High-speed cryptography, DNSSEC, and DNSCurve

D. J. Bernstein
University of Illinois at Chicago

NSF ITR–0716498

Stealing Internet mail: easy!

Given a mail message:
Your mail software
sends a DNS request,
receives a server address,
makes an SMTP connection,
sends the From/To lines,
sends the mail message.

Attackers can easily
see all of these packets
and change the packets.

Forging web pages: easy!

Starting from a URL:
Your browser
sends a DNS request,
receives a server address,
makes an HTTP connection,
sends an HTTP request,
receives a web page.

Attackers can easily
see all of these packets
and change the packets.

Solved by cryptography?

In theory:

Cryptography stops these attacks.

Solved by cryptography?

In theory:
Cryptography stops these attacks.

In practice:
Am I using cryptography?
Are you using cryptography?

Solved by cryptography?

In theory:
Cryptography stops these attacks.

In practice:
Am I using cryptography?
Are you using cryptography?
Occasionally yes; usually no.

Solved by cryptography?

In theory:
Cryptography stops these attacks.

In practice:
Am I using cryptography?
Are you using cryptography?
Occasionally yes; usually no.

Problem 1:
Most Internet protocols
do not support cryptography.

Why not? Obvious answer:
Hard for protocol designers
to integrate cryptography.

Some popular Internet protocols *do* have cryptographic options. Important example: HTTPS.

Some popular Internet protocols *do* have cryptographic options. Important example: HTTPS.

Problem 2:
Most *implementations* of these protocols do not support cryptography.

Why not? Obvious answer: Hard for software authors to integrate cryptography. Much easier to implement the non-cryptographic option.

Some popular implementations
do support cryptography.
Example: Apache.

Some popular implementations
do support cryptography.
Example: Apache.

Problem 3:
Most *installations*
of these implementations
do not support cryptography.
$\approx$99% of the Apache servers on
the Internet do not enable SSL.

Why not? Obvious answer:
Hard for site administrators
to turn on the cryptography.

Some important installations *do* support cryptography.

Example: SourceForge has paid for an SSL certificate and set up SSL servers. Try `https://sourceforge.net/account`.

Some important installations *do* support cryptography.

Example: SourceForge has paid for an SSL certificate and set up SSL servers. Try `https://sourceforge.net/account`.

Problem 4: Cryptography is not enabled for most *data* at these installations.

Example: Try `https://sourceforge.net/community`. SourceForge redirects your browser to `http://sourceforge.net/community`.

Why does SourceForge actively *turn off* cryptographic protection?

Why does SourceForge actively *turn off* cryptographic protection?

Obvious answer: Enabling SSL for more than a small fraction of SourceForge connections would massively overload the SourceForge servers.

SourceForge doesn't want to pay for a bunch of extra computers.

Many companies sell SSL-acceleration hardware, but that costs money too.

Making progress

Obvious speed questions:

Why are cryptographic computations so expensive?

Can crypto be faster, without being easy to break?

Can crypto be fast enough to solidly protect all of SourceForge's communications?

Can crypto be fast enough to protect every Internet packet?

And questions beyond speed:

Can universal crypto be easy to use and administer?

Can universal crypto be easy to implement in software?

Can universal crypto be easy to add to protocols?

Can universal crypto be *usable*?

U.S. government, last century:

"Encryption is dangerous!
It can be used by terrorists,
drug dealers, pedophiles,
and money launderers!"

U.S. government, last century:
“Encryption is dangerous!
It can be used by terrorists,
drug dealers, pedophiles,
and money launderers!”

I say: Criminals have been using
encryption for a long time.
Low speed? Hard to use?
They use it anyway.
We cannot stop them.

U.S. government, last century:
"Encryption is dangerous!
It can be used by terrorists,
drug dealers, pedophiles,
and money launderers!"

I say: Criminals have been using
encryption for a long time.
Low speed? Hard to use?
They use it anyway.
We cannot stop them.

