Trusted Execution Environments…

…for Privacy-preserving Cloud Applications

Institut d'informatique
Université de Neuchâtel

Prof. P. Felber
Pascal.Felber@unine.ch
http://iiun.unine.ch/

Cloud computing and security

- Cloud is an appealing paradigm
  - Cost savings due to sharing (economies of scale)
  - Easy/ubiquitous access to data
  - Widely applicable: IaaS, PaaS, SaaS, DaaS, ?aaS
  - Affordable for SMEs
- Tempting to attack
  - Resources are accessible online, remotely
  - One service provider holds data from multiple companies
  - Financial gain from selling/trading sensitive data

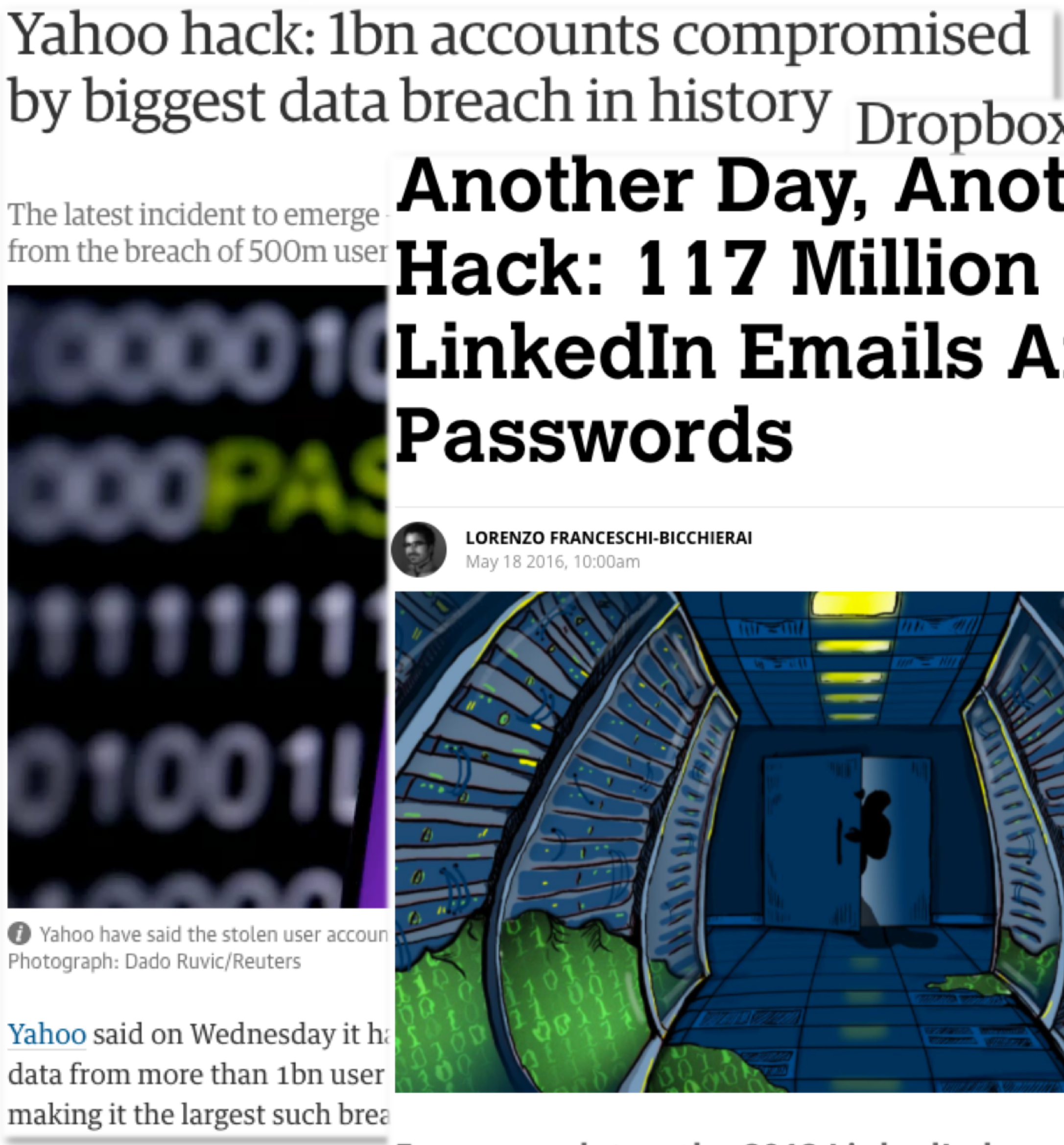
Why is data so important?

- Data is key asset for businesses
    - Moving data offsite is inherent security risk
- Storing data in the cloud
    - Encryption helps
- Using data stored in the cloud
    - Hard if data is encrypted
    - Some tasks (e.g., queries, matching) possible
- Processing data in the cloud
    - Transformations of encrypted data is very hard
    - Cryptographic techniques are not practical (yet)

Securing data

- Challenges
  - Data should be searchable, range queries
  - Must not leak information (e.g., statistical attack knowing the distribution of values)
  - Tradeoffs between functionality, performance and confidentiality, privacy
- Tools
  - Encryption
    - Deterministic or non-deterministic, order-preserving, homomorphic
  - Trusted computing (e.g., SGX)

Why is Cloud security important?

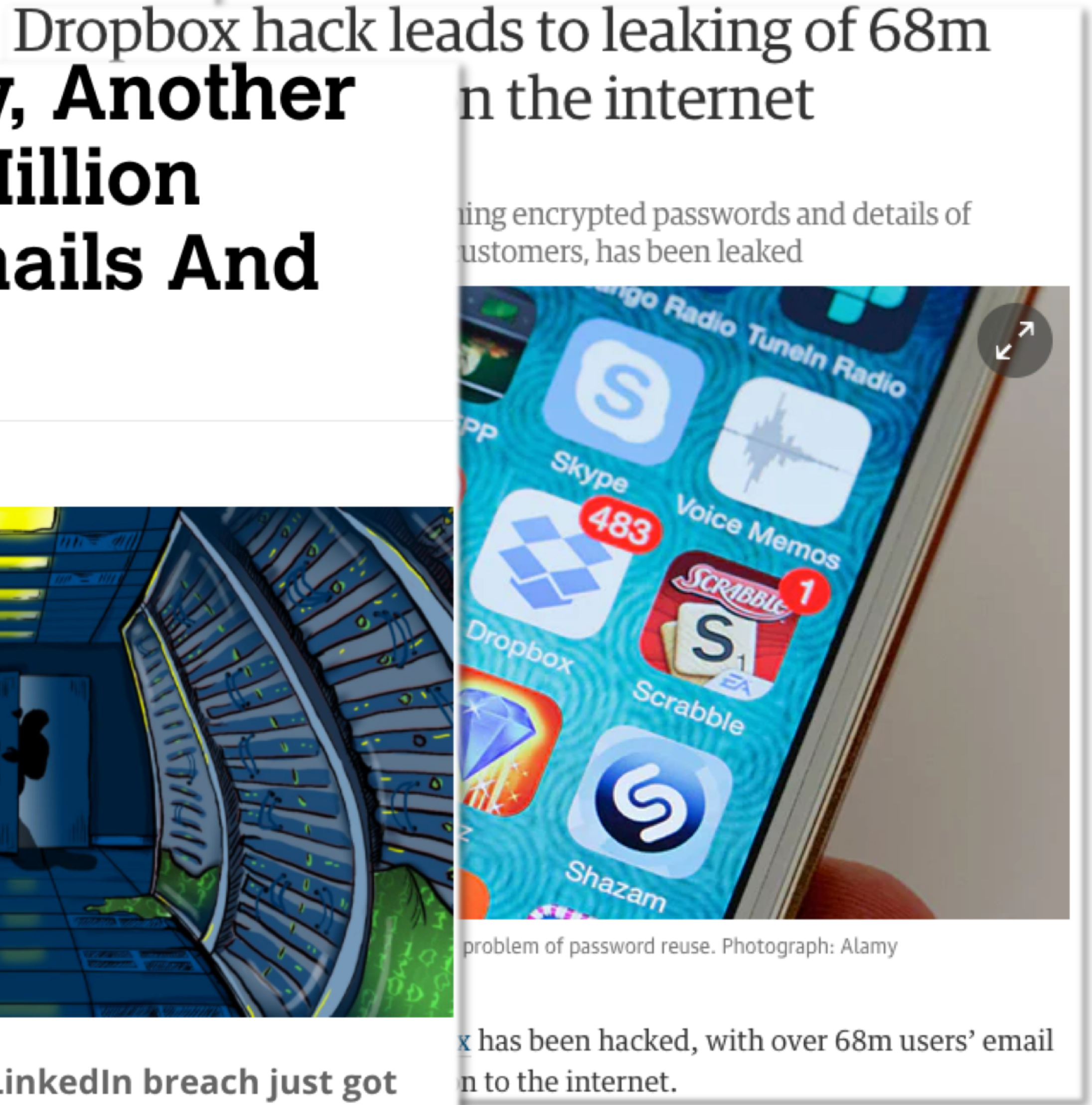


Yahoo hack: 1bn accounts compromised by biggest data breach in history

The latest incident to emerge
from the breach of 500m user

Yahoo have said the stolen user accoun
Photograph: Dado Ruvic/Reuters

Yahoo said on Wednesday it ha
data from more than 1bn user
making it the largest such brea

