Generation of Elliptic Curve Points in Tandem

Armando Faz Hernández\(^1,2\), Julio López\(^1\)

\(^1\)Institute of Computing, University of Campinas  
\(^2\)Cloudflare, Inc.

XX Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais – SBSeg 2020  
October 14, 2020 – Brazil
Elliptic Curves

Let $\mathbb{F}$ be a finite field, an elliptic curve $E$ over $\mathbb{F}$ is defined by the equation

$$E/\mathbb{F} : y^2 = x^3 + ax + b$$

where $4a^3 + 27b^2 \neq 0$.

The points $(x, y)$ on this curve form an abelian group, denoted $E(\mathbb{F})$, with $\mathcal{O}$ as identity element.
Given a bit string $m$, we want a hash function $H$ that takes $m$ and produces a point $P$ on the elliptic curve.

$$H: \{0, 1\}^* \rightarrow E(\mathbb{F})$$

$$m \downarrow$$

$$P = H(m)$$

$$H(m) = P \in E(\mathbb{F})$$
Methods for Generating Points

First approach

- Trial-and-Error -> requires randomization.
Methods for Generating Points

First approach

- Trial-and-Error \(\rightarrow\) requires randomization.
- Skalba equations [2] \(\rightarrow\) deterministic, but expensive.

Deterministic Encodings

Rational maps that, given a field element, deterministically produce valid candidates for the coordinates of a point.

\[ f: \mathbb{F} \rightarrow E(\mathbb{F}) \]

Examples:

- Icart [3]
- SW [4]
- SWU [5]
- Elligator2 [6]
Let 
\[ f : \mathbb{F} \rightarrow E(\mathbb{F}) \]
be a deterministic encoding and 
\[ h : \{0, 1\}^* \rightarrow \mathbb{F} \]
be a cryptographic hash function.

Brier et al. [5] and Farashahi et al. [7] showed that a secure approach to construct a hash to curve function \( H \) is

\[ H(m) = f(h_0(m)) + f(h_1(m)) \]  

(1)

where \( h_0, h_1 \) are independent hash functions into \( \mathbb{F} \).
Let

\[ f : \mathbb{F} \rightarrow E(\mathbb{F}) \] be a deterministic encoding and
\[ h : \{0, 1\}^* \rightarrow \mathbb{F} \] be a cryptographic hash function.

Brier et al. [5] and Farashahi et al. [7] showed that a secure approach to construct a hash to curve function \( H \) is

\[ H(m) = f(h_0(m)) + f(h_1(m)) \]  \hspace{1cm} (1)

where \( h_0, h_1 \) are independent hash functions into \( \mathbb{F} \).

We are interested on optimizing the execution of this function.
Sequential Evaluation of Hashing

Given $m \in \{0, 1\}^*$, calculate $H(m) = f(h_0(m)) + f(h_1(m))$.
Parallel Evaluation of Hashing

\[ m \]

\[ u_0 = h_0(m) \]

\[ u_1 = h_1(m) \]

\[ P_0 = f(u_0) \]

\[ P_1 = f(u_1) \]

\[ P = P_0 + P_1 \]

\[ H(m) = P \in E(\mathbb{F}) \]
Parallel Evaluation of Hashing

\[ m \]

\[ u_0 = h_0(m) \]

\[ u_1 = h_1(m) \]

\[ P_0 = f(u_0) \]

\[ P_1 = f(u_1) \]

\[ P = P_0 + P_1 \]

\[ H(m) = P \in E(\mathbb{F}) \]
Parallel Evaluation of Hashing

\[ m \]

\[ u_0 = h_0(m) \]
\[ u_1 = h_1(m) \]

\[ P_0 = f(u_0) \]
\[ P_1 = f(u_1) \]
\[ P = P_0 + P_1 \]

\[ H(m) = P \in E(\mathbb{F}) \]
Parallel Evaluation of Hashing

\[ u_0 = h_0(m) \]
\[ u_1 = h_1(m) \]
\[ P_0 = f(u_0) \]
\[ P_1 = f(u_1) \]
\[ P = P_0 + P_1 \]
\[ H(m) = P \in E(F) \]
Parallel Evaluation of Hashing

\[
\begin{align*}
    m &
    \quad \Rightarrow \quad u_0 = h_0(m) \\
    &\Rightarrow u_1 = h_1(m) \\
    &\Rightarrow P_0 = f(u_0) \\
    &\Rightarrow P_1 = f(u_1) \\
    &\Rightarrow P = P_0 + P_1 \\
    H(m) &= P \in E(\mathbb{F})
\end{align*}
\]
Vectorized Implementation

**SHA-512**

- All operations are on 64-bit words.
- A 128-bit vector register carries the calculation of two hashes in parallel.