What we can do is improve
the speed and usability of
cryptography for normal people.

My current mission:
Cryptographically protect
every Internet packet
against espionage,
corruption, and sabotage.

Confidentiality despite espionage:
Spies cannot understand packets.

Integrity despite corruption:
Forged packets are detected.
User does not see wrong data.

Availability despite sabotage:
User *does* see *correct* data.

Securing DNS

DNSCurve cryptographically protects DNS packets against espionage, corruption, and sabotage.

DNSCurve is only for DNS, but same ideas can be adapted to many other protocols.

Warning: DNSCurve *does not* hide packet length, sender, etc. But it does provide confidentiality for contents of packets, plus strong integrity, availability.

Packet from DNSCurve client
to DNSCurve server:

- Here's my public key.
- Here's an encrypted DNS query.

Client encrypts, authenticates
using client's secret key,
server's public key.

Server verifies, decrypts
using server's secret key,
client's public key.

Packet from DNSCurve server
to DNSCurve client:

- Here's an encrypted response.

Server encrypts, authenticates
using server's secret key,
client's public key.

Client verifies, decrypts
using client's secret key,
server's public key.

Every packet is authenticated.

Client verifies every packet
immediately upon receipt.

If packet fails verification,
client discards packet
and waits for correct packet.

Attacker can stop correct packet
by flooding the network,
but this consumes many more
attacker resources than
sending a few forged packets.
$\Rightarrow$ Many fewer victims.

How does DNSCurve client retrieve server's public key?

Does it send more packets? No!

DNS architecture: DNS client learns IP address of `.ubuntu.com` DNS server from `.com` DNS server.

The `.com` server says:
"The `ubuntu.com` DNS server is named `ns3` and has IP address 209.6.3.210."

The name `ns3` was selected by the `ubuntu.com` administrator and given to `.com`.

To announce
his DNSCurve server's public key,
the `ubuntu.com` administrator
changes the name `ns3` to
an encoding of the public key.

The DNSCurve client
sees the public key, begins
cryptographically protecting
communication with that server.

An older approach

1993.11 Galvin: "The DNS Security design team of the DNS working group met for one morning at the Houston IETF."

1994.02 Eastlake–Kaufman, after months of discussions on `dns-security` mailing list: "DNSSEC" protocol specification.

Continued DNSSEC efforts have received millions of dollars of government grants: e.g., DISA to BIND; NSF to UCLA; DHS to Secure64.

The Internet has nearly
80000000 `*.com` names.

The Internet has nearly 80000000 `*.com` names.

Surveys by DNSSEC developers, last updated 2009.08.04, have found 274 `*.com` names with DNSSEC signatures.

116 on 2008.08.20; $274 > 116$.
"Wow, exponential growth!"

The Internet has nearly
80000000 `*.com` names.

Surveys by DNSSEC developers, last updated 2009.08.04, have found 274 `*.com` names with DNSSEC signatures.

116 on 2008.08.20; $274 > 116$.
"Wow, exponential growth!"

The same surveys show 941 IP addresses worldwide running DNSSEC servers.

DNSSEC's design is driven by fear of cryptographic overload.

Basic assumption: Busy servers cannot afford per-query crypto.

Consequences:
DNSSEC has no encryption.
DNSSEC has no DoS protection.
DNSSEC precomputes signatures.

Signature is computed once; saved; sent to many clients. Hopefully the server can afford to sign each DNS record once.

DNSSEC signatures do not depend on fresh client data.

Consequences:
To limit replay attacks, DNSSEC has to put expiration times on signatures. Normally 30 days; short intervals cause problems.

Attackers can still replay data for 30 days; replay across clients; etc.

DNSCurve: every response is freshly encrypted, authenticated.

To avoid punishing sysadmin, DNSSEC requires new code in every DNS-management tool.

Whenever a tool adds or changes a DNS record, it also has to precompute DNSSEC signature; store DNSSEC signature; arrange for re-signature before expiration.