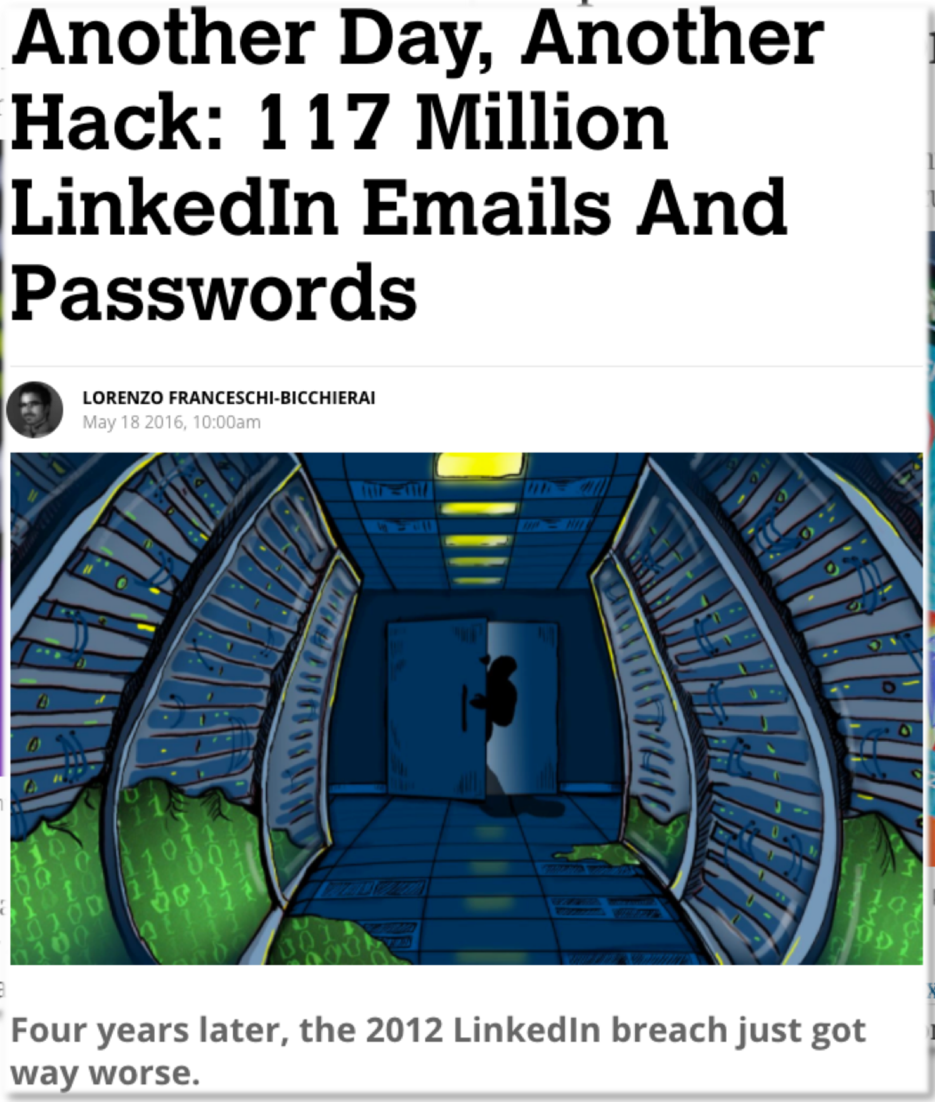


Dropbox hack leads to leaking of 68m
n the internet

ing encrypted passwords and details of
ustomers, has been leaked

problem of password reuse. Photograph: Alamy

x has been hacked, with over 68m users' email
n to the internet.



Another Day, Another Hack: 117 Million LinkedIn Emails And Passwords

LORENZO FRANCESCHI-BICCHIERAI
May 18 2016, 10:00am

Four years later, the 2012 LinkedIn breach just got way worse.

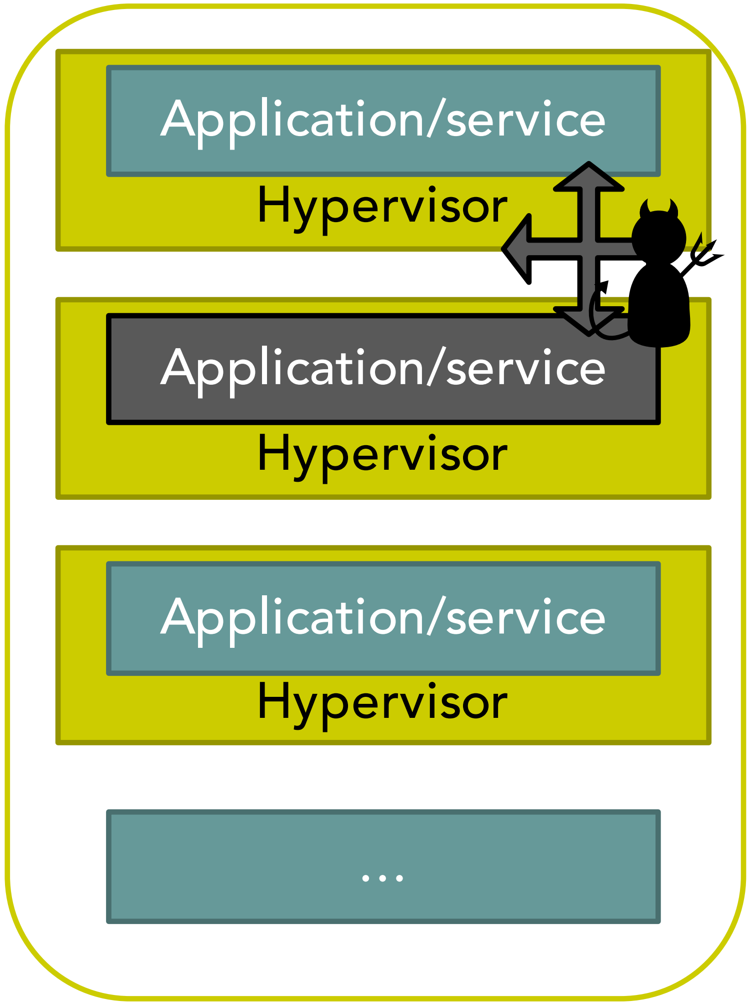
Cloud security

- Data confidentiality becomes a real problem
  - Storage, processing take place *off premises*
- No control over hardware
  - Must deal with physical attacks
- Hardware is shared between customers
  - Vulnerability in one service can affect others
- Less control over software stack
  - Managed and operated by Cloud provider
- Insider attacks
  - Malicious employee with root access

Provider's perspective

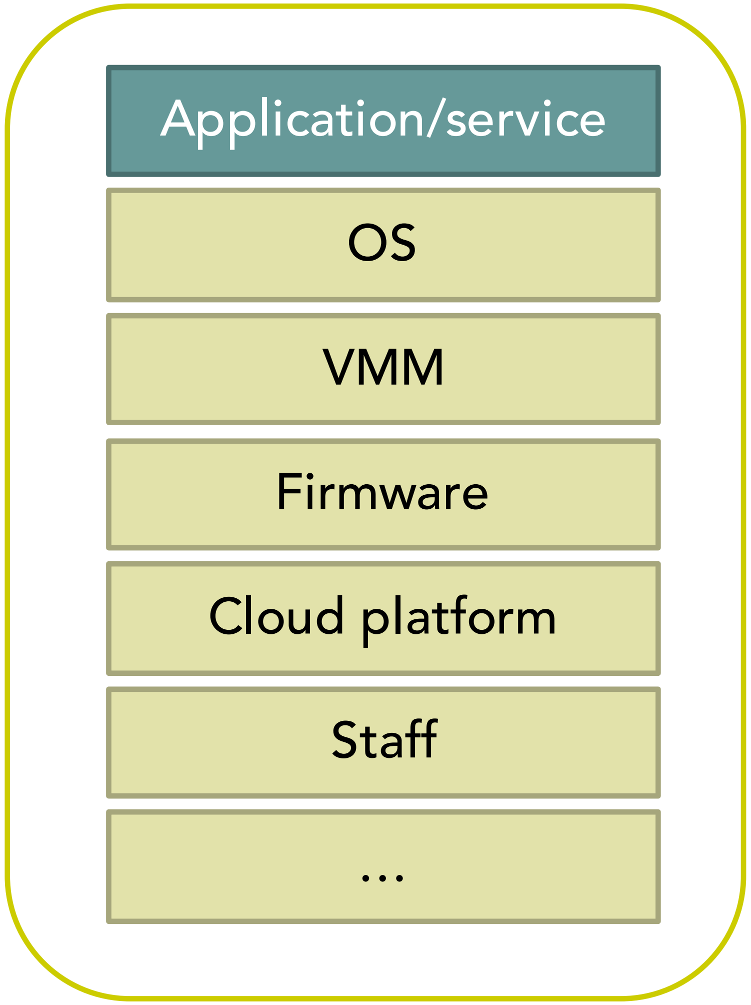
- Cloud provider needs to protect against malicious customers
  - Hypervisor-based isolation
  - Both security and performance

- One-way protection



Client's perspective

- Cloud tenant is forced to trust the provider…
  - … Including personnel
  - … Including every software component
- Ideally, we want to trust only our service



The software stack

- Cloud platforms contain enormous amounts of code that must be trusted
  - Linux: ~20 MLOC
  - KVM: ~13 MLOC
  - OpenStack: ~2 MLOC
- Cloud platforms are effectively a trusted computing base (TCB): all components of the system are critical to security
  - Software, hardware

