Implementing TLS with Verified Cryptographic Security

Cédric Fournet

with Karthik Bhargavan, Antoine Delignat-Lavaud, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, Santiago Zanella

Microsoft Research, Cambridge INRIA and MSR-INRIA, Paris

https://www.mitls.org
Transport Layer Security (1994—)

The most widely deployed cryptographic protocol?
HTTPS, 802.1x (EAP), VPN, files, mail, VoIP, ...

20 years of attacks, fixes, and extensions
1994 – Netscape’s Secure Sockets Layer
1995 – SSL3
1999 – TLS1.0 (RFC2246, ≈SSL3)
2006 – TLS1.1 (RFC4346)
2008 – TLS1.2 (RFC5246)

Many implementations
• SChannel, OpenSSL, NSS, GnuTLS, JSSE, PolarSSL, ...
• Several patches every year
• Specific Snowden claims

Many papers
• Well-understood, detailed specs
• Formal security theorems... mostly for small simple models of TLS
What can still possibly go wrong?

- **Protocol Logic**
  - e.g. ambiguous messages
  - cause clients and server to negotiate older, weaker TLS

- **Cryptography**
  - e.g. no fresh IV
  - write applet to realize adaptive attack (BEAST)

- **Implementation Errors**
  - many critical bugs

- **Weak Algorithms**
  - MD5, PKCS1, RC4, ...
Cryptographers Demonstrate New Crack For Common Web Encryption

It's long been known that one of the oldest and most widely used standards for encrypting websites has some serious weaknesses. But one group of researchers has found a method that downgrades that security scheme from vaguely flawed to demonstrably breakable.

At the Fast Software Encryption conference in Singapore earlier this week, University of Illinois at Chicago Professor Dan Bernstein presented a method for breaking Transport Layer Security, (TLS) as well as its predecessor, Secure Sockets Layer or SSL. (Slides here.) Specifically, Bernstein showed serious cracks in TLS and SSL when they're combined with another encryption scheme known as RC4, a system invented in 1980 that remains one of the most widely used in the field.

Weak Algorithms
MD5, PKCS1, RC4, ...
<table>
<thead>
<tr>
<th>Application</th>
<th>Infrastructure</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTPS clients &amp; servers</td>
<td>certificate management</td>
</tr>
</tbody>
</table>

### Protocol Logic
- e.g. ambiguous messages
  - cause clients and server to negotiate older, weaker TLS

### Cryptography
- e.g. no fresh IV
  - write applet to realize adaptive attack (BEAST)

<table>
<thead>
<tr>
<th>TLS DESIGN</th>
</tr>
</thead>
</table>

### Implementation Errors
- many critical bugs

### Weak Algorithms
- MD5, PKCS1, RC4, ...
The Most Dangerous Code in the World: Validating SSL Certificates in Non-Browser Software

Martin Georgiev
The University of Texas
at Austin

Subodh Iyengar
Stanford University

Suman Jana
The University of Texas
at Austin

Rishita Anubhai
Stanford University

Dan Boneh
Stanford University

Vitaly Shmatikov
The University of Texas
at Austin

ABSTRACT

SSL (Secure Sockets Layer) is the de facto standard for secure Internet communications. Security of SSL connections against an active network attacker depends on correctly validating public-key certificates presented when the connection is established.

We demonstrate that SSL certificate validation is completely broken in many security-critical applications and libraries. Vulnerable software includes Amazon’s EC2 Java library and all cloud clients based on it; Amazon’s and PayPal’s merchant SDKs responsible for transmitting payment details from e-commerce sites to payment gateways; integrated shopping carts such as osCommerce, ZenCart, Ubercart, and PrestaShop; AdMob code used by mobile websites; Chase mobile banking and several other Android apps and libraries; Java Web-services middleware—including Apache Axis, Axis 2, Codehaus XFIRE, and Pusher library for Android—and all applications employing this middleware. Any SSL connection from any of these programs is insecure against a man-in-the-middle attack.

The root causes of these vulnerabilities are badly designed APIs of SSL implementations (such as JSSE, OpenSSL, and GnuTLS) and data-transport libraries (such as cURL) which present developers with a confusing array of settings and options. We analyze perils and pitfalls of SSL certificate validation in software based on these APIs and present our recommendations.