Any mistakes destroy your domain ("DNSSEC suicide"). 2009: This happened to all ISC DLV DNSSEC users. UCLA admin: "The solution in all cases was to disable DNSSEC validation."

2009.06.02: “Today we reached a significant milestone in our effort to bolster online security . . . [.ORG is] the first open generic Top-Level Domain to successfully sign our zone with Domain Name Security Extensions (DNSSEC). To date, the .ORG zone is the largest domain registry to implement this needed security measure. . . . This process adds new records to the zone, which allows verification of the origin authenticity and integrity of data.”

Verification! Authenticity!
Integrity! Sounds great!

Verification! Authenticity!
Integrity! Sounds great!

Now I simply configure the new `.org` public key into my DNS software. Because the `.org` servers have implemented “this needed security measure” (signing with DNSSEC), it is no longer possible for attackers to forge data from those servers!

Verification! Authenticity! Integrity! Sounds great!

Now I simply configure the new `.org` public key into my DNS software. Because the `.org` servers have implemented "this needed security measure" (signing with DNSSEC), it is no longer possible for attackers to forge data from those servers!

... or is it?

Suppose an attacker forges a DNS packet from `.org`, including exactly the same DNSSEC signatures but *changing the NS+A records* to point to the attacker's servers.

Suppose an attacker forges a DNS packet from `.org`, including exactly the same DNSSEC signatures but *changing the NS+A records* to point to the attacker's servers.

Fact: DNSSEC clients will *accept the forgery*. The signatures say *nothing* about the NS+A records.

Suppose an attacker forges a DNS packet from `.org`, including exactly the same DNSSEC signatures but *changing the NS+A records* to point to the attacker's servers.

Fact: DNSSEC clients will *accept the forgery*. The signatures say *nothing* about the NS+A records.

`.org` can't actually handle signing complete database, so it sends "opt-out" signatures saying "Sorry, no security here."

Novice protocol-design error,
not forced by precomputation:
DNSSEC public keys are
distributed through an
ad-hoc channel.

Supporting this channel requires
changes in even more software:
registrar web interfaces,
registrar database tools, etc.
Even farther from being done
than the basic DNSSEC changes.

DNSCurve: reuse existing
server-name channel;
no changes to tools.

DNSSEC protocol details allow astonishing DDoS amplification, a giant step backwards in the fight against amplifiers.

http://cr.yp.to/talks/2009.08.10/slides.pdf explains how 200 sites, each sending just 3Mbps, trigger a 20000Mbps flood from the 941 DNSSEC servers against any desired target.

DNSSEC signatures don't exist for names not on server.

When asked about nonexistent `ixyz.clegg.com`, the `clegg.com` server returns signed statement "There are no names between `imogene.clegg.com` and `jennifer.clegg.com`."

The `clegg.com` administrator disabled DNS "zone transfers" — but then leaked the same data by installing DNSSEC! Also wrote a guide "DNSSEC in 6 minutes."

Myth: These privacy violations were fixed by NSEC3 (proposed standard, 2008).

Myth: These privacy violations were fixed by NSEC3 (proposed standard, 2008).

Reality: DNSSEC+NSEC3 leaks private information *much* more quickly than classic DNS.

DNSSEC+NSEC3 gives away *hashes* of existing names.
I currently have 9 computers (9 2.4GHz Core 2 Quad CPUs; part of `www.win.tue.nl/cccc/`) hashing

Myth: These privacy violations were fixed by NSEC3 (proposed standard, 2008).

Reality: DNSSEC+NSEC3 leaks private information *much* more quickly than classic DNS.

DNSSEC+NSEC3 gives away *hashes* of existing names. I currently have 9 computers (9 2.4GHz Core 2 Quad CPUs; part of `www.win.tue.nl/cccc/`) hashing 580000000000 name guesses per day.

Client has to verify DNSSEC signature for each response.

DNSSEC tries to reduce client-side costs through choice of crypto primitive.