Bugs are a reality

- More code ⇒ more bugs
  - Exploited vulnerability may lead to complete disclosure of confidential data
- Xen hypervisor
  - 184 vulnerabilities (2012-2016)
    [http://www.cvedetails.com/product/23463/XEN-XEN.html?vendor_id=6276]
- Linux kernel
  - 721 vulnerabilities (2012-2016)
    [http://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33]
- Especially bad in privileged software as it may result in unrestricted access to the system

Protected mode not sufficient

- Protected mode (rings) protects OS from applications, and applications from one another…
  - … until a malicious applications exploits a flaw to gain full privileges and then tampers with the OS or other applications
  - Applications not protected from privileged code attacks
- The attack surface is the whole software stack
  - Applications, OS, VMM, drivers, BIOS…

Software attacks in the Cloud

- Performed by executing software on the victim computer
  - Can be done remotely
- Vast majority of attacks exploit vulnerabilities in software components
  - E.g., memory safety violations in C/C++
- Note: applicable not only to the cloud environment

Software attacks in the Cloud

- Control-flow hijacking
  - Goal: execute arbitrary code on the target machine
  - Modify application's control flow
- Code injection attack
  - Overwrite return address by writing beyond allocated buffer on the stack (inject code)
  - Jump to the injected code
- Return-oriented programming
  - Hijack control flow by corrupting stack (no injection)
  - Jump to sequences of instructions (gadgets) already present in memory (e.g., `libc`) ending with a return
  - Chain gadgets to execute arbitrary code

Hardware attacks in the Cloud

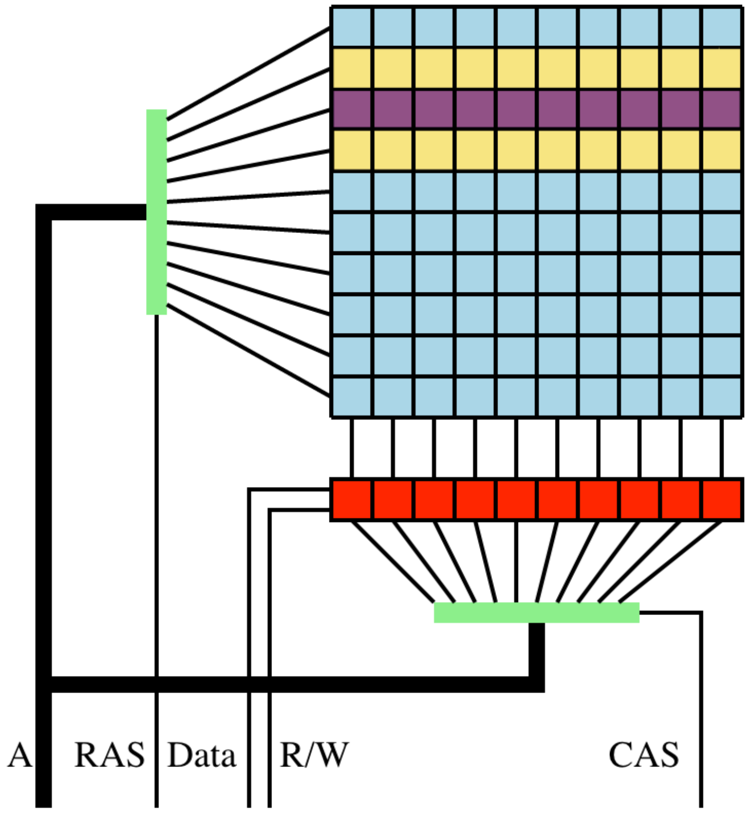
- Require physical access to the machine

- Bus snooping
  - Dump CPU ↔ memory communication
- Cold boot attacks
  - Power cycle the machine, boot to a lightweight OS, dump memory contents…

  … or remove memory modules, plug into another machine, dump memory contents
  - DRAM retains its state for a short period of time

Example: "*Row hammer*" attack

- Attack the system by causing bit-flips in memory
  - Accessing physical bits causes neighboring bits to flip
  - Carefully chosen addresses can result in privilege escalation
- Effect
  - Sandbox escape
  - Corrupted page table



Rapid row activations (yellow rows) may change the values of bits stored in victim row (purple row). [wikipedia]

```
code1a:
  mov (X), %eax // read from address X
  mov (Y), %ebx // read from address Y
  clflush (X)   // flush cache for address X
  clflush (Y)   // flush cache for address Y
  jmp code1a
```

Example: Heartbleed bug

- Serious vulnerability in the popular OpenSSL cryptographic software library
  - Very widely used: apache/nginx (66% of Web servers), email servers, chat servers, VPN, etc.
- Buffer overrun when replying to a heartbeat message
- Allows anyone on the Internet to read the memory of the systems protected by the vulnerable versions of the OpenSSL software
  - The attacker can obtain sensitive data from server’s memory: passwords, private keys, …

More examples

Meltdown

- Allows a program to access the memory and secrets of other programs and the operating system

Spectre

- Allows an attacker to trick error-free programs into leaking their secrets

Goals: security in the Cloud

- Confidentiality
  - Information is not made available or disclosed to unauthorized individuals, entities, or processes

  ⇒ Encryption
- Integrity
  - Data cannot be modified in an undetected manner

  ⇒ MAC, digital signature
- The problem: what mechanisms can we use to protect data confidentiality and integrity in untrusted environments (such as clouds)?

Ensuring data confidentiality

- Data-at-rest protection
  - Encrypt data before storing on disk
  - Encrypted file systems, full-disk encryption
  - Application-level protection with encrypted databases
- Communication protection
  - Well established end-to-end encryption mechanisms
  - Transport layer security (TLS)
- Trusted platform module (TPM)
  - Tamper-resistant chip external to the CPU
  - Facilities for secure generation of cryptographic keys, remote attestation, sealed storage
  - Limited protection, susceptible to physical attacks

Ensuring data confidentiality

- How to ensure confidentiality during computation?
  - Need to decrypt data before processing
  - Encryption keys/plaintext data in main memory/registers
- Memory dump will reveal all secrets
- No (practical) solution until recently
  - Homomorphic encryption: too slow, not general enough

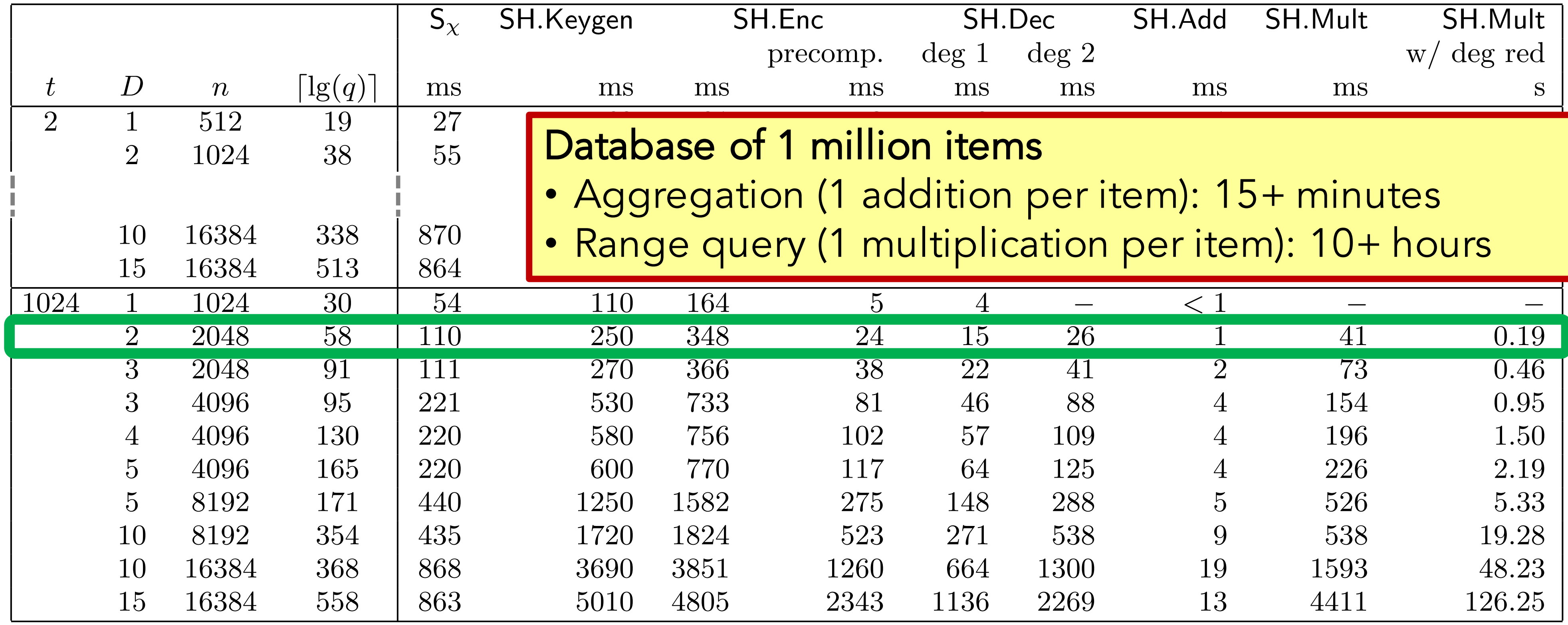
Encrypted data processing

- Homomorphic encryption

  "a form of encryption which allows specific types of computations to be carried out on ciphertext and generate an encrypted result which, when decrypted, matches the result of operations performed on the plaintext" [wikipedia]

- Fully homomorphic encryption [Gentry 2010]
  - Supports arbitrary functions on encrypted data
  - Addition, multiplication, binary operations

Homomorphic encryption [CCSW '11]



