A Trusted Cloud

A Cryptographic and a Language Perspective
Outline

• The Network is the attacker
• The Memory is the attacker

• The Server is the attacker
  • Confidentiality of distributed data and computation
  • Authenticity of distributed data and computation

Everyone could be the attacker, but not everyone at the same time to everyone, allegiances shift.
Part I

• The Network is the attacker
  • TLS/SSL verified

• The Memory is the attacker
  • Javascript
Classic view on Network Security
Implementing TLS with Verified Cryptographic Security

Karthikeyan Bhargavan*, Cédric Fournet†, Markulf Kohlweiss†, Alfredo Pironti*, Pierre-Yves Strub‡

*INRIA Paris-Rocquencourt, {karthikeyan.bhargavan,alfredo.pironti}@inria.fr
†Microsoft Research, {fournet,markulf}@microsoft.com
‡IMDEA Software, pierre-yves@strub.nu

http://mitls.rocq.inria.fr
Transport Layer Security (1994—)

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

18 years of attacks, fixes, and extensions
1994 – Netscape’s Secure Sockets Layer (SSL2)
1995 – SSL3
1999 – TLS1.0 (RFC2246, ≈SSL3)
2006 – TLS1.1 (RFC4346)
2008 – TLS1.2 (RFC5246)

Many implementations
• SChannel, OpenSSL, NSS, GnuTLS, JSSE, PolarSSL, ...
• Security patches every year, NSA Bullrun claims.

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

<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>• Attacker can use signature signing DH parameters for Elliptic Curve DH.</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>
It’s long been known that one of the oldest and most widely used standards for encrypting web sites 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 1987 that remains one of the most popular and most widely recommended mechanisms for protecting traffic on banking, email, and other private sites.

“A bunch of us have been sitting in the background scratching our heads, knowing that RC4 is weak in all kinds of ways,” says Kenny Paterson, a computer scientist at the University of Edinburgh. Bernstein and his colleagues have, for the most part, “cracked it.”
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)

<table>
<thead>
<tr>
<th>Application</th>
<th>Infrastructure</th>
</tr>
</thead>
<tbody>
<tr>
<td>protocol configuration</td>
<td>certificate management</td>
</tr>
</tbody>
</table>

**TLS DESIGN**

**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
Rishita Anubhai
Stanford University
Subodh Iyengar
Stanford University
Suman Jana
The University of Texas at Austin
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 PrestoShop; 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.

cations. The main purpose of SSL is to provide end-to-end security against an active, man-in-the-middle attacker. Even if the network is completely compromised—DNS is poisoned, access points and routers are controlled by the adversary, etc.—SSL is intended to guarantee confidentiality, authenticity, and integrity for communications between the client and the server.

Authentication of the server is a critical part of SSL connection establishment. This authentication takes place during the SSL handshake, when the server presents its public-key certificate. In order for the SSL connection to be secure, the client must carefully verify that the certificate has been issued by a valid certificate authority, has not expired (or been revoked), the name(s) listed in the certificate match(s) the name of the domain that the client is connecting to, and perform several other checks [14, 15].

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 users to mobile
<table>
<thead>
<tr>
<th>Application</th>
<th>Infrastructure</th>
</tr>
</thead>
<tbody>
<tr>
<td>protocol configuration</td>
<td>certificate management</td>
</tr>
</tbody>
</table>

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

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

**Implementation Errors**
- many critical bugs

**Weak Algorithms**
- MD5, PKCS1, RC4, ...

**Combining all of the above**
Recent cryptographic attacks exploit side channels in protocol logic (errors) and implementation (timing)
To get application security, we need to capture all these aspects within a precise model of TLS
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.**”

We develop and verify **mi-TLS**: 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)
   - formal computational security theorems for a 5000-line functionality (automation required)

3. **Experimental platform**: for testing corner cases, trying out attacks, analysing new extensions and patches, …
We develop and verify **mi-TLS**: a reference implementation for SSL 3.0—TLS 1.2

(Demo)

http://mitls.rocq.inria.fr
## Interoperability & Performance

<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>
Verification method [ACM CCS’11]
Modular Type-Based Cryptography

1. We program in F#
2. We specify in F7,
   using abstract type, refinements, ...
3. We verify modules against interfaces:
   F7 typechecks, calls Z3
   on each proof obligation

- We plan to upgrade to RF*
- We formalized the F7, F*, RF* type systems in Coq/SSReflect
Main result: concrete TLS & ideal TLS are computationally indistinguishable

