Massively Parallel Hardware Security Platform

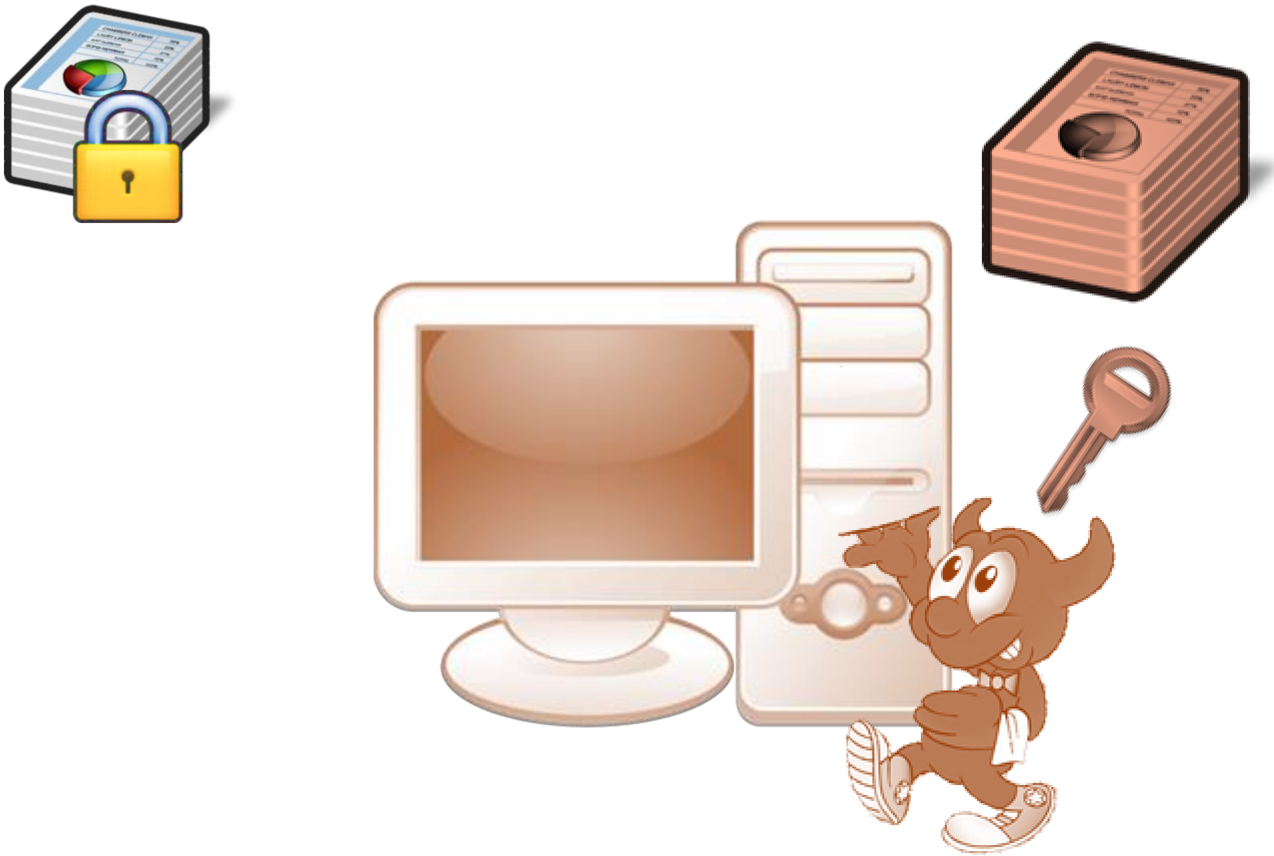
Dan Cvrček, Enigma Bridge, UK
dan@enigmabridge.com
Petr Švenda, CRoCS, Masaryk University, CZ
svenda@fi.muni.cz

Centre for Research on Cryptography and Security

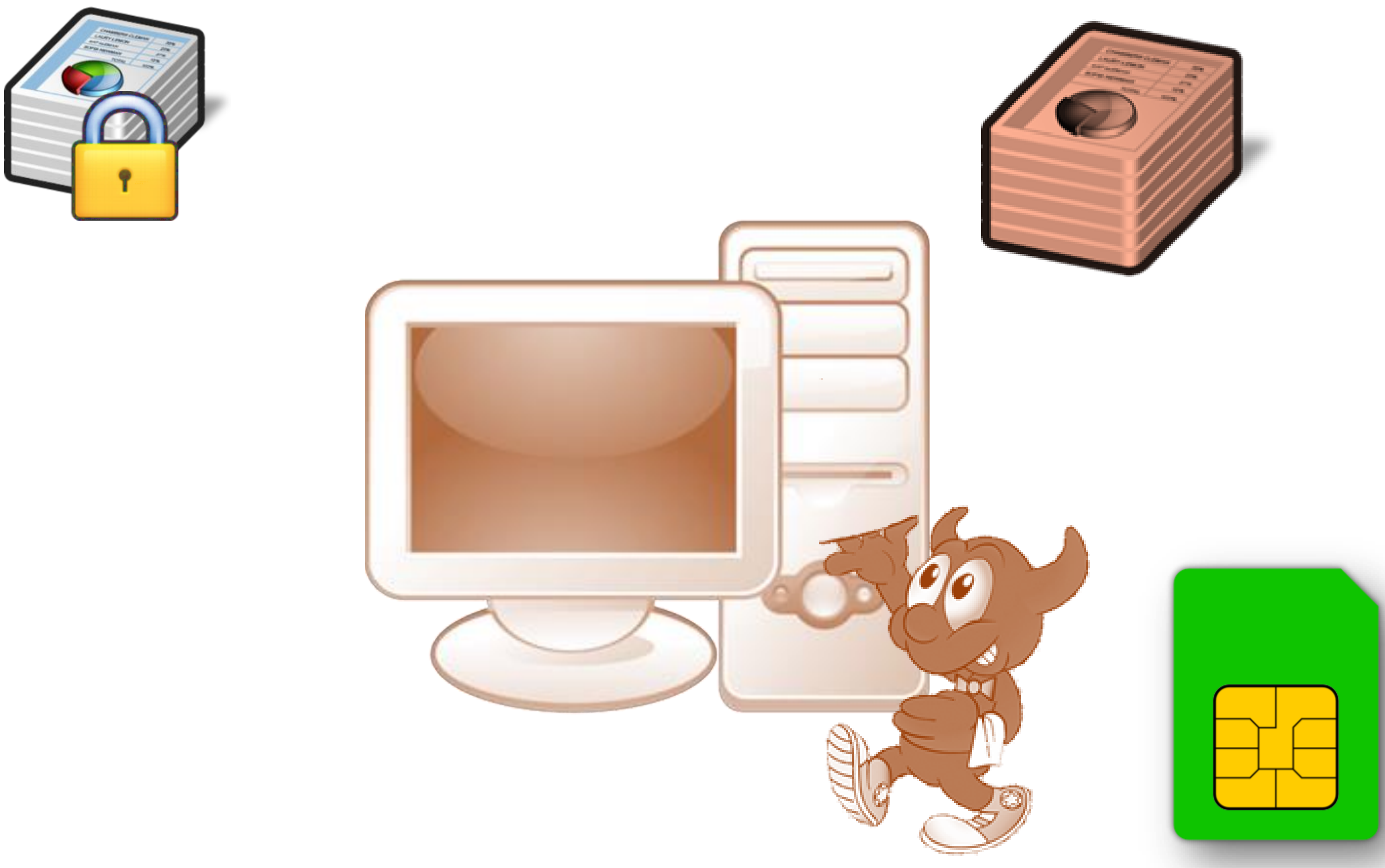
Overview

1. Cryptography as a Service
2. Usage scenarios, implication for hardware
3. Options for computational platforms
4. Secure parallel multi-processor design
5. Prototype results and experience
6. Open issues

Cryptography on client



On client, but with secure hardware



Is this enough?

- You don’t need full performance?
- Have peak demands in performance?
- What if you already use cloud servers?