**Prime field arithmetic**

- All operations are on 255-bit elements.
- A set of 256-bit vector registers handles two field operations in parallel.

\[
\langle Z_0, Z_1 \rangle = \langle X_0, X_1 \rangle + \langle Y_0, Y_1 \rangle
\]

- Large time savings because exponentiations can run in parallel.

**Software Library**

- [https://github.com/armfazh/fld-ecc-vec](https://github.com/armfazh/fld-ecc-vec)
Instance Parameters

- For $E(\mathbb{F}_p)$: we use the edwards25519 curve

\[ E/\mathbb{F}_{2^{255}-19} : -x^2 + y^2 = 1 - \frac{121665}{121666} x^2 y^2 \]

- For $h_0, h_1$: we use SHA-512 hash function as recommended in IETF draft [8].

- For $f$: we send the output of the Elligator2 map [6] to a point on edwards25519 through the birational equivalence.

- **Point addition** is performed using complete formulas in extended projective coordinates [9].
Latency of hashing 64-byte strings to points on the edwards25519 curve.

<table>
<thead>
<tr>
<th>Description</th>
<th>Operation</th>
<th>Seq&lt;sup&gt;1&lt;/sup&gt;</th>
<th>Par&lt;sup&gt;1&lt;/sup&gt;</th>
<th>Δ&lt;sup&gt;2&lt;/sup&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hash</td>
<td>$u_i = h_i(m) \mid i=0,1$</td>
<td>3.3</td>
<td>1.8</td>
<td>44.9 %</td>
</tr>
</tbody>
</table>

<sup>1</sup> Entries are 10<sup>3</sup> clock cycles of a Core i7-6700K Skylake processor.
<sup>2</sup> $\Delta = 1 - \frac{\text{Par}}{\text{Seq}}$. 
Latency of hashing 64-byte strings to points on the \texttt{edwards25519} curve.

<table>
<thead>
<tr>
<th>Description</th>
<th>Operation</th>
<th>Seq$^1$</th>
<th>Par$^1$</th>
<th>$\Delta^2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hash</td>
<td>$u_i = h_i(m)</td>
<td>i=0,1$</td>
<td>3.3</td>
<td>1.8</td>
</tr>
<tr>
<td>Encoding</td>
<td>$P_i = \Phi(f(u_i))</td>
<td>i=0,1$</td>
<td>27.6</td>
<td>15.7</td>
</tr>
</tbody>
</table>

1. Entries are $10^3$ clock cycles of a Core i7-6700K Skylake processor.
2. $\Delta = 1 - \frac{\text{Par}}{\text{Seq}}$. 
Performance Benchmark on Skylake

Latency of hashing 64-byte strings to points on the edwards25519 curve.

<table>
<thead>
<tr>
<th>Description</th>
<th>Operation</th>
<th>Seq(^1)</th>
<th>Par(^1)</th>
<th>Δ(^2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hash</td>
<td>(u_i = h_i(m))(\mid i=0,1)</td>
<td>3.3</td>
<td>1.8</td>
<td>44.9 %</td>
</tr>
<tr>
<td>Encoding</td>
<td>(P_i = \Phi(f(u_i)))(\mid i=0,1)</td>
<td>27.6</td>
<td>15.7</td>
<td>43.0 %</td>
</tr>
<tr>
<td>Point addition</td>
<td>(P = P_0 + P_1)</td>
<td>0.5</td>
<td>0.5</td>
<td>0.0 %</td>
</tr>
<tr>
<td>Point to affine</td>
<td>((x, y) = P)</td>
<td>11.9</td>
<td>11.9</td>
<td>0.0 %</td>
</tr>
</tbody>
</table>

\(^1\) Entries are \(10^3\) clock cycles of a Core i7-6700K Skylake processor.
\(^2\) \(\Delta = 1 - \frac{\text{Par}}{\text{Seq}}\).
Latency of hashing 64-byte strings to points on the \texttt{edwards25519} curve.