				S_χ	SH.Keygen	SH.Enc		SH.Dec		SH.Add	SH.Mult	SH.Mult
							precomp.	deg 1	deg 2			w/ deg red
t	D	n	$\lceil \lg(q) \rceil$	ms	ms	ms	ms	ms	ms	ms	ms	s
2	1	512	19	27								
	2	1024	38	55								
	10	16384	338	870								
	15	16384	513	864								
1024	1	1024	30	54	110	164	5	4	–	< 1	–	–
	2	2048	58	110	250	348	24	15	26	1	41	0.19
	3	2048	91	111	270	366	38	22	41	2	73	0.46
	3	4096	95	221	530	733	81	46	88	4	154	0.95
	4	4096	130	220	580	756	102	57	109	4	196	1.50
	5	4096	165	220	600	770	117	64	125	4	226	2.19
	5	8192	171	440	1250	1582	275	148	288	5	526	5.33
	10	8192	354	435	1720	1824	523	271	538	9	538	19.28
	10	16384	368	868	3690	3851	1260	664	1300	19	1593	48.23
	15	16384	558	863	5010	4805	2343	1136	2269	13	4411	126.25

Database of 1 million items
- Aggregation (1 addition per item): 15+ minutes
- Range query (1 multiplication per item): 10+ hours

Table 2: Timings for the somewhat homomorphic encryption scheme using the example parameters given in Table 1. The column labeled S_χ gives timing for sampling an element from the discrete Gaussian distribution χ. In the second column for SH.Enc, labeled prec., encryption is measured without sampling from χ, which is instead done as a precomputation. The two columns for SH.Dec correspond to decryption of a degree-1 and a degree-2 ciphertext, respectively. The last column gives the time taken for a ciphertext multiplication of two linear ciphertexts including the degree reduction resulting in a degree-1 ciphertext for the product. Measurements were done on a 2.1 GHz Intel Core 2 Duo using the computer algebra system Magma [BCP97].

Suitable parameters are given in Table 1 as $t = 1024$, $D = 2$, and $n = 2048$ with the 58-bit prime $q = 144115188076060673$.

Homomorphic encryption

- HELib: open-source homormophic encryption library in C++ by IBM [Shoup and Halevi, 2012]
  - Many optimizations to make HE “practical”, i.e., make homomorphic evaluation run faster
  - Low-level routines (set, add, multiply, shift, etc.)
- Still far from being practical
  - Addition: ~1+ ms
  - Multiplication: ~10/100+ ms
  - Evaluated the AES-128 circuit in 36 hours in 2012 (vs. 2 ms in the clear)

[https://mpclounge.files.wordpress.com/2013/04/hespeed.pdf]

Intel SGX

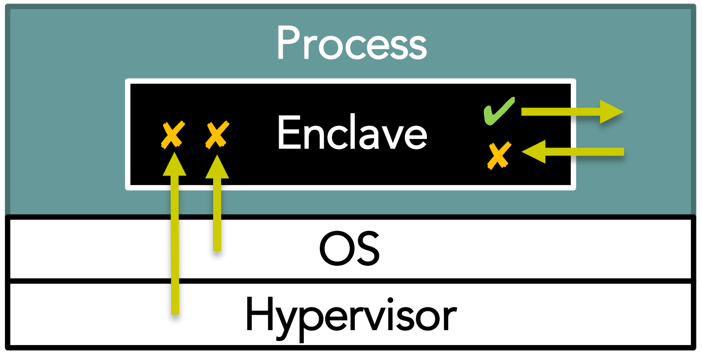
- "Software guard extensions"
- Hardware extension in recent Intel CPUs
  - Skylake (2015), Kaby lake (2016)
- Protects confidentiality and integrity of code and data in untrusted environments
  - Platform owner is considered malicious
  - Only the CPU chip and the isolated region are trusted

Enclaves

- SGX introduces the notion of "enclave"
  - Isolated memory region for code and data
  - New CPU instructions to manipulate enclaves and a new enclave execution mode
- Enclave memory is encrypted and integrity-protected by the hardware
  - Memory encryption engine (MEE)
  - No plaintext secrets in main memory
- Enclave memory can be accessed only by the enclave code
  - Protection from privileged code (OS, hypervisor)

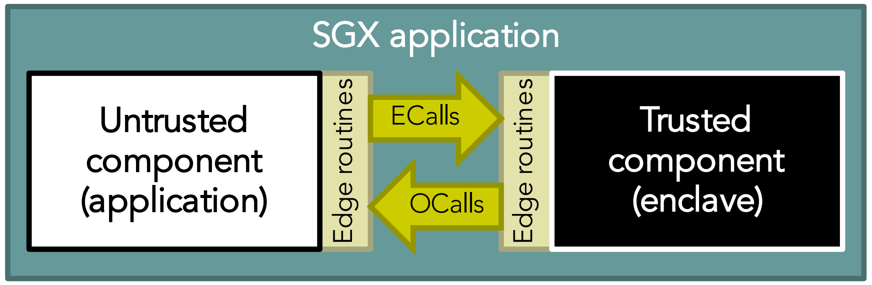
Enclave memory

- Enclave memory is not accessible to other software
    - Can access memory within its process
- Application has ability to defend its secrets
    - Attack surface reduced to just enclaves and CPU
    - Compromised software cannot steal application secrets

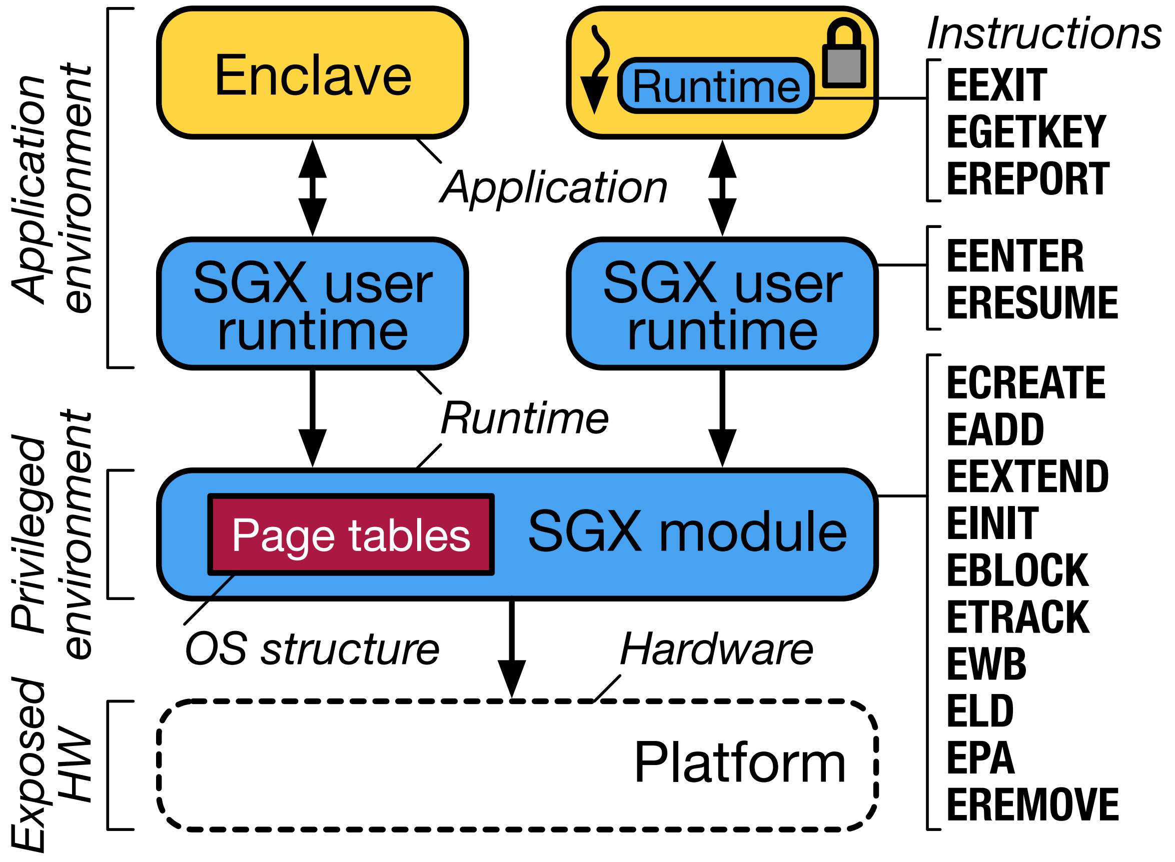


Enclave memory

- Enclave define APIs
  - Enclave interface functions: ECalls to provide input data to the enclave
  - Calls outside the enclave: OCalls to return results from the enclave
  - Constitute the enclave boundary interface

