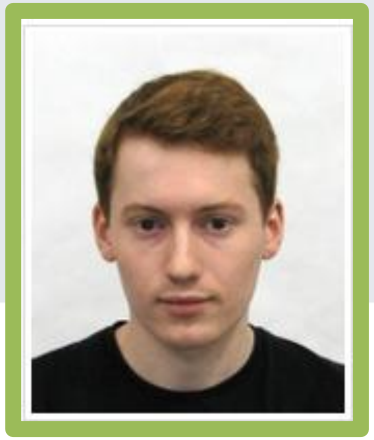
Two-party ECDSA with JavaCard-based smartcards

Antonín Dufka[1], Peeter Laud[2], Petr Švenda[1]

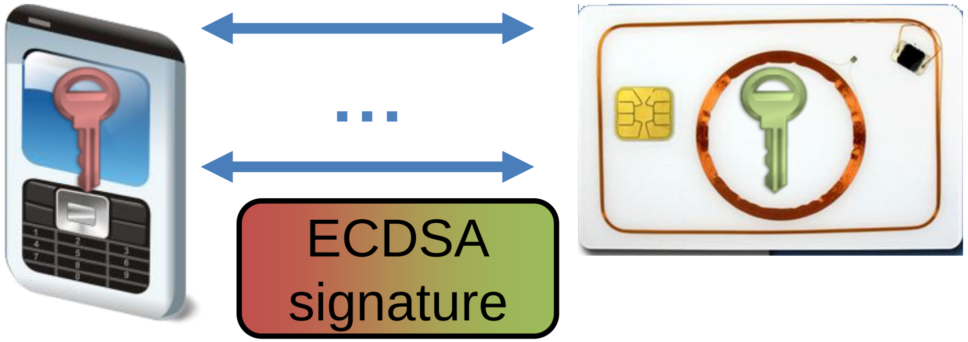


[1]Centre for Research on Cryptography and Security, Masaryk University, Czechia
[2]Cybernetica, Estonia

*Funded by the European Union under Grant Agreement No. 101087529 (CHESS project)

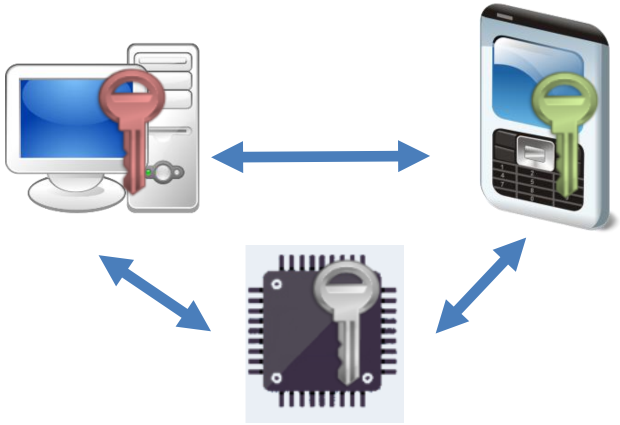
Overview

- Overall goal: 2-of-2 ECDSA with one party being cryptographic smartcard, second party being mobile phone/PC/server…
  1. Better protection of private key,
  2. Possibility of enforcement of a signing policy,
  3. Easy key revocation functionality
  4. …



- Gennaro, Goldfeder, Narayanan proposed k-of-n ECDSA [ACNS'16]
- Lindell proposed efficient computation for 2-of-2 ECDSA [CRYPTO'17]
- This paper shows how to do it on secure, but very restricted smartcard
  – 2-of-2, but with some trust assumptions, k-of-n ECDSA also possible

Threshold cryptography



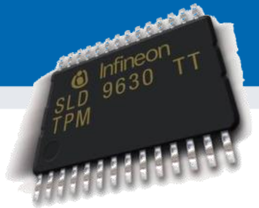
- Formulated by Yvo Desmedt [Crypto'87]
- Principle
  - Private key split into multiple parts ("shares")
  - Shares used (independently) by separate parties during a protocol to perform desired cryptographic operation
  - If enough shares are available, operation is finished successfully
- Properties
  - Better protection of private key (single point of failure removed)
  - Key shares can be distributed to multiple parties (check policy condition)
  - Resulting signature may be indistinguishable from a standard one (e.g., ECDSA)
- Significant research progress made in the cryptocurrency context

