qDSA: Compact Kummer signatures for IoT and other small devices

Benjamin Smith

X/Cisco Symposium, Issy-les-Moulineaux. April 9, 2018

INRIA + Laboratoire d'Informatique de l'École polytechnique (LIX)

Joint work with ***Joost Renes****, Radboud Universiteit Nijmegen*

The Internet of Insecure Things

The **Internet of Things**: a ubiquitous, pervasive, embedded, decentralised distributed computing platform.

Almost entirely **unsecured**, and mostly **unmaintained**.

To setup new connections, authenticate devices, and provide **software updates**, we need basic asymmetric (public-key) crypto.

Public-key cryptosystems give us

- **key exchange** protocols to establish secure connections
- and **digital signatures** to authenticate data.

Size matters

Unfortunately, embarking asymmetric cryptography on a microcontroller is like “carrying a sofa on a motorbike”.

Example: basic **RSA signature verifcation**: a bit of cheap hashing, then cube one 384-byte integer modulo another 384-byte integer.

On the internet, this is **easy**. But if you only have, say, 1K of RAM, then this kind of thing may be practically **impossible**.

There is no "half a sofa"

Small devices need **full-sized security**, because adversaries don't have the same constraints on power, time, memory, or access.

Pre-shared keys

Public-key crypto gives us **key exchange** and **digital signatures**.

Current software: consider **wolfSSL**. A (relatively) lightweight TLS library targeting embedded and constrained environments.

- Small code size: 20-100kB
- Runtime memory: 1-36kB
- 20x smaller than OpenSSL

Where does this variation in size come from?

How do you get down to 20kB code and 1kB runtime?

...You remove the public-key crypto and use **pre-shared keys** (i.e. PSK in TLS).

Pre-shared keys

Using **pre-shared keys** removes the need for expensive key exchange software (when you can get away with it), but it doesn't make signatures any cheaper.

qDSA: a more streamlined, aerodynamic sofa

qDSA (Renes–S., Asiacrypt 2017):
Efficient high-security key exchange and signatures in **well under 1K** of RAM.

Keypairs

Asymmetric crypto **keys** come in (public,private) **pairs**:

public: poses a **problem** in computational mathematics

private: gives the solution to the problem.

Context: **prequantum** crypto for small, **low-memory** devices:
e.g. microcontrollers with only 1 or 2Kbytes of RAM.

This limits us to the most compact public-key systems:

1. **Elliptic Curve Cryptography** and
2. **Genus 2 Cryptography**.

The setting

We work in a group $\mathcal{G}$, which is either an **elliptic curve** or a **genus-2 Jacobian**.

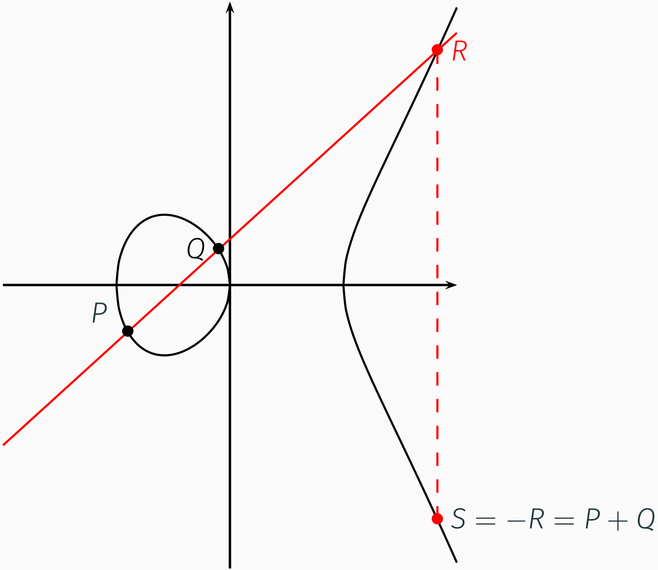
Elements are tuples of bigints mod some prime, with a "group operation" $+$ defined by polynomial formulae.