- Cryptography as a Service (CaaS)

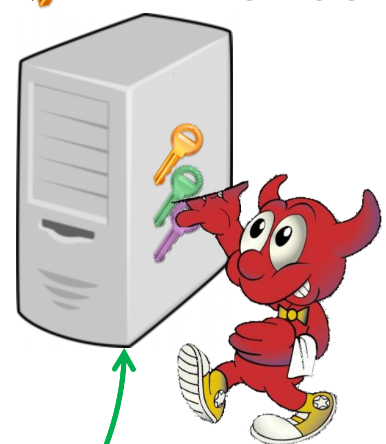
Offloading security operations…

… into secured environment

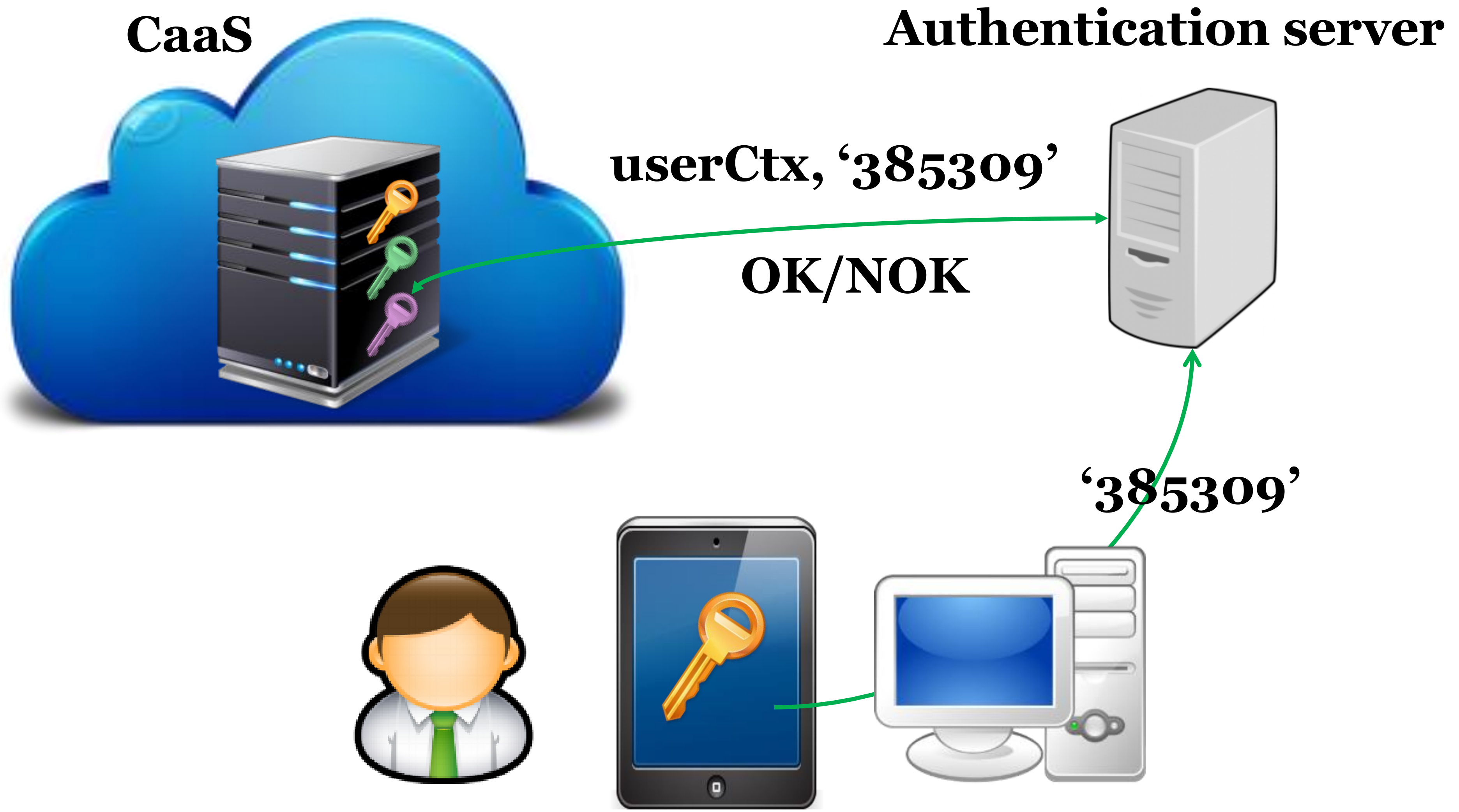
How to import key(s) securely?
Which hardware platform to use?
What about high number of clients?

HMAC-based One-Time Password

- Improves protection of client side
- Increases risk at Auth. server

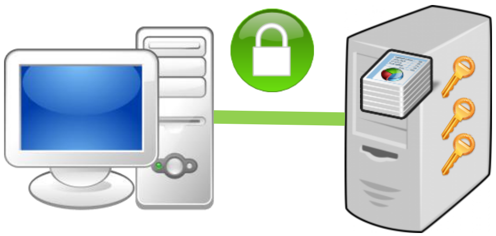


HOTP with CaaS



HOTP = HMAC(ctr++, 🔑) = '385309'

Different levels of trust



- CaaS with trusted CaaS provider
  - Software operation only, HTTPS for in/out
  - Trust to provider => valid target, insider attack…

- CaaS with semi-trusted CaaS provider
  - HTTPS for in/out, decrypted by server
  - Data sent for processing into trusted hardware
  - CaaS platform still target (data visible)

- CaaS with untrusted provider
  - HTTPS for in/out + inner protection
  - Data decrypted/processed/encrypted inside device

Problem scoping

Requirements – client view

- Untrusted CaaS provider (handling secrets)
- Secure import of app's secrets - enrollment
- Client $\leftrightarrow$ CaaS communication security
  - Confidentiality/integrity of input and output data
  - Authentication of input/output requests
- Key use control
  - Use constraints – e.g., number of allowed ops
- Easy recovery from client-side compromise

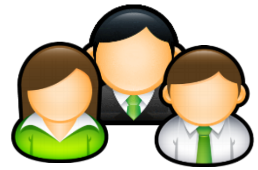
Requirements – CaaS provider view

- Massive scalability
  - W.r.t. users, keys, transactions…
- Low latency of responses
- Robust audit trail of key usage
- Tolerance and recovery from failures
  - hardware/software failures
- Easy to use API
  - also easy to use securely

Usage scenarios

Steps of cryptographic operation

1. Transfer input data
2. Transfer wrapped key in
3. Initialize unwrap engine
4. Unwrap data/key (decrypt/verify)
5. Initialize key object with key value
6. Initialize cryptographic engine with key
7. Start, execute and finalize crypto operation
8. Initialize wrap engine
9. Wrap data/key (encrypt/sign)
10. Erase key(s)/engine(s)
11. Transfer output data
12. Transfer wrapped key out

Usage scenarios (users vs. keys)

- S1: One user, few keys
  - No sharing, all engines fully prepared
- S2: One user, many keys
  - No sharing, frequent crypto context change
- S3: Few users, few keys
  - Device is shared → isolation of users
- S4:Few users, many keys
  - Limited sharing, frequent crypto context change
- S5: Many users, many keys
  - High sharing, frequent crypto context change

S1: One user, few keys

- No sharing, all engines fully prepared

1. Transfer input data
2. Transfer wrapped key in
3. Initialize unwrap engine
4. Unwrap data/key (decrypt/verify)
5. Initialize key object with key value
6. Initialize cryptographic engine with key
7. Start, execute and finalize crypto operation
8. Initialize wrap engine
9. Wrap data/key (encrypt/sign)
10. Erase key(s)/engine(s)
11. Transfer output data
12. Transfer wrapped key out

S5: Many users, many keys

- High sharing, frequent crypto context change

1. Transfer input data
2. Transfer wrapped key in
3. Initialize unwrap engine
4. Unwrap data/key (decrypt/verify)
5. Initialize key object with key value
6. Initialize cryptographic engine with key
7. Start, execute and finalize crypto operation
8. Initialize wrap engine
9. Wrap data/key (encrypt/sign)
10. Erase key(s)/engine(s)
11. Transfer output data
12. Transfer wrapped key out

