Efficient Post-Quantum Undeniable Signature on 64-bit ARM

Amir Jalali ¹  Reza Azarderakhsh ¹  Mehran Mozaffari-Kermani ²

Department of Computer and Electrical Engineering and Computer Science, Florida Atlantic University, FL, USA

Department of Computer Science and Engineering, University of South Florida, FL, USA

August 2017
Outline

1 Introduction

2 SIUS Protocol

3 Proposed Choice of Implementation-Friendly Primes

4 SIUS Protocol Implementation

5 Implementation Results

6 Conclusions
Isogeny-based Crypto History

- The first suggestions to use isogenies in crypto by Couveignes in 1997
- Supersingular isogeny-based hash function by Charles, Lauter and Goren in 2005
- Isogeny-based public-key cryptosystems by Rostovtsev and Stolbunov in 2006
- The biggest impetus by David Jao and Luca De Feo in 2011.
Undeniable Signature and SIUS

- **Undeniable Signature**
  1. Invented by Chaum in 1989
  2. Allows the signer to choose to whom signatures are verified
  3. Interactive protocol between the signer and the verifier
  4. Applications: e-voting, e-auction, e-cash, ...

This work presents the first practical implementation of the Isogeny-based Undeniable Signature (SIUS) which was first introduced by Jao and Soukhen in 2014.

- Smallest keys and signature size compared to other post-quantum candidates
- Fast and optimized implementation
- Quantum-resistant undeniable signature scheme

Jalali, Azarderakhsh, Mozaffari-Kermani
Post-Quantum Undeniable Signature
August 2017
Undeniable Signature and SIUS

- **Undeniable Signature**
  1. Invented by Chaum in 1989
  2. Allows the signer to choose to whom signatures are verified
  3. Interactive protocol between the signer and the verifier
  4. Applications: e-voting, e-auction, e-cash, ...

- This work presents the **first** practical implementation of the Isogeny-based Undeniable Signature (SIUS) which was first introduced by Jao and Soukharev in 2014
  1. **Smallest** keys and signature size compared to other post-quantum candidates
  2. Fast and optimized implementation
  3. Quantum-resistant undeniable signature scheme
Isogenies on Elliptic Curves

**Definition**

Let $E$ and $E'$ be elliptic curves over $\mathbb{F}$.

An isogeny $\phi : E \to E'$ is a non-constant algebraic morphism (defined by polynomials)

$$
\phi(x, y) = \left( \frac{p(x)}{q(x)}, \frac{s(x)}{t(x)} y \right)
$$

satisfying $\phi(\infty) = \infty$ and $\phi(P + Q) = \phi(P) + \phi(Q)$.

The kernel $H$ determines the image curve $E'$ up to isomorphism

$$
E/H := E'
$$

deg(\phi)$ is its degree as an algebraic map
Public Parameters

- $p = \ell_A^{e_A} \ell_B^{e_B} \ell_C^{e_C} f \pm 1$, where $\ell_A$, $\ell_B$, and $\ell_C$ are small primes, $e_A$, $e_B$, and $e_C$ are positive integers, and $f$ is a small cofactor to make the number prime.
- Starting supersingular elliptic curve, $E_0/\mathbb{F}_p^2$
- Torsion bases $\{P_A, Q_A\}$, $\{P_B, Q_B\}$, and $\{P_C, Q_C\}$ over $E_0[\ell_A^{e_A}]$, $E_0[\ell_B^{e_B}]$, and $E_0[\ell_C^{e_C}]$, respectively.
SIUS Protocol

SIUS Overview

- **Public Parameters**
  - $p = \ell_A^{e_A} \ell_B^{e_B} \ell_C^{e_C} f \pm 1$, where $\ell_A$, $\ell_B$, and $\ell_C$ are small primes, $e_A$, $e_B$, and $e_C$ are positive integers, and $f$ is a small cofactor to make the number prime.
  - Starting supersingular elliptic curve, $E_0/\mathbb{F}_p^2$
  - Torsion bases $\{P_A, Q_A\}$, $\{P_B, Q_B\}$, and $\{P_C, Q_C\}$ over $E_0[\ell_A^{e_A}]$, $E_0[\ell_B^{e_B}]$, and $E_0[\ell_C^{e_C}]$, respectively.

