Massively Parallel Hardware Security Platform

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

WS API: JSON
... 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

\[
\text{HOTP} = \text{HMAC}(\text{ctr}++,\text{\text{'}}) = '385309'\]

\[
\text{HMAC}(\text{ctr}++,\text{\text{'}}) == '385309'?\]
HOTP with CaaS

CaaS

Authentication server

userCtx, ‘385309’

OK/NOK

‘385309’

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 ↔ 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
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 cryptographic operation
8. Initialize wrap engine
9. Wrap data/key (encrypt)
10. Erase key(s)/engine(s)
11. Transfer output data
12. Transfer wrapped output key

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)
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
Controller

Secure processors

User key (encrypted)

Input data: user key #1

Input data: user key #2
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](https://github.com/crocs-muni/JCAlgTest)

<table>
<thead>
<tr>
<th>javacard.security.MessageDigest</th>
<th>introduced in JavaCard version</th>
<th>c0</th>
<th>c1</th>
<th>c2</th>
<th>c3</th>
<th>c4</th>
<th>c5</th>
<th>c6</th>
<th>c7</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALG_SHA</td>
<td>&lt;=2.1</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>ALG_MD5</td>
<td>&lt;=2.1</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>ALG_RIPEMD160</td>
<td>&lt;=2.1</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>ALG_SHA_256</td>
<td>2.2.2</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>ALG_SHA_384</td>
<td>2.2.2</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>ALG_SHA_512</td>
<td>2.2.2</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>ALG_SHA_224</td>
<td>3.0.1</td>
<td>no</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>javacard.security.RandomData</th>
<th>introduced in JavaCard version</th>
<th>c0</th>
<th>c1</th>
<th>c2</th>
<th>c3</th>
<th>c4</th>
<th>c5</th>
<th>c6</th>
<th>c7</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALG_PSEUDO_RANDOM</td>
<td>&lt;=2.1</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>ALG_SECURE_RANDOM</td>
<td>&lt;=2.1</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>
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](https://github.com/crocs-muni/JCAlgTest) (ms)
- (excerpt from large tables, will be public soon)
- 256B of data processed

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

- (Raw performance of crypto engines)

<table>
<thead>
<tr>
<th>Card type</th>
<th>AES-128 CBC encrypt</th>
<th>RSA-1024 sign</th>
<th>RSA-2048 sign</th>
</tr>
</thead>
<tbody>
<tr>
<td>NXP CJ2A081 (2012)</td>
<td>36.5kB/sec</td>
<td>10.5 signs/sec</td>
<td>2.3 signs/sec</td>
</tr>
<tr>
<td>Infineon CJTOP 80K (2012)</td>
<td>10.2kB/sec</td>
<td>9.8 signs/sec</td>
<td>4.1 signs/sec</td>
</tr>
<tr>
<td>NXP CJ3A080 v2.4.1 (2011)</td>
<td>17.6kB/sec</td>
<td>6.3 signs/sec</td>
<td>1.6 signs/sec</td>
</tr>
<tr>
<td>Gemalto GXP R4 72K (2008)</td>
<td>10.8kB/sec</td>
<td>2.5 signs/sec</td>
<td>0.6 signs/sec</td>
</tr>
<tr>
<td>NXP JCOP4.1 v2.2.1 72K (2008)</td>
<td>N/A</td>
<td>9.3 signs/sec</td>
<td>1.6 signs/sec</td>
</tr>
</tbody>
</table>
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/sigin)
10. Erase key(s)/engine(s)
11. Transfer output data
12. Transfer wrapped key out
Crypto context change is a problem

- 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

CaaS

UserCtx, ‘385309’

OK/NOK

Authentication server

‘385309’