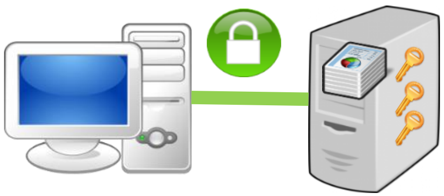
Frequent exchange of cryptographic context → implications for computation platform

Platform options for CaaS

Performance perspective

- Use of general-purpose hardware (CPU/GPU)
  - Great code base and library support
- Use of generic programmable hardware (FPGA)
  - Flexible for new algorithms, fast reconfiguration
- Use of dedicated cryptographic circuits (ASIC)
  - Fastest, but fixed to pre-specified design

Security perspective

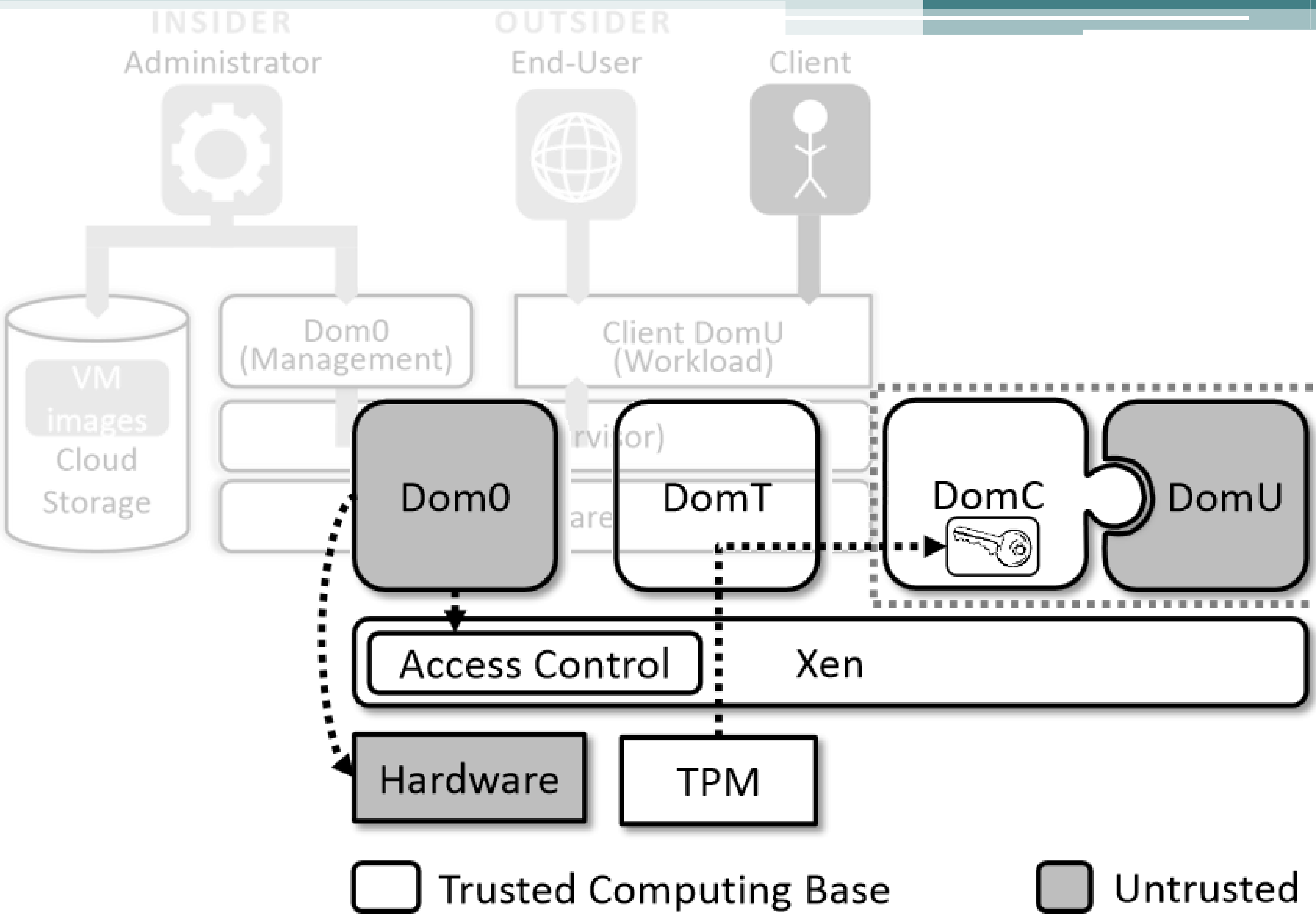


- Fully trusted provider
  - No additional protection of data/code
  - (Additional tamper protection of device)

- Use of secure hardware
  - Trusted boot (TPM-based)
  - Intel's Software Guard Extensions (SGX)
  - Use of Hardware Security Module (HSM)
- Use of software protection techniques
  - Fully homomorphic encryption
  - (promising, but not fully practical yet)

Client-controlled CaaS in the Cloud

- Bleikertz et. al., 2013 (IBM, TU Darmstadt)
- Protection against attacker on logical level
    - Administrator without physical access
- Modification of Xen hypervisor by standard Trusted Computing (based on TPM)
    - Establishment of a separate security-domain (DomC) for critical cryptographic operations
- (No protection against attacker with physical access)



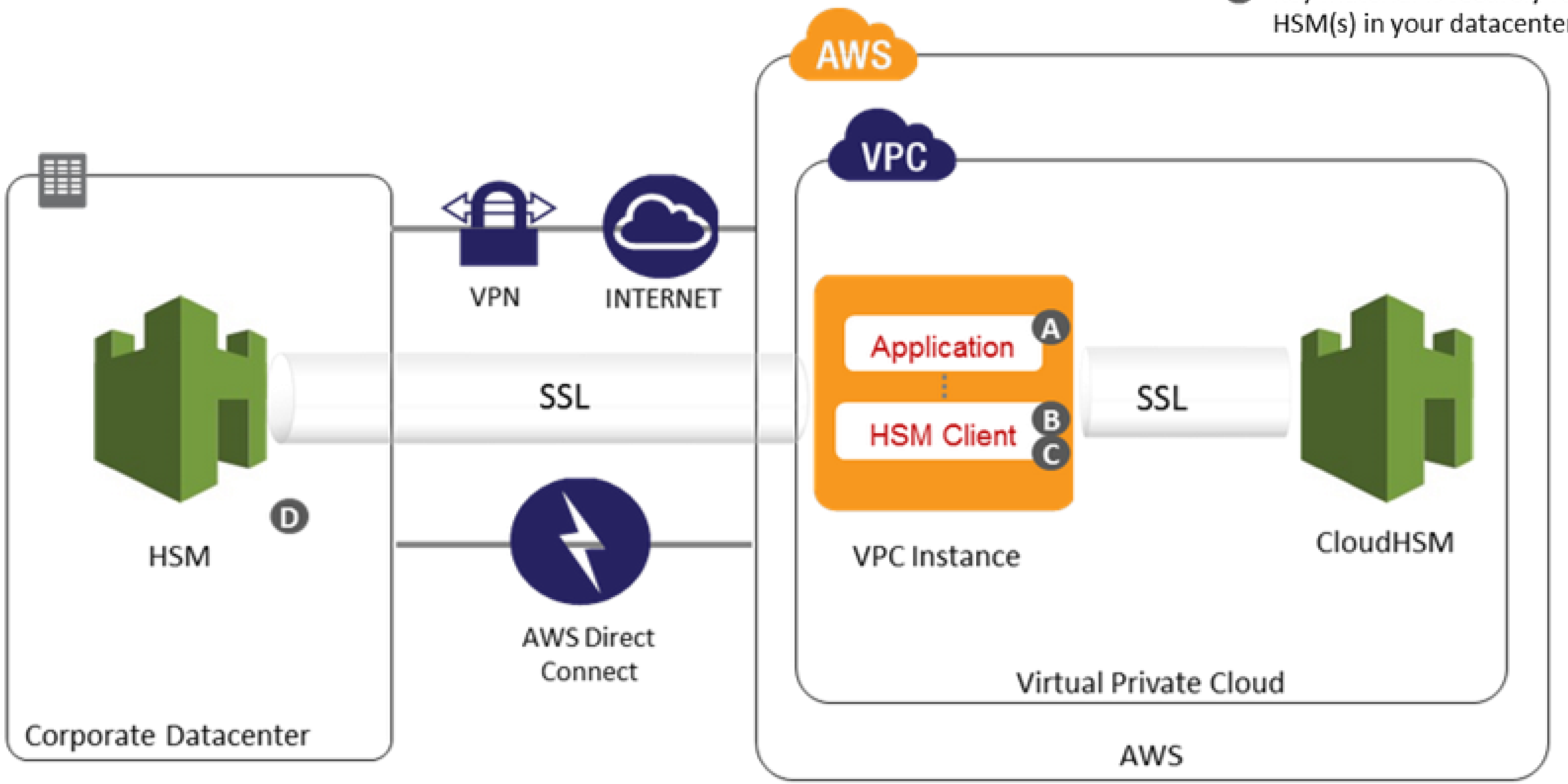
https://www.infsec.cs.uni-saarland.de/~bugiel/publications/pdfs/bugiel13-acns.pdf