SSL implementations in Web browsers are constantly evolving through “penetrate-and-patch” testing, and many SSL-related vulnerabilities in browsers have been repaired over the years. SSL, however, is also widely used in non-browser software whenever secure Internet connections are needed. For example, SSL is used for (1) remotely administering cloud-based virtual infrastructure and sending local data to cloud-based storage, (2) transmitting customers’ payment details from e-commerce servers to payment processors such as PayPal and Amazon, (3) logging instant messenger clients into online services, and (4) authenticating servers to mobile devices.
<table>
<thead>
<tr>
<th>Application</th>
<th>Infrastructure</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTPS clients &amp; servers</td>
<td>certificate management</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Protocol Logic</th>
<th>Cryptography</th>
</tr>
</thead>
<tbody>
<tr>
<td>e.g. ambiguous messages</td>
<td>e.g. no fresh IV</td>
</tr>
<tr>
<td>• cause clients and server to negotiate older, weaker TLS</td>
<td>• write applet to realize adaptive attack (BEAST)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Implementation Errors</th>
<th>Weak Algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>many critical bugs</td>
<td>MD5, PKCS1, RC4, ...</td>
</tr>
</tbody>
</table>

**Combining all of the above**
Recent cryptographic attacks exploit side channels in protocol logic (errors) and implementation (timing)
Combining all of the above
Recent cryptographic attacks exploit side channels in protocol logic (errors) and implementation (timing)
On the (provable) security of TLS

“A handful of works have attempted to analyse the entirety of SSL or TLS using machine-assisted proof techniques.
This is incredibly ambitious, and moreover it's probably the only real way to tackle the problem.
Unfortunately, the proofs hugely simplify the underlying cryptography, and thus don't cover the full range of attacks.
Moreover, only computers can read them.”

• Matthew Green: http://blog.cryptographyengineering.com/
To get application security, we must capture all of these aspects within the same model

- We build a verified reference implementation
- We use formal automated tools to scale up
We develop and verify a **reference implementation** for SSL 3.0—TLS 1.2

1. **Standard compliance**: we closely follow the RFCs
   - concrete message formats
   - support for multiple ciphersuites, sessions and connections, re-handshakes and resumptions, alerts, message fragmentation,…
   - interop with other implementations such as web browsers and servers

2. **Verified security**: we structure our code to enable its modular verification, from its main API down to concrete assumptions on its base cryptography (e.g. RSA)
   - probabilistic computational security theorems for a 5000-line functionality (automation required)

3. **Experimental platform**: for testing corner cases, trying out attacks, studying application-level protocols, analysing new extensions and patches, …
Method:
Type-Based Cryptographic Verification
Models: Formal vs Computational Cryptography

• Two approaches for verifying protocols
  
  **Symbolic models** (Needham-Schroeder, Dolev-Yao, ... late 70’s)
  – Structural view of protocols, using formal languages and methods
  – Many automated verification tools, scales to large systems including full-fledged implementations of protocol standards

  **Computational models** (Yao, Goldwasser, Micali, Rivest, ... early 80’s)
  – Concrete, algorithmic view, using probabilistic polynomial-time machines
  – New wave of formal tools: CryptoVerif, Certicrypt, Easycrypt

• Can we get the best of both worlds? Much ongoing work on computational soundness for symbolic cryptography

• Can we directly verify real-world protocols? Surprisingly, type-based verification can be more effective and more compositional computationally than symbolically.
Modular Type-Based Cryptographic Verification

- symmetric encryption (AES-CBC)
- symmetric encryption (RC4)
- message authentication (SHA1)

Types express cryptographic assumptions

Types express cryptographic constructions

Types express security guarantees

Types express attacker models

Active adversaries

Authenticated encryption

Encrypt then-MAC

Fragment-MAC-encode-then-encrypt

Secure channel

Secure RPC

TLS 1.2

Some attacks

Application code
Basis for Verification: **Refinement types**

A *refinement type* is a base type qualified with a logical formula; the formula can express invariants, preconditions, postconditions, ... 

Refinement types are types of the form $x : T \{ C \}$ where

- $T$ is the base type,
- $x$ refers to the result of the expression, and
- $C$ is a logical formula

The values of this type are the values $M$ of type $T$ such that $C\{M/x\}$ holds.
F7: refinement typechecking for F#

- We program in F#
- We specify in F7
- We verify modules against interfaces: F7 typechecks & calls Z3, an SMT prover, on each logical proof obligation

- We formalized the F7 & F* type systems in Coq/SSReflect
Sample modular verification (protocol)

RPC protocol using Authenticated Encryption

- some cryptographic implementation
- authenticated encryption
- Adversary Model
- active adversaries
- any typed F# program
- Secure RPC
- RPC API
- Formatting
- message format
- any typed F7 program
- application code
- system libraries
- security protocols