For example: for the usual 128-bit security level (matching AES) we would use an **elliptic curve**

$$\mathcal{E} : y^2 = x^3 + Ax + B$$

over a 256-bit finite field; so elements are solutions (x, y) to the equation, taken modulo some 256-bit prime: that is, 32-byte values.

Elliptic curve addition: $P + Q = -R = S$



Most cryptographic operations involve **scalar multiplication**:

$$(m \in \mathbb{Z}, P \in \mathcal{G}) \longmapsto [m]P := \underbrace{P + P + \cdots + P}_{m \text{ copies}}$$

Keypairs present instances of the **Discrete Logarithm Problem**:

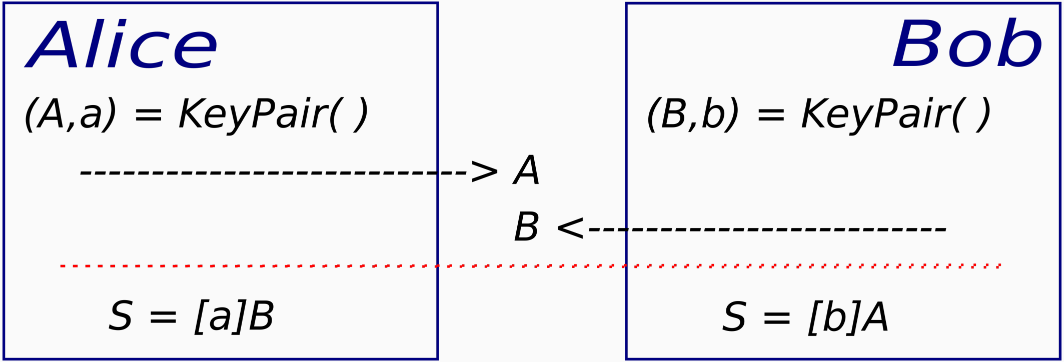
$$(\text{Public}, \text{Private}) = (Q, x) \quad \text{where} \quad Q = [x]P$$

Recovering x from Q and P is **extremely hard** (if $\mathcal{E}$ well chosen).

It seems that using 256-bit numbers for coordinates produces DLPs that are as hard to solve as breaking 3072-bit RSA keys (while being 12× smaller!)

Diffie–Hellman key exchange ($\leq$ 1976)

KeyPair: generates a new pair (Q, x) with $Q = [x]P$; here P is a fixed “base point”.



Alice & Bob now derive a shared cryptographic key from the shared secret S.

Correctness: $[a]B = [a][b]P = [ab]P = [b][a]P = [b]A$ for all $a, b \in \mathbb{Z}$.

Modern Diffie–Hellman key exchange

In modern ECDH (eg. X25519), Alice and Bob **work "up to sign"**:

- Alice's public key is $\pm A = [\pm a]P$ instead of $[a]P$
- Bob's public key is $\pm B = [\pm b]P$ instead of $[b]P$
- they share the secret $\pm[ab]P$ instead of $[ab]P$.

Why do we do this?

- If $P = (x_P, y_P)$ then $-P = (x_P, -y_P)$, so $\pm P \longleftrightarrow x_P$
- We can efficiently compute $\pm[m]P = x_{[m]P}$ from m and x_P.

No need for *y*-coordinates: we save considerable space, time, and energy by completely ignoring them.

Result: we have several practical implementations of X25519 (TLS 1.3-style) key exchange for microcontrollers.

Signatures for microcontrollers

Problem: we also want **signatures**.

Most of the work in ECDSA or (better) Schnorr signatures is in scalar multiplication, which is similar to Diffie–Hellman.

But **verifying signatures** means checking an equation like

$$R = [s]P + [e]Q \qquad \text{for } R, P, Q \in \mathcal{G}\,.$$

In the x-only setting, we should check $\pm R = \pm([s]P + [e]Q)$.

Mathematical problem:
$\pm[s]P$ and $\pm[e]Q$ do not uniquely determine $\pm([s]P + [e]Q)$.

Conventional wisdom: to compute $+$, we need y-coordinates...