Cloud service with HSM

- Hardware Security Module (HSM)
  - Hardened secure device (tamper protection…)
  - Cryptographic accelerators (9000 RSA1024/sec)
- Example: AWS CloudHSM
  - Dedicated HSM (SafeNet Luna) in AWS cloud
  - Pricing (2015-09-16)
    - $5000 upfront, $1.88 per hour
- Possibility for custom firmware plugins
  - But not possible to move generic app inside HSM

Example: AWS CloudHSM

- **A** Your applications continue to use standard crypto APIs (PKCS#11, MS CAPI, JCA/JCE, etc.).
- **B** SafeNet HSM client replaces existing crypto service provider libraries and connects to the HSM to implement API calls in hardware
- **C** SafeNet HSM Client can share load and store keys redundantly across multiple HSMs
- **D** Key material is securely replicated to HSM(s) in your datacenter

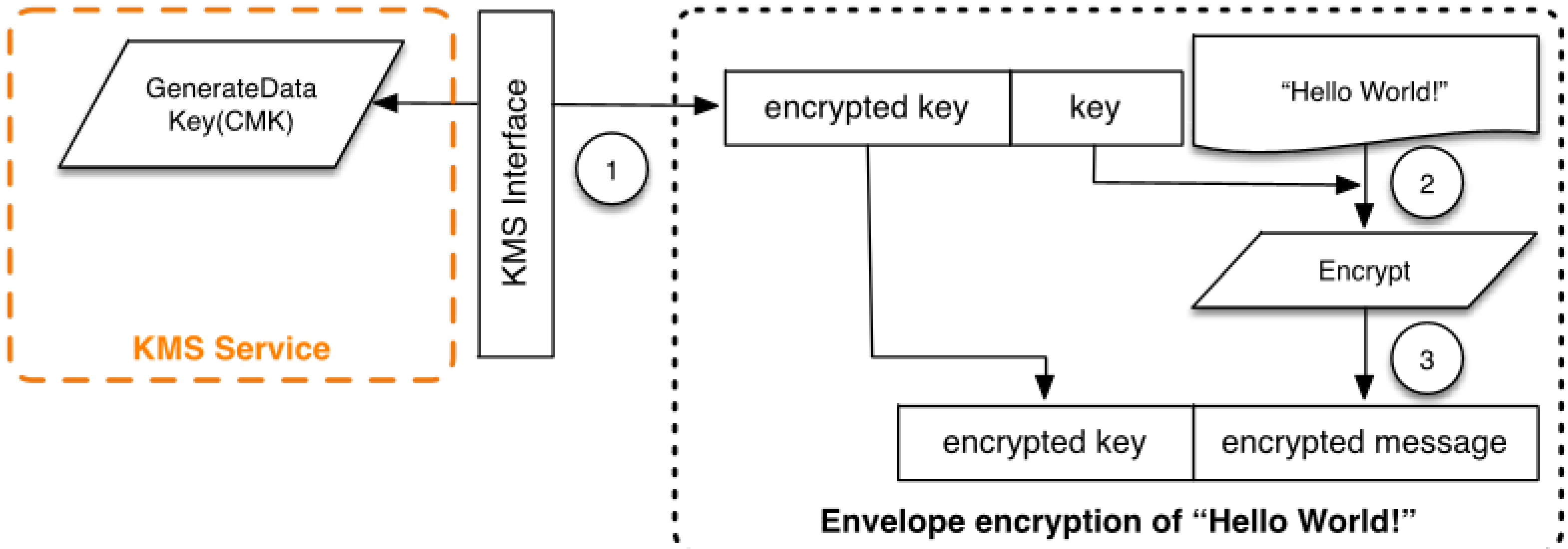


https://aws.amazon.com/cloudhsm/details/

Cloud service with HSM-based KMS

- HSM used only to provide key management service
  - Key generation and distribution center
- Protection of keys, not application/data itself
  - Still running in standard computation platform
- Example: AWS Key Management Service
  - User master key stored inside multiple HSM(s)
  - New key(s) generated for data blobs as needed
  - Wrapped by master key for transfer between HSM(s)
  - Transfer of necessary keys between different AWS locations
  - Pricing (2015-09-16): $0.03 per 10,000 requests

Example: AWS Key Management Service



https://d0.awsstatic.com/whitepapers/KMS-Cryptographic-Details.pdf

Security enclave via Intel's SGX

- New set of CPU instructions intended for future cloud server CPUs
- Protection against privileged attacker
  - server admin with physical access, privileged malware
- Application requests private region of code and data
  - Security enclave (4KB for heap, stack, code)
  - Encrypted enclave is stored in main RAM memory, decrypted only inside CPU
  - Access from outside enclave is prevented on CPU level
  - Code for enclave is distributed as part of application