<table>
<thead>
<tr>
<th>Description</th>
<th>Operation</th>
<th>Seq(^1)</th>
<th>Par(^1)</th>
<th>(\Delta)^2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hash</td>
<td>(u_i = h_i(m)\big</td>
<td>_{i=0,1})</td>
<td>3.3</td>
<td>1.8</td>
</tr>
<tr>
<td>Encoding</td>
<td>(P_i = \Phi(f(u_i))\big</td>
<td>_{i=0,1})</td>
<td>27.6</td>
<td>15.7</td>
</tr>
<tr>
<td>Point addition</td>
<td>(P = P_0 + P_1)</td>
<td>0.5</td>
<td>0.5</td>
<td>0.0 %</td>
</tr>
<tr>
<td>Point to affine</td>
<td>((x, y) = P)</td>
<td>11.9</td>
<td>11.9</td>
<td>0.0 %</td>
</tr>
<tr>
<td>Hash to curve</td>
<td>(H(m))</td>
<td>46.0</td>
<td>32.7</td>
<td>28.8 %</td>
</tr>
</tbody>
</table>

\(^1\) Entries are \(10^3\) clock cycles of a Core i7-6700K Skylake processor.

\(^2\) \(\Delta = 1 - \frac{\text{Par}}{\text{Seq}}\).
Analysis of Speedup Factor

\[
\text{Speedup Factor} = \frac{\text{Exec Time Sequential}}{\text{Exec Time Parallel}} \\
\text{Ideal Factor} = 2
\]
Analysis of Speedup Factor

Speedup Factor = \frac{\text{Exec Time Sequential}}{\text{Exec Time Parallel}}

Ideal Factor = 2
Analysis of Speedup Factor

\[
\text{Speedup Factor} = \frac{\text{Exec Time Sequential}}{\text{Exec Time Parallel}}
\]

Ideal Factor = 2
Analysis of Speedup Factor

Speedup Factor = $\frac{\text{Exec Time Sequential}}{\text{Exec Time Parallel}}$

Ideal Factor = 2
Analysis of Speedup Factor

Speedup Factor = \frac{\text{Exec Time Sequential}}{\text{Exec Time Parallel}}

Ideal Factor = 2
Analysis of Speedup Factor

![Graph showing speedup factor vs input length in bytes for different input sizes, comparing Haswell and Skylake architectures. The y-axis represents speedup factor, and the x-axis represents input length in bytes (1, 4, 16, 64, 1K, 4K, 16K, 64K, 256K, 1M). The graph shows that as input size increases, the speedup factor also increases, with a noticeable difference between Haswell and Skylake. The ideal factor is 2.](image)

Speedup Factor = \( \frac{\text{Exec Time Sequential}}{\text{Exec Time Parallel}} \)

Ideal Factor = 2
Analysis of Speedup Factor

![Graph showing speedup factor vs input length in bytes for Haswell and Skylake processors. The y-axis represents speedup factor, and the x-axis represents input length in bytes. The graph compares the execution time (sequential vs parallel) for encoding and hash operations. The ideal factor is 2.]

\[
\text{Speedup Factor} = \frac{\text{Exec Time Sequential}}{\text{Exec Time Parallel}}
\]

Ideal Factor = 2
Summary

• A parallel strategy for hash to curve functions.
• This work covers:
  • Sequential and parallel software implementations.
  • Use of the edwards25519 curve.
  • Optimized performance using AVX2 vector instructions.
• Performance benchmark shows a 1.4 speedup factor for short inputs, a larger factor is observed as the input’s size grows.
Final Remarks

Summary

• A parallel strategy for hash to curve functions.
• This work covers:
  • Sequential and parallel software implementations.
  • Use of the *edwards*25519 curve.
  • Optimized performance using AVX2 vector instructions.
• Performance benchmark shows a 1.4 speedup factor for short inputs, a larger factor is observed as the input’s size grows.

Future Work

• Evaluate performance on hardware implementations.
• Use of alternative vector instruction sets.
• Consider other elliptic curves or deterministic encodings.


Generation of Elliptic Curve Points in Tandem

Armando Faz Hernández$^{1,2}$, Julio López$^{1}$

$^{1}$Institute of Computing, University of Campinas
$^{2}$Cloudflare, Inc.

XX Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais – SBSeg 2020

October 14, 2020 – Brazil