Conventional approach

Conventional wisdom: to compute $+$, we need y-coordinates, so we use

1. dedicated x-only software for key exchange,
2. with a second, complete (x, y)-based implementation for signatures.

Example: the NaCl library (`http://nacl.cr.yp.to`)

Disadvantages:

- much slower execution for signatures;
- more stack space for full elliptic curve coordinate systems;
- much bigger trusted code base;
- separate key formats for key exchange and signatures.

Alternative approach: new verification

In fact, **the only place** where an isolated $+$ appears in the protocol is in the signature verification equation

$$R = [s]P + [e]Q \ .$$

Only **one thing stopping us** from using x-only algorithms:
we can't unambiguously compute $\pm([s]P + [e]Q)$ from $\pm[s]P$ and $\pm[e]Q$.

Solution: instead, verify the slightly weaker equation

$$\pm R = \pm[s]P \pm [e]Q$$

using a quadratic polynomial from the classical literature.

Mike Hamburg's elliptic STROBE library already verifies this way.

quotient Digital Signature Algorithm

qDSA = the **q**uotient **D**igital **S**ignature **A**lgorithm (Renes–S., 2017).

- Formalizes this verification hack in an EdDSA-style scheme,
- With a proper security proof.

Now any practical Diffie–Hellman implementation can be cheaply extended into a secure practical signature scheme.

*...So you could also say that qDSA stands for **q**uick and **D**irty **S**ignature **A**lgorithm.*

From theory to practice

Two **free** high-speed **software implementations**:

1. one conservative **elliptic** version (extending Curve25519)
2. one cool **genus-2** version, using **Kummer surfaces**.

Download the code: `http://www.cs.ru.nl/~jrenes/`

Coming soon: a package for RIOT OS.

Experimental results: elliptic implementation

		ATmega (8-bit)		Cortex M0 (32-bit)	
System	Function	Cycles	Stack	Cycles	Stack
Ed25519	sign	19048	1473	—	—
	verify	30777	1226	—	—
Four$\mathbb{Q}$	sign	5175	1590	—	—
	verify	11468	5050	—	—
qDSA-$\mathcal{E}$	sign	14070	**412**	3889	**660**
	verify	25375	**644**	6799	**788**

Ed25519 = Nascimento–López–Dahab (2015). Four$\mathbb{Q}$ = Liu–Longa–Pereira–Reparaz–Seo (2017). qDSA-$\mathcal{E}$ = qDSA on Curve25519 (Renes–S. 2017).

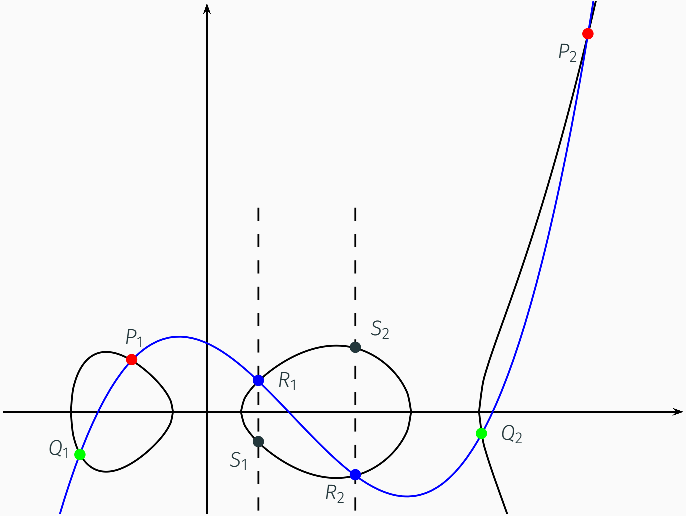
qDSA-$\mathcal{E}$ faster and smaller than Ed25519. Four$\mathbb{Q}$ faster still, but costs *a lot* of stack.

Genus-2 cryptography: a drop-in replacement for elliptic curve crypto (ECC) with the same security-to-keysize ratio.