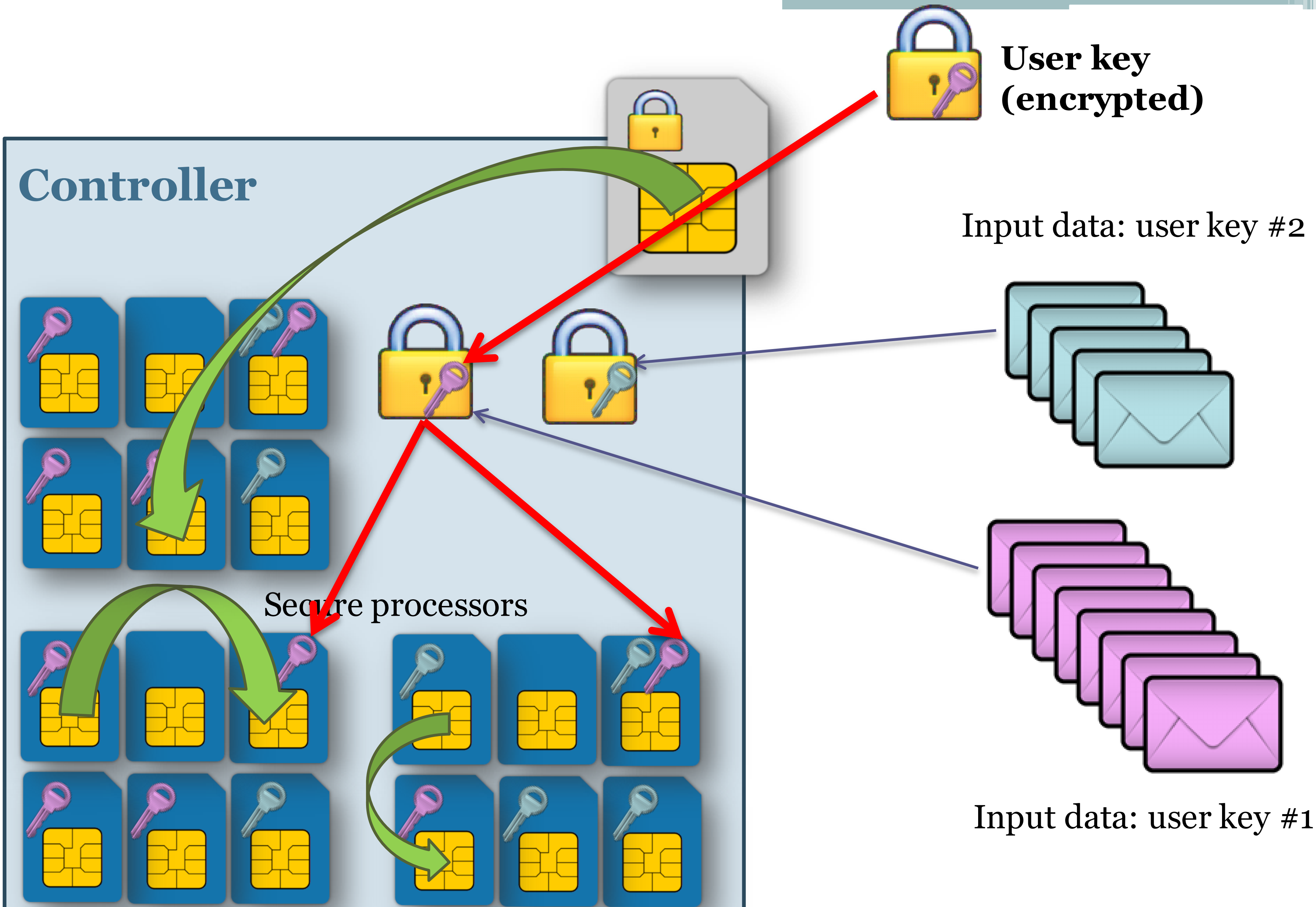
Intel's SGX – some details

- EGETKEY instruction generates new enclave key
  - SGX security version numbers
  - Device ID (unique number of CPU)
  - Owner epoch – additional entropy from user
    - ? Why not also random part generated inside CPU?
- EREPORT instruction generates signed report
  - Local/remote attestation of target platform
- Debugging possible if application opt in
- Enclave cannot be emulated by VM

Secure multi-processor

Secure parallel multi-processor

1. High number of secure processors (100s-10000s)
   - Secure memory, secure execution, crypto engines
   - FIPS140-2 Level 3/4, CC EAL 5+
2. Small trusted computing base
   - Everything outside facilitated in untrusted controller
3. Secure channels between secure processors
4. Technical and logical structure to facilitate:
   - Efficient requests processing
   - Efficient inter-key distribution
5. High robustness due to high redundancy
   - If one processor locks or dies, another serves a request


User key
(encrypted)
Controller
Input data: user key #2
Secure processors
Input data: user key #1

Do we have such secure processors?

- Cryptographic smartcards

1. Programmable, secure runtime environment
2. Dedicated cryptographic coprocessors
3. Secure on-card TRNG generator
4. Secure on-card storage (but limited in size)
5. Reasonable price per unit
6. High-level of tamper protection (FIPS140-2...)

Cryptographic operations

- Supported algorithms (JCAlgTester, 43+ cards)
  - https://github.com/crocs-muni/JCAlgTest

javacard.security.MessageDigest	introduced in JavaCard version	c0	c1	c2	c3	c4	c5	c6	c7
ALG_SHA	<=2.1	yes	yes	yes	yes	yes	yes	yes	yes
ALG_MD5	<=2.1	no	yes	yes	yes	yes	yes	yes	no
ALG_RIPEMD160	<=2.1	no	no	no	yes	yes	yes	no	no
ALG_SHA_256	2.2.2	yes	no	no	suspicious yes	yes	no	no	yes
ALG_SHA_384	2.2.2	no	no	no	no	no	no	no	yes
ALG_SHA_512	2.2.2	no	no	no	no	no	no	no	yes
ALG_SHA_224	3.0.1	no	-	-	-	no	no	no	no
javacard.security.RandomData	**introduced in JavaCard version**	**c0**	**c1**	**c2**	**c3**	**c4**	**c5**	**c6**	**c7**
ALG_PSEUDO_RANDOM	<=2.1	yes	yes	yes	yes	yes	yes	yes	yes
ALG_SECURE_RANDOM	<=2.1	yes	yes	yes	yes	yes	yes	yes	yes

Common algorithms

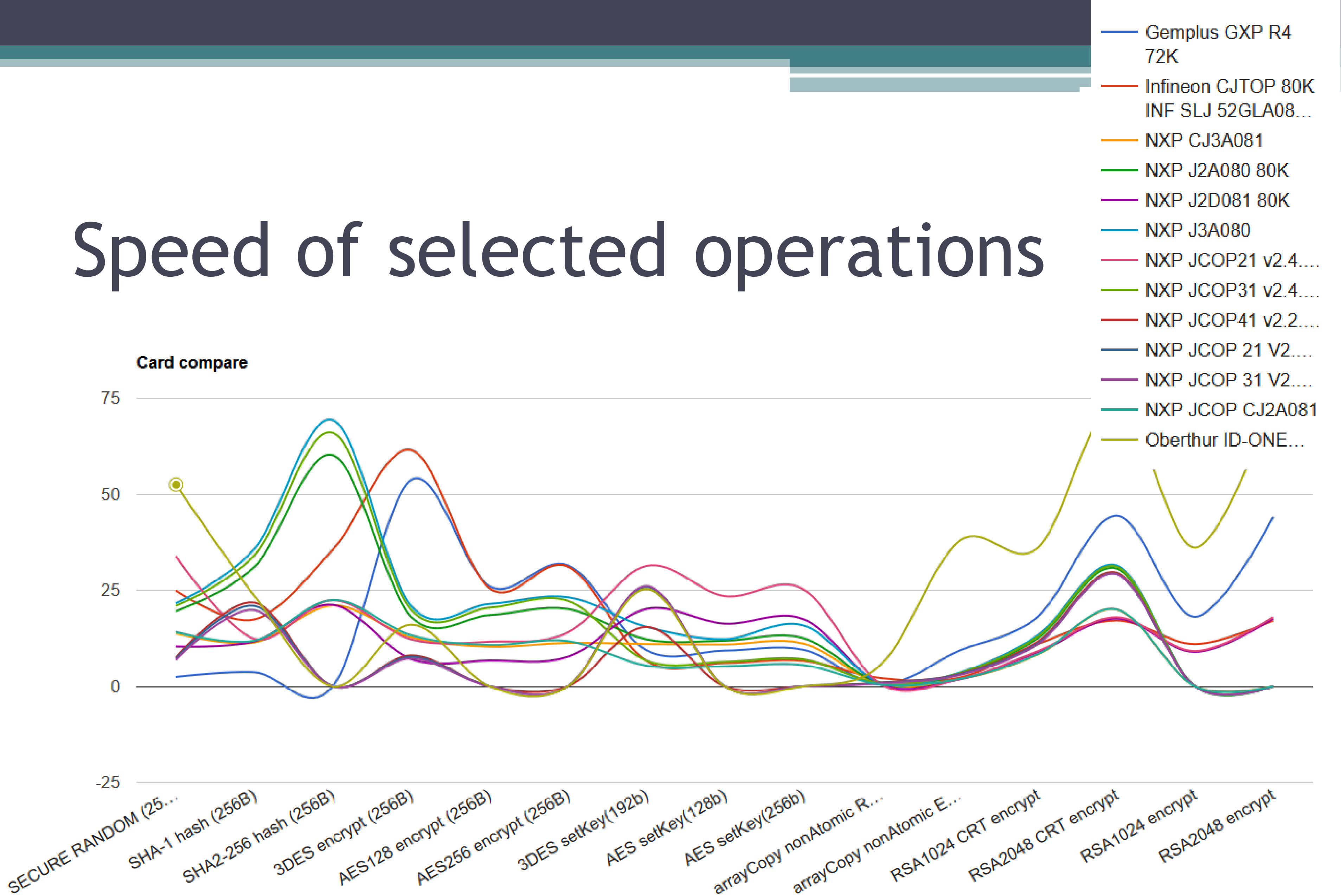
- Basic - cryptographic co-processor
  - TRNG
  - 3DES, AES128/256
  - MD5, SHA1, SHA-2 256/512
  - RSA (up to 2048b common, 4096 possible)
  - ECC (up to 192b common, 384b possible)
  - Diffie-Hellman key exchange
- Composite crypto operations (JavaCard VM)
  - Custom code running in secure environment
  - E.g. HMAC, OTP code, re-encryption

What is the performance?