Our TLS typed API thus yields application security by typechecking

<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>2,400</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></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>5,134</strong></td>
<td><strong>2,527</strong></td>
<td><strong>949</strong></td>
</tr>
</tbody>
</table>
Classic view on Network Security

Memory is the attacker

Network is the attacker
JavaScript Security

Gradual Typing Embedded Securely in JavaScript

N. Swamy  C. Fournet  A. Rastogi  K. Bhargavan  J. Chen  P.-Y Strub  G. Bierman
MSR  INRIA  IMDEA

http://research.Microsoft.com/fstar
http://rise4fun.com/FStar/tutorial/tsStar
Writing Security Critical JavaScript Code is Hard!
JavaScript Security

Web pages mix various code & data in the same runtime environment

• our scripts, using capabilities, tokens... with safety & security goals
• others, of dubious origins & intents

Type error
≈ security exploit

They found new attacks in:
• Facebook API
• OWASP anti-CSRF library (Cross Site Request Forgery)
• MEGA client-side password
How to protect good scripts from bad ones?

1. good = compiled from F* [POPL’13]
   • heap separation & dynamic wrappers
   • full abstraction
   • not many F* web developers
   • hard to interact with the DOM

2. good = in a safe subset of JavaScript [TS*]
   • remove offensive JS features
   • re-use TypeScript syntax — but we do need type safety!
   • enforce gradual typing, relying on compile-time & run-time mechanisms
   • secure Web APIs by rewriting/re-compiling their security-critical scripts.
Part II

• The Server is the attacker
  • Confidentiality of distributed data and computation
    • Oblivious RAM, Zero Knowledge Proofs, Multi-party Computation
  • Authenticity of distributed data and computation
    • Verifiable Computation

Everyone could be the attacker, but not everyone at the same time to everyone, allegiances shift.
Classic view on Network Security
Snowden leaks could cost US IT industry 22 billion-35 billion in lost revenues between now and 2016

[Information Technology and Innovation Foundation]
Computational services get centralized

Trust remains inherently distributed
We shouldn’t expect customers just to trust our cloud.

Users & services want to share information, with their own privacy & integrity requirements.
The big picture

David Bernhard
Do x-rays cause cancer?
Today:
early experiments

safety concerns

privacy concerns

secure?
Storage Security
Storage Security: Restrictions

• Encryption is not enough.
  • (Always use authenticated encryption)
  • Access patterns reveal a lot of information. [Pinkas & Reinman, Islam et. al] (80% of search queries in encrypted email database revealed)

• Server cannot do anything useful with the data.
  • Only party storing the data can retrieve it.

• How can other parties retrieve the data?
  • Functional encryption (See previous talk.)
  • Private information retrieval.
  • More general computation on encoded data.
Read-Write Patterns

Buys IBM stock

Buys MSFT stock
A dumb solution
The smart solution (Oblivious RAM)

Server uses smart data structure

read and write patterns look random

Small amount of client storage (Stash)
The smart solution (Oblivious RAM)

- **read and write** patterns look random
- Server uses smart data structure
- Small amount of client storage (Stash)
The smart solution (Oblivious RAM)

- **read and write** patterns look random
- Small amount of client storage (Stash)

Server uses smart data structure
Path ORAM: An Extremely Simple Oblivious RAM Protocol

Emil Stefanov†, Marten van Dijk‡, Elaine Shi*, Christopher Fletcher○, Ling Ren○, Xiangyao Yu○, Srinivas Devadas○

† UC Berkeley  ‡ UConn  · UMD  ○ MIT CSAIL
Path ORAM

1. Every data element a, b, c, d, e, f assigned random leafs 1, 2, 3, 4.
2. Read all blocks along path to leave.
3. Return block on a read or update block on a write.
4. Replace leaf with a new random leaf.
5. Evict as many blocks as possible from the stash.
Path ORAM (read b)

1. Every data element a, b, c, d, e, f assigned random leafs 1, 2, 3, 4.
2. Read all blocks along path to leave.
3. Return block on a read or update block on a write.
4. Replace leaf with a new random leaf.
5. Evict as many blocks as possible from the stash.
Path ORAM (read b)

1. Every data element a, b, c, d, e, f assigned random leaves 1, 2, 3, 4.
2. Read all blocks along path to leave.
3. Return block on a read or update block on a write.
4. Replace leaf with a new random leaf.
5. Evict as many blocks as possible from the stash.
Path ORAM (read b)

1. Every data element a,b,c,d,e,f assigned random leafs 1,2,3,4.
2. Read all blocks along path to leave.
3. Return block on a read or update block on a write.
4. Replace leaf with a new random leaf.
5. Evict as many blocks as possible from the stash.
Path ORAM (read b)

