Verifiable Side-Channel Security of Cryptographic Implementations: Constant-Time MEE-CBC.

José Bacelar Almeida[1,2] Manuel Barbosa[2,3]
Gilles Barthe[4] François Dupressoir[4]

[1]Universidade do Minho [2]HASLab – INESC Tec
[3]Universidade do Porto [4]IMDEA Software Institute

March 21, 2016 – FSE'16
RU Bochum

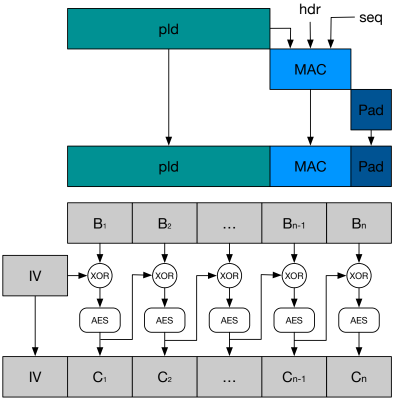
Breaking that Title Down

Our main practical contribution: A *machine-checked* proof of IND$-CPA and INT-PTXT security for x86 code implementing *MAC-then-Encode-then-CBC-Encrypt* (MEE-CBC) against some *timing adversaries*.

- Why *MEE-CBC*? Simple crypto, but very difficult to implement securely.
- Why *machine-checked*? Necessary to take implementation details into account, and verify implementations for properties not easily testable...
- Such as their *timing behaviour*, which has been exploited in the past to break MEE-CBC. We show a new attack on AWS Labs's implementation of MEE-CBC in s2n.

To achieve this, we present a framework to break such proofs down into simpler problems.

MEE-CBC: An Overview



- Payload is fed through MAC with additional data;
- Payload and tag are concatenated and padded to multiple of block length;
- The result is fed through AES-CBC.

On the Side-Channel Security of MEE-CBC

When decrypting:

- Length of padding must be known to check the MAC;
- Padding validity needs to be checked.

The problem: AES-CBC provides only CPA security.

- Decrypted ciphertext is sensitive until MAC has been checked.

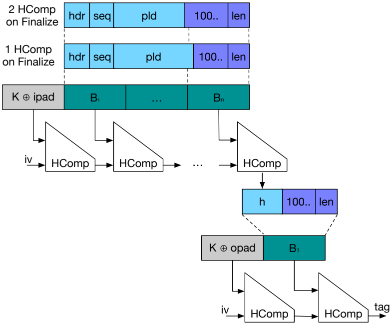
Countermeasures and attacks:

- Both padding and MAC computation must be performed *always* [Vaudenay, 2002];
- Number of compression function queries in MAC computation must be independent from padding length or validity [AlFardan and Paterson, 2013];

Countermeasures in Practice

- After Lucky Thirteen [AlFardan and Paterson, 2013], many switch to “constant-time” programming policy.
  - No secret-dependent branching (prevents coarse leaks via overall execution time, some leaks via branch prediction);
  - No secret-dependent memory accesses (prevents precise leakage via cache timing).
- In s2n, AWS Labs do limited mitigation in MEE-CBC *and* hide whatever leakage is left behind a random delay.
  - Randomization is insufficient in practice [Albrecht and Paterson, 2016];
  - More mitigation was added (and noise increased).

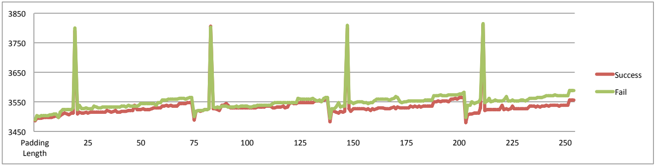
HMAC and s2n's Additional Mitigation



- Mitigation aimed at better balancing number of compression function calls.
- Finalize call for inner hash may make 1 or 2 compression queries depending on length of final message block;
- 9 bytes are reserved for SHA-X padding (8 payload length bytes + 1 0x80 byte).

An Off-by-One Error, a Leak and an Attack

- When deciding whether or not to make a dummy compression query, s2n checks whether there are 8 bytes left instead of 9.
- This leads to large timing discrepancies for interesting values of the payload length:



- Without randomized delay, this leads to plaintext recovery, following Lucky Thirteen.