- https://github.com/crocs-muni/JCAlgTest **(ms)**
- (excerpt from large tables, will be public soon)
- 256B of data processed

CARD/FUNCTION (ms/op)	SECURE RANDOM	SHA-1 hash	SHA2-256 hash	AES128 encrypt
NXP J2D081 80K	10.4	11.73	21.18	6.73
NXP CJ3A081	13.8	11.45	21.05	10.33
NXP JCOP CJ2A081	14.14	11.9	22.46	10.78
NXP JCOP21 v2.4.2R3	33.77	12.35	22.39	11.65
NXP J2A080 80K	19.59	31.09	60.16	18.57
NXP JCOP31 v2.4.1 72K	20.97	34.1	66.02	20.44
NXP J3A080	21.64	35.78	69.32	21.41
Infineon CJTOP 80K INF SLJ 52GLA080AL M8.4	24.9	17.42	35.58	25.53
Gemplus GXP R4 72K	2.45	3.69	-	26.05

Speed of selected operations



What is the performance?

- (Raw performance of crypto engines)

Card type	AES-128 CBC encrypt	RSA-1024 sign	RSA-2048 sign
NXP CJ2A081 (2012)	36.5kB/sec	10.5 signs/sec	2.3 signs/sec
Infineon CJTOP 80K (2012)	10.2kB/sec	9.8 signs/sec	4.1 signs/sec
NXP CJ3A080 v2.4.1 (2011)	17.6kB/sec	6.3 signs/sec	1.6 signs/sec
Gemalto GXP R4 72K (2008)	10.8kB/sec	2.5 signs/sec	0.6 signs/sec
NXP JCOP4.1 v2.2.1 72K (2008)	N/A	9.3 signs/sec	1.6 signs/sec

Recall: steps of cryptographic operation

1. Transfer input data
2. Transfer wrapped key in
3. Initialize unwrap engine
4. Unwrap data/key (decrypt/verify)
5. Initialize key object with key value
6. Initialize cryptographic engine with key
7. Start, execute and finalize crypto operation
8. Initialize wrap engine
9. Wrap data/key (encrypt/sign)
10. Erase key(s)/engine(s)
11. Transfer output data
12. Transfer wrapped key out

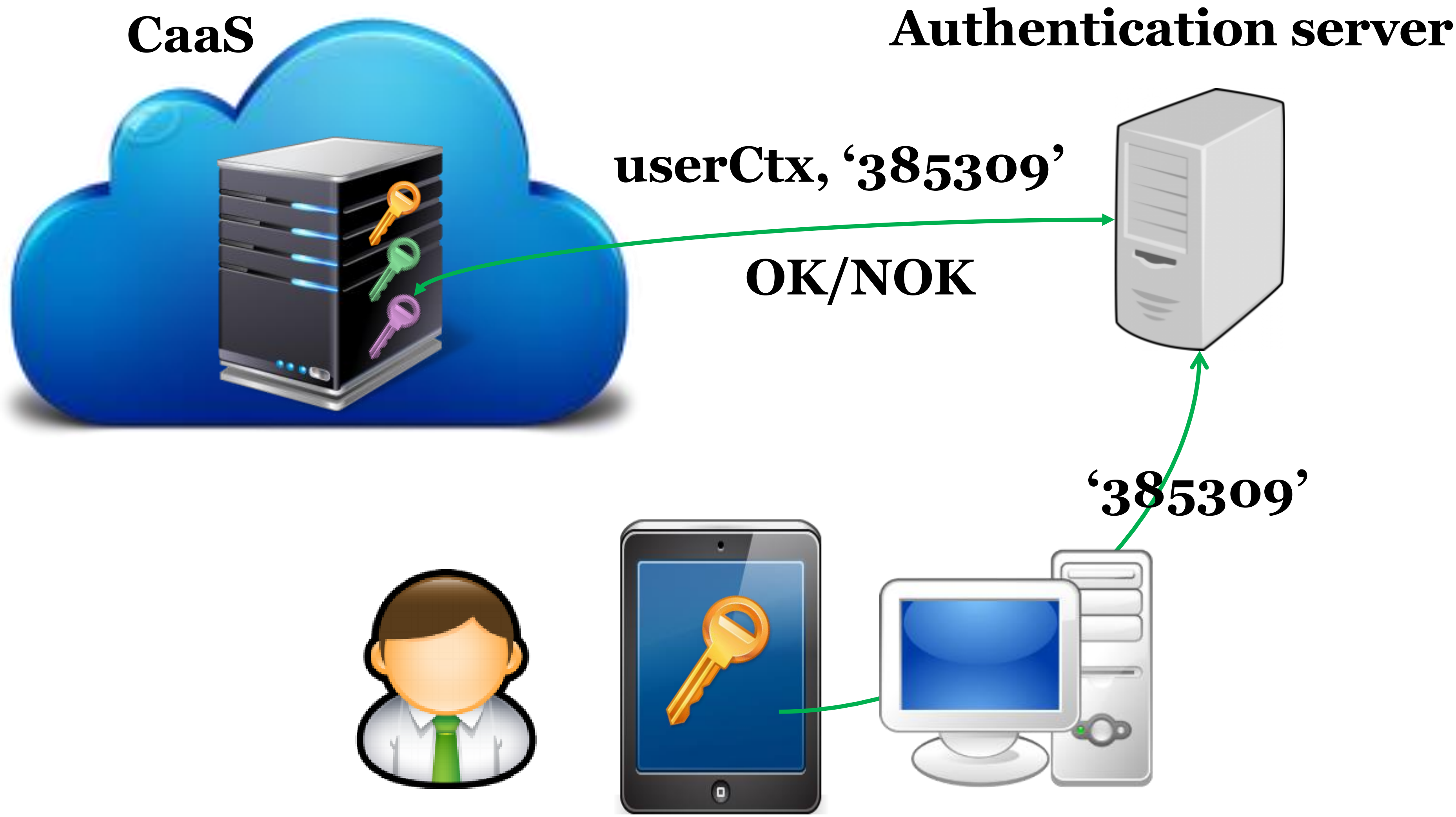
Crypto context change is a problem

CARD/FUNCTION (ms/op)	AES setKey(128b)	AES128 init	AES128 encrypt
NXP JCOP CJ2A081	5.22	11.56	10.78
Infineon CJTOP 80K INF SLJ 52GLA080AL M8.4	6.08	2.85	25.53
NXP JCOP31 v2.4.1 72K	6.38	12.34	20.44
NXP JCOP21 v2.4.2R3	23.48	11.62	11.65

CARD/FUNCTION (ms/op)	AES setKey(128b)	AES128 init	AES128 encrypt
NXP JCOP CJ2A081	5.22	11.56	10.78

- E.g., theoretical AES128 speed → 36.5KB/s
  - complete engine init + encrypt 256B → 10.4KB/s
- Performance penalty factor up to 100x for small blocks on some cards!