Bytes
Networking
Sample modular verification (crypto)

RPC using Encrypt-then-MAC

cryptographic schemes

cryptographic constructions

probabilistic computational indistinguishability

AES-CBC encryption ≈ IDEAL IND-CPA
MAC authentication ≈ IDEAL INT-CMA

Encrypt-then-MAC

authenticated encryption

Secure RPC

RPC API

Adversary Model

system libraries

networking

Bytes

Networking

Formatting

message format

secure protocols

/application code

any typed F7 program

application code

any typed F# program

active adversaries
Modular Architecture for miTLS

Base
- CoreCrypto
- Bytes
- TCP
- TLSConstants
- TLSInfo
- Error
- Range

Handshake/CCS
- Sig
- RSAKey
- DHGroup
- Cert
- CRE
- Nonce
- Extensions
- SessionDB
- Handshake (and CCS)

Alert Protocol
- Alert

AppData Protocol
- Datastream
- AppData

TLS API
- Dispatch
- TLS

TLS Record
- MAC
- Encode
- LHAEPiPlain
- LHAE
- StPlain
- StAE
- TLSFragment
- Record

Application
- AuthPlain
- RPCPlain

Adversary
- Untyped API
- Untyped Adversary

Nonces:
1. Nonce
2. Sig
3. RSA
4. DH
5. CRE
6. PRF
7. StPlain
8. StAE
9. MAC
Modular Architecture for miTLS

Base
- CoreCrypto
- Bytes
- TCP
- TLSConstants
- TLSInfo
- Error
- Range

Handshake/CCS
- Sig
- RSAKey
- DHGroup
- CRE
- PRF
- Cert
- Nonce
- Extensions
- SessionDB
- Handshake (and CCS)

Alert Protocol
- Alert

AppData Protocol
- Datastream
- AppData

TLS API
- Dispatch
- TLS

Application
- AuthPlain
- RPCPlain

Adversary
- Untyped API
- Untyped Adversary

TLS Record
- MAC
- Encode
- LHAEP
- LHAE
- StPlain
- StAE
- TLSFragment
- Record
Sample Typed Interface for Cryptography

Message Authentication Code : integrity
Sample functionality:
Message Authentication Codes

```fsharp
define module MAC

define type text = bytes
define type key = bytes
define type mac = bytes

define val KEYGEN : unit -> key
define val MAC : key -> text -> mac
define val VERIFY : key -> text -> mac -> bool
```

This plain interface says nothing on security
module MAC

type text = bytes

type key

type mac = bytes

val KEYGEN: unit -> key
val MAC : key -> text -> mac
val VERIFY: key -> text -> mac -> bool

MAC keys are abstract
Module `MAC`

type `text` = bytes

type `key` = bytes

type `mac` = bytes

Predicate `Msg` of `key` * `text`

Val `KEYGEN`: `unit` -> `key`

Val `MAC`: `key` -> `text`{`Msg(k,t)`} -> `mac`

Val `VERIFY`: `key` -> `text` -> `mac` -> `bool`{ `b=true` ⇒ `Msg(k,t)` }

**Ideal F7 Interface**

“All verified messages have been MACed”

**Sample functionality:**

Message Authentication Codes

MAC keys are abstract

`Msg` is specified by protocols using MACs

Precondition of MAC creation

Postcondition of MAC verify
Sample functionality:

Message Authentication Codes

```fsharp
module MAC

open System.Security.Cryptography

let macsize = 20

let KEYGEN() = randomBytes 16

let MAC k t = (new HASHMACSHA1(k)).ComputeHash t

let VERIFY k t m = (MAC k t = m)
```

**Message Authentication Codes**

**module** MAC

type text = bytes

type key = bytes

type mac = bytes

**predicate** Msg of key * text

**val** KEYGEN: unit -> key

**val** MAC : k:key -> t:text{Msg(k,t)} -> mac

**val** VERIFY: k:key -> t:text -> mac

-> b:bool{ b=true ⇒ Msg(k,t)}

**module** MAC

open System.Security.Cryptography

let macsize = 20

let KEYGEN()= randomBytes 16

let MAC k t = (new HASHMACSHA1(k)).ComputeHash t

let VERIFY k t m = (MAC k t = m)

**MAC keys are abstract**

** Msg is specified by protocols using MACs**

**ideal F7 interface**

**precondition of MAC creation**

**postcondition of MAC verify**

"All verified messages have been MACed"