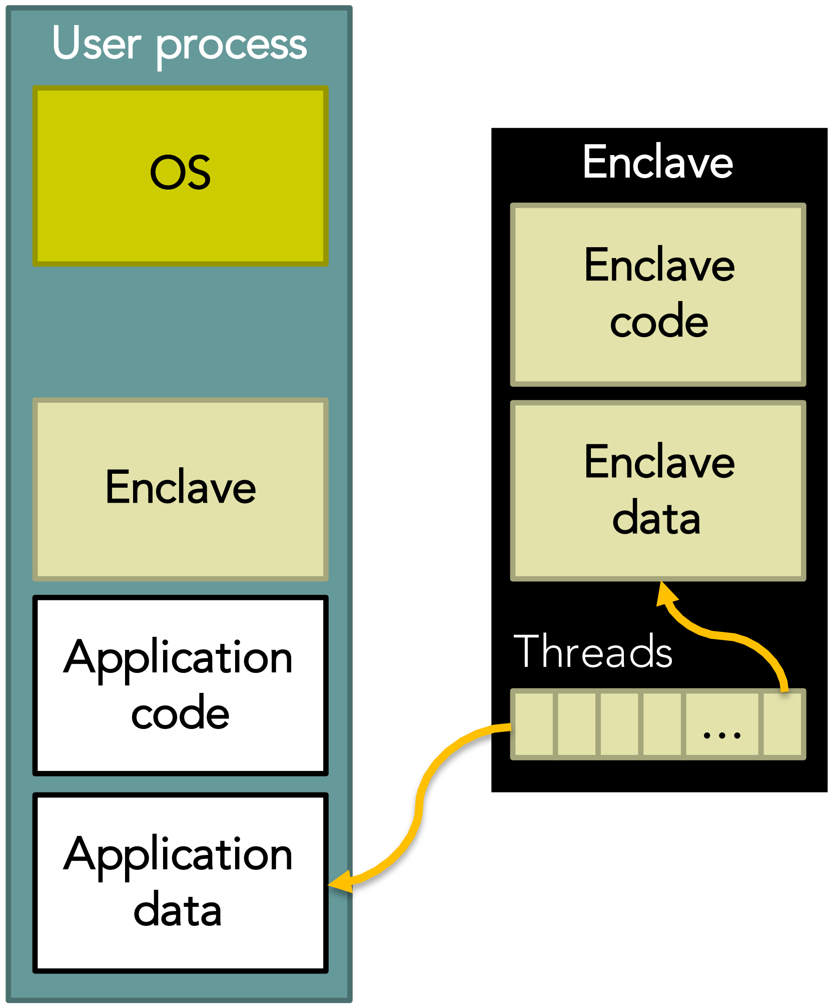

SGX application
Untrusted component (application)
Edge routines
ECalls
OCalls
Edge routines
Trusted component (enclave)

SGX architecture and API

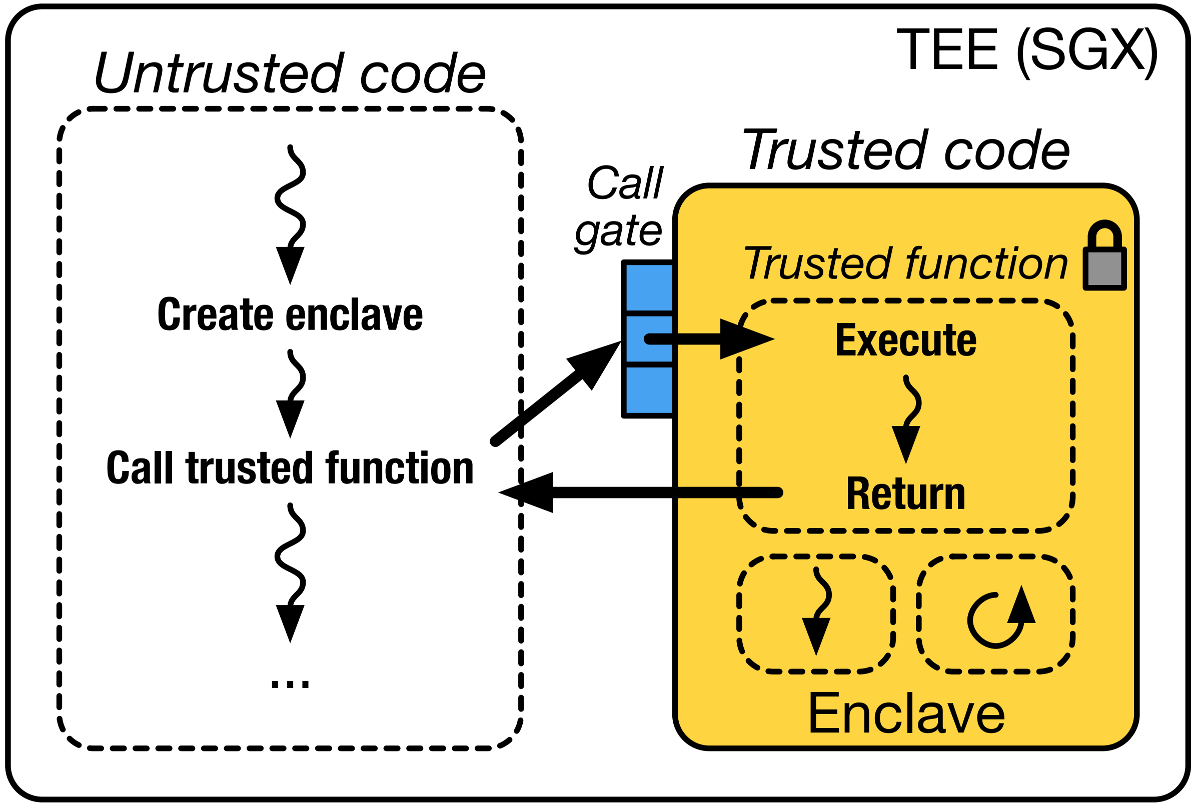


SGX execution model

- Trusted execution environment in a process
  - With its own code and data
  - With controlled entry points
  - Provides confidentiality
  - Provides integrity
  - Supporting multiple threads
  - With full access to application memory



SGX operation



Enclave page cache (EPC)

- Physical memory region protected by the MEE
  - EPC holds enclave contents
- Shared resource between all enclaves running on a platform
  - Currently only 128MB
  - ~96MB available to the user, the rest is for metadata
- Content encrypted while in DRAM, decrypted when brought to CPU
  - Plaintext in CPU caches

Example

SGX application: unstrusted code

```
char request_buf[BUFFER_SIZE];
char response_buf[BUFFER_SIZE];

int main()
{
  ...
  while(1)
  {
    receive(request_buf);
    ret = EENTER(request_buf, response_buf);
    if (ret < 0)
      fprintf(stderr, "Corrupted message\n");
    else
      send(response_buf);
  }
  ...
}
```

Enclave: trusted code

```
char input_buf[BUFFER_SIZE];
char output_buf[BUFFER_SIZE];

int process_request(char *in, char *out)
{
  copy_msg(in, input_buf);
  if(verify_MAC(input_buf))
  {
    decrypt_msg(input_buf);
    process_msg(input_buf, output_buf);
    encrypt_msg(output_buf);
    copy_msg(output_buf, out);
    EEXIT(0);
  } else
    EEXIT(-1);
}
```

Server:
- Receives encrypted requests
- Processes them in enclave
- Sends encrypted responses

Example

SGX application: unstrusted code

```
char request_buf[BUFFER_SIZE];
char response_buf[BUFFER_SIZE];

int main()
{
  ...
  while(1)
  {
1   receive(request_buf);
2   ret = EENTER(request_buf, response_buf);
    if (ret < 0)
3     fprintf(stderr, "Corrupted message\n");
    else
4     send(response_buf);
  }
  ...
}
```

Enclave: trusted code

```
char input_buf[BUFFER_SIZE];
char output_buf[BUFFER_SIZE];

int process_request(char *in, char *out)
{
  copy_msg(in, input_buf);
  if(verify_MAC(input_buf))
  {
    decrypt_msg(input_buf);
    process_msg(input_buf, output_buf);
    encrypt_msg(output_buf);
    copy_msg(output_buf, out);
    EEXIT(0);
  } else
    EEXIT(-1);
}
```

1. Receive a requests
2. Enter the enclave with two arguments: pointer to a buffer with encrypted request and pointer to a response buffer (EENTER instruction switches CPU to the enclave mode and transfers control to predefined location in enclave)
3. Print error message if enclave returns an error
4. Send the response provided by the enclave

Example

SGX application: unstrusted code

```
char request_buf[BUFFER_SIZE];
char response_buf[BUFFER_SIZE];

int main()
{
  ...
  while(1)
  {
    receive(request_buf);
    ret = EENTER(request_buf, response_buf);
    if (ret < 0)
      fprintf(stderr, "Corrupted message\n");
    else
      send(response_buf);
  }
  ...
}
```

Enclave: trusted code

```
char input_buf[BUFFER_SIZE];
char output_buf[BUFFER_SIZE];
1
int process_request(char *in, char *out)
{
2 copy_msg(in, input_buf);
3 if(verify_MAC(input_buf))
  {
4   decrypt_msg(input_buf);
    process_msg(input_buf, output_buf);
    encrypt_msg(output_buf);
    copy_msg(output_buf, out);
    EEXIT(0);
  } else
    EEXIT(-1);
}
```

1. Enclave entry point
2. Copy request into enclave memory (enclave can access `in` buffer in untrusted memory while untrusted part cannot access `input_buf` buffer in trusted memory)
3. Check the MAC (assuming keys were already exchanged)
4. Decrypt request

Example

SGX application: unstrusted code

```
char request_buf[BUFFER_SIZE];
char response_buf[BUFFER_SIZE];

int main()
{
  ...
  while(1)
  {
    receive(request_buf);
    ret = EENTER(request_buf, response_buf);
    if (ret < 0)
      fprintf(stderr, "Corrupted message\n");
    else
      send(response_buf);
  }
  ...
}
```

