies the result to various concrete implementations of $f^{-1}$,
- but the final security result may be very weak (or even vacuous).
Proving Security

We use EasyCrypt (Barthe et al, CRYPTO’11), an SMT-based computer-aided tool for cryptographic proofs. We extend it with variable-length bytestrings.

- First step: abstract away low-level implementation details
  - Imperative arrays into functional bitstrings,
    \[
    \begin{array}{l}
    i \leftarrow 0; \\
    \text{while } (i < hLen + 1) \\
    \{ s[i] \leftarrow t[i] \oplus h[i]; i \leftarrow i + 1; \}
    \end{array} \implies s \leftarrow t \oplus h;
    \]
  - Separate computation and leakage,
  - ~3000 lines of proof - but we know how to automate this.

- Then: a variant of Fujisaki et al.’s proof
  - 6 main games, some intermediate games,
  - ~3000 lines of proof - This is normal.
Reductionist proof
Theorem (Abstract Security)

For every IND-CCA adversary $A$ against $(\mathcal{K}, \mathcal{E}_{PKCS}, \mathcal{D}_{PKCS})$, there exists an sPDOW adversary $I$ against $(\mathcal{K}, f, f^{-1})$ s.t.

$$\left| \Pr_{\text{IND-CCA}_I}(A)[b' = b] - \frac{1}{2} \right| \leq \Pr_{\text{sPDOW}_I(I)}[s \in S] + \frac{2q_D q_G + q_D + q_G}{2^{k_0}} - \frac{2q_D}{2^{k_1}}$$

- The first part of the proof reduces the security of $(\mathcal{K}, \mathcal{E}_{PKCS-C}, \mathcal{D}_{PKCS-C})$ to that of $(\mathcal{K}, \mathcal{E}_{OAEP}, \mathcal{D}_{OAEP})$ with the same bound.
- Is the result preserved through compilation?
- Even with the side-channels?
Going from “EasyCrypt C-mode” to C is a syntactic transformation.

- “C-mode” arrays are “base-offset” representation and match subset of C arrays (no aliasing or overlap possible, pointer arithmetic only within an array).
- Very little care needed so leakage traces correspond.

Going from C to ASM is more complicated.

- We use CompCert.
CompCert

- CompCert is a certified optimizing C compiler (in Coq).
- It comes with a proof of semantic preservation expressed in terms of (potentially infinite) traces of events.
  - Only terminating programs.
  - Only “safe” programs (no undefined behaviours).
- A trace of events is possible in compiled program iff it is possible in the source program.
- Events can be
  - system calls (“external calls”),
  - I/O from and to the environment, and
  - user-defined events (parameterized by base-typed values).
Compiling computationally-secure code

- Probabilistic operations pushed into the environment:
  - ideal random sampling of bitstrings,
  - hash function (random oracle),
- Trusted multi-precision arithmetic libraries modelled as external calls:
  - extensions needed to let external calls modify memory,
  - CompCert and proof extended with this mechanism,
- This is all we would need if we didn’t care about the side-channels.
Compiling PC-secure Programs using CompCert

- User-defined events sufficient to model PC traces.
- They would be sufficient to model more complex leakage models as well.
- Given the chance, compilation could introduce observable differences in the PC trace.
  - A simple static analysis on ASM programs: “There is at least one branching annotation between any two conditional statements.”
  - A Coq proof that this is sufficient to efficiently reconstruct PC traces in a 1-1 way.
Our extensions to CompCert can be interpreted as extensions to our subset of C.

- “trusted-lib” extends C with a datatype and functions for arbitrary precision integer arithmetic.
  - Implemented using a cut-down version of LIP,
  - augmented with (formally verified) PC countermeasures,
  - but without functional verification.

- the new environment functions (random sampling and hash functions) can be seen as new system calls.
  - Implemented using NaCl.
What is \texttt{gcc -O1} doing to the PC and cache countermeasures in the trusted library?
We use EasyCrypt to prove cryptographic security of C-like code with leakage.

We use CompCert to preserve the security guarantees through compilation and down to assembly.

We illustrate the technique on an implementation of PKCS#1.

Mind the Gap

Still a model.

Adversary and execution models are still somewhat idealized:

- Adversary is *not* in the same virtual address space,
- We do not (yet) capture value-dependent leakage
Going Further: cache attacks and StealthMem
Work by G. Barthe, G. Betarte, J.D. Campo, C. Luna, and D. Pichardie

- Array accesses with secret indices is a recipe for disaster
- Want obliviousness: sequence of memory accesses does not leak secrets

Problems
- compilers might break obliviousness
- algorithms might fail to satisfy obliviousness

Solution
- information flow analysis on CompCert lower levels
- tag arrays accessed with secret indices
- use stealth memory to protect these arrays
What next?

- Consider lower-level adversaries and counter-measures
- Consider more active adversaries (fault injection)
- Keep moving towards a 'verified cryptographic software toolchain' going from algorithms to executable code.
  - Automation and modularity in and between the proofs,
  - Adding more language extensions (DH, EC, polys ...),
  - Verify functional correctness and leakage properties of the trusted libraries.

http://www.easycrypt.info

New

- Formal support for compositional proofs and abstraction.
- Ability to prove high-level transformations once and for all and instantiate them in concrete proofs.