This can’t be true! (collisions)

concrete F# implementation (using .NET)
Sample computational assumption:

Resistance to Chosen-Message Existential Forgery Attacks (INT-CMA)

```fsharp
module Game
open Mac

let private k = KEYGEN()
let private log = ref []

let mac t =
    log := t::!log
    MAC k t

let verify t m =
    let v = VERIFY k t m in
    if v && not (mem t !log) then FORGERY
    v
```

**Computational Safety**
a probabilistic polytime program calling `mac` and `verify` forges a MAC only with negligible probability $\epsilon$
Computational Safety for MACs

ideal system

- Mac
  - Ideal filter
    - Ideal MAC
- RPC protocol
  - secure RPC
- Any p.p.t. adversary

perfectly safe by typing

concrete system

- Mac
  - F# interface
- RPC protocol
- Any p.p.t. adversary

concrete algorithm assumed INT-CMA computationally

error correction making VERIFY returns false on forgeries

safe too, with probability $\geq 1 - \epsilon$
Sample Typed Interface for Cryptography

encryption : secrecy
Perfect Secrecy by Typing

- Secrecy is expressed using observational equivalences between systems that differ on their secrets.
- We prove (probabilistic, information theoretic) secrecy by typing, relying on type abstraction.

\[ I_\alpha = \alpha, \ldots, x : T_\alpha, \ldots \]

\( P_\alpha \) range over pure modules such that \( \vdash P_\alpha \sim I_\alpha \).

**Theorem** (Secrecy by Typing).
Let \( A \) such that \( I_\alpha \vdash A : bool \).
For all \( P_\alpha^0 \) and \( P_\alpha^1 \), we have \( P_\alpha^0 \cdot A \approx P_\alpha^1 \cdot A \).
Encryption is parameterized by a module that abstractly define plaintexts, with interface

```plaintext
module Plaintext

val size: int
type plain
type repr = b:bytes{Length(b)=size}

val coerce : repr -> plain // turning bytes into secrets
val leak : plain -> repr // breaking secrecy!

val respond: plain -> plain // sample protocol code
```

If we remove the `leak` function, we get secrecy by typing

If we remove the `coerce` function, we get integrity by typing

`Plain` may also implement any protocol functions that operates on secrets
Towards TLS: adding **Type Indexes**

- Within TLS, we keep track of many keys, for different algorithms & sessions
- We use finer ideal functionalities that provide *conditional security* only for “good” keys
  - generated by algorithms assumed *computationally strong*; and
  - for sessions between **honest** participants (not those with the adversary)
Verifying our TLS codebase
Transport Layer Security (Review)

- Interleaving of four protocols on top of the record layer

- Four protocols:
  - Handshake protocol
  - Change ciphersuite protocol
  - Alert protocol
  - Application data protocol

- Handshake protocol:
  - Fresh keys for each new epoch

- Record Layer:
  - Fragment; compress
  - Stateful authenticated encryption
  - Authenticated encryption with additional data

- I/O bytestreams:
  - Application
  - Web pages

- TCP/IP:
  - Plain fragments
  - Encrypted fragments
Ciphersuites & crypto agility

- Not all algorithms are equal!

- Cautionary tale [RSA 2012]
  - ECDH part of the latest strongest ciphersuites using elliptic curves
  - OpenSSL vulnerable to a bug in its multiplication algorithm (attack takes 635 handshakes) widely deployed & left open for 2 years

- Intuitively, clients and servers should get the security for the ciphersuites they use, not for the weakest supported ones
  - Non trivial: there is a circular dependency, as TLS relies on the ciphersuites being negotiated

- We verify TLS generically, for multiple ciphersuites: formally, the adversary provides the weaker algorithms.

<table>
<thead>
<tr>
<th>Ciphersuites</th>
<th>Hexadecimal Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS_NULL_WITH_NULL_NULL</td>
<td>0x00,0x00</td>
</tr>
<tr>
<td>TLS_RSA_WITH_NULL_SHA256</td>
<td>0x00,0x3B</td>
</tr>
<tr>
<td>TLS_RSA_WITH_RC4_128_MD5</td>
<td>0x00,0x04</td>
</tr>
<tr>
<td>TLS_DH_anon_WITH_AES_256_CBC_SHA</td>
<td>0x00,0x3A</td>
</tr>
</tbody>
</table>