Recall: HOTP with CaaS



HOTP = HMAC(ctr++,) = ‘385309’

Verify HOTP (OAuth)

Operation	Length (bytes)	Clean call	Repeat call
Verify HOTP code	**I/O:157/66B**	**288ms**	**134ms**
1. Transfer authentication server context, input data and user state into card	5+88+40+24	34ms	34ms
2. Unwrap authentication server context – use: $K_{authServerCtxEnc}$ and $K_{authServerCtxMAC}$	88	14ms	14ms
3. Unwrap user state (HOTP counter, failed attempts, settings, HMAC key) – prepare&use: $K_{stateEnc}$ and $K_{stateMAC}$	40	65ms	11ms
4. Unwrap input data (HOTP code provided by user) – prepare&use: $K_{commEnc}$ and $K_{commMAC}$	24	63ms	10ms
5. Compute HMAC&truncation over current value of counter obtained from user state– prepare&use: K_{auth}	-	20ms	20ms
6. att		4ms	4ms
7. ca K		33ms	10ms
8. $K_{stateMAC}$		36ms	12ms
9. Transfer output data and user state outside card	40+24+2	19ms	19ms

Length (bytes)	Clean call	Repeat call
I/O:157/66B	**288ms**	**134ms**

How to minimize contexts change?

1. Cache keys/engines on card

Type of object	NXP CJ2A081	NXP CJ2D081 80K	NXP JCOP21 v2.4.2R3 145KB
AESKey 128	877	729	678
AESKey 256	658	607	565
DESKey 196	748	607	565
Cipher AES	79	74	74
Cipher DES	147	136	136
RSA CRT PRIVATE 1024	72	93	86
RSA PRIVATE 1024	203	152	141
RSA CRT PRIVATE 2048	61	51	47
RSA PRIVATE 2048	108	82	77

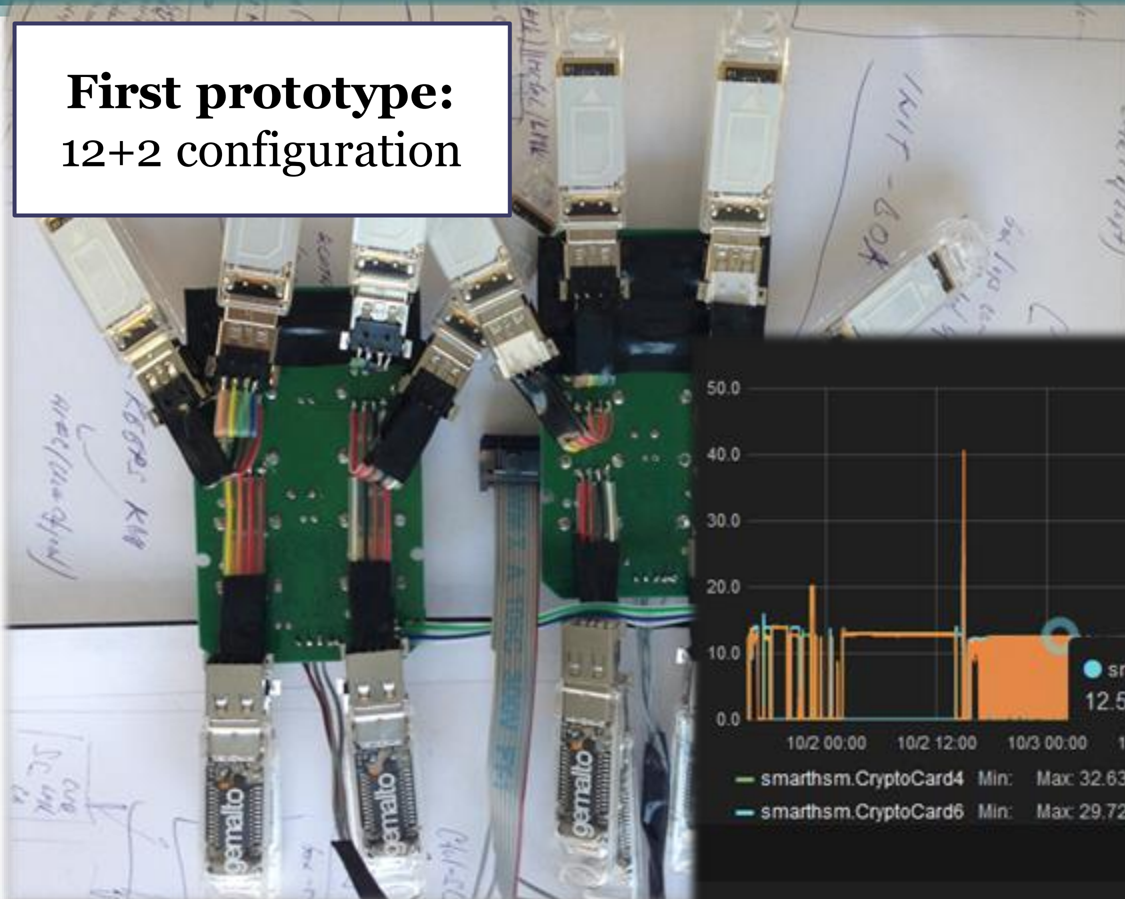
How to minimize contexts change?

2. Proper load-balancing on the controller side
   - Which card should serve the request?
   - When context should be removed from card?
   - How required throughput can be guaranteed?

