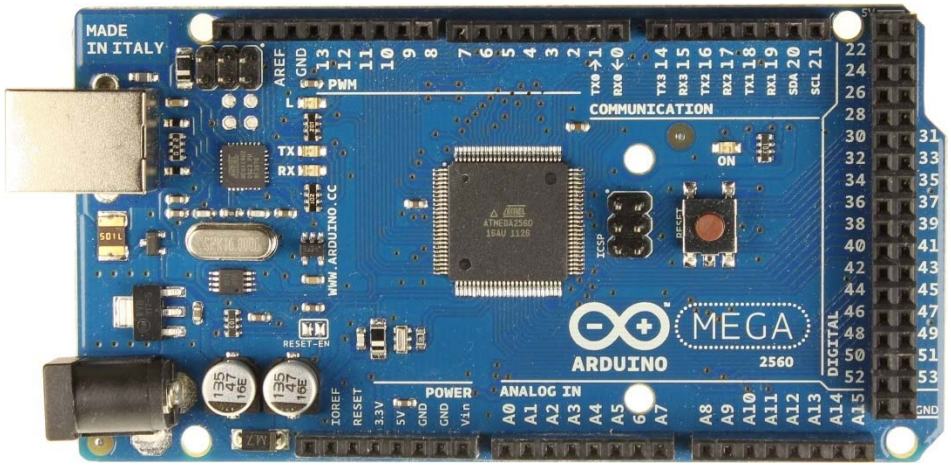
Practical Experiences with crypto on 8-bit

draft-aks-lwig-crypto-sensors-01

Mohit Sethi, Jari Arkko, Ari Keranen, Heidi-Maria Back

Public Key Experiences

- Can we do Public key crypto on (really) small 8-bit devices 2-5 kB of RAM (Class 0/1)
  - What is available off-the shelf to a developer?
  - How hard it is to run these? (How much time and hacking does it need)
  - How is the performance?