- Classical and quantum security is approximately $\sqrt[6]{p}$ and $\sqrt[9]{p}$, respectively.
  - Based on the difficulty of computing isogenies between supersingular elliptic curves.
SIUS Overview

Key-generation:

- The signer securely generates two random integers $m_A, n_A \in \mathbb{Z}/\ell_A^{e_A}\mathbb{Z}$ and computes $K_A = [m_A]P_A + [n_A]Q_A$.
- The signer computes isogeny map $\phi_A : E \rightarrow E_A/\langle K_A \rangle$ and also evaluates $\phi_A(P_C)$ and $\phi_A(Q_C)$ using $\phi_A$.
- The signer publishes the public-key as: $E_A, \phi_A(P_C)$, and $\phi_A(Q_C)$, while the private-key is $(m_A, n_A)$. 
Key-generation:
- The signer securely generates two random integers $m_A, n_A \in \mathbb{Z}/\ell_A^e \mathbb{Z}$ and computes $K_A = [m_A]P_A + [n_A]Q_A$.
- The signer computes isogeny map $\phi_A : E \rightarrow E_A/\langle K_A \rangle$ and also evaluates $\phi_A(P_C)$ and $\phi_A(Q_C)$ using $\phi_A$.
- The signer publishes the public-key as: $E_A, \phi_A(P_C), \phi_A(Q_C)$, while the private-key is $(m_A, n_A)$.

Signature:
- The signer computes the message hash $h = H(M)$, $K_M = P_M + [h]Q_M$.
- The signer first computes $\phi_M : E \rightarrow E_M = E/\langle K_M \rangle$. 

Jalali, Azarderakhsh, Mozaffari-Kermani | Post-Quantum Undeniable Signature | August 2017 7 / 21
SIUS Overview

- **Key-generation:**
  - The signer securely generates two random integers \( m_A, n_A \in \mathbb{Z}/\mathbb{Z}^e \) and computes \( K_A = [m_A]P_A + [n_A]Q_A \).
  - The signer computes isogeny map \( \phi_A : E \rightarrow E_A/\langle K_A \rangle \) and also evaluates \( \phi_A(P_C) \) and \( \phi_A(Q_C) \) using \( \phi_A \).
  - The signer publishes the public-key as: \( E_A, \phi_A(P_C), \) and \( \phi_A(Q_C) \), while the private-key is \( (m_A, n_A) \).

- **Signature:**
  - The signer computes the message hash \( h = H(M), K_M = P_M + [h]Q_M \).
  - The signer first computes \( \phi_M : E \rightarrow E_M = E/\langle K_M \rangle \).

- The signature:
  \[
  [E_A M, \phi_M, AM(\phi_M(P_C)), \phi_M, AM(\phi_M(Q_C))] 
  \]
The signer secretly selects random integers $m_C, n_C \in \mathbb{Z}/\ell_C^{eC} \mathbb{Z}$ and computes the kernel $K_C = [m_C]P_C + [n_C]Q_C$ to blind the signature and computes $\phi_C, \phi_C, MC, \phi_A, AC, \phi MC, AMC$. 

---

**Confirmation Protocol**
Confirmation Protocol

- The signer secretly selects random integers $m_C, n_C \in \mathbb{Z}/\ell^e_C \mathbb{Z}$ and computes the kernel $K_C = [m_C]P_C + [n_C]Q_C$ to blind the signature and computes $\phi_C, \phi_{C,MC}, \phi_{A,AC}, \phi_{MC,AMC}$
- The signer commits $E_C, E_{AC}, E_{MC}, E_{AMC}$, and $\text{ker}(\phi_{C,MC}) = \phi_C(K_M)$.
- The verifier randomly selects a bit $b \in \{0, 1\}$
Confirmation Protocol

- The signer secretly selects random integers $m_C, n_C \in \mathbb{Z}/\ell_C^e\mathbb{Z}$ and computes the kernel $K_C = [m_C]P_C + [n_C]Q_C$ to blind the signature and computes $\phi_C, \phi_{C,MC}, \phi_{A,AC}, \phi_{MC,AMC}$.

- The signer commits $E_C, E_{AC}, E_{MC}, E_{AMC}$, and $\ker(\phi_{C,MC}) = \phi_C(K_M)$.

- The verifier randomly selects a bit $b \in \{0, 1\}$

  - If $b = 0$
    - The signer outputs $\ker(\phi_C)$
    - The verifier computes $\ker(\phi_{A,AC}), \phi_{M,MC}, \phi_{AM,AMC}, \phi_{C,MC}$.
    - Verifier checks the correctness of all the committed information by signer.