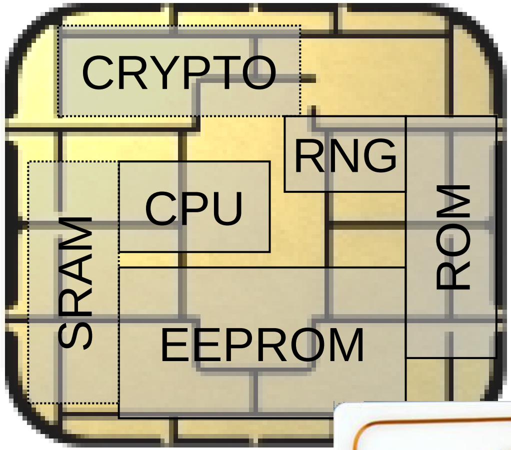
Threshold multiparty signatures

- “Easy” with RSA or Schnorr-based signatures
  - RSA: Shoup [EUROCRYPT’00] … Buldas et.al. [ESORICS’17] (Smart-ID, 4M users)
  - Schnorr: first threshold protocols already in 199x [CRYPTO’95]
    - But Schnorr sig. protocol was patented (expired 2010) => small overall uptake
    - Recent non-interactive and practical FROST [SAC’20], MuSig2 [CRYPTO’21]
- “Efficient” with pairing-based cryptosystems (BLS, non-interactive)
- Threshold variant of ECDSA is significantly more complicated
  - Significant breakthroughs and speedups recently (mainly due to cryptocurrencies)
  - Uses computing of Oblivious Linear Evaluation (OLE) for mult. of secret shares
    - Fully practical on common CPUs, but complicated on smartcards

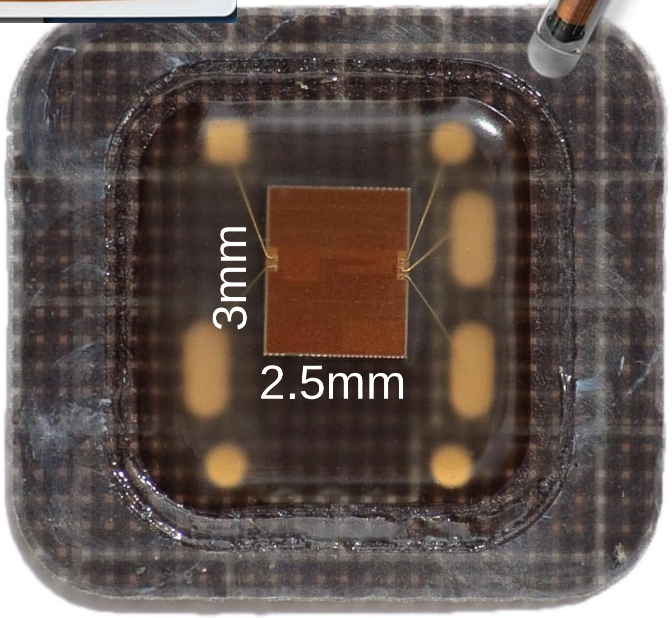
Credit Wikimedia Commons

Cryptographic smart cards





- SC is quite powerful device (for its size)
  - 8-32 bit processor @ 5-50MHz
  - persistent memory 32-200+kB (EEPROM)
  - volatile fast RAM, usually <<10kB
  - truly random generator, cryptographic coprocessors
  - JavaCard Virtual Machine (bytecode, but very slow)
- For environments where attacker has physical access
  - FIPS140-2/3, security Level 4; Common Criteria EAL4-6+
- 9-10 billion units shipped every year (EUROSMART)
- Three main things to wrestle with:
  - Lack of algorithmic and large types (BigInt) support on card
  - Low memory (~kBs of RAM) and CPU performance (@50MHz)
  - Restricted access to card's cryptographic API (no BigInt multiply)



(k-of-n) ECDSA (on smartcards)

- Single-party ECDSA directly supported by almost all modern smartcards
  – ALG_ECDSA_SHA, accelerated by dedicated coprocessor (scalar multiplication)
- k-of-n ECDSA requires multiplication of two (or more) secretly shared values computing Oblivious Linear Evaluation (OLE)
  – OLE is commonly constructed using:
    1. Oblivious transfer (OT), computation and high communication overhead
    2. Or partially homomorphic encryption (e.g., Paillier's cryptosystem)
       – Allows for hidden addition of plaintexts by multiplication of ciphertexts
       – Requires computation of numerous large public-key operations (~RSA)
       – Expensive range proofs required to prevent overflow/wrap-around attacks
- Deemed too expensive for smartcards ☹

Lindell's 2p-ECDSA (exactly two parties) [CRYPTO'17]