End-to-End Verification of Cryptographic Security with Side-Channels

Cut the problem of proving security of implementation against side-channel adversary into three tasks:

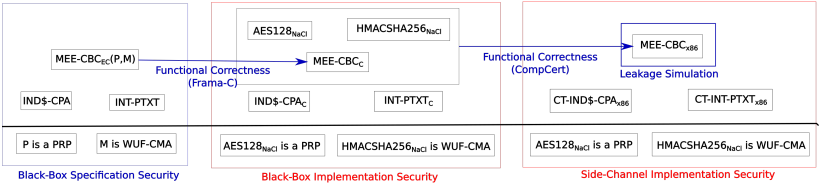
Black-box specification security usual notion of provable security;

Functional correctness of implementation: the input-output behaviour of the implementation is the same as that of the specification;

Leakage simulation for all inputs, the leakage produced during execution of the algorithm can be efficiently and perfectly simulated given only public inputs.

Framework Theorem: black-box specification security $\wedge$ functional correctness $\wedge$ leakage simulation $\Rightarrow$ side-channel implementation security.

Application to MEE-CBC



- We formalize a black-box security proof in EasyCrypt.
- We prove equivalence of a new C implementation of MEE-CBC to a functional spec extracted from EasyCrypt.
  - EasyCrypt specification is generic in block and tag lengths and (length-regular and invertible) padding function.
  - We instantiate it with relevant values (and discharge proofs) before extraction.
- We compile it using CompCert (formally proved C compiler).
- We verify leakage simulation of the compiled code using the certified constant-time verifier by [Barthe et al., 2014].

Black-Box Specification Security Formally

Game IND\$-CPA$_\Pi^{\mathcal{A}}$():

$b \xleftarrow{\$} \{0,1\}$
key $\xleftarrow{\$}$ Gen()
$b' \xleftarrow{\$} \mathcal{A}^{\textbf{RoR},\textbf{Dec}}()$
Return $(b' = b)$

proc. RoR(m):

If (b = Ideal)
Then c $\xleftarrow{\$}$ Enc(m, key)
Else c $\xleftarrow{\$} \{0,1\}^{|\mathsf{m}|}$
Return c

proc. Dec(c):

m $\leftarrow$ Dec(c, key)
Return $\perp$

- We also prove some weak length hiding.
  - Not shown here: we don't transfer it to implementation.

Side-Channel Implementation Security Formally

Game $\mathbb{M}$-IND\$-CPA$^{\mathcal{A}}_{\Pi^*,\phi}(b)$:

key $\xleftarrow{\$} \mathbb{M}(\text{Gen}^*)_{\rightsquigarrow \ell_g}$
$b' \xleftarrow{\$} \mathcal{A}^{\textbf{RoR}\ \textit{Decrypt}}(\ell_g)$
Return $(b' = b)$

proc. RoR(m):

c $\xleftarrow{\$} \mathbb{M}(\text{Enc}^*, \text{m}, \text{key})_{\rightsquigarrow \ell_e}$
If $(b = \text{Ideal})$ Then c $\xleftarrow{\$} \{0,1\}^{|\text{m}|}$
Return (c, ℓ_e)

proc. Dec(c):

m $\leftarrow \mathbb{M}(\text{Dec}^*, \text{c}, \text{key})_{\rightsquigarrow \ell_d}$
Return $(\perp, \ell_d)$

- Applies to implementations of the primitive in a language $\mathcal{L}$...
- ... whose leaky semantics are animated by a machine $\mathbb{M}$.
- We use the same $\mathbb{M}$ as [Barthe et al., 2014]:
  - language is x86,
  - semantics are those of CompCert,
  - leakage trace reveals ordered sequence of branching operations and memory accesses.

Total Functional Correctness Formally

Game $\mathbf{Corr}^{\mathcal{A}}_{\mathbb{M},\Pi,\Pi^*}()$:

bad $\leftarrow$ false
$\mathcal{A}^{\mathbf{Eval}}$
Return $\neg$ bad

proc. Eval$(k, i, \mathbf{r})$:

$o \leftarrow \Pi[k](i; \mathrm{r})$
$o' \leftarrow \mathbb{M}(\Pi^*[k], i; \mathrm{r})_{\rightsquigarrow \ell}$
If $o \neq o'$ then bad $=$ true

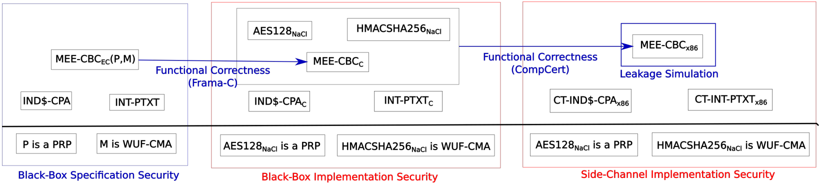
- Captures perfect (rather than probabilistic) correctness.
  - Prevents algorithm substitution attacks IF the property can be checked before running.
  - Some weakening may be possible if a proof of resilience against ASA exists on the specification.
- This is trivially implied by standard notions of correctness in program verification:
  - Functional correctness; or
  - When lifted to the compiler: semantic preservation.

