Imperial College London

LSDS
Large-Scale Data & Systems Group

Maru: SGX-Spark Deep Dive

Florian Kelbert, Peter Pietzuch, Jon Crowcroft

Imperial College London, University of Cambridge

http://lsds.doc.ic.ac.uk
<prp@imperial.ac.uk>

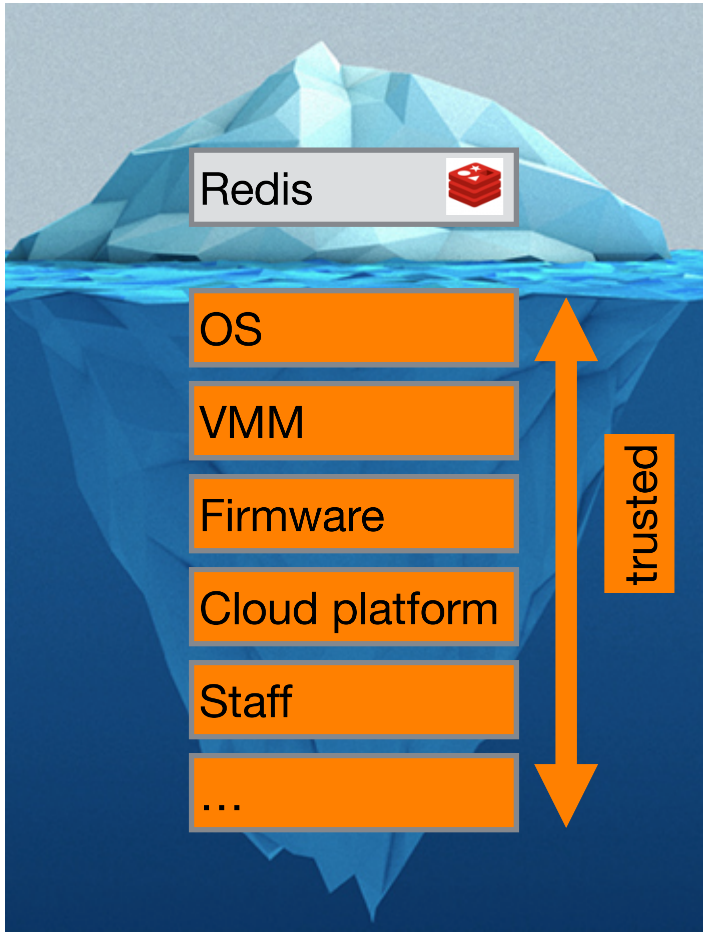
Maru Meeting – Imperial College London – November 2017