- Partially homomorphic encryption of inputs using Paillier's cryptosystem
  - But does not require costly zero-knowledge range proofs
  - Lindell's trick: in the two-party setting, the party that would otherwise need to verify the range proof can instead compose and verify the resulting signature
- Paillier's homomorphic encryption still too memory heavy for smartcard
  - Only few kilobytes in data transmission total (card communication speed relevant)
  - Decryption of Paillier's ciphertext can be accelerated using the modular exponentiation coprocessor available on smartcards
    - faster than if done on main slow CPU@50MHz) + few optimization trick

Responding Initiating

Lindell’s 2p-ECDSA on smartcards

- Lindell’s protocol is asymmetric
  - Initiating party only does one P.’s decryption
  - Responding party several operations with Paillier’s ciphertexts
- Make card to be initiating party
  - But… card only respond to requests
  - Synthetic message to start protocol

The Lin…-party protocol

Host(x…$_2$), n, g, m) **Sma**… …$, \lambda, \mu$)

$\xrightarrow{m}$

$k_2 \leftarrow\$ \mathbb{Z}_r$

$R_2 = k_2 G$

$\pi_2 = \mathrm{DLPoK}(k_2, G)$

$c = H(R_2, \pi_2)$

$\xleftarrow{c}$

$k_1 \leftarrow\$ \mathbb{Z}_r$

$R_1 = k_1 G$

$\pi_1 = \mathrm{DLPoK}(k_1, G)$

$\xrightarrow{R_1, \pi_1}$

verify π_1

$\xleftarrow{R_2, \pi_2}$

verify $c \stackrel{?}{=} H(R_2, \pi_2) \wedge \pi_2$

$R = k_1 R_2$

$\rho \leftarrow\$ \mathbb{Z}_r$

$\boxed{e = E(k_1^{-1} \cdot R_x \cdot x_1 \cdot x_2 + k_1^{-1} \cdot m + \rho \cdot r)}$

$\xrightarrow{e}$

$R = k_2 R_1$

$\boxed{s_1 = D(e)}$

$s = k_2^{-1} s_1 \mod r$

$\xleftarrow{s}$

verify s

What real cards openly support?

- Not large numbers operations
- Definitely not Paillier’s scheme ☺

- JCAlgTest project (since 2007) https://github.com/crocs-muni/JCAlgTest/
  – Support tested directly on real cards (100+)
  – Community-provided results in open db
  – RSA, ECDSA, ECDH… (called in full, no intermediate access or code change)