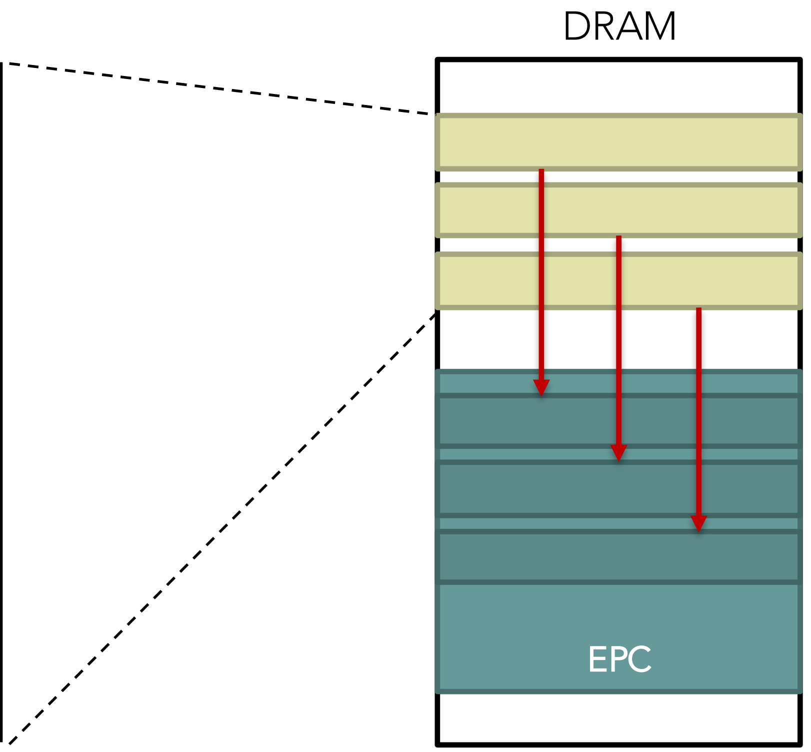
Enclave: trusted code

```
char input_buf[BUFFER_SIZE];
char output_buf[BUFFER_SIZE];

int process_request(char *in, char *out)
{
  copy_msg(in, input_buf);
  if(verify_MAC(input_buf))
  {
    decrypt_msg(input_buf);
5   process_msg(input_buf, output_buf);
6   encrypt_msg(output_buf);
7   copy_msg(output_buf, out);
8   EEXIT(0);
  } else
    EEXIT(-1);
}
```

5. Process request and store result in output buffer
6. Encrypt result and add MAC
7. Write result to untrusted memory for access from outside
8. Exit the enclave (EEXIT instruction switches from enclave to normal mode and transfer control to the next location after EENTER, similar to regular return)

Enclave construction

```
1 { char input_buf[BUFFER_SIZE];
2 { char output_buf[BUFFER_SIZE];

    int process_request(char *in, char *out)
    {
      copy_msg(in, input_buf);
      if(verify_MAC(input_buf))
      {
        decrypt_msg(input_buf);
3 {     process_msg(input_buf, output_buf);
        encrypt_msg(output_buf);
        copy_msg(output_buf, out);
        EEXIT(0);
      } else
        EEXIT(-1);
    }
```



Enclave is populated using a special instruction (EADD)

- Contents are initially in untrusted memory
- Copied into the EPC in 4KB pages

Both data and code are copied before starting execution in enclave

Enclave construction

- Enclave contents are distributed in plaintext
  - Can be inspected
  - Must not contain any (plaintext) confidential data
- Secrets are provisioned after the enclave was constructed and its integrity verified
- Problem: what if someone tampers with the enclave?
  - Contents are initially in untrusted memory

Enclave construction

- Someone may tamper with the enclave by modifying the code

```
int process_request(char *in, char *out)
{
  copy_msg(in, input_buf);
  if(verify_MAC(input_buf))
  {
    decrypt_msg(input_buf);
    process_msg(input_buf, output_buf);
    encrypt_msg(output_buf);
    copy_msg(output_buf, out);
    EEXIT(0);
  } else
    EEXIT(-1);
}
```

```
int process_request(char *in, char *out)
{
  copy_msg(in, input_buf);
  if(verify_MAC(input_buf))
  {
    decrypt_msg(input_buf);
    process_msg(input_buf, output_buf);
    copy_msg(output_buf, external_buf);
    encrypt_msg(output_buf);
    copy_msg(output_buf, out);
    EEXIT(0);
  } else
    EEXIT(-1);
}
```

Write unencrypted response to outside memory

Enclave construction

- CPU calculates enclave's measurement hash during enclave construction
  - Each new page extends the hash with the page content and attributes (read/write/execute)
  - Hash computed with SHA-256
- Measurement can then be used to attest the enclave to a local or remote entity

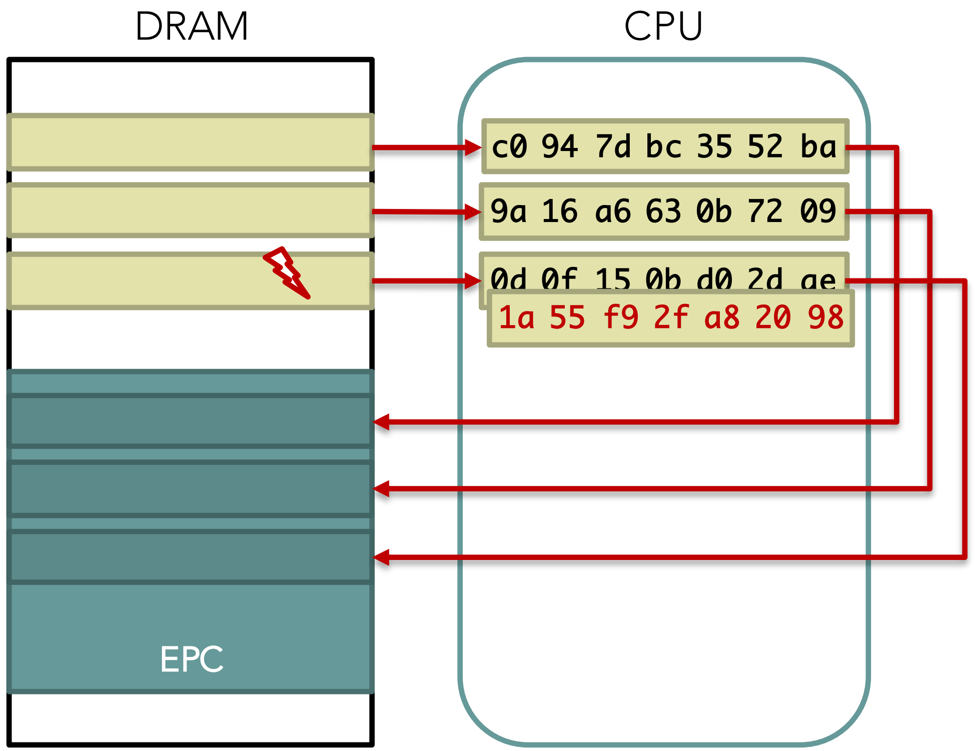
Enclave attestation

- Is my code running on remote machine intact?
- Is code really running inside an SGX enclave?

- Local attestation
  - Prove enclave's identity (=measurement) to another enclave on the same CPU
- Remote attestation
  - Prove enclave's identity to a remote party

- Once attested, an enclave can be trusted with secrets

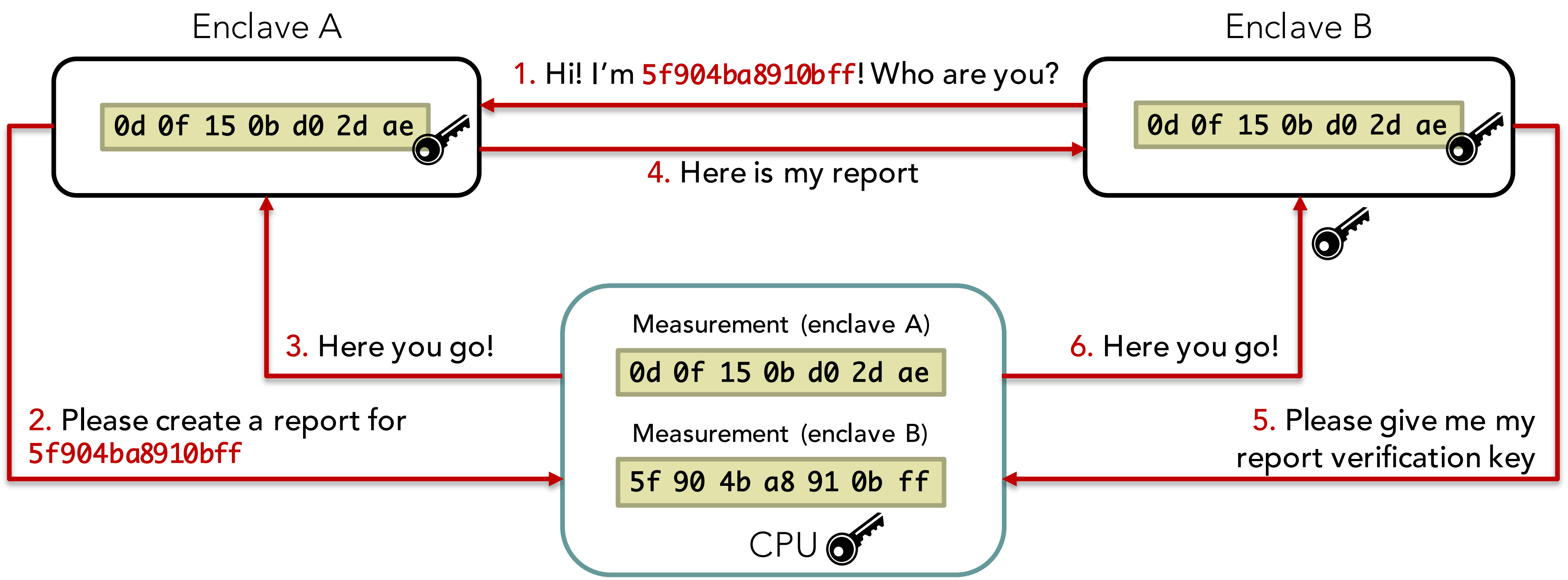
Enclave construction



CPU calculates enclave's measurement hash during enclave construction
Different measurement if enclave is modified