Trust Issues: Provider Perspective

Cloud provider does not trust users

Use virtual machines to isolate users from each other and the host

VMs only provide one way protection

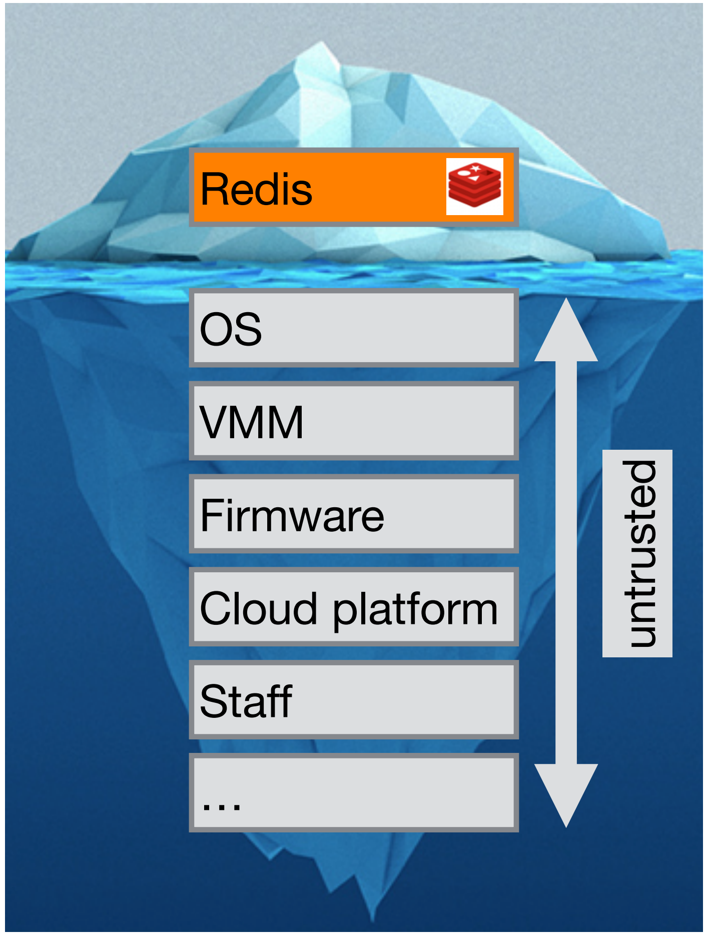


Trust Issues: User Perspective

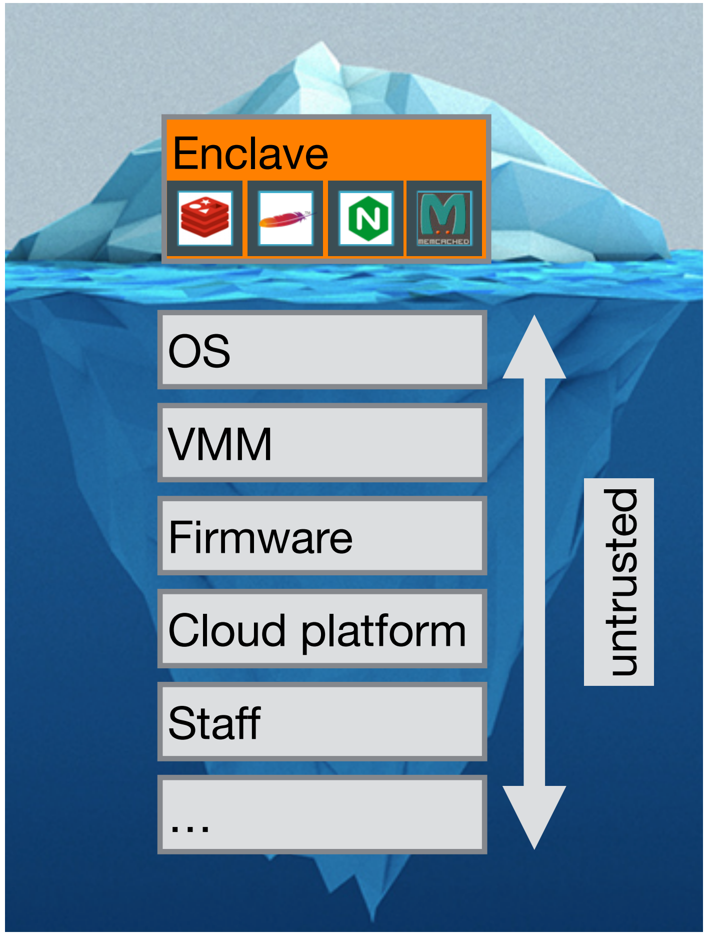
Users trust their applications

Users must implicitly trust cloud provider

Existing applications implicitly assume trusted operating system



Trusted Execution Support with Intel SGX



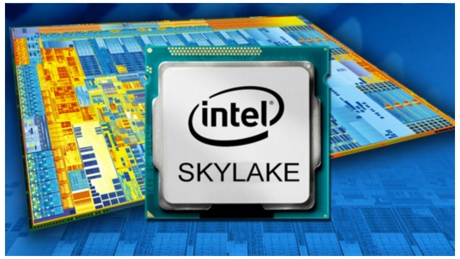
Users create HW-enforced trusted environment (enclave)

Supports unprivileged user code

Protects against strong attacker model

Remote attestation

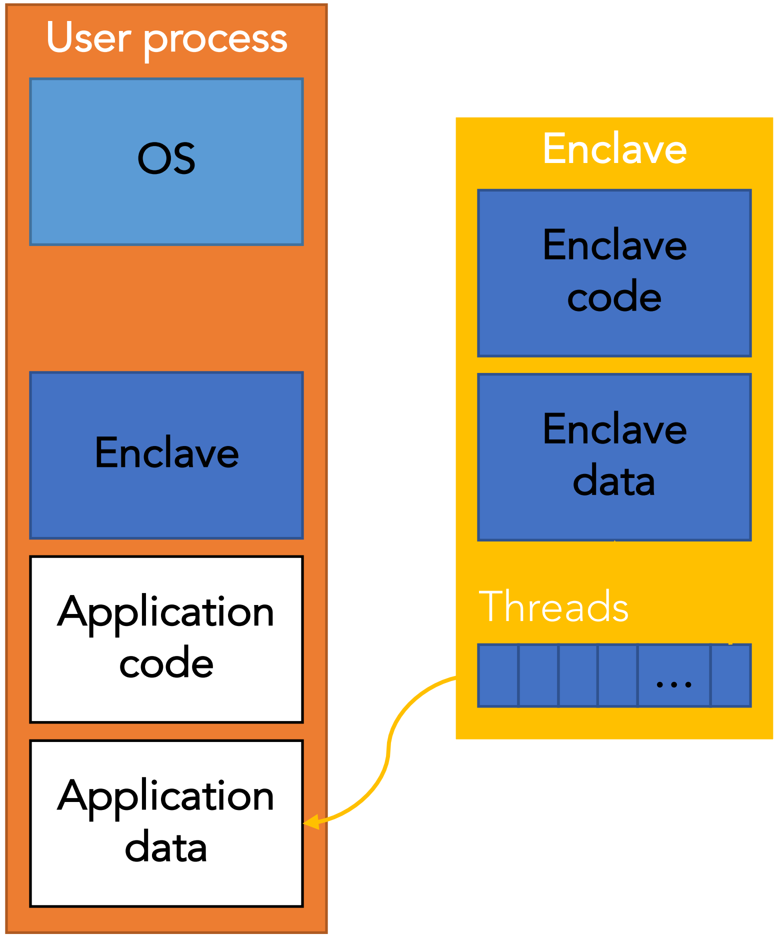
Available on commodity CPUs



Trusted Execution Environments

Trusted execution environment (TEE) in process

- Own code & data
- Controlled entry points
- Provides confidentiality & integrity
- Supports multiple threads
- Full access to application memory