Closely related: if you can break ECC, then you should be able to break a large chunk of genus 2 (and vice versa).

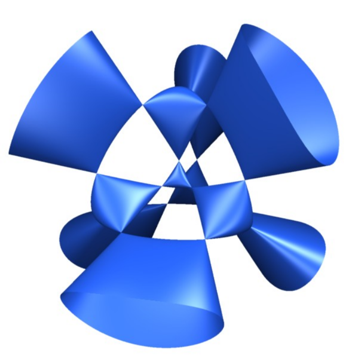
- We replace the elliptic curve $\mathcal{E} : y^2 = x^3 + ax + b$ with a **genus-2 curve** $\mathcal{C} : y^2 = x^5 + \cdots$.
- The group **elements** ($\Longrightarrow$ **keys**) are no longer single points (x, y) on $\mathcal{E}$, but **pairs** of points $\{(x_1, y_1), (x_2, y_2)\}$ on $\mathcal{C}$.
- The group operation $+$ involves interpolating a cubic through two pairs, rather than a line through two points...

Genus 2 group law: $\{P_1, P_2\} + \{Q_1, Q_2\} = -\{R_1, R_2\} = \{S_1, S_2\}$



Kummer surfaces: the analogue of x-coordinates

When we **work up to sign in genus 2**, we don't get pairs of x-coordinates. Instead, we get points on the **Kummer surface** $\mathcal{K}_{\mathcal{C}}$:



...This is the genus-2 analogue of what is just a line in the elliptic world, which says a lot about the jump in mathematical complexity...

Why bother with genus 2?

Genus-2 is obviously much more complicated than ECC.

Why bother?

1. The underlying finite field $\mathbb{F}_p$ (where the coordinates live) has **half the bitsize**: we work with eg. 128-bit integers instead of 256-bit integers.
2. High symmetry of the Kummer surface gives remarkably fast and simple operations (also easily vectorizable).

These qualities already give speed advantages for PC/server implementations, but they are **even more valuable in low-memory contexts**.

Kummer surfaces in practice

Kummers turn out to be **faster** than elliptic *x*-lines for the same security level: $\mathcal{K}_{\mathcal{C}}$ over 128-bit field beats $\mathcal{E}$ over 256-bit field.

Kummers are already used for speed-record Diffie–Hellman software on PCs.
eg. Bernstein–Chuengsatiansup–Lange–Schwabe, 2014

μKummer (Renes–Schwabe–S.–Batina, CHES 2016):
Open crypto lib for **8-bit** and **32-bit microcontrollers**.

		AVR ATmega (8-bit)		ARM Cortex M0 (32-bit)	
Software	Type	KCycles	Stack bytes	KCycles	Stack bytes
NIST P-256	elliptic	34930	590	10730	540
Curve25519	elliptic	13900	494	3590	548
μKummer	**genus-2**	**9739**	**99**	**2644**	**248**

NIST P-256 = Wenger–Unterluggauer–Werner (2013). Curve25519 = Düll et al. (2015)

Experimental results: elliptic vs genus-2

			ATmega (8-bit)		Cortex M0 (32-bit)	
System	Type	Function	Cycles	Stack bytes	Cycles	Stack bytes
Ed25519	elliptic	sign	19048	1473	—	—
		verify	30777	1226	—	—
Four$\mathbb{Q}$	elliptic	sign	5175	1590	—	—
		verify	11468	5050	—	—
qDSA-$\mathcal{E}$	elliptic	sign	14070	**412**	3889	**660**
		verify	25375	**644**	6799	**788**
μKummer	genus-2	sign	10404	926	2865	1360
		verify*	16240	992	4454	1432
qDSA-$\mathcal{K}_{\mathcal{C}}$	genus-2	sign	10477	**417**	2908	**580**
		verify	20423	**609**	5694	**808**

qDSA-$\mathcal{K}_{\mathcal{C}}$ = qDSA on a fast Kummer (Renes–S. 2017), using arithmetic from μKummer.

Details!

For further detail, see the preprint:
`https://eprint.iacr.org/2017/518`