Local attestation

- Prove identity of A to a local enclave B



1. Target enclave B measurement is required for key generation
2. Report contains information about target enclave B, including its measurement
3. CPU fills in the report and creates a MAC using the report key, which depends on random CPU fuses and the target enclave B measurement
4. Report sent back to target enclave B
5. Verify report by CPU to check that it was generated on the same platform, i.e., its MAC was created with the same report key (available only on the same CPU)
6. Check the MAC received with the report and do not trust A upon mismatch

Remote attestation

- Transform a local report to a remotely verifiable "quote"
- Based on provisioning enclave (PE) and quoting enclave (QE)
  - Architectural enclaves provided by Intel
  - Execute locally on user platform
- Each SGX-enabled CPU has a unique key fused during manufacturing
  - Intel maintains a database of these keys

Remote attestation

- PE communicates with Intel attestation service
  - Proves it has a key installed by Intel
  - Receives asymmetric attestation key
- QE performs local attestation for enclave
  - QE verifies report and signs it using attestation key
  - Creates a quote that can be verified outside platform
- The quote and signature are sent to the remote attester, which communicates with Intel attestation service to verify quote validity

Multithreading support

- SGX allows multiple threads to enter the same enclave simultaneously
  - One thread control structure (TCS) per thread
  - Part of the enclave, reflected in measurement
- TCS limits the number of enclave threads
  - Upon thread entry a TCS is blocked and cannot be used by another thread
- Each TCS contains address of the entry point
  - Prevents jumps into random locations inside of enclave

SGX paging

- SGX provides a mechanism to evict an EPC page to unprotected memory
  - EPC is very limited in size
- Paging performed by the OS
  - Validated by the HW to prevent attacks on address translations
  - Metadata (MAC, version) kept within EPC
- Accessing evicted page results in page fault
  - The page is brought back into the EPC by the OS
  - Hardware verifies integrity of the page
  - Another page might be evicted if the EPC is full

SGX limitations

- Amount of memory the enclave can use needs to be known in advance
  - Dynamic memory support in SGX v2
- Security is not perfect
  - Vulnerabilities within the enclave can still be exploited
  - Side-channel attacks are possible
- Performance bottlenecks
  - Enclave entry/exit is costly
  - Paging is very expensive
- Application partitioning? Legacy code? …

Summary

- The cloud is attractive for many reasons
  - Simplicity, availability, cost, economies of scale, performance, etc.
  - Great for computations, non-sensitive data processing
- Also an attractive target for attacks!
  - Data is power, must be protected
  - Encryption helps but limits opportunities for querying, processing data
    - Homomorphic encryption great but slow
  - TEEs like SGX provide a first practical solution that combines efficiency and security
    - Requires trusting Intel, no HW bugs/backdoors

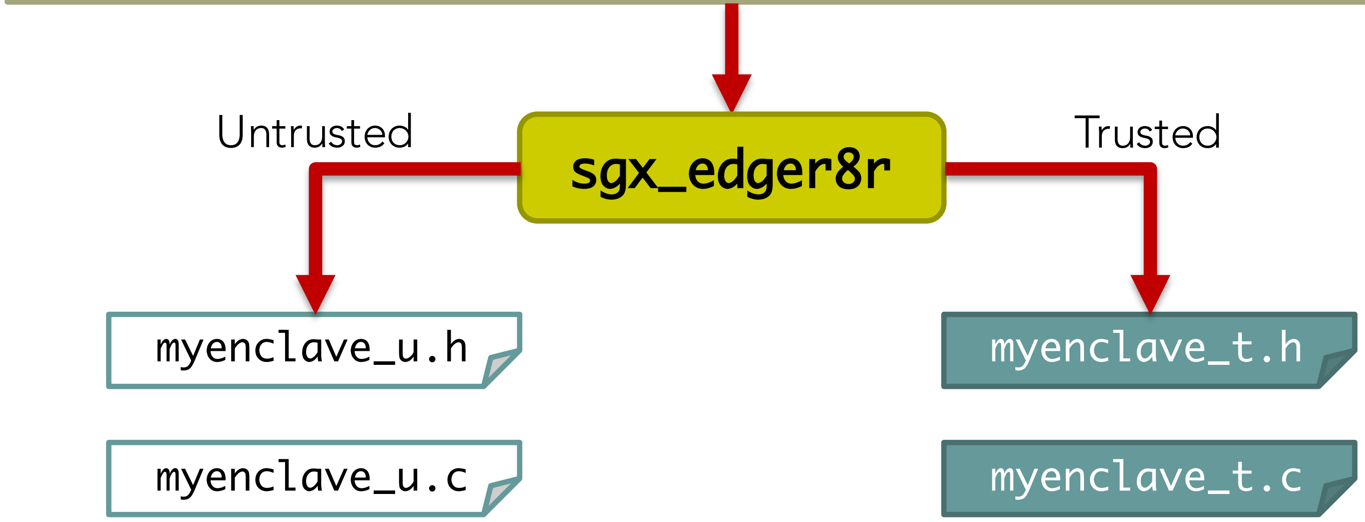
Writing your first SGX application…

SGX PRIMER

1. Define interface

Enclave definition language: myenclave.edl

```
enclave {
    untrusted{
        void ocall_error([in,string] const char *msg);
    };

    trusted {
        public int ecall_compute(int a, int b);
    };
};
```



2. Write code

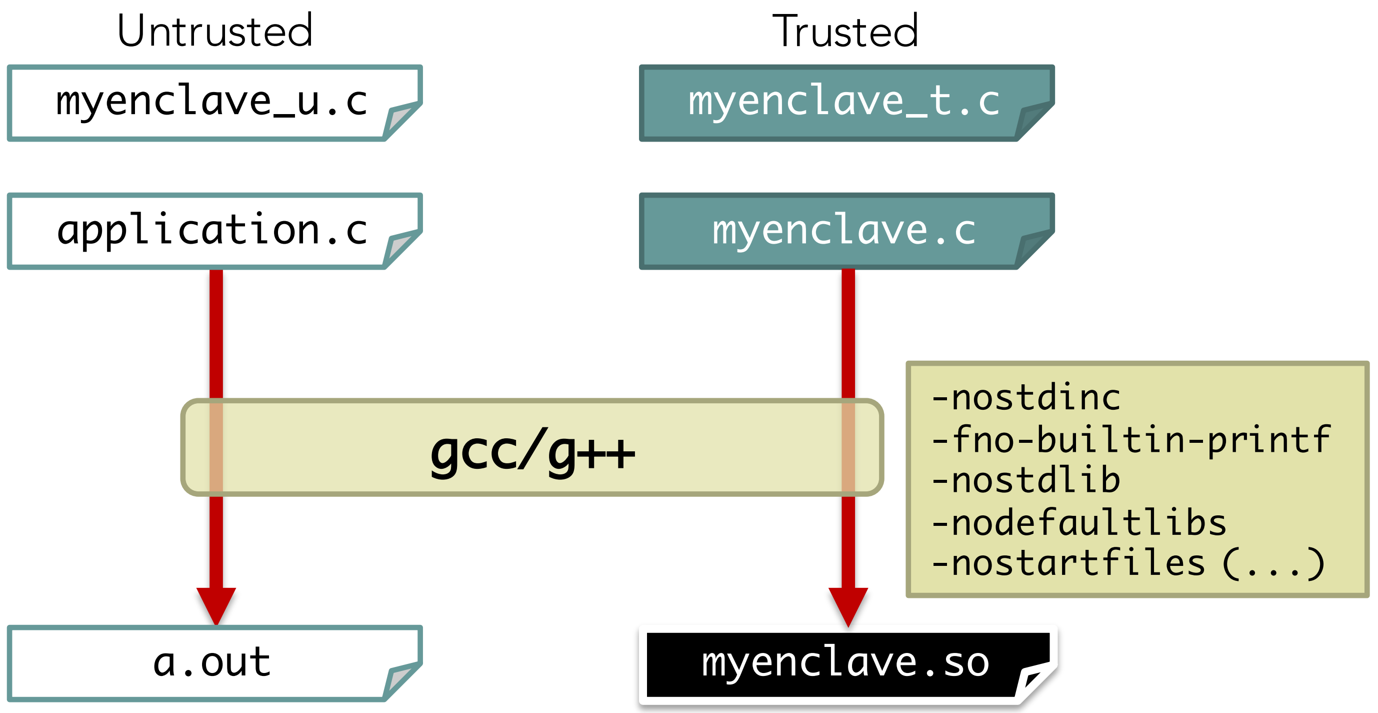
Application: application.c

```
#include <stdio.h>
#include <myenclave_u.h>

void ocall_error( const char *msg ) {
  printf( msg ); // syscall
}

int main() {
  sgx_enclave_id_t eid = 0;
  // initalize enclave

  int ret;
  ecall_compute(eid, &ret, 4, -5);
  printf("Result: %d\n", ret);

  // destroy enclave
  return 0;
}
```