Overview of Intel SGX

Intel Software Guard Extensions (SGX)

Extension of Instruction Set Architecture (ISA) in recent Intel CPUs

- Skylake (2015), Kaby lake (2016)

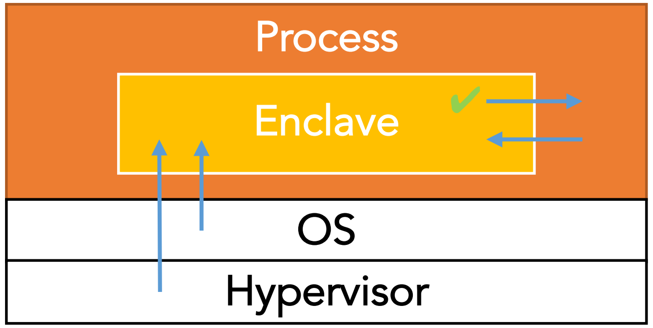
Protects confidentiality and integrity of code & data in untrusted environments

- Platform owner considered malicious
- Only CPU chip and isolated region trusted

SGX Enclaves

SGX introduces notion of **enclave**

- Isolated memory region for code & data
- New CPU instructions to manipulate enclaves and new enclave execution mode



Enclave memory **encrypted** and **integrity-protected** by hardware

- Memory encryption engine (MEE)
- No plaintext secrets in main memory

Enclave memory can be accessed only by enclave code

- Protection from privileged code (OS, hypervisor)

Application has ability to defend secrets

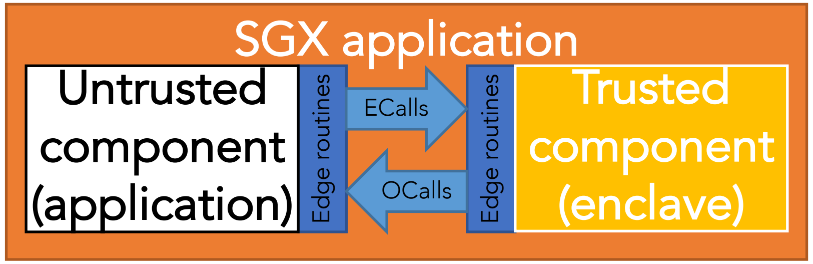
- Attack surface reduced to just enclaves and CPU
- Compromised software cannot steal application secrets

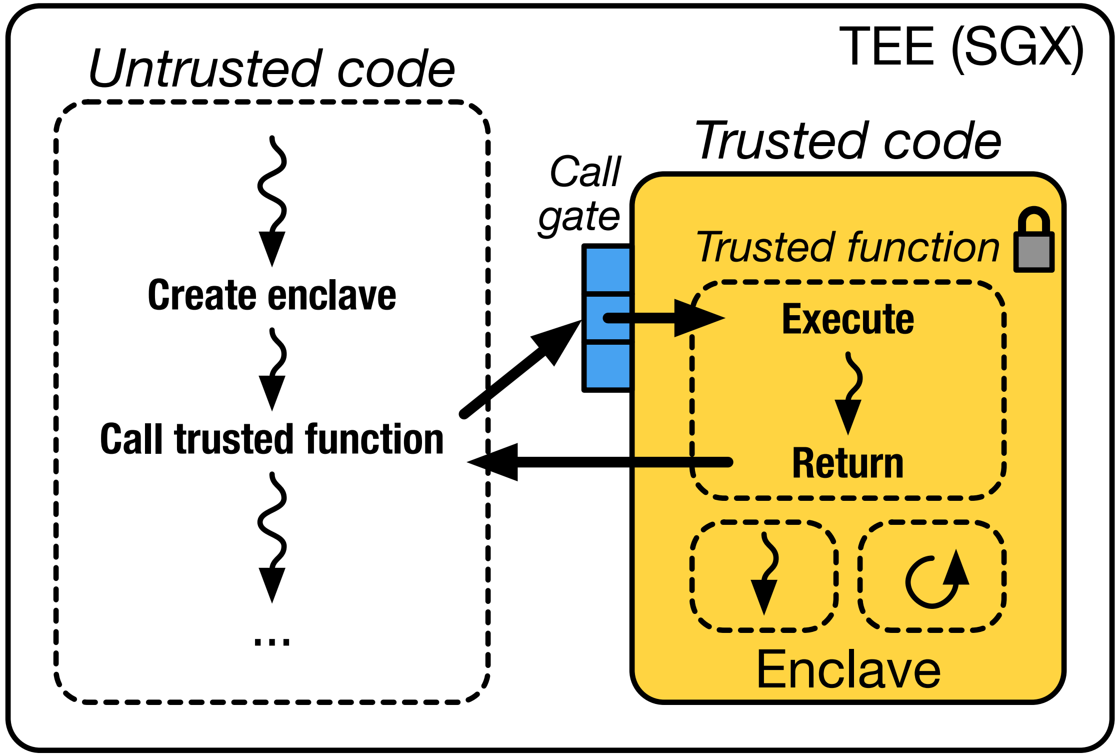
SGX Enclave API & Operation

Enclave interface functions: **ECalls** to provide input data to enclave

Calls outside enclave: **OCalls** to return results from enclave

Constitute enclave boundary interface


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



Enclave Page Cache (EPC)