1. Every data element a,b,c,d,e,f assigned random leafs 1,2,3,4.
2. Read all blocks along path to leave.
3. Return block on a read or update block on a write.
4. Replace leaf with a new random leaf.
5. Evict as many blocks as possible from the stash.
Path ORAM (write b)

1. Every data element a,b,c,d,e,f assigned random leafs 1,2,3,4.
2. Read all blocks along path to leave.
3. Return block on a read or update block on a write.
4. Replace leaf with a new random leaf.
5. Evict as many blocks as possible from the stash.
Path ORAM (write b)

1. Every data element a,b,c,d,e,f assigned random leafs 1,2,3,4.
2. Read all blocks along path to leave.
3. Return block on a read or update block on a write.
4. Replace leaf with a new random leaf.
5. Evict as many blocks as possible from the stash.
Path ORAM (write b)

1. Every data element a, b, c, d, e, f assigned random leaves 1, 2, 3, 4.
2. Read all blocks along path to leave.
3. Return block on a read or update block on a write.
4. Replace leaf with a new random leaf.
5. Evict as many blocks as possible from the stash.
Path ORAM (write b)

1. Every data element a,b,c,d,e,f assigned random leafs 1,2,3,4.
2. Read all blocks along path to leave.
3. Return block on a read or update block on a write.
4. Replace leaf with a new random leaf.
5. Evict as many blocks as possible from the stash.
Storage Security: Restrictions

• Encryption is not enough.
  • (Always use authenticated encryption)
  • Access patterns reveal a lot of information. [Pinkas & Reinman, Islam et. al] (80% of search queries in encrypted email database)

• Server cannot do anything useful with the data.
  • Only party storing the data can retrieve it.

• How can other parties retrieve the data?
  • Functional encryption (See previous talk.)
  • Private information retrieval.
  • More general computation on encoded data.
Storage Security
Trust but distribute and verify
Bikelock App
key : 0120
key: 0120

3281 (random)

7390 (random)

9449 = 0120 - 3281 - 7390
independent of secret:
still $10^4$ trials
Survey App

Average lunch satisfaction 9/3=3
Smart Meter Aggregation

Smart Meter Aggregation via Secret-Sharing

George Danezis, Cédric Fournet, Markulf Kohlweiss, Santiago Zanella-Béguelin
Microsoft Research, Cambridge, UK
{gdane,fournet,markulf,santiago}@microsoft.com
Motivating problem: smart-meter statistics

- Trust is centralized
- Queries are highly sensitive
Motivating problem: smart-meter statistics

Use secret shares that add up to zero

$$\sum_j s_{i,j} = 0$$

$$b_i = r_i + s_{i,0}$$

$$t_j = \sum_i w_i s_{i,j}$$

$$\overrightarrow{w} = \sum_i w_i b_i + \sum_j t_j$$
\[ r_1 + s_{1,0} \]
\[ r_2 + s_{2,0} \]
\[ r_3 + s_{3,0} \]

\[ w_1 \ s_{1,1} \]
\[ w_2 \ s_{2,1} \]
\[ w_3 \ s_{3,1} \]

\[ w_i \sum_i (r_i + s_{i,0}) \sum_i w_is_{i,1} \]

\[ \sum_i w_is_{i,2} = \sum_i w_ir_i \]
Beyond linear computations

• We know how to compute linear aggregates efficiently and privately
• We need a way to compute products
• Can do this with some pre-shared material, triplets of the form 
  \[(a, b, c)\] where \(a\) and \(b\) are random and \(c = a \cdot b\)

• Each multiplication consumes a triplet
• To compute \(x \cdot y\), reveal \(\epsilon = x - a\) and \(\delta = y - b\)

\[
x \cdot y = c + \epsilon b + \delta a + \epsilon \delta \\
= c + (x - a)b + (y - b)a + (x - a)(y - b) \\
= c + xb - ab + ya - ab + xy - xb - ya + ab \\
= \epsilon + xb - ab + ya - ab + xy - xb - ya + ab
\]
Integrity

Integrity can be protected using shared information theoretic Message Authentication Codes (MACs)

• Share a single MAC key $\alpha$ and the MAC $\gamma(x) = \alpha x$
• Perform two computations in parallel, one on values other on MACs
• Chicken-and-egg problem: to check a result, need to reveal $\alpha$ to check the MAC. But revealing $\alpha$ allows to forge MACs!
• Smart Solution: participants commit to result before $\alpha$ is revealed
• Smarter Solution: only reveal that $x \cdot \alpha - \gamma(x) = 0$
The SPDZ Protocol Family

