<https://www.securetechalliance.org/resources/pdf/OPACITY_Overview%203.8.pdf>

OPACITY: Open Protocol for Access Control and Ticketing with PrivacY

Modification of ECDSA?

Protocol suite for authentication and key agreement optimized for contactless transactions.

NIST SP 800-56A compliant.

Only one command/response pair

Relies on Elliptic Curve Cryptography ECDH/ECDSA, as well as AES and SHA256.

Supports 2 protocols:

* Zero Key management (ZKM aka SMAv3), doesn’t require any secrets stored on a terminal.
* Forward Secrecy (FS) (or is it Full Secrecy?) requires static PKI credential stored on terminal, but enhanced privacy protection

Reference implementation provides Secure Authentication Module (SAM) applet/library. SAM is smartcard device embedder in the terminal. (For storing secrets)

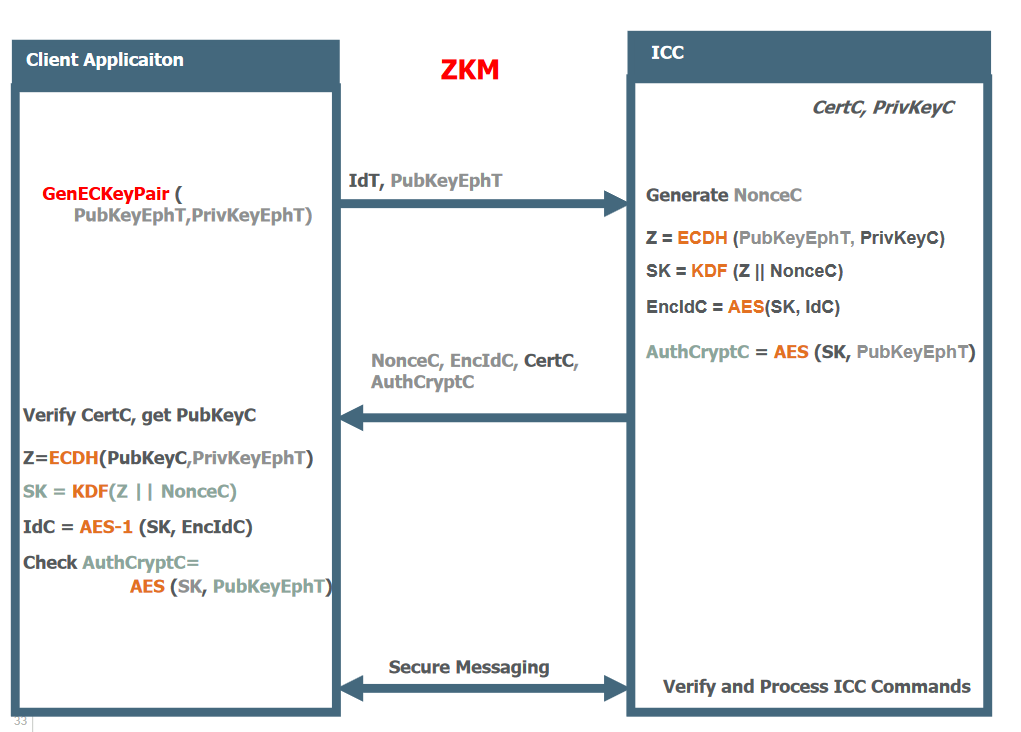
ZKM doesn’t require a SAM. Only requires root public key of CVC Digital Signatory to be protected by the terminal. (rather than MASTER and private keys). SAM could still be used to store public key.

SAM software library can never exceed fips140-2 level 1. Applet on specialised hardware can reach fips140-2 level 3.

SAM activation: SAM operations protected by an access control rule. Admin operations protected by GlobalPlatform secure channels. Usage is protected by PIN/OPACITY based auth by control panel to SAM. Policy-based deactivation of SAM makes it lose its state.

Go with ZKM.

Persistent Binding: pre-session keys cached by User applet and Terminal applet, allows for faster transactions for that user card next time.



Result of joint effort by HID global and the US DoD.

Full secrecy – ensure identity of cardholder is not compromised. Many readers (esp. old readers) can’t contain secrets, don’t have dedicated security module.

ZKM provides card authentication but not terminal authentication and should only be used where terminals are known and trusted. Do any/all Cambridge terminals not have a SAM?

Opacity protocols are a single command-response transaction. (What would generic EC algo look like?)

OPACITY keys not strictly defined. Assumed they are unique authentication keys likely generated on the ICC and never shared.

ZKM is mainly an ICC internal authentication protocol and key agreement using EC cryptography.

Key agreement:

* Key establishment prerequisite must be executed on both sides prior to executing the protocol. Each party has copy of same set of domain parameters and obtain assurance of these parameters.
* Prerequisites for C(1,1) ECC CDH mode applied. Requirements of one pass Diffie-Hellman must be applied.
* Key derivation function is concatenation of KDF, specified in NIST SP800-56A 5.8.1.
* Key confirmation follows NIST SP 800-56A 8.4.9

Initial host state:

* Client app on host that can implement crypto functions.
* Access to registry of host-ICC pairing records to support persistent binding capability. Each record includes ICC record identifier (key) and one-time shared secret valid for next communication.
* If auth settings are variable, client ensures auth method and parameters are set.
* Etc.