Physical memory region protected by MEE

- EPC holds enclave contents

Shared resource between all enclaves running on platform

- Currently only 128 MB
- ~96 MB available to user, rest for metadata

Content encrypted while in DRAM, decrypted when brought to CPU

- Plaintext in CPU caches

SGX Multithreading Support

SGX allows multiple threads to enter same enclave simultaneously

- One thread control structure (TCS) per thread
- Part of enclave, reflected in measurement

TCS limits number of enclave threads

- Upon thread entry TCS is blocked and cannot be used by another thread

Each TCS contains address of entry point

- Prevents jumps into random locations inside of enclave

SGX Paging

SGX provides mechanism to evict EPC page to unprotected memory

- EPC limited in size

Paging performed by OS

- Validated by HW to prevent attacks on address translations
- Metadata (MAC, version) kept within EPC

Accessing evicted page results in page fault

- Page is brought back into EPC by OS
- Hardware verifies integrity of page
- Another page might be evicted if EPC is full

SGX SDK Code Sample

SGX application: untrusted 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);
}
```

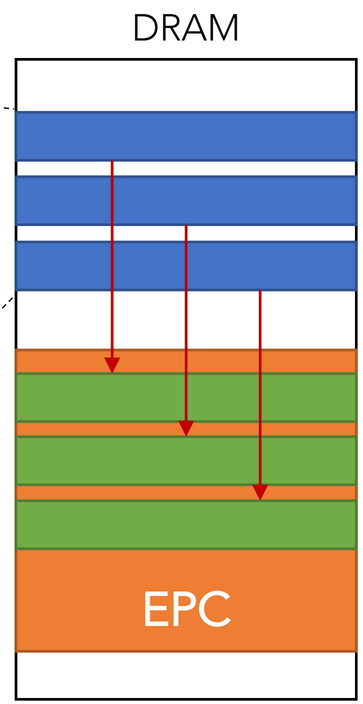
Server:

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

SGX 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 populated using special instruction (EADD)

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

Both data & code copied before execution commences in enclave

SGX Enclave Construction

Enclave contents distributed in plaintext

- Must not contain any (plaintext) confidential data

Secrets provisioned after enclave constructed and integrity verified

Problem: what if someone tampers with enclave?

- Contents initially in untrusted memory