Multiparty Computation from Somewhat Homomorphic Encryption

Ivan Damgård¹, Valerio Pastro¹, Nigel Smart², and Sarah Zakarias¹

Practical Covertly Secure MPC for Dishonest Majority – or: Breaking the SPDZ Limits

Ivan Damgård¹, Marcel Keller², Enrique Larraia², Valerio Pastro¹, Peter Scholl², and Nigel P. Smart²

¹ Department of Computer Science, Aarhus University
² Department of Computer Science, University of Bristol

An architecture for practical actively secure MPC with dishonest majority

Marcel Keller, Peter Scholl, and Nigel P. Smart

Department of Computer Science, University of Bristol, UK,
{m.keller, Peter.Scholl}@bristol.ac.uk, nigel@cs.bris.ac.uk
A heterogeneous cloud
Client verifies Cloud computation

Your medicine has no side effects

```c
void main(int argc, char **argv) {
    sleep(10000);
    return "No";
}
```
Pinocchio

Pinocchio: Nearly Practical Verifiable Computation

Bryan Parno
Jon Howell
Microsoft Research

Craig Gentry
Mariana Raykova
IBM Research
Performance

Verifier Latency

Worker Latency

100x100 matrix mult.

Cost fell 23 orders of magnitude in 6 years

15 ms

≈ 12 minutes
Verifiable Computation

[GGP – CRYPTO 2010]

GenKeys(F) → EK_F, VK_F

void main(int argc, char **argv) {
    ...
    x = b[i] + d[j];
    y *= x;
    ...
    return y;
}

y) → \Pi_{y=F(x)}

Key Constraint
Alice’s efforts must be less than computing F
Verifiable Computation

[GGP – CRYPTO 2010]

\[
\text{GenKeys}(F) \rightarrow \text{EK}_F, \text{VK}_F
\]

\[
F(x) \rightarrow y, \prod_{y=F(x)}
\]

\[
\text{Prove}(\text{EK}_F, x, y) \rightarrow \prod_{y=F(x)}
\]

\[
\text{Verify}(\text{VK}_F, x, y, \prod_{y=F(x)}) \rightarrow \{\text{Yes, No}\}
\]

Key Constraint
Alice’s efforts must be \textit{less than} computing F
void main() {
    ...
    x = b[i] + d[j];
    y *= x;
    ...
    return y;
}
Compiling C to Circuits

- Compiler understands a subset of C
- Outputs an *arithmetic* circuit with wire values $C_i \in \mathbb{F}_p$

\[
C_1 + C_2 \mod p \\
C_3 \cdot C_4 \mod p \\
C_5 == 0 ? 1 : 0 \\
X = 01011
\]
Compiling C to Circuits

• Compiler understands a subset of C
• Outputs an arithmetic circuit with wire values $C_i \in \mathbb{F}_p$

```
C_1 + C_2 \mod p
```

```
C_1 + C_2 \mod p
```

```
C_3 *
```

```
C_1 + C_2 \mod p
```

```
C_3 *
```

<table>
<thead>
<tr>
<th>Boolean</th>
<th>Arithmetic</th>
</tr>
</thead>
<tbody>
<tr>
<td>And</td>
<td>$A \cdot B$</td>
</tr>
<tr>
<td>Or</td>
<td>$(A + B) - A \cdot B$</td>
</tr>
<tr>
<td>Not</td>
<td>$1 - A$</td>
</tr>
<tr>
<td>Xor</td>
<td>$(A + B) - 2 A \cdot B$</td>
</tr>
</tbody>
</table>
void main(){
    ...
    x = b[i] + d[j];
    y *= x;
    ...
    return y;
}

Pinocchio’s Verification Pipeline

Compile

Compile

Compile

EK_F, VK_F ← GenKeys(F)

Π_y ← Prove(EK_F, x, y)

{Yes, No} ← Verify(VK_F, x, y, Π_y)
Quadratic Arithmetic Program Intuition

\[ V \cdot c \ast W \cdot c - Y \cdot c = 0 \]

\[ v_j(i) = V[i, j] \text{ same for } W \text{ and } Y \]

\[ \forall i \sum c_j v_j(i) \ast \sum c_j w_j(i) - \sum c_j w_j(i) = 0 \]