  - If $b = 1$
    - The signer outputs $\ker(\phi_{C,AC})$
    - The verifier computes $\phi_{MC,AMC}, \phi_{AC,AMC}$ and checks the corresponding curves in the commitment.
Figure: Signature and confirmation protocol in SIUS scheme
Disavowal Protocol

- The signer is presented with a fake signature \((E_F, F_P, F_Q)\) instead of the real signature \((E_{AM}, \phi_{M,AM}(\phi_M(P_C)), \phi_{M,AM}(\phi_M(Q_C)))\)
Disavowal Protocol

- The signer is presented with a fake signature \((E_F, F_P, F_Q)\) instead of the real signature \((E_{AM}, \phi_{M,AM}(\phi_{M}(PC)), \phi_{M,AM}(\phi_{M}(QC)))\).
- The signer secretly selects random integers \(m_C, n_C \in \mathbb{Z}/\ell_C^{ec} \mathbb{Z}\) and computes the kernel \(K_C = [m_C]P_C + [n_C]Q_C\) along with all the curves and isogeny maps as shown before.
Disavowal Protocol

- The signer is presented with a fake signature \((E_F, F_P, F_Q)\) instead of the real signature \((E_{AM}, \phi_{M,AM}(\phi_M(P_C)), \phi_{M,AM}(\phi_M(Q_C)))\).
- The signer secretly selects random integers \(m_C, n_C \in \mathbb{Z}/\ell_C^e \mathbb{Z}\) and computes the kernel \(K_C = [m_C]P_C + [n_C]Q_C\) along with all the curves and isogeny maps as shown before.
- The signer commits \(E_C, E_{AC}, E_{MC}, E_{AMC}\), and \(\ker(\phi_{C,MC}) = \phi_C(K_M)\).
Disavowal Protocol

- The signer is presented with a fake signature \((E_F, F_P, F_Q)\) instead of the real signature \((E_{AM}, \phi_{M, AM}(\phi_M(P_C)), \phi_{M, AM}(\phi_M(Q_C)))\).
- The signer secretly selects random integers \(m_C, n_C \in \mathbb{Z}/\ell_C^e \mathbb{Z}\) and computes the kernel \(K_C = [m_C]P_C + [n_C]Q_C\) along with all the curves and isogeny maps as shown before.
- The signer commits \(E_C, E_{AC}, E_{MC}, E_{AMC}\), and \(\ker(\phi_{C, MC}) = \phi_C(K_M)\).
- The verifier randomly generates a bit \(b \in \{0, 1\}\).
Disavowal Protocol

- The signer is presented with a fake signature \((E_F, F_P, F_Q)\) instead of the real signature \((E_{AM}, \phi_{M, AM}(\phi_M(P_C)), \phi_{M, AM}(\phi_M(Q_C)))\).
- The signer secretly selects random integers \(m_C, n_C \in \mathbb{Z}/\ell^e_{C}\mathbb{Z}\) and computes the kernel \(K_C = [m_C]P_C + [n_C]Q_C\) along with all the curves and isogeny maps as shown before.
- The signer commits \(E_C, E_{AC}, E_{MC}, E_{AMC}\), and \(\ker(\phi_{C, MC}) = \phi_C(K_M)\).
- The verifier randomly generates a bit \(b \in \{0, 1\}\).
- The verifier computations are all the same as before except in case of \(b = 0\) which requires one more isogeny computation:
\[
\phi_F : E_F \to E_{FC} = E_F/\langle [m_C]F_P + [n_C]F_Q \rangle.
\]
- The verifier computes this isogeny and compares it with \(E_{AMC}\) (committed value by signer). These values should be different.
SIUS-Friendly Primes

- Smooth Isogeny Prime: \( p = \ell_A^{e_A} \ell_B^{e_B} \ell_C^{e_C} \cdot f \pm 1 \), where \( \ell_A, \ell_B, \) and \( \ell_C \) are small primes, \( e_A, e_B, \) and \( e_C \) are positive integers, and \( f \) is a small cofactor to make the number prime.
SIUS-Friendly Primes

- **Smooth Isogeny Prime:** \( p = \ell_A^{e_A} \ell_B^{e_B} \ell_C^{e_C} \cdot f \pm 1 \), where \( \ell_A, \ell_B, \) and \( \ell_C \) are small primes, \( e_A, e_B, \) and \( e_C \) are positive integers, and \( f \) is a small cofactor to make the number prime.