```
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);
}
```

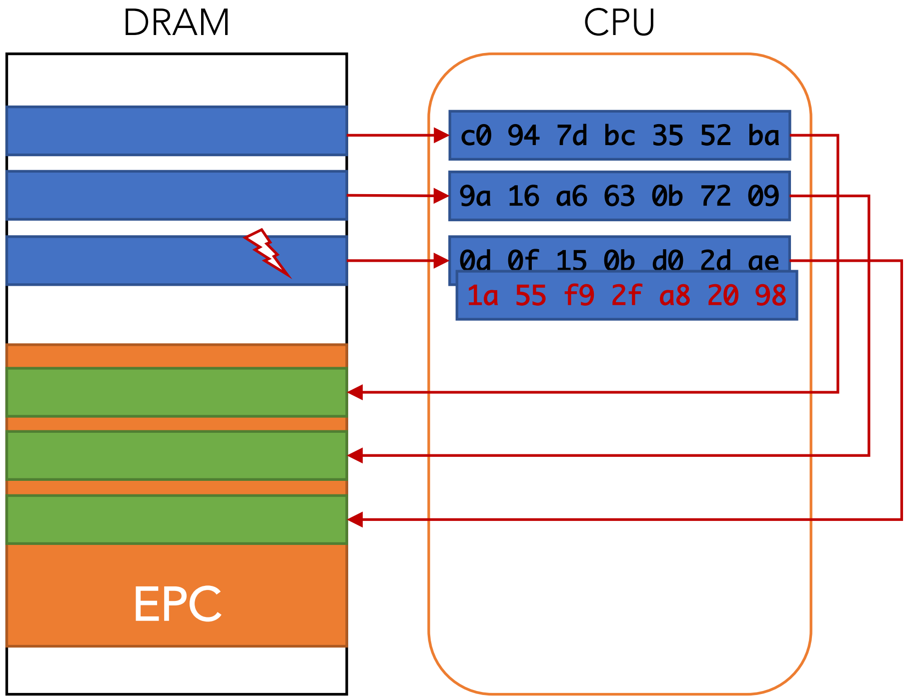
Write unencrypted response to outside memory

SGX Enclave Measurement

CPU calculates enclave measurement hash during enclave construction

- Each new page extends hash with page content and attributes (read/write/execute)
- Hash computed with SHA-256

Measurement can be used to attest enclave to local or remote entity



CPU calculates enclave measurement hash during enclave construction

Different measurement if enclave modified

SGX 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 same CPU

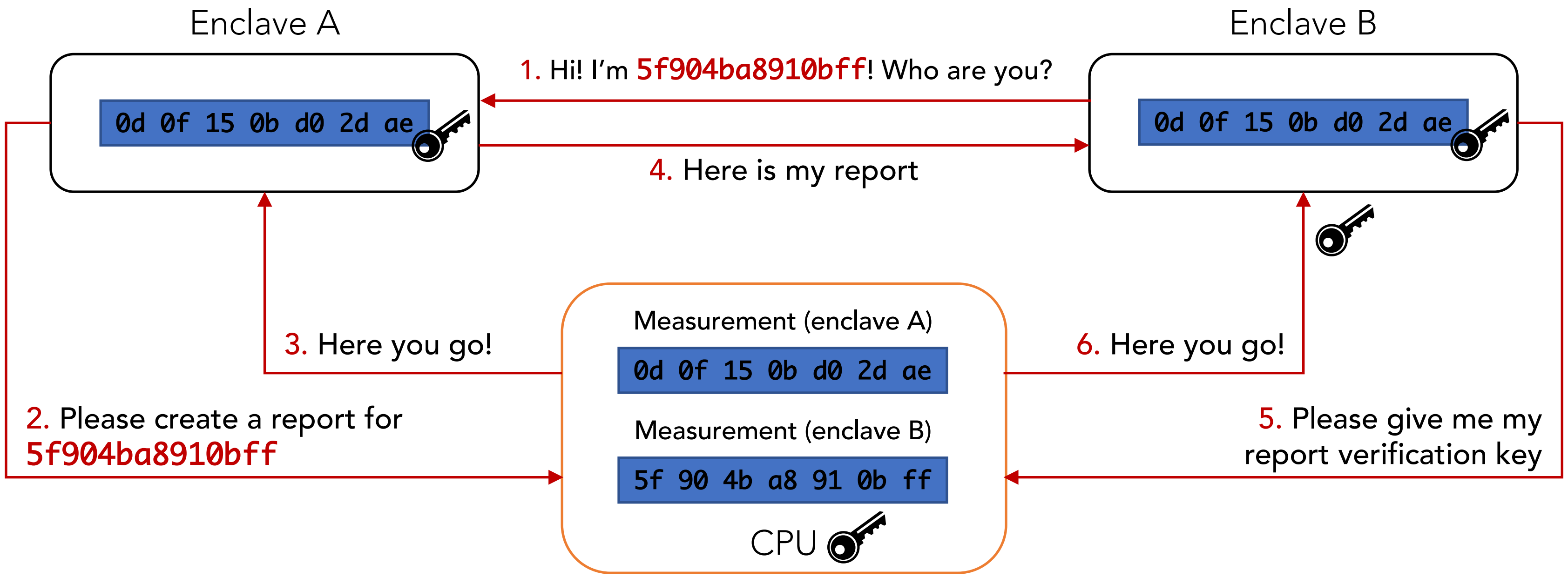
Remote attestation

- Prove enclave's identity to remote party

Once attested, enclave can be trusted with secrets

Local Attestation

Prove identity of A to local enclave B



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

Remote Attestation

Transform local report to 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 unique key fused during manufacturing

- Intel maintains database of keys

Remote Attestation