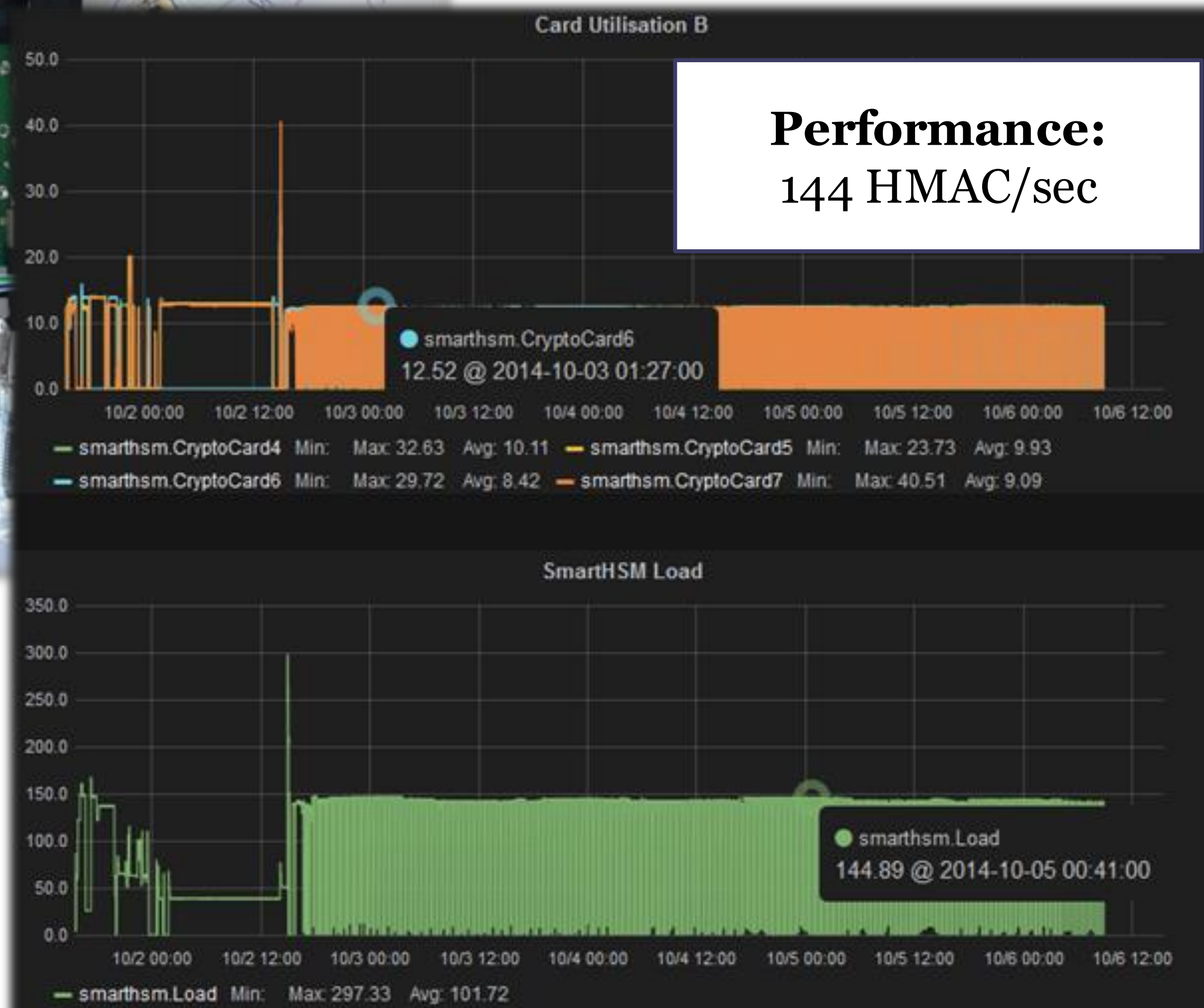
ENIGMA BRIDGE

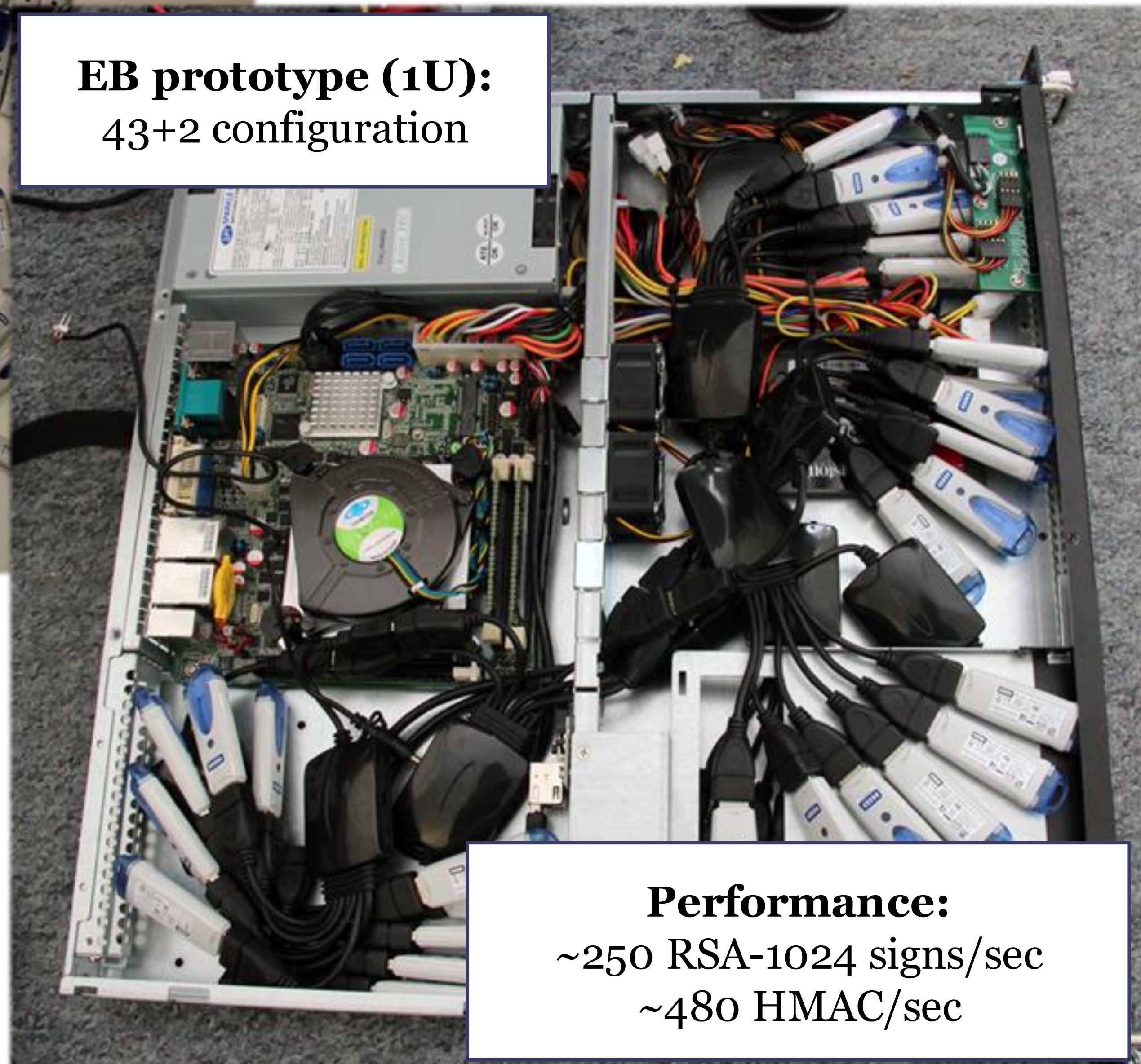
Building the device

First prototype:
12+2 configuration



Performance:
144 HMAC/sec





EB prototype (1U):
43+2 configuration

Performance:
~250 RSA-1024 signs/sec
~480 HMAC/sec

Not just "send and encrypt fast"

- Device is shared – load/unload user ctxs
- Protection of incoming/outgoing data
  - Additional crypto context initializations
- Hierarchical control of loaded keys
  - Efficient secure distribution of keys
- User specifies limited use for its key (credits)
  - No more then specified uses allowed
- Signed audit trail collected from processors
  - independently verifiable, control over uses

Some interesting problems

Some development issues

- Many common sw/hw components fail when used in uncommon “extreme” settings
  - Many readers/cards used, high peak load, long-term usage…
- Task should be processed in given time frame
  - Assigned card may fail to deliver result
  - Several timeouts must be implemented

Some development issues

- 1000 / 1 < 1000 / 11
  - Adding more cards may not speed up anything
  - just one smart card - 93s to process 1000 packets
  - 11 smart cards - 123s required instead
- "Thread hell" inevitable
  - Serialized assignment of tasks is inefficient
  - Task assignment must be highly parallelized
- Lock-free programming
  - Prevent by hard lock or detect and respond?

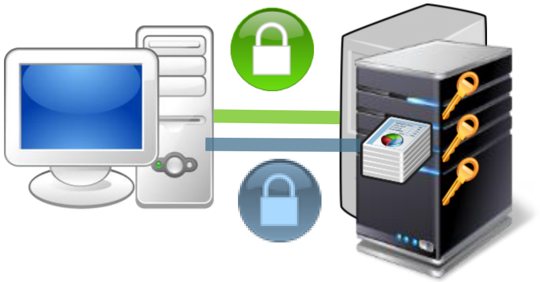
Freshness of distributed state

- Counter, time/logical time, challenge/response…
- Freshness of data blobs, when:
  - Secure processors can't communicate too often,
  - can't store too much (limited memory)
  - and controller is not trusted
- Inter-device communication via secure channel
- User-to-device communication

Trade communication for initialization

- Initializing crypto engines introduces overhead
  - setKey, initEngine, startEngine → 10-30ms
  - communication is faster → 5-15B/ms
- Secure crypto schemes which trades (higher) data transfer for simpler operation (on card)
  - E.g., send precomputed keystream as an input

Platform / functionality	Performance of crypto functions	Perf. of crypto fncs (many users/keys)	Performance of generic functions	Protection of application keys	Protection of application data	Protection against side-channel attacks
Generic HW	✓✓	✓✓	✓✓✓	✗	✗	✗
Trusted boot	✓✓	✓✓	✓✓✓	✓	✓	✗
Intel GCX	✓✓	✓✓	✓✓	✓✓	✓✓	✗
HSM	✓✓✓	✓✓	✓✓	✓✓✓	✓✓✓	✓✓✓
Parallel smart cards	✓✓✓	✓✓✓	✓	✓✓✓	✓✓✓	✓✓✓



Conclusions

- Cryptography as a Service
  - Data/keys moved to untrusted provider
  - Different hardware platforms available
  - Different performance vs. security tradeoff
- Usage scenario is important for performance
  - Number of users, number of keys
  - Frequency of key exchanges
- Highly parallel grid of secure processors
  - High performance and scalability
  - Based on secure cryptographic smartcards

ENIGMA BRIDGE

Thank you for your attention!

Questions

Contact me at svenda@fi.muni.cz