- Fast known point multiplications and isogeny formulas for \( \ell_A = 2 \) and \( \ell_B = 3 \) in affine and projective coordinates.
SIUS-Friendly Primes

- Smooth Isogeny Prime: \( p = \ell_A^{e_A} \ell_B^{e_B} \ell_C^{e_C} \cdot f \pm 1 \), where \( \ell_A, \ell_B, \) and \( \ell_C \) are small primes, \( e_A, e_B, \) and \( e_C \) are positive integers, and \( f \) is a small cofactor to make the number prime.

- Fast known point multiplications and isogeny formulas for \( \ell_A = 2 \) and \( \ell_B = 3 \) in affine and projective coordinates.

- We propose new set of formulas for \( \ell_C = 5 \) in projective coordinates.
SIUS-Friendly Primes

- Smooth Isogeny Prime: \( p = \ell_A^{e_A} \ell_B^{e_B} \ell_C^{e_C} \cdot f \pm 1 \), where \( \ell_A, \ell_B, \) and \( \ell_C \) are small primes, \( e_A, e_B, \) and \( e_C \) are positive integers, and \( f \) is a small cofactor to make the number prime.

- Fast known point multiplications and isogeny formulas for \( \ell_A = 2 \) and \( \ell_B = 3 \) in affine and projective coordinates.

- We propose new set of formulas for \( \ell_C = 5 \) in projective coordinates.

- Security of a large-degree isogeny is \( 3^{\sqrt{\ell^e}} \)
  - Quantum claw finding problem by Childs in 2014.
SIUS-Friendly Primes

- Find two different primes at different security levels for a variety of optimizations
SIUS-Friendly Primes

- Find two different primes at different security levels for a variety of optimizations
- Prime search criteria:
  - **Security:**
    - The relative security of SIUS over a prime is based on \( \min(\ell^a, \ell^b, \ell^c) \).
SIUS-Friendly Primes

- Find two different primes at different security levels for a variety of optimizations

Prime search criteria:

- **Security:**
  - The relative security of SIUS over a prime is based on \( \min(\ell_A, \ell_B, \ell_C) \).

- **Speed:**
  - Primes of the form \( p = 2^a \ell_B \cdot f - 1 \) → Montgomery-friendly property
  - Prime search: efficiency parameter \( \theta \) for a prime of the form \( p = \ell_A^{e_A} \ell_B^{e_B} \ell_C^{e_C} - 1 \)
    
    \[
    \theta = \frac{\text{nbits}(p)}{\min(\text{nbits}(\ell_A^{e_A}, \ell_B^{e_B}, \ell_C^{e_C}))/3}
    \]

  - Recall: security of a large-degree isogeny is \( 3\sqrt{\ell_e} \)
  - We are interested in the primes with the smaller value of \( \theta \)
Table: Proposed smooth implementation-friendly primes for SIUS scheme

<table>
<thead>
<tr>
<th>Prime expression</th>
<th>Prime size (bits)</th>
<th>Quantum Security</th>
<th>Classical Security</th>
<th>Prev. Signature (bytes)</th>
<th>Signature (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p = \ell_A^{e_A} \ell_B^{e_B} \ell_C^{e_C} - 1 )</td>
<td>764</td>
<td>83</td>
<td>125</td>
<td>764</td>
<td>573</td>
</tr>
<tr>
<td>( 2^{25031635110} - 1 )</td>
<td>1014</td>
<td>110</td>
<td>165</td>
<td>1014</td>
<td>761</td>
</tr>
</tbody>
</table>

- By ignoring the curve coefficient \( B \) and using projective coordinates, each element of the signature, i.e., curve and auxiliary points is represented by only one field element in \( \mathbb{F}_{p^2} \).
- Therefore SIUS signature and public-key in our implementation are 25% smaller than the original signature sizes reported in the original scheme by Jao and Soukharev.
Projective Isogeny costs

- **Projective 3 Isogenies**
  1. Isogeny map: \((6M + 2S + 5a)\)
  2. Isogeny eval.: \((3M + 3S + 8a)\)

- **Projective 4 Isogenies**
  1. Isogeny map: \((5S + 7a)\)
  2. Isogeny eval.: \((3M + 3S + 8a)\)