PE communicates with Intel attestation service

- Proves it has key installed by Intel
- Receives asymmetric attestation key

QE performs local attestation for enclave

- QE verifies report and signs it using attestation key
- Creates quote that can be verified outside platform

Quote and signature sent to remote attester, which communicates with Intel attestation service to verify quote validity

SGX Limitations & Research Challenges

Amount of memory enclave can use needs to be known in advance

- Dynamic memory support in SGX v2

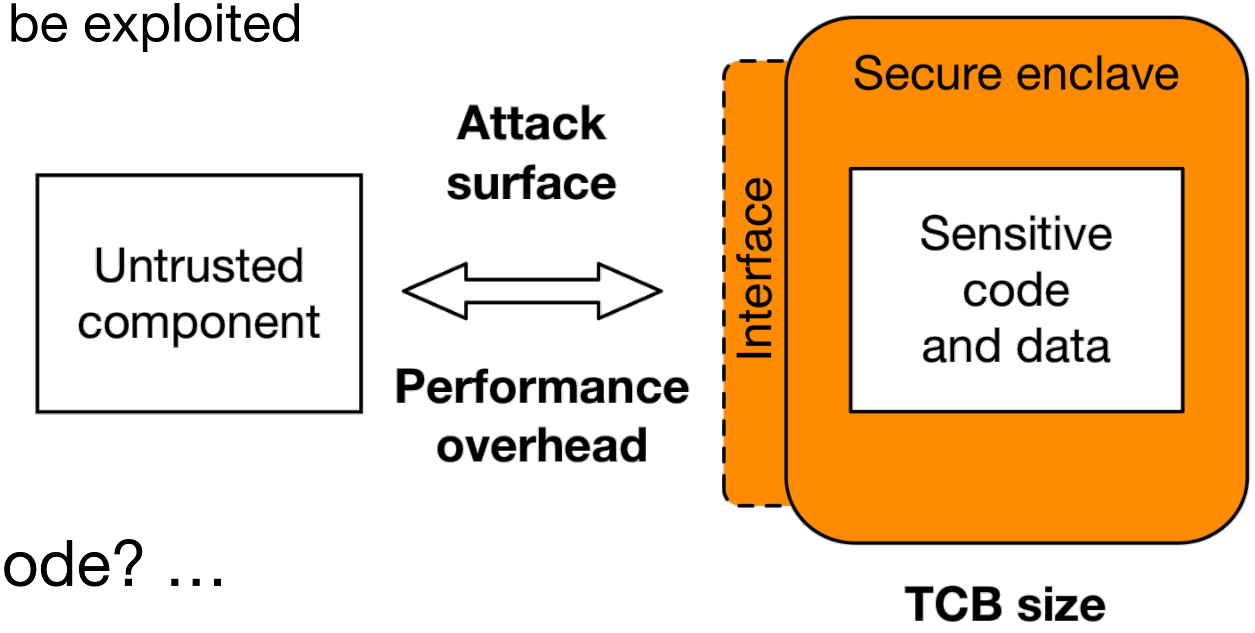
Security guarantees not perfect

- Vulnerabilities within enclave can still be exploited
- Side-channel attacks possible

Performance overhead

- Enclave entry/exit costly
- Paging very expensive

Application partitioning? Legacy code? …



SGX-Spark

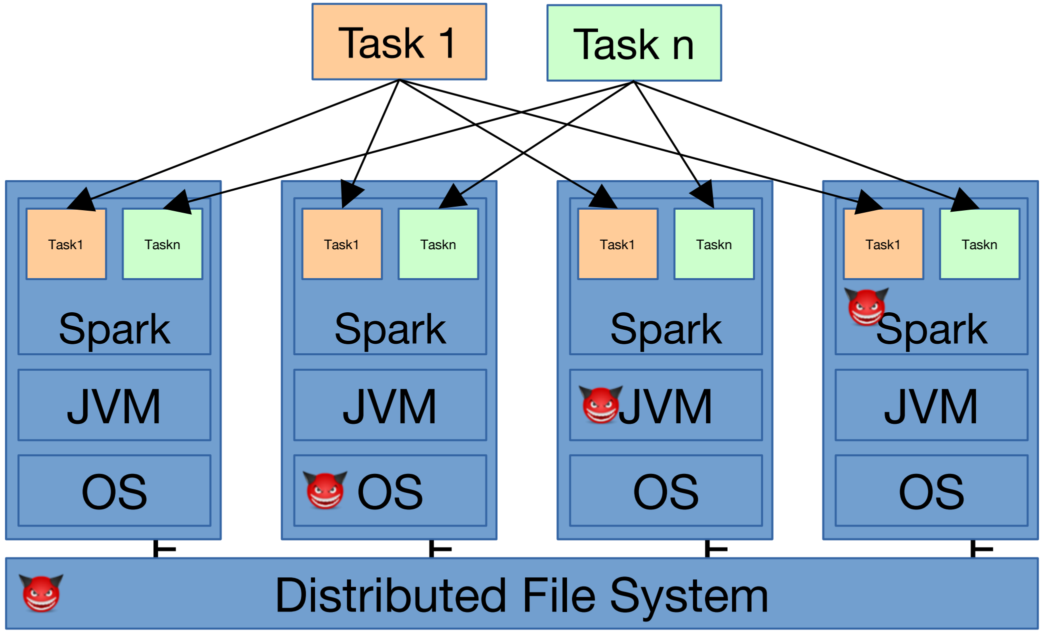
Secure Big Data Processing

Processing of large amounts of sensitive information

Outsourcing of data storage and processing

Cloud provider can access processed data

- Not acceptable for number of industries



