Error-prone cryptographic designs

Daniel J. Bernstein
University of Illinois at Chicago &
Technische Universiteit Eindhoven

"*The poor user is given enough rope with which to hang himself—something a standard should not do.*"
—1992 Rivest,
commenting on nonce generation inside Digital Signature Algorithm (1991 proposal by NIST, 1992 credited to NSA, 1994 standardized by NIST)

Crypto horror story #1

2010 Bushing–Marcan–Segher–Sven "failOverflow" demolition of Sony PS3 security system: Sony had ignored requirement to generate new random nonce for each ECDSA signature.
$\Rightarrow$ Sony's signatures leaked Sony's secret code-signing key.

Crypto horror story #1

2010 Bushing–Marcan–Segher–Sven "failOverflow" demolition of Sony PS3 security system: Sony had ignored requirement to generate new random nonce for each ECDSA signature.
$\Rightarrow$ Sony's signatures leaked Sony's secret code-signing key.

Traditional response: Blame Sony.
Blame the crypto implementor.

Crypto horror story #1

2010 Bushing–Marcan–Segher–Sven “failOverflow” demolition of Sony PS3 security system: Sony had ignored requirement to generate new random nonce for each ECDSA signature.
$\Rightarrow$ Sony’s signatures leaked Sony’s secret code-signing key.

Traditional response: Blame Sony.
Blame the crypto implementor.

Rivest’s response: Blame DSA.
Blame the crypto designer.

Crypto horror story #1

2010 Bushing–Marcan–Segher–Sven "failOverflow" demolition of Sony PS3 security system: Sony had ignored requirement to generate new random nonce for each ECDSA signature.
$\Rightarrow$ Sony's signatures leaked Sony's secret code-signing key.

Traditional response: Blame Sony.
Blame the crypto implementor.

Rivest's response: Blame DSA.
Blame the crypto designer.
Change DSA to avoid this pitfall!

Crypto horror story #2

2005 Osvik–Shamir–Tromer:
65ms to steal Linux AES key
used for hard-disk encryption.
Attack process on same CPU
but without privileges.

Almost all AES implementations
use fast lookup tables.
Kernel's secret AES key
influences table-load addresses,
influencing CPU cache state,
influencing measurable timings
of the attack process.
65ms to compute influence^{-1}.

2012 Mowery–Keelveedhi–Shacham: "We posit that any data-cache timing attack against x86 processors that does not somehow subvert the prefetcher, physical indexing, and massive memory requirements of modern programs is doomed to fail."

2012 Mowery–Keelveedhi–Shacham: “We posit that any data-cache timing attack against x86 processors that does not somehow subvert the prefetcher, physical indexing, and massive memory requirements of modern programs is doomed to fail.”

2014 Irazoqui–Inci–Eisenbarth–Sunar “Wait a minute! A fast, Cross-VM attack on AES” recovers “the AES keys of OpenSSL 1.0.1 running inside the victim VM” in 60 seconds despite VMware virtualization.

After many, many, many papers on implementations and attacks, today we still have an ecosystem plagued with AES vulnerabilities.

Warning: more papers $\neq$ security.

After many, many, many papers on implementations and attacks, today we still have an ecosystem plagued with AES vulnerabilities. Warning: more papers $\neq$ security.

AES has a serious conflict between security, simplicity, speed. It's tough to achieve security while insisting on the AES design —i.e., blaming the implementor.

After many, many, many papers on implementations and attacks, today we still have an ecosystem plagued with AES vulnerabilities. Warning: more papers $\neq$ security.

AES has a serious conflict between security, simplicity, speed. It's tough to achieve security while insisting on the AES design —i.e., blaming the implementor.

Allowing the *design* to vary makes security much easier. Next-generation ciphers are naturally constant-time and fast.

The big picture

Crypto
designs

more complexity
less review
more attacks
less security

Crypto
implementations

The big picture

Primitive designs

more complexity
less review
more attacks
less security

Primitive implementations

Protocol designs

more complexity
less review
more attacks
less security

Protocol implementations

Public review is naturally biased towards the simplest targets, ignoring complications of protocols and implementations.

There's much more public review of, e.g., discrete logarithms than of ECDSA signatures.

Public review is naturally biased towards the simplest targets, ignoring complications of protocols and implementations.

There's much more public review of, e.g., discrete logarithms than of ECDSA signatures.

There's much more public review of the ECDSA design than of ECDSA implementations.

Public review is naturally biased towards the simplest targets, ignoring complications of protocols and implementations.

There's much more public review of, e.g., discrete logarithms than of ECDSA signatures.

There's much more public review of the ECDSA design than of ECDSA implementations.

There's much more public review of ECDSA implementations than of ECDSA applications.

What about security proofs?

The fundamental goal
of “provable security”:
Prove that the whole system
is as secure as the primitive.

i.e.: Prove that the protocol
is as secure as the primitive.
Prove that the implementation
is as secure as the design.
Then it’s safe for reviewers
to focus on the primitive design.