- **Projective 5 Isogenies**
  1. Isogeny map: \((10M + 2S + 7a)\) → **slow**
  2. Isogeny eval.: \((30M + 4S + 16a)\) → **very slow**
Confirmation Protocol Mechanism

- Interactive procedure (both parties should involve)
- The verifier’s computations depend on the value of $b$
- Disavowal protocol mechanism is almost the same

**Figure**: The SIUS confirmation protocol mechanism.
Finite-Field Arithmetic on ARMv8

- A64 or Advanced SIMD?
Finite-Field Arithmetic on ARMv8

- A64 or Advanced SIMD?
  - **A64**: General-purpose register file with thirty one 64-bit registers (radix-$2^{64}$)

![Figure: 8 × A64 multiplications](image)
Finite-Field Arithmetic on ARMv8

A64 or Advanced SIMD?
- **A64**: General-purpose register file with thirty one 64-bit registers (radix-\(2^{64}\))
- **Adv. SIMD**: 256-bit vectors which can be used to implement \(32 \times 32\)-bit multiplication in parallel (radix-\(2^{32}\))
A64 or Advanced SIMD?

- **A64**: General-purpose register file with thirty one 64-bit registers (radix-$2^{64}$)
- **Adv. SIMD**: 256-bit vectors which can be used to implement $32 \times 32$-bit multiplication in parallel (radix-$2^{32}$)
- Both take the same number of multiplications for the implementation of field multi-precision multiplication
- **A64** implementation is faster because ASIMD multiplications are more expensive!

\[
\begin{array}{c}
\text{MUL}(a_0, b_0) \\
\text{MUL}(a_0, b_1) \\
\text{MUL}(a_1, b_0) \\
\text{MUL}(a_1, b_1)
\end{array}
\quad
\begin{array}{c}
\text{UMULH}(a_0, b_0) \\
\text{UMULH}(a_0, b_1) \\
\text{UMULH}(a_1, b_0) \\
\text{UMULH}(a_1, b_1)
\end{array}
\]

\[
\begin{array}{c}
\text{MUL}(a_0, b_0) \\
\text{MUL}(a_0, b_1) \\
\text{MUL}(a_1, b_0) \\
\text{MUL}(a_1, b_1)
\end{array}
\quad
\begin{array}{c}
\text{UMULL}(a_0, a_1, b_0) \\
\text{UMULL}(a_0, a_1, b_1) \\
\text{UMULL}(a_0, a_1, b_2) \\
\text{UMULL}(a_0, a_1, b_3)
\end{array}
\quad
\begin{array}{c}
\text{UMULL2}(a_2, a_3, b_0) \\
\text{UMULL2}(a_2, a_3, b_1) \\
\text{UMULL2}(a_2, a_3, b_2) \\
\text{UMULL2}(a_2, a_3, b_3)
\end{array}
\]

**Figure**: $8 \times$ A64 multiplications

**Figure**: $8 \times$ ASIMD multiplications
Finite-Field Multiplication

- $A \times B = C$, where $A, B, C \in \mathbb{F}_p$
- Requires a reduction from $2m$ bits to $m$ bits, so Montgomery reduction was used
- Perform separated multiply and reduce with Cascade Operand Scanning (COS) method
Finite-Field Multiplication

- $A \times B = C$, where $A, B, C \in \mathbb{F}_p$
- Requires a reduction from $2m$ bits to $m$ bits, so Montgomery reduction was used
- Perform separated multiply and reduce with Cascade Operand Scanning (COS) method
  - Utilizes ARMv8 A64 registers in radix-$2^{64}$ representation
  - With choice of primes, we reduce the complexity from $k^2 + k$ to $k^2$ single-precision multiplications, where $k$ is the number of words in the field
  - Also reduction over $\hat{p} = p + 1$ which eliminates several single-precision multiplications by “0” limbs:
    - $p764 + 1$ and $p1014 + 1$ have three and five 64-bit words equal to “0” in the lower half.
Finite-Field Inversion

- Finds some $A^{-1}$ such that $A \cdot A^{-1} = 1$, where $A, A^{-1} \in \mathbb{F}_p$

- Fermat’s little theorem performs $A^{-1} = A^{p-2}$
  - Complexity $O(\log^3 n)$
Finite-Field Inversion