```
def main(args: Array[String]) {
  new SparkContext(new SparkConf())
    .textFile(args(0))
    .flatMap(line => {line.split(" ")})
    .map(word => {(word, 1)})
    .reduceByKey{case (x, y) => x + y}
    .saveAsTextFile(args(1))
}
```

Secure Machine Learning

Secure **machine learning (ML)** killer application for Maru

- Resource-intensive thus good use case for cloud usage
- Raw training data comes with security impliations

Complex implementations of ML algorithms cannot be adapted for SGX

- Consider Spark MLlib with 100s of algorithms

Challenges

- Extremely **data-intensive** domain
- Must support **existing frameworks** (Spark, TensorFlow, MXNet, CNTK, …)
- ML requires **accelerators** support (GPUs, TPUs, …)
- Prevention of **side-channel** attacks

State of the Art

Protect confidentiality and integrity of tasks and input/output data

Opaque [Zheng, NSDI 2017]

- Hide access patterns of distributed data analytics (Spark SQL)
- Introduces new oblivious relational operators
- Does not support arbitrary/existing Scala Spark jobs

VC3 [Schuster, S&P 2015]

- Protects MapReduce Hadoop jobs
- Confidentiality/integrity of code/data; correctness/completeness of results
- No support for existing jobs → Re-implement for VC3

SGX Support for Spark

SGX-Spark

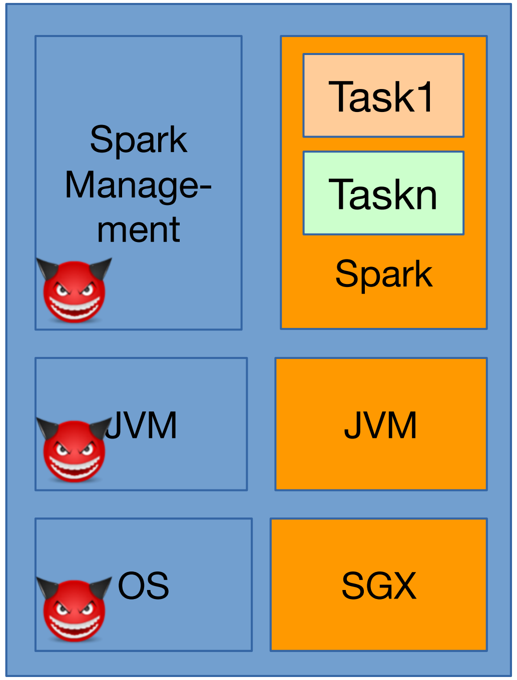
- Protect data processing from infrastructure provider
- Protect confidentiality & integrity of existing jobs
- No modifications for end users
- Acceptable performance overhead

Idea:
Execute only sensitive parts of Spark inside enclave

- Code that accesses/processes sensitive data

Code outside of enclave only accesses encrypted data

- Partition Spark
- Run two collaborating JVMs, inside enclave and outside of enclave



Supporting Managed Runtimes in SGX Enclaves

Many applications need runtime support