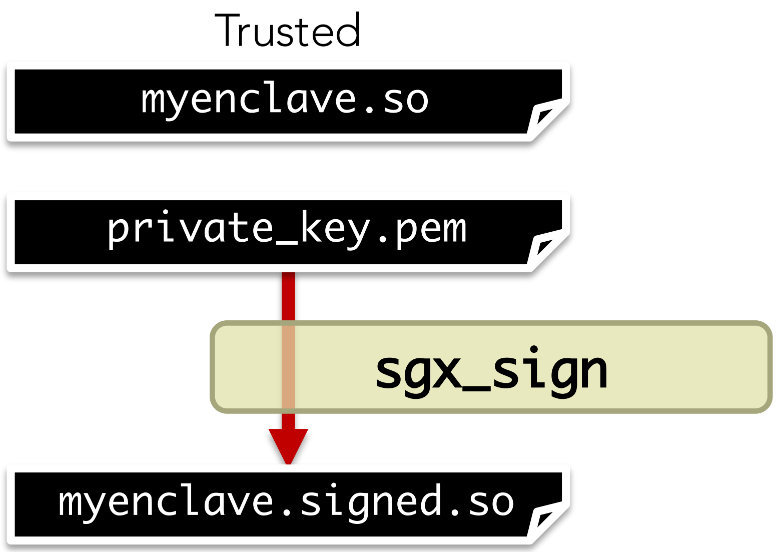
Enclave: myenclave.c

```
#include <myenclave_t.h>

int ecall_compute( int a, int b ) {
  int res = a + b;
  if( res < 0 )
    ocall_error("SGX says: I do not like "
                "negative results\n");
  return res;
}
```

3. Compile code



4. Sign code and run



```
$ ls
a.out    myenclave.signed.so

$ ./a.out
SGX says: I do not like negative results
Result: -1
```

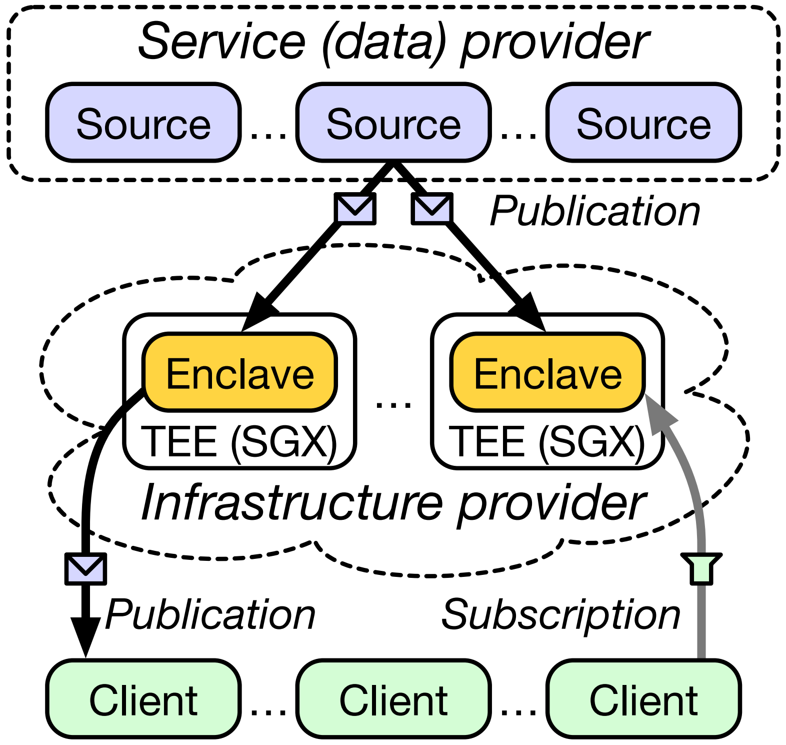
Thanks!

QUESTIONS?

EXTRA SLIDES

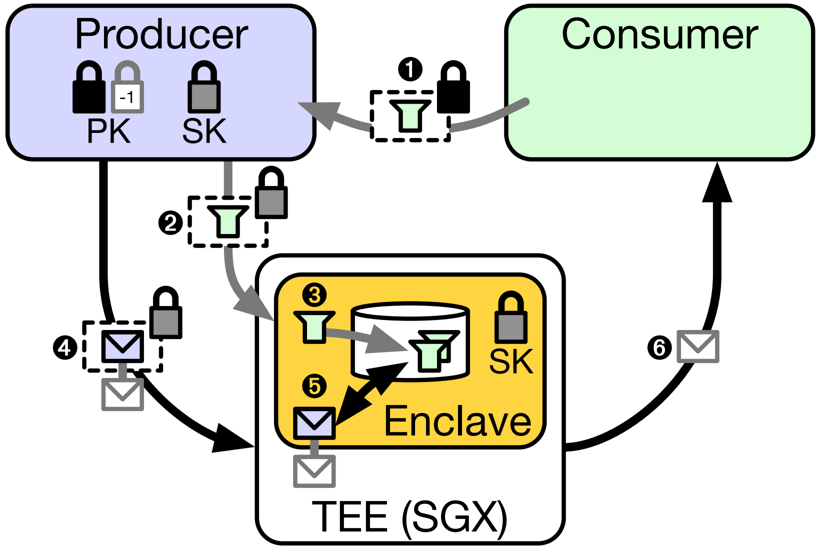
Secure content-based router

- Pub/sub model
  - Decoupled many-to-many communication
  - Routing based on subscriptions (interest of consumers) and message content
- CBR matches filters against data
  - Must inspect messages (threat to privacy)
  - Filtering done in SGX enclave



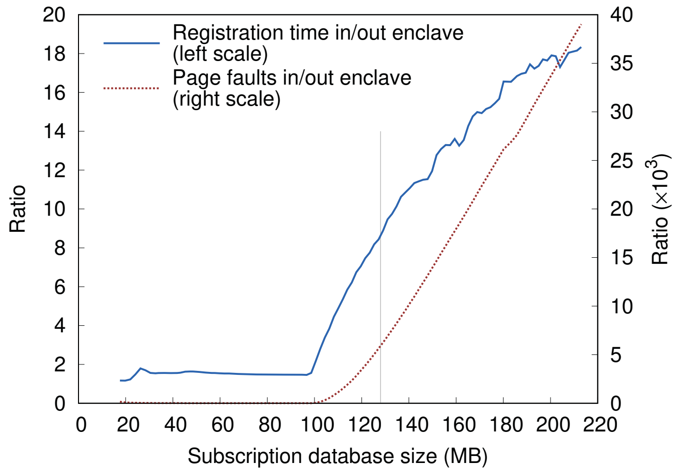
SCBR subscription/publication

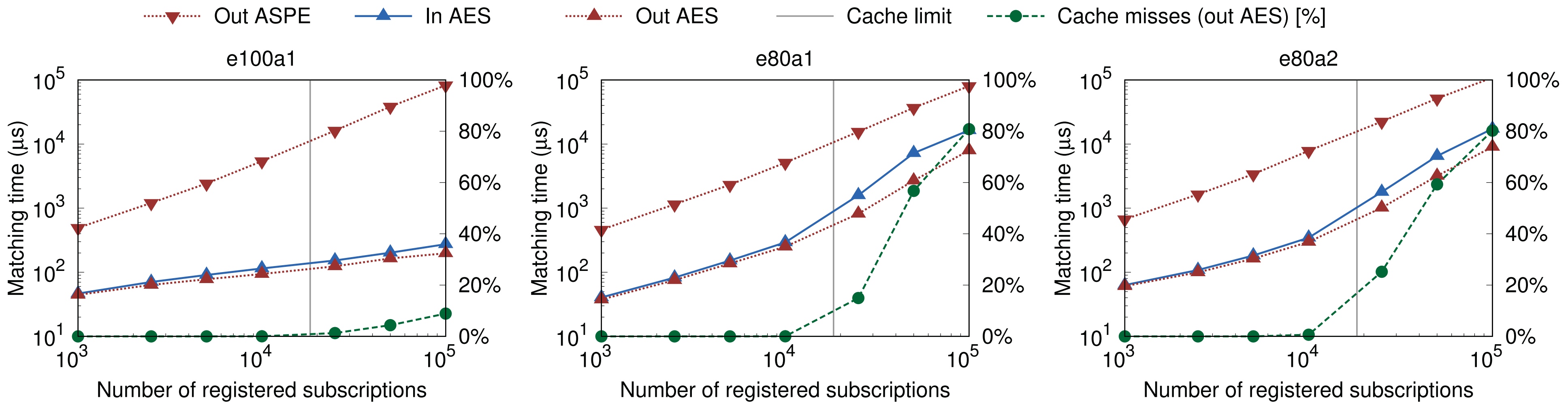
- To register subscription S (1-3)
  - Consumer encrypts S with publisher's PK and sends $\{S\}_{PK}$ to producer
  - Producer checks client's status, re-encrypts S and sends $\{S\}_{SK}$ to routing engine
  - Routing engine decrypts and stores S
- To publish message M (4-6)
  - Publisher encrypts header of M with SK and sends $\{M\}_{SK}$ to routing engine, payload encrypted with group key
  - Routing engine decrypts header and matches it against index, payload remains encrypted outside enclave and is forwarded to all matching clients



SCBR evaluation

- Limited memory in enclave
  - Paging is costly
- Matching faster than ASPE
  - Cache miss effects


Registration time in/out enclave (left scale)
Page faults in/out enclave (right scale)
Ratio
Ratio (×10^3)
Subscription database size (MB)


Out ASPE
In AES
Out AES
Cache limit
Cache misses (out AES) [%]
e100a1
e80a1
e80a2
Matching time (µs)
Number of registered subscriptions