Feature	First in version	JC ⩽ 2.2.1 (21 cards)	JC 2.2.2 (26 cards)	JC 3.0.1/2 (12 cards)	JC 3.0.4 (29 cards)	JC 3.0.5 (11 cards)
Truly random number generator						
TRNG (ALG_SECURE_RANDOM)	⩽ 2.1	100%	100%	100%	100%	100%
Block ciphers used for encryption or MAC						
DES (ALG_DES_CBC_NOPAD)	⩽ 2.1	100%	100%	100%	100%	100%
AES (ALG_AES_BLOCK_128_CBC_NOPAD)	2.2.0	52%	96%	100%	100%	100%
KOREAN SEED (ALG_KOREAN_SEED_CBC_NOPAD)	2.2.2	5%	62%	75%	34%	0%
Public-key algorithms based on modular arithmetic						
1024-bit RSA (ALG_RSA(_CRT) LENGTH_RSA_1024)	⩽ 2.1	76%	96%	100%	93%	82%
2048-bit RSA (ALG_RSA(_CRT) LENGTH_RSA_2048)	⩽ 2.1	67%	96%	100%	93%	82%
4096-bit RSA (ALG_RSA(_CRT) LENGTH_RSA_4096)	3.0.1	0%	0%	0%	3%	0%
1024-bit DSA (ALG_DSA LENGTH_DSA_1024)	⩽ 2.1	5%	8%	8%	10%	0%
Public-key algorithms based on elliptic curves						
192-bit ECC (ALG_EC_FP LENGTH_EC_FP_192)	2.2.1	5%	62%	83%	66%	82%
256-bit ECC (ALG_EC_FP LENGTH_EC_FP_256)	3.0.1	0%	50%	75%	66%	82%
384-bit ECC (ALG_EC_FP LENGTH_EC_FP_384)	3.0.1	0%	12%	17%	62%	82%
521-bit ECC (ALG_EC_FP LENGTH_EC_FP_521)	3.0.4	0%	4%	8%	45%	82%
ECDSA SHA-1 (ALG_ECDSA_SHA)	2.2.0	24%	84%	100%	69%	82%
ECDSA SHA-2 (ALG_ECDSA_SHA_256)	3.0.1	5%	12%	100%	69%	82%
ECDH IEEE P1363 (ALG_EC_SVDP_DH)	2.2.1	29%	81%	100%	69%	82%
IEEE P1363 plain coord. X (ALG_EC_SVDP_DH_PLAIN)	3.0.1	5%	4%	67%	48%	82%
IEEE P1363 plain c. X,Y (ALG_EC_SVDP_DH_PLAIN_XY)	3.0.5	0%	0%	0%	17%	82%
Modes of operation and padding modes						
ECB, CBC modes	⩽ 2.1	100%	100%	100%	100%	100%
CCM, GCM modes (CIPHER_AES_CCM, CIPHER_AES_GCM)	3.0.5	0%	0%	0%	0%	0%
PKCS1, NOPAD padding	⩽ 2.1	95%	100%	100%	100%	100%
PKCS1 OAEP scheme (ALG_RSA_PKCS1_OAEP)	⩽ 2.1	14%	31%	8%	41%	82%
PKCS1 PSS sheme (ALG_RSA_SHA_PKCS1_PSS)	3.0.1	14%	19%	83%	41%	100%
ISO14888 padding (ALG_RSA_ISO14888)	⩽ 2.1	14%	12%	8%	0%	0%
ISO9796 padding (ALG_RSA_SHA_ISO9796)	⩽ 2.1	81%	100%	100%	86%	100%
ISO9797 padding (ALG_DES_MAC8_ISO9797_M1/M2)	⩽ 2.1	90%	100%	100%	100%	100%
Hash functions						
MD5 (ALG_MD5)	⩽ 2.1	90%	77%	92%	62%	0%
SHA-1 (ALG_SHA)	⩽ 2.1	95%	100%	100%	100%	100%
SHA-256 (ALG_SHA_256)	2.2.2	14%	88%	100%	97%	100%
SHA-512 (ALG_SHA_512)	2.2.2	5%	23%	25%	90%	100%
SHA-3 (ALG_SHA3_256)	3.0.5	0%	0%	0%	0%	0%

…fied in JavaCard API. For a given feature, the *version* column specifies … the subsequent columns show its availability in cards reporting particular …ethod and maximally supported version of the *javacard.framework* pack-… n were not included.

P. Svenda, R. Kvasnovsky, I. Nagy, A. Dufka: JCAlgTest: Robust identification metadata for certified smartcards https://crocs.fi.muni.cz/papers/jcalgtest_secrypt22

JCMathLib: open JavaCard library for low-level operations

- JCMathLib extended to support Paillier's decryption
  - JCMathLib typically works with ~256bit integers, Paillier decryption uses integers up to 4096b
  - Will (typically) not fit into RAM, needs utilization of card's persistent storage
- Modular exponentiation in current smartcards support at most 4096-bit moduli ☹
  - Not enough as we need larger modulo operations (for n=4096)
  - Chinese remainder theorem (CRT) used to split into smaller steps with knowledge of **`n=p*q`**
    - With CRT, plaintext first computed with p and q separately then Garner-style combination
  - Lindell's protocol with 256-bit curve (ECDSA) and 4096-bit n (Paillier), valid plaintexts are always smaller than the factors (**`p`** and **`q`**) => combination is unnecessary => computation performed with only one factor (say **`p`**)

Performance on NXP JCOP4 J3R180

- Full Lindell's 2-of-2 protocol (with 4096-bit Paillier)
  - Low trust requirements (everything computed on-card)
  - Requires 5932 ms of on-card computation

	Supported setups	Trust required	Performance
Lindell's protocol [22]	2-out-of-2	low	5932 ms
Multiplication triples	2-out-of-2	medium	1800 ms
Presignatures	t-out-of-n	high[8]	203 ms

- Heavily utilizes (open-source) JCMathLib library
  - significant further speedup possible if native access to platform is available (especially the last step)

