Isogeny-Based Cryptography on Mobile Devices

Dieter Fishbein $^1$
joint work with D. Jao$^1$

$^1$University of Waterloo

September 27$^{th}$, 2013, ETSI Quantum-Safe-Crypto Workshop
Elliptic curves

As long as we are concerned in this talk, *elliptic curves* are

- Algebraic **groups** defined over a (finite) field.
- Their group law is easy to compute (say, in constant time).
- Any curve $E$ is (almost) uniquely defined by its *$j$-invariant* $j(E)$ up to isomorphism (just a change of coordinates).

\[ E : y^2 = x^3 + ax + b \quad a, b \in k \]

\[ j(E) = 1728 \frac{4a^3}{4a^3 + 27b^2} \]
Isogenies

*Isogenies are morphisms* of elliptic curves (we only deal with elliptic curves in this talk)

- Surjective group morphism
- Algebraic map (i.e. defined by polynomials)

\[ \phi : E \rightarrow E' \]

The kernel \( H \) determines the image curve \( E' \) up to isomorphism

\[ E / H := E' \]

- \( \text{deg} \phi \) is its degree as an algebraic map
Computational Isogenies

In practice: an isogeny \( \phi \) is just a rational fraction

\[
\frac{N(x)}{D(x)} = \frac{x^n + \cdots + n_1x + n_0}{x^{n-1} + \cdots + d_1x + d_0} \in k(x), \quad \text{with } n = \deg \phi,
\]

and \( D(x) \) vanishes on \( \ker \phi \).

We are interested in (separable) isogenies over finite fields. In this case there are other possible ways to represent an isogeny (up to isomorphism):

- A finite subgroup \( H \) of \( E \) specifies an isogeny \( E \to E/H \), up to isomorphism.
- A list of generators of \( H \) also specifies an isogeny.
Supersingular curves

Key exchange only uses *supersingular* elliptic curves

**Propositions**

- Every supersingular curve is defined over $\mathbb{F}_{p^2}$.
- There are $\sim \frac{p+1}{12}$ supersingular curves up to isomorphism.
- $E(\mathbb{F}_{p^2}) \cong (\mathbb{Z}/(p + 1)\mathbb{Z})^2$.
**Fixed parameters:**

- Prime $p$ such that $p + 1 = \ell^a_A \ell^b_B$;
- Supersingular curve $E \simeq (\mathbb{Z}/(p + 1)\mathbb{Z})^2$;
- $E[\ell^b_B] = \langle P_A, Q_A \rangle$;
- $E[\ell^a_A] = \langle P_B, Q_B \rangle$.

**Secret data:**

- $R_A = m_A P_A + n_A Q_A$;
- $R_B = m_B P_B + n_B Q_B$.

**Public data:**

- $E/\langle R_A \rangle$, $\phi_A(P_B)$, $\phi_A(Q_B)$
- $E/\langle R_B \rangle$, $\phi_B(Q_A)$
- $E/\langle R_A \rangle \simeq E/\langle R_B \rangle$, $\phi_A(R_B)$, $\phi_B(R_A)$
- $E/\langle R_B \rangle \simeq E/\langle R_A, R_B \rangle$, $\phi_B(Q_A)$
Computing $\phi : E \rightarrow E/\langle R \rangle$

We have $\text{ord}(R) = \ell^a$ and $\phi = \phi_0 \circ \phi_1 \circ \cdots \circ \phi_{a-1}$, each of degree $\ell$.

For each $i$, one needs to compute $[\ell^{a-1-i}]R_i$ in order to compute $\phi_i$. 
What’s the best strategy?

• Right edges are $\ell$-isogeny evaluation;
• Left edges are multiplications by $\ell$ (about twice as expensive);

The best strategy can be precomputed offline and hardcoded in an embedded system.

Remark: Strategies are in one-to-one correspondence with certain instances of Gelfand-Tsetlin polytopes [OEIS, Sequence A130715].
Implementation

Original Implementation Available at http://www.prism.uvsq.fr/~dfl/

- Original Implementation in mixed C, Cython, Python, Sage architecture
- We offloaded parameter generation (Python, Sage) from the main key exchange software and put the key exchange itself in pure C
- Implementation suitable for iOS and Android devices
- Attempted to optimize by coding parts of underlying field arithmetic in X86 and ARM Assembly
# Timings (Unoptimized)

<table>
<thead>
<tr>
<th>Prime Size</th>
<th>512 bits</th>
<th>768 bits</th>
<th>1024 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quantum Security</td>
<td>85 bits</td>
<td>128 bits</td>
<td>170 bits</td>
</tr>
<tr>
<td>Original (Mac OS$^1$)</td>
<td>0.113 s</td>
<td>0.303 s</td>
<td>0.529 s</td>
</tr>
<tr>
<td>C (Mac OS$^1$)</td>
<td>0.093 s</td>
<td>0.226 s</td>
<td>0.429 s</td>
</tr>
<tr>
<td>iOS$^2$</td>
<td>1.06 s</td>
<td>2.68 s</td>
<td>5.30 s</td>
</tr>
<tr>
<td>Android$^3$</td>
<td>0.629 s</td>
<td>1.77 s</td>
<td>3.81 s</td>
</tr>
</tbody>
</table>

- Field addition written in ARM Assembly gave savings of 1.5% on 512 bit Android platform
- Field addition and multiplication written in X86 Assembly gave savings of 4% on C 768 bit (Mac OS) platform

---

$^1$ Macbook Pro Intel Core i5 @ 2.4 GHz

$^2$Ipad 2 ARM Cortex-A9 @ 1 GHz dual-core

$^3$Arnale Board ARM Cortex-A15 @ 1.7 GHz dual-core