Initial ICC state:

* CVC root public key for on-card verification of client CVC. May be determined by client ID number.
* Access to registry of host pairing records needed for persistent binding. Includes host ID, one-time ICC ID used last session, and one-time secret valid for next session.
* Access to card auth key, card CVC
* Execute establishment preparations prior to launching protocol defined in NIST 800-56A figure 1

Protocol:

Base protocol client-side:

* Generate ephemeral key pair. SAM does GEN\_KEY\_PAIR(DeH;QeH), returns QeH. Ephemeral means only used for this session. Do this via elliptic curve crypto.
* Send ICC auth request IDsH||QeH.

ICC:

* IDsC = T8(HASH(CC)) (may already be computed)
* Validate QeH belongs to EC domain.
* Z = EC\_DH(dC;QeH)
* Generate nonce NC.
* SKCFRM||SKMAC||SKENC||SKRMAC||NextZ = KDF(Z, len, info(IDsC, IDsH, T16(QeH), NC))
* Zeroize Z. (?)
* Etc, not going to copy entire thing. Page 31 of OPACITY spec.

Found python ecdsa package. Should I look into fastecdsa? Potentially 25x faster but appears to use Unix-specific library so maybe can’t have it on Windows. C++ implementation about 30-40x faster than ecdsa. Look into optimisation as optional extension.

DSA faster to sign, slower to verify than RSA. Good because card does signing. (True?)

Note: talk about quantum computing implications for this (and other) algorithms.

Use ECDSA 256, SHA 256, AES 128, 16B nonces.

C\_icc format from annex B opacity spec:

* 1B Credential Profile Identifier = 0x80
* 8B Certificate issuer identification number = IssuerID (6B) || IssuerKeyID (2B). IssuerID a set number e.g. 6B of hash of name. keyID is signature verification public key number.
* L1 <= 10B Globally Unique Identifier. Application specific, identify card/cardholder.
* L2 CardHolderPublicKey. 76B total (2+8+2+65) 6B object identifier of algorithm, see annex for possible values. 86B public key encoded as 04||X||Y where (X,Y) are coordinates on curve.
* L3 DigitalSignature
* 1B RoleIdentifier. Type of key this is.

Single-step **Key Derivation Function** spec:

input includes stared secret Z and other info. Kdf(Z, …)

Auxiliary hash function H – either approved hash(x) function (800-56A 5.1 and 5.8.1), or HMAC-hash(salt, x) function, instantiation of HMAC defined in FIPS 198, employing approved hash. Salt could be protocol nonce, shared value, or pre-determined constant.

Input: Z, byte string representing shared secret z. keydatalen: length of secret keying material. Less than or equal to hashlen x (232-1).

Process:

* Repetitions = keydatalen / hashlen
* Initialise 32b counter 0x00000001.
* For i=1 to reps, do:
  + Compute K(i)=H(counter||Z||otherInfo)
  + Increment counter
* If (keydatalen/hashlen) is integer, K\_last = K(reps), else set to (keydatalen mod hashlen) leftmost bits of K(reps).
* derivedKeyingMaterial = K(1)||K(2)||…||K(reps-1)||K\_last

Output derivedKeyingMaterial of length keydatalen.

Optimisations defined in section 5. Perhaps complete as optional extension.

PLAID:

<https://www.humanservices.gov.au/organisations/about-us/publications-and-resources/protocol-lightweight-authentication-identity-plaid>

Best not use it: <https://www.schneier.com/blog/archives/2015/10/weaknesses_in_t.html>

NIST 800-73-4:

<http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-73-4.pdf>

Some other protocol published 2004:

<https://link.springer.com/content/pdf/10.1007%2F1-4020-8143-X_23.pdf>

Points from the paper:

Previously the main prohibitor to smartcard public key cryptography was limited card processing power.

* [www.dice.ucl.ac.be/cascade](http://www.dice.ucl.ac.be/cascade)
* Generation of prime numbers or pairs for elliptic curves was slow in software.

Public key based architecture requires existence of public key infrastructure (PKI) [ISOIIEC 11770-3, "Information technology -- Security techniques – Key management -- Part 3: Mechanisms using asymmetrie techniques", ISO 1999]

Most public key cryptography secure channel protocols not specifically designed for smart cards. E.g. limited communication buffers usually between 190-255B so the number of messages exchanged should be minimal.

GlobalPlatform:

* Card specification. Provides functionality e.g. key management/storage etc.
* Secure multi-application card management.
* Closely coupled with Java Card technology (but no necessary dependency)
* Secure channel – mechanism allowing card and host to authenticate each other and establish session keys to protect subsequent communication. GlobalPlatform spec defines 2 protocols for this: SCP01 and SCP02. Both symmetric protocols.
* GlobalPlatform has Security Domains. On-card representative of card issuer or app provider. Allows issuers to share control over portion of the card with approved partners. Security domains also responsible for cryptographic functions and key handling

Note: FIPS140-2 is a US government security standard for cryptographic modules.