Leakage Simulation Formally

Game $\mathsf{LeakSim}^{\mathcal{A}}_{\mathbb{M},\Pi^*,\Sigma}()$:	**proc. Leak**(alg, i, **r**):
bad $\leftarrow$ false	$o \leftarrow \mathbb{M}(\Pi^*[\mathsf{alg}], i; \mathsf{r})_{\leadsto \ell}$
$\mathcal{A}^{\mathbf{Leak}}$	$\ell' \leftarrow \Sigma[\mathsf{alg}](\tau_{\mathsf{alg}}(i; \mathsf{r}))$
Return $\neg$ bad	If $\ell \neq \ell'$ then bad = true

- τ_{alg} is determined by the black-box security experiment for each algorithm:
  - $\tau_{\mathsf{Gen}} = \emptyset$,
  - $\tau_{\mathsf{Enc}} = \{|\mathsf{key}|, |\mathsf{m}|\}$,
  - $\tau_{\mathsf{Dec}} = \{|\mathsf{key}|, \mathsf{c}\}$.
- Corresponds exactly to the standard language-based security notion of *non-interference*.
  - Easily and efficiently verified using type systems.
- Can be weakened by allowing simulator to use *public outputs* while retaining results.

Application to MEE-CBC



- We formalize a black-box security proof in EasyCrypt.
- We prove equivalence of a new C implementation of MEE-CBC to a functional spec extracted from EasyCrypt.
  - EasyCrypt specification is generic in block and tag lengths and (length-regular and invertible) padding function.
  - We instantiate it with relevant values (and discharge proofs) before extraction.
- We compile it using CompCert (formally proved C compiler).
- We verify leakage simulation of the compiled code using the certified constant-time verifier by [Barthe et al., 2014].

Performance

Implementation	Compiler	Time
s2n	GCC -O2	5μs
OpenSSL	GCC -O2	9μs
MEE-CBC$_{\mathsf{C}}$ (AES-NI)	CompCert*	21μs
MEE-CBC$_{\mathsf{C}}$	GCC -O2	25ms
MEE-CBC$_{\mathsf{C}}$	GCC -O1	26ms
MEE-CBC$_{\mathrm{x86}}$	CompCert	42ms
MEE-CBC$_{\mathsf{C}}$	GCC -O0	99ms

- Time taken to decrypt a 1.5kB TLS record.
- A very large part of the cost is due to constant-time AES.
  - Vector instructions not supported by CompCert
  - AES-NI gives reasonable results even with modified CompCert
  - But not all proofs have been adapted
- Some is due to CompCert (typically ca. $2\times$ w.r.t. GCC -O2).
- A small part is due to constant-time MEE-CBC.

Summary

- Some formal guarantees can be obtained in realistic settings.
- We propose a framework that breaks the problem down into more manageable parts, essentially by *successive refinements*.
- There is still a cost to pay for formal guarantees.
- In proof effort:
  - in practice, most effort expended in top two levels;
  - twisting the implementation to guarantee leakage simulation makes it harder to verify functional correctness.
- In performance:
  - in practice, most of that cost comes from primitive design;
  - in theory, most of what's left could be absorbed by proof effort.
- Our framework would support this, among other things.

(Some) Advantages of Successive Refinements

- Modular trust:
  - Trust [Paterson, Ristenpart and Shrimpton, 2011]? Get black-box LH-AEAD and side-channel INT-CTXT for free on the compiled code.
  - Trust the C code? No need to verify its equivalence with the functional specification.
- Proof Reuse:
  - Black-box specification security can be used for many implementations;
  - C-level equivalence proof is valid (almost) independently of the compiler;
- Tool (and Language) Independence. Leverage advances and expertise in each subtask.
  - [Beringer et al., 2015]: FCF, Verified-C and CompCert to prove properties of HMAC implementation.
  - [Bernstein and Schwabe, 2016]: GFVerif for automatic proofs of correctness for finite field arithmetic implemented in C.