- Finds some \( A^{-1} \) such that \( A \cdot A^{-1} = 1 \), where \( A, A^{-1} \in \mathbb{F}_p \)
- Fermat’s little theorem performs \( A^{-1} = A^{p-2} \)
  - Complexity \( O(\log^3 n) \)
- Since we implemented the whole point arithmetic in projective coordinates, the number of filed inversions are scarce
Finite-Field Inversion

- Finds some $A^{-1}$ such that $A \cdot A^{-1} = 1$, where $A, A^{-1} \in \mathbb{F}_p$
- Fermat’s little theorem performs $A^{-1} = A^{p-2}$
  - Complexity $O(\log^3 n)$
- Since we implemented the whole point arithmetic in projective coordinates, the number of field inversions are scarce
- We implemented constant-time FLT field inversion with fixed-window method
  - We prioritize security over a small amount of performance improvement in using non-constant time algorithms
Benchmark Targets

- The first empirical implementation of a quantum-resistant undeniable signature
The first empirical implementation of a quantum-resistant undeniable signature

Target processor: Huawei Nexus 6P smartphone with a 2.0 GHz Cortex-A57 and a 1.55 GHz Cortex-A53 processors running Android 7.1.1
Benchmark Targets

- The first empirical implementation of a quantum-resistant undeniable signature
- Target processor: Huawei Nexus 6P smart phone with a 2.0 GHz Cortex-A57 and a 1.55 GHz Cortex-A53 processors running Android 7.1.1
- Portable version is benchmarked on:
  - 2.3 GHz NVIDIA Jetson TK1 equipped with a 32-bit ARMv7 Cortex-A15 running Ubuntu 14.04 LTS
  - 2.1 GHz Intel x64 i7-6700 running Ubuntu 16.04 LTS
Results

- Verifier’s operations (server-side) are more computationally intensive
  - Performance bottleneck $\rightarrow b = 0$
- More efficient degree 5 isogenies formulas $\rightarrow$ significant performance improvement (future work)

Table: Performance results ($\times 10^6$ CPU clock cycles)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>764</td>
<td>83</td>
<td>C</td>
<td>1,068/230</td>
<td>1,416/290</td>
<td>2,638/544</td>
<td>2,980/614</td>
</tr>
<tr>
<td>1014</td>
<td>110</td>
<td>C</td>
<td>2,646/512</td>
<td>3,592/684</td>
<td>6,854/1,310</td>
<td>7,726/1,466</td>
</tr>
</tbody>
</table>

Huawei Nexus 6P ARMv8-A57 at 2.0 GHz

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>764</td>
<td>83</td>
<td>C</td>
<td>2,024/516</td>
<td>2,595/652</td>
<td>4,834/1,213</td>
<td>5,463/1,378</td>
</tr>
<tr>
<td>1014</td>
<td>110</td>
<td>C</td>
<td>4,515/1,227</td>
<td>6,142/1,671</td>
<td>11,724/3,199</td>
<td>13,153/3,585</td>
</tr>
</tbody>
</table>

Huawei Nexus 6P ARMv8-A53 at 1.55 GHz

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>764</td>
<td>83</td>
<td>C</td>
<td>493/1,136</td>
<td>655/1,545</td>
<td>1,222/2,973</td>
<td>1,379/3,357</td>
</tr>
<tr>
<td>1014</td>
<td>110</td>
<td>C</td>
<td>3,433/8,052</td>
<td>4,549/10,957</td>
<td>8,473/20,913</td>
<td>9,574/23,453</td>
</tr>
</tbody>
</table>

Desktop PC Intel x64 i7-6700 at 2.1 GHz

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>764</td>
<td>83</td>
<td>C</td>
<td>3,433/8,052</td>
<td>4,549/10,957</td>
<td>8,473/20,913</td>
<td>9,574/23,453</td>
</tr>
<tr>
<td>1014</td>
<td>110</td>
<td>C</td>
<td>3,433/8,052</td>
<td>4,549/10,957</td>
<td>8,473/20,913</td>
<td>9,574/23,453</td>
</tr>
</tbody>
</table>
Conclusions

- **Efficient** implementation of SIUS on ARMv8 platforms
- Proposed SIUS-friendly primes with an efficiency parameter
- Hand-optimized finite-field arithmetic → up to 5 times faster than generic C implementation
- Analysis of the ARMv8 capabilities for finite field arithmetic implementation
- Implementations on Huawei Nexus 6P → practical benchmark on a smartphone
- We reduce the signature and public-key sizes of SIUS protocol by 25% compared to the original scheme
- Thank you!