PK Experiences - RSA

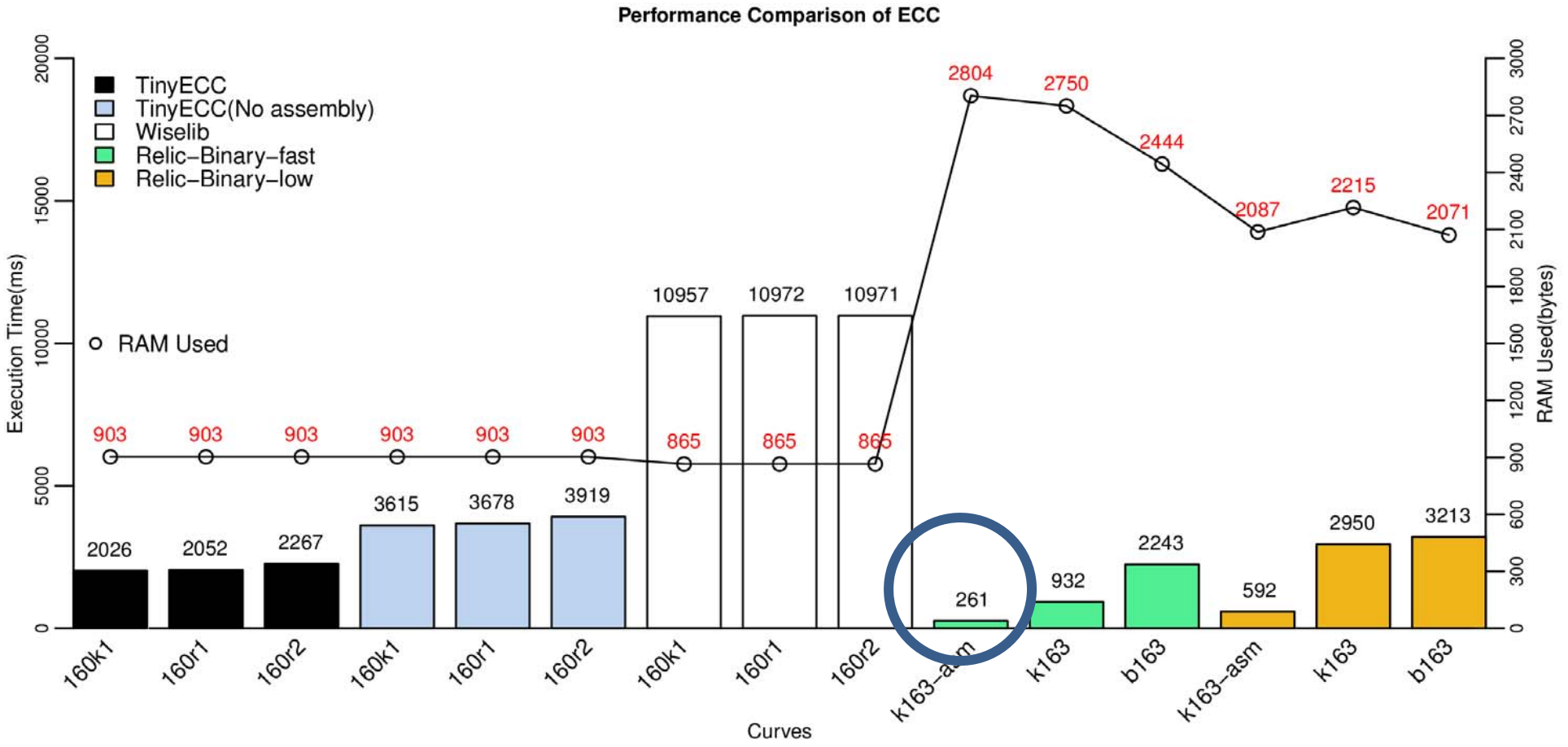
- http://www.emsign.nl/
- AVRCryptoLib

Key Length	Execution Time (ms): Keys in SRAM	Memory footprint (bytes): Keys in SRAM	Execution Time (ms): Keys in ROM	Memory footprint (bytes): Keys in ROM
64	64	40	69	32
128	434	80	460	64
256	3516	160	3818	128
512	25076	320	27348	256
1024	199688	640	218367	512
2048	1587567	1280	1740258	1024

ECDSA libraries

- ECDSA (Elliptic curve digital signature algorithm)
  - Relic (https://github.com/relic-toolkit/)
  - µNaCl (http://munacl.cryptojedi.org/index.shtml)
  - Wiselib (http://www.ibr.cs.tu-bs.de/users/tbaum/wiselib/doxygen/testing/html/index.html)
  - TinyECC (http://discovery.csc.ncsu.edu/software/TinyECC/)