Many DNSSEC crypto options: 640-bit RSA, original specs; 768-bit RSA, many docs; 1024-bit RSA, current RFCs (for “leaf nodes in the DNS”); DSA, “10 to 40 times as slow for verification” but faster for signatures.

I say:
Using RSA-1024 is irresponsible.

2003: Shamir–Tromer et al. concluded that 1024-bit RSA was already breakable by large companies and botnets.
$10 million: 1 key/year.
$120 million: 1 key/month.

2003: RSA Laboratories recommended a transition to 2048-bit keys "over the remainder of this decade." 2007: NIST made the same recommendation.

Will be a few years before
1024-bit RSA is breakable
by academics in small labs.
They're finishing RSA-768 now.

Will be a few years before
1024-bit RSA is breakable
by academics in small labs.
They're finishing RSA-768 now.

"RSA-1024: still secure
against honest attackers."

Will be a few years before
1024-bit RSA is breakable
by academics in small labs.
They're finishing RSA-768 now.

"RSA-1024: still secure
against honest attackers."

What about serious attackers
using many more computers?
e.g. botnet operators?

Government is mandating
at least 2048-bit RSA
by the end of next year.

DNSSEC made breakable choices
such as 640-bit RSA
for no reason other than
fear of cryptographic overload.

DNSSEC needed more options
to survive the inevitable breaks.

Profusion of options made
DNSSEC crypto complicated,
hard to review for bugs.

2009: Emergency BIND upgrade.
Minor software bug meant
that DNSSEC DSA signatures
had always been trivial to forge.

Cryptography in DNSCurve

Critical cryptographic operations:

Encrypt and authenticate packet
using server's secret key
and client's public key.

Verify and decrypt packet
using client's secret key
and server's public key.

Need serious security,
not something breakable
today by Storm, NSA, . . .
(and next decade by academics).

Could use public-key encryption
(e.g., 4096-bit RSA encryption)
and public-key signatures
(e.g., 4096-bit RSA signatures).

But why use two separate
public-key operations?
Combined operations are faster.

Why use *signatures*
that everyone can verify?
Better to use *authenticators*
verifiable by the recipient.

When client and server
exchange several messages,
why use several separate
public-key operations?

Classic “hybrid” speedup:
Client and server
use public-key operations
to share a secret,
and use secret-key cryptography
to protect many messages.

Elliptic-curve cryptography:
Client has secret key c,
public key Curve(c).
Server has secret key s,
public key Curve(s).
Client, server can cache
shared secret Curve(cs),
use secret-key cryptography
to protect many messages.

Introduced in 1985.
Today's best attacks
against random elliptic curves
use as much computer power
as 1985's best attacks.

1990s: ECC security criteria were standardized by IEEE P1363.

NIST used IEEE P1363 procedure to create several standard curves, such as the "P-256" curve.

More recent research recommends extra criteria to simplify and acclerate secure implementations. NIST P-256 flunks those criteria. The new "Curve25519" curve passes the IEEE P1363 criteria and the extra criteria. DNSCurve uses Curve25519.

So how fast is it?

New public-domain "Networking and Cryptography library", http://nacl.cace-project.eu:

crypto_box encrypts and authenticates a packet.

Can split crypto_box into
crypto_box_beforenm,
crypto_box_afternm
to cache and reuse shared secret.

crypto_box_open verifies and decrypts a packet.

Using this software, a low-cost PC
with a 2.4GHz Core 2 Quad CPU
can encrypt and authenticate
50 billion packets/day
to 500 million clients.

Also highly space-efficient:
32 bytes for a public key;
similar overhead per packet.

Major code contributions
from Adam Langley (Google)
and Matthew Dempsky
(Mochi Media, now OpenDNS).

The *total* load on `.com`
is 38 billion packets/day
from 5 million clients.

"Project Titan":
The `.com` operators
are spending $100000000
to be ready for a 200Gbps flood.
A worst-case 200Gbps
cryptographic flood
can be handled by a few thousand
PCs running this software.