Smartcard$(x_2, X, n, g, \lambda, \mu)$

$\xrightarrow{m}$ $k_2 \leftarrow\$ \mathbb{Z}_r$

627ms

$R_2 = k_2 G$

$\pi_2 = \mathrm{DLPoK}(k_2, G)$

$c = H(R_2, \pi_2)$

$\xleftarrow{c}$

$\xrightarrow{R_1, \pi_1}$

186ms verify π_1

$\xleftarrow{R_2, \pi_2}$

$\xrightarrow{e}$ $R = k_2 R_1$

5119ms

$s_1 = D(e)$

$s = k_2^{-1} s_1 \mod r$

$\xleftarrow{s}$ verify s

FURTHER SPEEDUPS

Approach 2: Two-party protocol with multiplication triples

- The most expensive part is multiplication of secretly shared values
  - Speedup using well-known Beaver's trick (additive secret shares triples)
    - Independent of the multiplied values => no need for secrets knowledge
    - Generated by trusted dealer or in distributed way by SPDZ [ESORICS'13]
- Adaptation of Dalskov et al. [ESORICS'20] k-ECDSA to 2-of-2
- Offload of multiplication triples from card to host using authenticated encryption (to overcome card's limited memory)
- 1800ms sign

	Supported setups	Trust required	Performance
Lindell's protocol [22]	2-out-of-2	low	5932 ms
Multiplication triples	2-out-of-2	medium	1800 ms
Presignatures	t-out-of-n	high[8]	203 ms

Approach 3: Presignatures

- Most efficient way of constructing threshold ECDSA signatures
  - Pre-signature: $[k^{-1}]$ and $[z]=[k^{-1}\cdot x_i]$ and x-coordinate r of point $R=[k]G$
  - Final signature (r,s) for message m : $[s] = [k^{-1}]\cdot H(m) + [z]$ (203ms)
- Who will compute presignatures?
  1. Very efficiently by trusted dealer given access to all parties' key shares (x_i)
  2. Trust-minimized distributed protocol for generating presignatures (e.g., Doerner et al. [EPRINT'23] – but if smartcard involved => very long computation
  3. Usage of Beaver's multiplication triples to speedup distributed computation of presignatures for option 2. (tradeoff between trust and speed)
- Allows for generic t-of-n ECDSA! (t,n fixed during precomputation)

Conclusions

- Even 2-of-2 ECDSA signatures have many practical usage scenarios
  - Less parties involved => higher requirements for share protection
  - Yet execution on smartcard previously deemed (very ☺) impractical
- Careful selection of a protocol, ordering of parties and operations used make it efficiently computable even without proprietary interfaces
  - 2-of-2 ECDSA, Paillier's decryption on-card (for up to 256-bits values)
- Open-source code available
  - Fully on card: https://github.com/crocs-muni/JC2pECDSA/
  - With precomputations: https://github.com/crocs-muni/JCPreECDSA/

Questions 

References

[CRYPTO’95] Langford, S.K. Threshold DSS Signatures without a Trusted Party. In CRYPTO’ 95. LNCS 963. Springer

[EUROCRYPT’00] Shoup, V. Practical Threshold Signatures. In EUROCRYPT 2000. LNCS 1807. Springer.

[PKC’03] A. Boldyreva, Threshold Signatures, Multisignatures and Blind Signatures Based on the Gap-Diffie-Hellman-Group Signature Scheme. In PKC 2003. LNCS 2567. Springer.

[ESORICS’13] Damgård, I., Keller, M., Larraia, E., Pastro, V., Scholl, P., Smart, N.P.: Practical covertly secure MPC for dishonest majority– or: breaking the SPDZ limits. In ESORICS 2013, Springer.

[ACNS’16] Gennaro, R., Goldfeder, S. and Narayanan, A., Threshold-optimal DSA/ECDSA signatures and an application to bitcoin wallet security. In ACNS 2016, Springer.

[CRYPTO’17] Lindell, Y., Fast secure two-party ECDSA signing. In Crypto 2017. Springer.

[ESORICS’17] Buldas, A., Kalu, A., Laud, P., Oruaas, M. Server-Supported RSA Signatures for Mobile Devices. In ESORICS 2017. LNCS 10492. Springer.

[SAC’20] Komlo, C., Goldberg, I. FROST: Flexible Round-Optimized Schnorr Threshold Signatures. In Selected Areas in Cryptography. SAC 2020. LNCS 12804. Springer.

[ESORICS’20] Dalskov, A., Orlandi, C., Keller, M., Shrishak, K., Shulman, H.: Securing DNSSECkeys via threshold ECDSA from generic MPC. In ESORICS2020, Springer (2020).

[CRYPTO’21] Nick, J., Ruffing, T., Seurin, Y. MuSig2: Simple Two-Round Schnorr Multi-signatures. In CRYPTO 2021. LNCS 12825. Springer.

[EPRINT’23] Doerner, J., Kondi, Y., Lee, E., et al.: Threshold ECDSA in three rounds. Cryptology ePrint Archive (2023)