What about security proofs?

The fundamental goal
of "provable security":
Prove that the whole system
is as secure as the primitive.

i.e.: Prove that the protocol
is as secure as the primitive.
Prove that the implementation
is as secure as the design.
Then it's safe for reviewers
to focus on the primitive design.

Maybe will succeed someday, but
needs to overcome huge problems.

Problem 1: “Proofs” have errors.
Proofs are increasingly complex,
rarely reviewed, rarely automated.

Problem 1: “Proofs” have errors.
Proofs are increasingly complex,
rarely reviewed, rarely automated.

Problem 2: Most proofs are of
security bounds that aren’t tight:
e.g. forking-lemma “security”
is pure deception for typical sizes.

Problem 1: “Proofs” have errors.
Proofs are increasingly complex,
rarely reviewed, rarely automated.

Problem 2: Most proofs are of
security bounds that aren’t tight:
e.g. forking-lemma “security”
is pure deception for typical sizes.

Problem 3: “Security” definitions
prioritize simplicity over accuracy.
e.g. is MAC-pad-encrypt secure?

Problem 1: "Proofs" have errors.
Proofs are increasingly complex, rarely reviewed, rarely automated.

Problem 2: Most proofs are of security bounds that aren't tight: e.g. forking-lemma "security" is pure deception for typical sizes.

Problem 3: "Security" definitions prioritize simplicity over accuracy. e.g. is MAC-pad-encrypt secure?

Problem 4: Maybe the only way to achieve the fundamental goal is to switch to weak primitives.

Some advice to crypto designers

Creating or evaluating a design?
Think about the implementations.
What will the implementors do?
What errors are likely
to appear in implementations?
Can you compensate for this?

Some advice to crypto designers

Creating or evaluating a design?
Think about the implementations.
What will the implementors do?
What errors are likely
to appear in implementations?
Can you compensate for this?

Is the design a primitive?
Think about the protocols.
Is the design a protocol?
Think about the higher-level protocols.
Will the system be secure?

Crypto horror story #3

HTTPS.

Crypto horror story #3

HTTPS.

`letsencrypt.org` is a huge improvement in HTTPS usability; will obviously be widely used.
But is HTTPS actually secure?

Crypto horror story #3

HTTPS.

`letsencrypt.org` is a huge improvement in HTTPS usability; will obviously be widely used.
But is HTTPS actually secure?

"It's not so bad
against passive eavesdroppers!"
—The eavesdroppers will start forging (more) packets.

Crypto horror story #3

HTTPS.

`letsencrypt.org` is a huge improvement in HTTPS usability; will obviously be widely used.
But is HTTPS actually secure?

"It's not so bad
against passive eavesdroppers!"
—The eavesdroppers will start forging (more) packets.

"Then we'll know they're there!"
—Yes, we knew that already.
What we want is *security*.

2013.01 Green:

State-of-the-art TLS proofs are obviously unsatisfactory.

e.g. Defense vs. Bleichenbacher requires "goofy made-up assumption" for proof.

2013.01 Green:

State-of-the-art TLS proofs are obviously unsatisfactory.

e.g. Defense vs. Bleichenbacher requires "goofy made-up assumption" for proof.

But that was "the *good news* ... The problem with TLS is that we are cursed with *implementations*."

e.g. Defense vs. Bleichenbacher is in wrong order in OpenSSL. Does this allow timing attacks?

2014.08 Meyer–Somorovsky–Weiss–Schwenk–Schinzel–Tews: Successful Bleichenbacher attacks, exploiting analogous timing variations in Java SSE, Cavium NITROX SSL accelerator chip.

2014.08 Meyer–Somorovsky–Weiss–Schwenk–Schinzel–Tews: Successful Bleichenbacher attacks, exploiting analogous timing variations in Java SSE, Cavium NITROX SSL accelerator chip.

The whole concept of a "public-key cryptosystem" is a historical accident, dangerously unauthenticated.

2014.08 Meyer–Somorovsky–Weiss–Schwenk–Schinzel–Tews: Successful Bleichenbacher attacks, exploiting analogous timing variations in Java SSE, Cavium NITROX SSL accelerator chip.

The whole concept of a "public-key cryptosystem" is a historical accident, dangerously unauthenticated.

Do we seriously believe that we'll make HTTPS secure by fixing the implementations?
Fix the bad crypto design.

Exercise: How many of these TLS failures can a *designer* address?

Renegotiation attack.

Diginotar CA compromise.

BEAST CBC attack.

Trustwave HTTPS interception.

CRIME compression attack.

Lucky 13 padding/timing attack.

RC4 keystream bias.

TLS truncation.

gotofail signature-verification bug.

Triple Handshake.

Heartbleed buffer overread.

POODLE padding-oracle attack.

Winshock buffer overflow.