ECDSA libraries

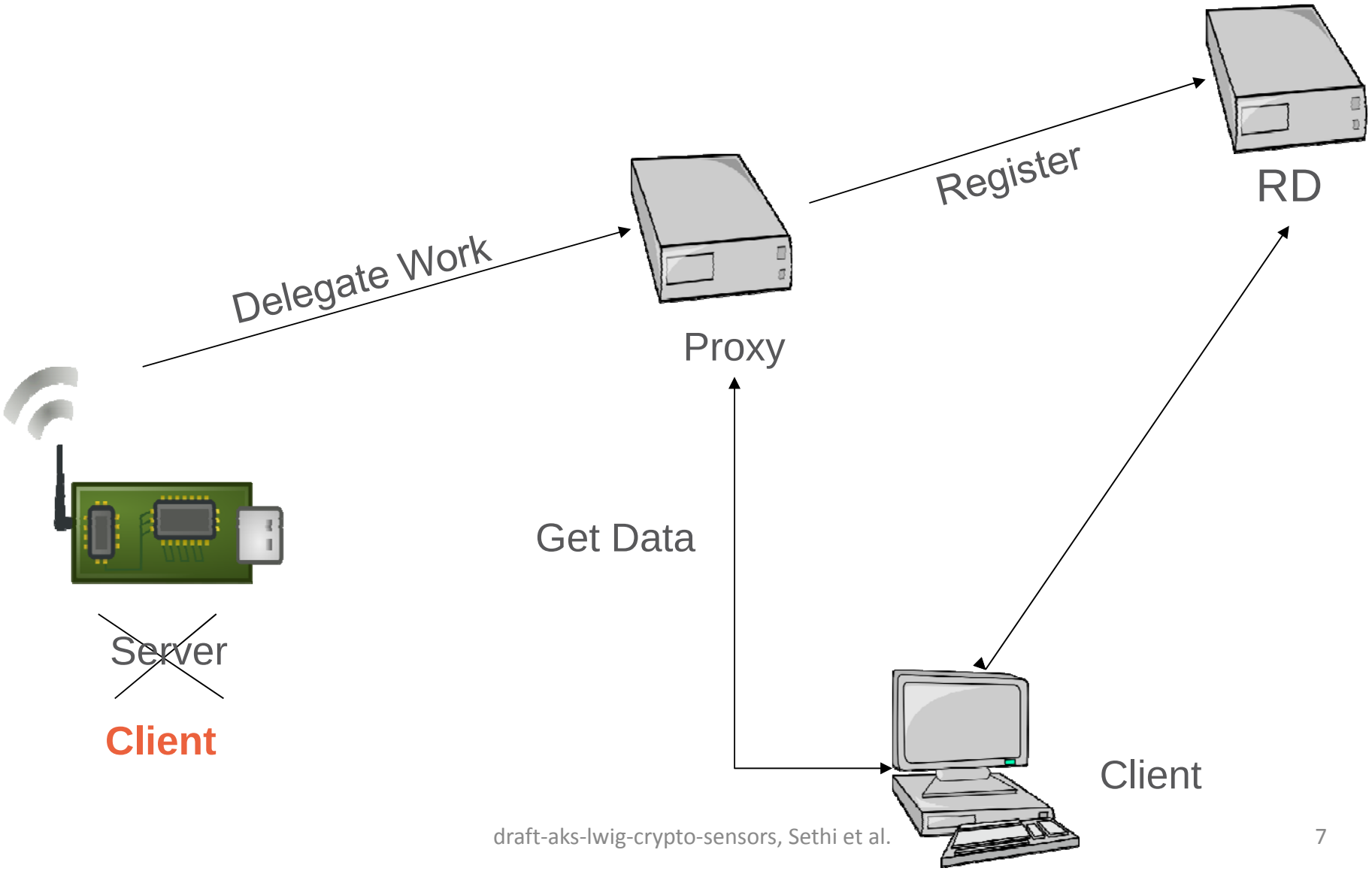


EdDSA libraries

- Edwards-curve Digital Signature Algorithm (EdDSA)
- https://tools.ietf.org/html/draft-irtf-cfrg-eddsa-05
- NaCl and µNaCl high-speed software library
- Public domain
- Signing* = 23,216,241 clock cycles ~ 1,4 sec
- Verification* = 32,634,713 clocks cycles ~ 2 sec

* NaCl on 8-Bit AVR Microcontrollers, Michael Hutter, Peter Schwabe, http://link.springer.com/chapter/10.1007/978-3-642-38553-7_9

Example application



Example application

- Sensor: Arduino-based device
  - Runs a CoAP client in roughly 3 kilobytes of flash
  - Uses relic-toolkit for signing
  - JSON Web Signature (JWS)
- Mirror Proxy: Linux x86_64
- Resource Directory: Linux x86_64, register resources
- Client application: retrieve both registrations from the resource directory and most recent sensor readings from the mirror proxy

Example application

Flash memory consumption (for the entire prototype (including Relic crypto + CoAP + Arduino UDP etc. libraries)	**51 kB**
SRAM consumption (for the entire prototype including client + key generation + signing the hash of message + COAP + UDP)	**4678 bytes**
Execution time for creating the key pair + sending registration message + time spent waiting for acknowledgement	**2030 ms**
time for signing the hash of message+ sending update	**987 ms**
Signature overhead on the wire	**42 bytes**

What we learnt

- Chosen prototype platform was unnecessarily restrictive in the amount of code space
  - we chose this platform on purpose to demonstrate something that is as small and difficult as possible
- Power requirements necessary to send or receive messages are far bigger than those needed to execute cryptographic operations
- No good reason to choose platforms that do not provide sufficient computing power to run the necessary operations

Trade offs

- Symmetric vs Asymmetric
  - Contrary to popular beliefs, a symmetric crypto solution can be deployed in large scale.
    - One of the largest deployment of cryptographic security, the cellular network authentication system, uses SIM cards that are based on symmetric secrets.
  - When there are very costly asymmetric operations, doing a key exchange followed by the use of generated symmetric keys would make sense.
    - This model works very well for DTLS and other transport layer solutions, but works less well for data object security, particularly when the number of communicating entities is not exactly two.

Trade offs – Link layer

- Common solution, security services are commonly available (WLAN secrets, cellular SIM cards)
- No security beyond the first hop
- Can be problematic, e.g., in many devices that communicate to a server in the Internet
- Often link layer security designs use group secrets. This allows any device within the local network (e.g., an infected laptop) to attack the communications

Trade offs – Network layer

- IPsec, PANA and others
- Work across forwarding hops but only as far as to the next middlebox or application entity.
- IPSec-> VPN. Difficult or impossible to tweak to be used on a per-application basis

Trade offs – Transport layer

- TLS and DTLS
- Work across forwarding hops but only as far as to the next middlebox or application entity

Trade offs – data object layer

- Protects individual data elements being sent
- Works particularly well when there are multiple application layer entities on the path of the data
- As long as the data is protected and checked upon every time it passes through an application level entity, it is not clear that there are attacks beyond denial-of-service

Message Freshness

- Essential to ensure that data updates include a freshness indicator:
  - Communication is mostly unidirectional to save energy.
  - Internal clocks might not be accurate and may be reset several times during the operation.
  - NTP is resource intensive.

Message Freshness

- What can be easily implemented:
  - Including sequence numbers in signed messages
    - Must be monotonic (write to flash)
    - Can wrap-around
    - Long sequence numbers increases packet sizes
  - Maintain and sign long sequence numbers of equal bit-lengths but transmit only the least significant bits.
  - Can end up in discordant state
    - Send confirmable (CON) signed updates to sync if RESET is received.
  - Attacker can induce inordinate delays to the communication of signed updates

Path forward

- Serves as an important data point
- People already using and referencing it
- Useful document
- Adopt as WG document?