(...): 38 ciphersuites in TLS 1.2
(...): many others in recent TLS extensions

Practical Realisation and Elimination of an ECC-Related Software Bug Attack

Billy B. Brumley¹, Manuel Barbosa², Dan Page³, and Frederik Vercauteren⁴
Authenticated Encryption for fragment streams with additional data
Fragment; MAC; Encode; then Encrypt

sent earlier

plaintext message sent by the application

fragment

to be sent later

fragmenting & padding are under-specified

content type & sequence number

content type & sequence number

header

sent/received on TCP connection

- TLS decodes the decrypted text before authentication; potentially leaking secret information (via “padding oracles”)
- Security relies on joint ciphertext integrity (INT-CTXT)
The proof is ad hoc (for CBC) and depends on |MAC| > |Block|
(recent attack & proof by Paterson et al. at ASIACRYPT’11)
Large messages are sliced into many fragments.
When encoded, each fragment is independently compressed.
An eavesdropper can record the sequence of fragment ciphertext lengths, and obtain precise message fingerprints – leaking much more than the total message length.

Bar charts showing:
- Max fragment length (16KB) in blue.
- Lengths observed on the network in red.
Fragment-then-Compress??

- Experimental data: downloading songs over HTTPS:
Our approach: disable compression, then

**Hide secret lengths within public ranges**

- The application chooses its own plaintext range, e.g. any secret URL of size 0..200 bytes

Formally, we index our type of plaintext fragments by their range & sequence number in the stream too. By typing, we check that

- Fragmentation and padding depends *only on the range & ciphersuite*, not on the secret message length & content
Main TLS API
The TLS API & ideal functionality

- Our API is similar but more informative than mainstream APIs
  - We run on the caller’s thread, letting the application do the scheduling & multiplexing
  - We give *more control* to the application code, and reflect *more information* from the underlying TLS state (lengths, fragmentation, authorization queries)
    - More precise security theorems
    - More flexibility for experiments & testing

- We can implement safe & simple APIs on top of it

- Sample applications using our API
  - Secure RPCs (with one connection per call)
  - Password-based client authentication
  - Basic HTTPS clients and servers (for interoperability testing)
Each application provides its own plaintext module for data streams:

- Typing ensures secrecy and authenticity at safe indexes
- Parameters select ciphersuites and certificates
- Results provide detailed information on the protocol state

```scala
// for each local instance of the protocol

import (
  cn // client and server instances
)

val connect: TcpStream -> params -> (Client) nullCn Result
val accept: TcpStream -> params -> (Server) nullCn Result

// triggering new handshakes, and closing connections
val rehandshake: c:cn{Role(c)=Client} -> cn Result
val request: c:cn{Role(c)=Server} -> cn Result
val shutdown: c:cn -> TcpStream Result

// writing data

type (; c:cn, data:(; c) msg_o) ioreresult_o =
| WriteComplete of c':cn
| WritePartial of c':cn * rest:(; c') msg_o
| MustRead of c':cn
val write: c:cn -> data:(; c) msg_o -> (; c, data) ioreresult_o

// reading data

type (; c:cn) ioreresult_i =
| Read of c':cn * data:(; c) msg_i
| CertQuery of c':cn
| Handshake of c':cn
| Close of TcpStream
| Warning of c':cn * a:alertDescription
| Fatal of a:alertDescription
val read : c:cn -> (; c) ioreresult_i
```
Main result: concrete TLS and ideal TLS are indistinguishable

Our typed API for TLS then yields application security by typing

<table>
<thead>
<tr>
<th>Component</th>
<th>F# (LOC)</th>
<th>F7 (LOC)</th>
<th>F7 (S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>945</td>
<td>581</td>
<td>11</td>
</tr>
<tr>
<td>TLS Record</td>
<td>826</td>
<td>511</td>
<td>77</td>
</tr>
<tr>
<td>Handshake/CCS</td>
<td>2400</td>
<td>777</td>
<td>413</td>
</tr>
<tr>
<td>Alert Protocol</td>
<td>184</td>
<td>119</td>
<td>105</td>
</tr>
<tr>
<td>AppData Protocol</td>
<td>139</td>
<td>113</td>
<td>34</td>
</tr>
<tr>
<td>TLS API</td>
<td>640</td>
<td>426</td>
<td>309</td>
</tr>
<tr>
<td>Total</td>
<td>5134</td>
<td>2527</td>
<td>949</td>
</tr>
</tbody>
</table>
Main result: concrete TLS and ideal TLS are indistinguishable