HOTP = HMAC(ctr++, ,) = ‘385309’
<table>
<thead>
<tr>
<th>Operation</th>
<th>Length (bytes)</th>
<th>Clean call</th>
<th>Repeat call</th>
</tr>
</thead>
<tbody>
<tr>
<td>Verify HOTP code</td>
<td>I/O:157/66B</td>
<td>288ms</td>
<td>134ms</td>
</tr>
<tr>
<td>1. Transfer authentication server context, input data and user state into card</td>
<td>5+88+40+24</td>
<td>34ms</td>
<td>34ms</td>
</tr>
<tr>
<td>2. Unwrap authentication server context – use: $K_{authServerCtxEnc}$ and $K_{authServerCtxMAC}$</td>
<td>88</td>
<td>14ms</td>
<td>14ms</td>
</tr>
<tr>
<td>3. Unwrap user state (HOTP counter, failed attempts, settings, HMAC key) – prepare&amp;use: $K_{stateEnc}$ and $K_{stateMAC}$</td>
<td>40</td>
<td>65ms</td>
<td>11ms</td>
</tr>
<tr>
<td>4. Unwrap input data (HOTP code provided by user) – prepare&amp;use: $K_{commEnc}$ and $K_{commMAC}$</td>
<td>24</td>
<td>63ms</td>
<td>10ms</td>
</tr>
<tr>
<td>5. Compute HMAC&amp;truncation over current value of counter obtained from user state – prepare&amp;use: $K_{auth}$</td>
<td>-</td>
<td>20ms</td>
<td>20ms</td>
</tr>
<tr>
<td>6. att</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7. call $K_a$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8. $K_{stateMAC}$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9. Transfer output data and user state outside card</td>
<td>40+24+2</td>
<td>19ms</td>
<td>19ms</td>
</tr>
</tbody>
</table>
How to minimize contexts change?

1. Cache keys/engines on card

<table>
<thead>
<tr>
<th>Type of object</th>
<th>NXP CJ2A081</th>
<th>NXP CJ2D081 80K</th>
<th>NXP JCOP21 v2.4.2R3 145KB</th>
</tr>
</thead>
<tbody>
<tr>
<td>AESKey 128</td>
<td>877</td>
<td>729</td>
<td>678</td>
</tr>
<tr>
<td>AESKey 256</td>
<td>658</td>
<td>607</td>
<td>565</td>
</tr>
<tr>
<td>DESKey 196</td>
<td>748</td>
<td>607</td>
<td>565</td>
</tr>
<tr>
<td>Cipher AES</td>
<td>79</td>
<td>74</td>
<td>74</td>
</tr>
<tr>
<td>Cipher DES</td>
<td>147</td>
<td>136</td>
<td>136</td>
</tr>
<tr>
<td>RSA CRT PRIVATE 1024</td>
<td>72</td>
<td>93</td>
<td>86</td>
</tr>
<tr>
<td>RSA PRIVATE 1024</td>
<td>203</td>
<td>152</td>
<td>141</td>
</tr>
<tr>
<td>RSA CRT PRIVATE 2048</td>
<td>61</td>
<td>51</td>
<td>47</td>
</tr>
<tr>
<td>RSA PRIVATE 2048</td>
<td>108</td>
<td>82</td>
<td>77</td>
</tr>
</tbody>
</table>
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?
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 contexts
- 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 than 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
<table>
<thead>
<tr>
<th>Platform / functionality</th>
<th>Performance of crypto functions</th>
<th>Perf. of crypto fncts (many users/keys)</th>
<th>Performance of generic functions</th>
<th>Protection of application keys</th>
<th>Protection of application data</th>
<th>Protection against side-channel attacks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generic HW</td>
<td>🟢</td>
<td>🟢</td>
<td>🟢</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Trusted boot</td>
<td>🟢</td>
<td>🟢</td>
<td>🟢</td>
<td>🟢</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Intel GCX</td>
<td>🟢</td>
<td>🟢</td>
<td>🟢</td>
<td>🟢</td>
<td>🟢</td>
<td>✗</td>
</tr>
<tr>
<td>HSM</td>
<td>🟢</td>
<td>🟢</td>
<td>🟢</td>
<td>🟢</td>
<td>🟢</td>
<td>🟢</td>
</tr>
<tr>
<td>Parallel smart cards</td>
<td>🟢</td>
<td>🟢</td>
<td>🟢</td>
<td>🟢</td>
<td>🟢</td>
<td>🟢</td>
</tr>
</tbody>
</table>
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
Thank you for your attention!

Questions

Contact me at svenda@fi.muni.cz