Check that divisible by \( \prod_i (z - i) \)

\[ (0,0,1,0,0,0) \ast (0,0,0,1,0,0) \ast (0,0,0,0,1,0) = 0 \]

\[ (1,1,0,0,0,0) \ast (0,0,0,0,1,0) \ast (0,0,0,0,0,6) = 0 \]
Quadratic Arithmetic Program Intuition

\[ V \cdot c \ast W \cdot c - Y \cdot c = 0 \]
\[ \nu_j(i) = V[i, j] \text{ same for } W \text{ and } Y \]
\[ \forall i \sum c_j \nu_j(i) \ast \sum c_j w_j(i) - \sum c_j w_j(i) = 0 \]
Check that divisible by \( \prod_i (z - i) \)

\[ \exists h, \text{ s.t. for random } s, \]
\[ h(s) \ast \prod_i (s - i) = \sum c_j \nu_j(s) \ast \sum c_j w_j(s) - \sum c_j w_j(s) \]

\[ (C_1 + C_2) \ast C_5 = C_6 \]
\[ (1,1,0,0,0) \ast (0,0,0,1,0) - (0,0,0,0,6) = 0 \]

Pinocchio’s Verification Pipeline

void main()
{
  ...
  x = b[i] + d[j];
  y *= x;
  ...
  return y;
}

Quadratic Program

$\Sigma_{1}(z) \ldots \Sigma_{m}(z)$
$\Pi_{1}(z) \ldots \Pi_{m}(z)$

$\Pi_{y} \leftarrow \text{Prove}(\text{EK}_F, x, y)$

$\{\text{Yes, No}\} \leftarrow \text{Verify}(\text{VK}_F, x, y, \Pi_y)$

$\text{EK}_F, \text{VK}_F \leftarrow \text{GenKeys}(F)$
Cryptographic Protocol (simplified)

**GenKeys(F)** → **EK_F, VK_F**
Generate the QAP for F
Pick random s
Compute **EK_F** = \{g_v^1(s), ..., g_v^m(s),
                          g_w^1(s), ..., g_w^m(s),
                          g_y^1(s), ..., g_y^m(s), g_s^i\}\

Compute **VK_F** = \{g_d(s) = g \Pi_i (s-i)\}

**Verify(VK_F, x, y, \Pi_y)** → {Yes, No}
Check: \(e(g^v(s), g^w(s))/e(g^y(s), g) =? e(g^h(s), g^d(s))\)

**Prove(EK_F, x, y)** → \(\Pi_y\)
Evaluate circuit. Get wire values \(c_1, ..., c_m\)
Compute:
- \(g^v(s) = \Pi (g^v_{-i}(s)c_i)\)
- \(g^w(s) = \Pi (g^w_{-i}(s)c_i)\)
- \(g^y(s) = \Pi (g^y_{-i}(s)c_i)\)

Find \(h(z)\) s.t. \(h(z) \cdot d(z) = v(z) \cdot w(z) - y(z)\)
Compute \(g^h(s) = \Pi (g^{s^i}h_i)\)
Proof is \((g^v(s), g^w(s), g^y(s), g^h(s))\)
\(e(\cdot, \cdot)\) is a pairing:
\(e(g^a, g^b) = e(g, g)^{ab}\)
Cloud verifies that Clients possesses credentials

I have the right access rights
The public key identifies the account. The signing key controls the account. Everyone checks the transaction before appending it to the ledger:
1. Check the signature
2. Check Kb “owns” $72.49 as the result of prior transactions
Zerocoin (Review)

- Still relying on a global ledger but with strong privacy guaranteed
- Private-coin transactions are **unlinkable**
- Transactions must remain **small & fast to verify**

The commitment **identifies** the account. The serial number & opening **control** the account. Everyone checks the transaction before appending it to the ledger:
1. check the zero-knowledge proof
2. check s does not appear earlier in the log (to prevent double spending)

Transactions now record large, random, pairwise-distinct values: $s$ and $C = g^s h^o$ are independent and prove knowledge of $o$ such that $C = g^s h^o$ appears in the log.
We mix

- C code for the core proof, using Pinocchio
- F# code for the rest of the protocol, a la ZQL

We experiment with extension-field commitments & with SHA1 hashes

We can (in principle) program advanced multiparty transactions, say privately pay some V.A.T.

Pinocchio (a long while)

arithmetic circuit (~300,000 gates) & quadratic arithmetic program
High-level security programming

The computation is written as a single global program, executed by a (fictional) trusted third party

PL tools usefully apply: types, static analyses, dynamic tracking...

This program deals with data at different levels of integrity and confidentiality:

- When to endorse untrusted information?
- When to release confidential information? How much is leaked?

I spend a coin with this serial number

Compile
Summer of Love & Privacy