Our typed API for TLS then yields application security by typing

<table>
<thead>
<tr>
<th>Component</th>
<th>F# (LOC)</th>
<th>F7 (LOC)</th>
<th>F7 (#)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>945</td>
<td>581</td>
<td>11</td>
</tr>
<tr>
<td>TLS Record</td>
<td>826</td>
<td>511</td>
<td>77</td>
</tr>
<tr>
<td>Handshake/CCS</td>
<td>2400</td>
<td>777</td>
<td>413</td>
</tr>
<tr>
<td>Alert Protocol</td>
<td>184</td>
<td>119</td>
<td>105</td>
</tr>
<tr>
<td>AppData Protocol</td>
<td>139</td>
<td>113</td>
<td>34</td>
</tr>
<tr>
<td>TLS API</td>
<td>640</td>
<td>426</td>
<td>309</td>
</tr>
<tr>
<td>Total</td>
<td>5134</td>
<td>2527</td>
<td>949</td>
</tr>
</tbody>
</table>
Security for RPC over TLS

- Cryptographic Provider
- our verified modular TLS implementation
- any typed F# program
- active adversaries
- application code

RPC

Bytes, Network

any typed F# program

RPC Payload

any typed F# program

Bytes, Network lib.fs

RPC plain.fs

Cryptographic Provider

cryptographic assumptions

TLS.fs7

any typed F# program

RpcAdv.fs

any typed F# program

TLS.fs7
Interoperability & Performance

We run clients against an OpenSSL 1.0.1e server for various ciphersuites

- How many handshakes per second?
- How much data transferred per second?

<table>
<thead>
<tr>
<th>KEX</th>
<th>Ciphersuite</th>
<th>F# (BC)</th>
<th>OpenSSL</th>
<th>Oracle</th>
<th>JSSE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>HS/s</td>
<td>MiB/s</td>
<td>HS/s</td>
<td>MiB/s</td>
</tr>
<tr>
<td>RSA</td>
<td>RC4</td>
<td>305.25</td>
<td>30.17</td>
<td>292.04</td>
<td>226.51</td>
</tr>
<tr>
<td>RSA</td>
<td>RC4</td>
<td>291.37</td>
<td>27.85</td>
<td>288.74</td>
<td>232.42</td>
</tr>
<tr>
<td>RSA</td>
<td>3DES</td>
<td>267.09</td>
<td>8.40</td>
<td>283.04</td>
<td>22.95</td>
</tr>
<tr>
<td>RSA</td>
<td>AES128</td>
<td>278.71</td>
<td>18.54</td>
<td>285.35</td>
<td>234.41</td>
</tr>
<tr>
<td>RSA</td>
<td>AES128</td>
<td>278.71</td>
<td>16.50</td>
<td>281.92</td>
<td>128.33</td>
</tr>
<tr>
<td>RSA</td>
<td>AES256</td>
<td>291.37</td>
<td>16.86</td>
<td>282.89</td>
<td>204.47</td>
</tr>
<tr>
<td>RSA</td>
<td>AES256</td>
<td>267.09</td>
<td>15.16</td>
<td>307.72</td>
<td>119.42</td>
</tr>
<tr>
<td>DHE</td>
<td>3DES</td>
<td>20.16</td>
<td>8.37</td>
<td>58.07</td>
<td>22.99</td>
</tr>
<tr>
<td>DHE</td>
<td>AES128</td>
<td>20.41</td>
<td>18.59</td>
<td>57.06</td>
<td>244.30</td>
</tr>
<tr>
<td>DHE</td>
<td>AES128</td>
<td>19.99</td>
<td>16.45</td>
<td>58.33</td>
<td>128.34</td>
</tr>
<tr>
<td>DHE</td>
<td>AES256</td>
<td>20.29</td>
<td>16.72</td>
<td>56.83</td>
<td>203.01</td>
</tr>
<tr>
<td>DHE</td>
<td>AES256</td>
<td>20.16</td>
<td>14.86</td>
<td>59.52</td>
<td>120.96</td>
</tr>
</tbody>
</table>
Implementing TLS with Verified Cryptographic Security

Our ideal API provides strong, modular, usable, type-based, \textbf{conditional} application security. We trust

- **automated typechecking**: F7 and Z3
  - Now: mechanized type theory
  - Next: certified F* typechecker [POPL’12] and SMT solver
- **cryptographic assumptions**, with handwritten proofs
  - New: better concrete reductions & mechanized proofs (CertiCrypt)
- **the F\# compiler and runtime**: Windows and .NET
- **core cryptographic providers**
  - Next: functional correctness for selected algorithms (elliptic curves)

We account for some side-channels, but not for timing analysis