- JVM
- .NET
- JavaScript/V8/Node.js

Requires complex system support

- Dynamic library loading
- Filesystem support
- Signal handling
- Complete networking stack

SGX-LKL Architecture

Runs **unmodified Linux applications** in SGX enclaves

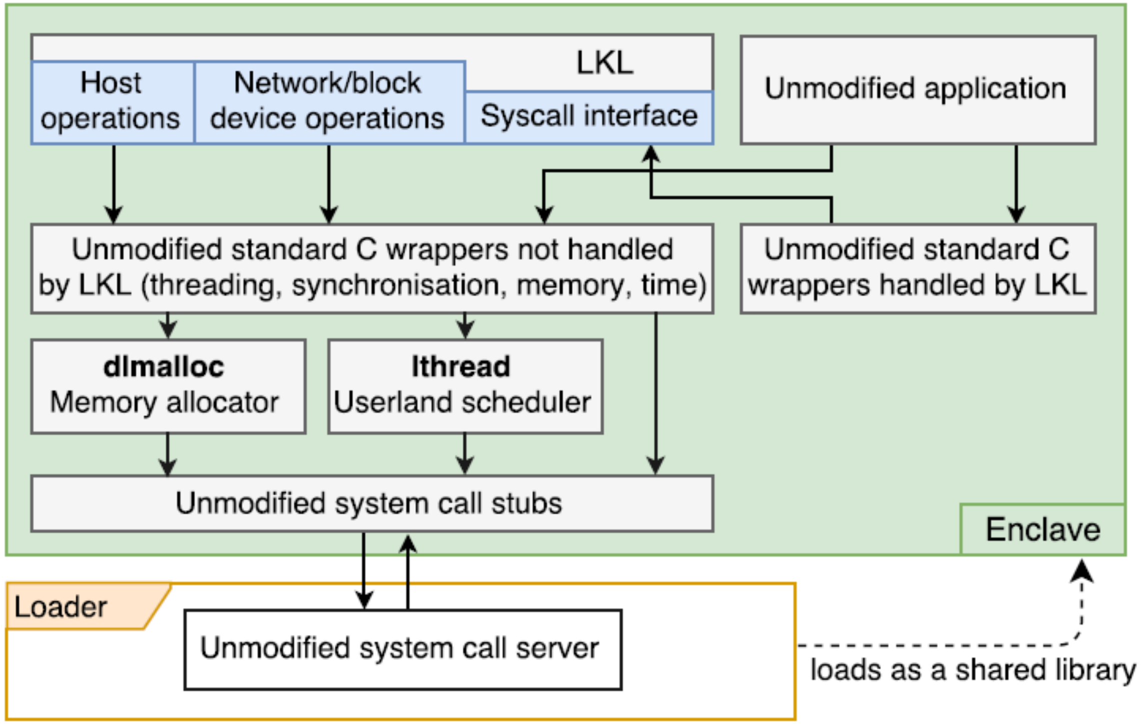
Applications and dependencies provided via **disk image**

Full Linux kernel functionality available

Custom memory allocator

User-level threading

- In-enclave synchronisation primitives



LKL
Host operations
Network/block device operations
Syscall interface
Unmodified application
Unmodified standard C wrappers not handled by LKL (threading, synchronisation, memory, time)
Unmodified standard C wrappers handled by LKL
dlmalloc
Memory allocator
lthread
Userland scheduler
Unmodified system call stubs
Enclave
Loader
Unmodified system call server
loads as a shared library

Challenges & Current State

1. **Partitioning Spark**

2. **Data movement between JVMs**

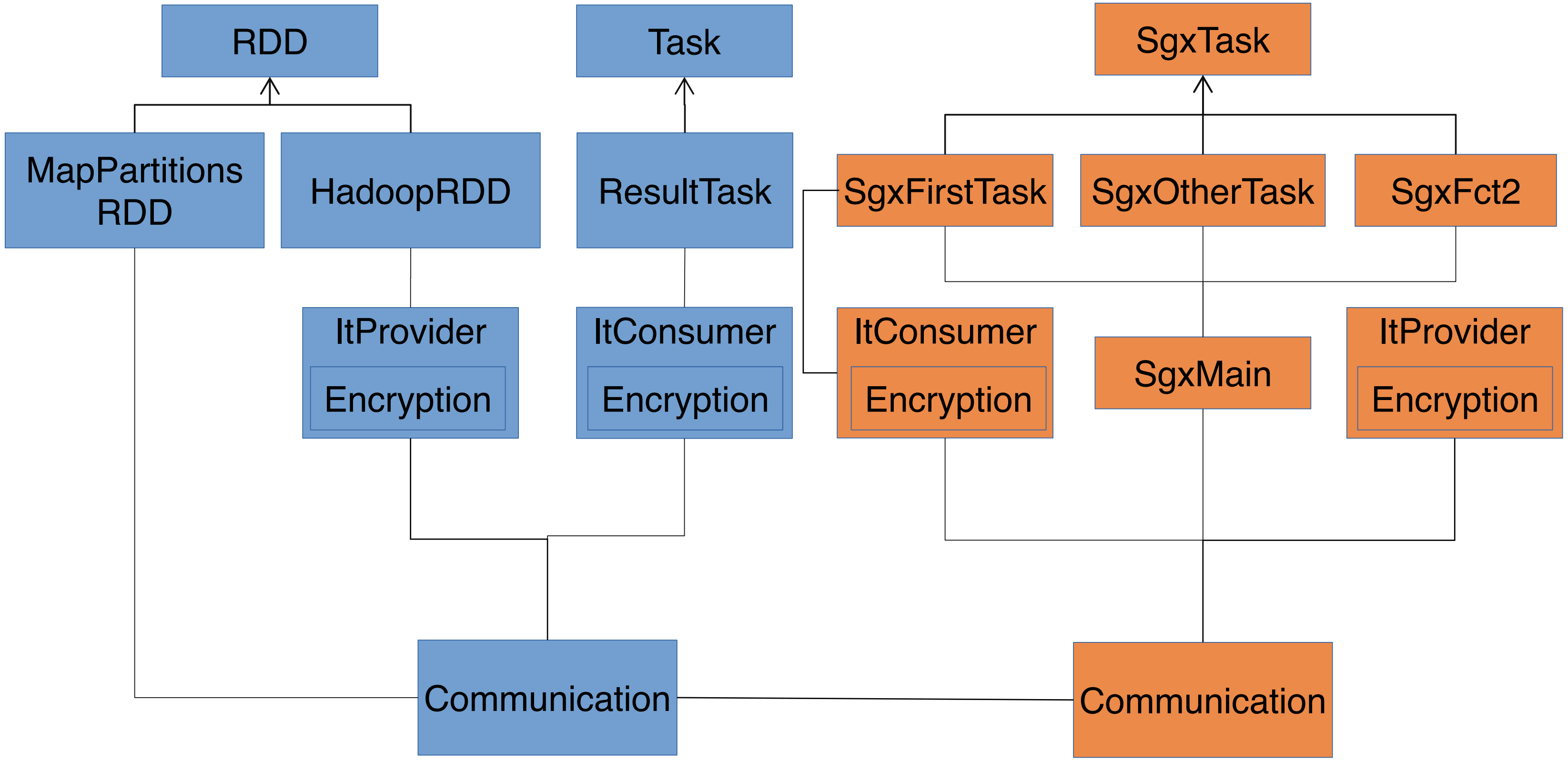
3. **Memory efficiency**

1. Partitioning Spark

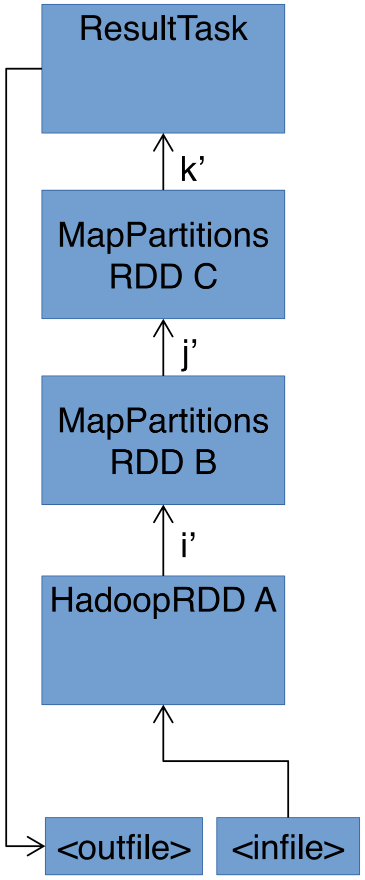
Goal: Move minimal amount of Spark code to enclave

Outside	Enclave
HadoopRDD Provide iterator over input data partition (encrypted)	
MapPartitionsRDD it Execute user-provided function (f) (eg `flatMap(line => {line.split(" ")})`) (i) Serialise user-provided function `f` (ii) Send `f` and `it` to enclave JVM f,it (iv) Receive result iterator `it_result` it2 = it_result	(iii) Decrypt input data (iv) Compute `f(it) = it_result` (v) Encrypt result
ExternalSorter Execute user-provided reduce function g g,it2 (eg `reduceByKey{case (x, y) => x + y}`) it2_result	(iii) Decrypt input data (iv) Compute `g(it2) = it2_result` (v) Encrypt result
ResultTask Output results	

1. Partitioning Spark



1. Partitioning Spark



🟦 Outside

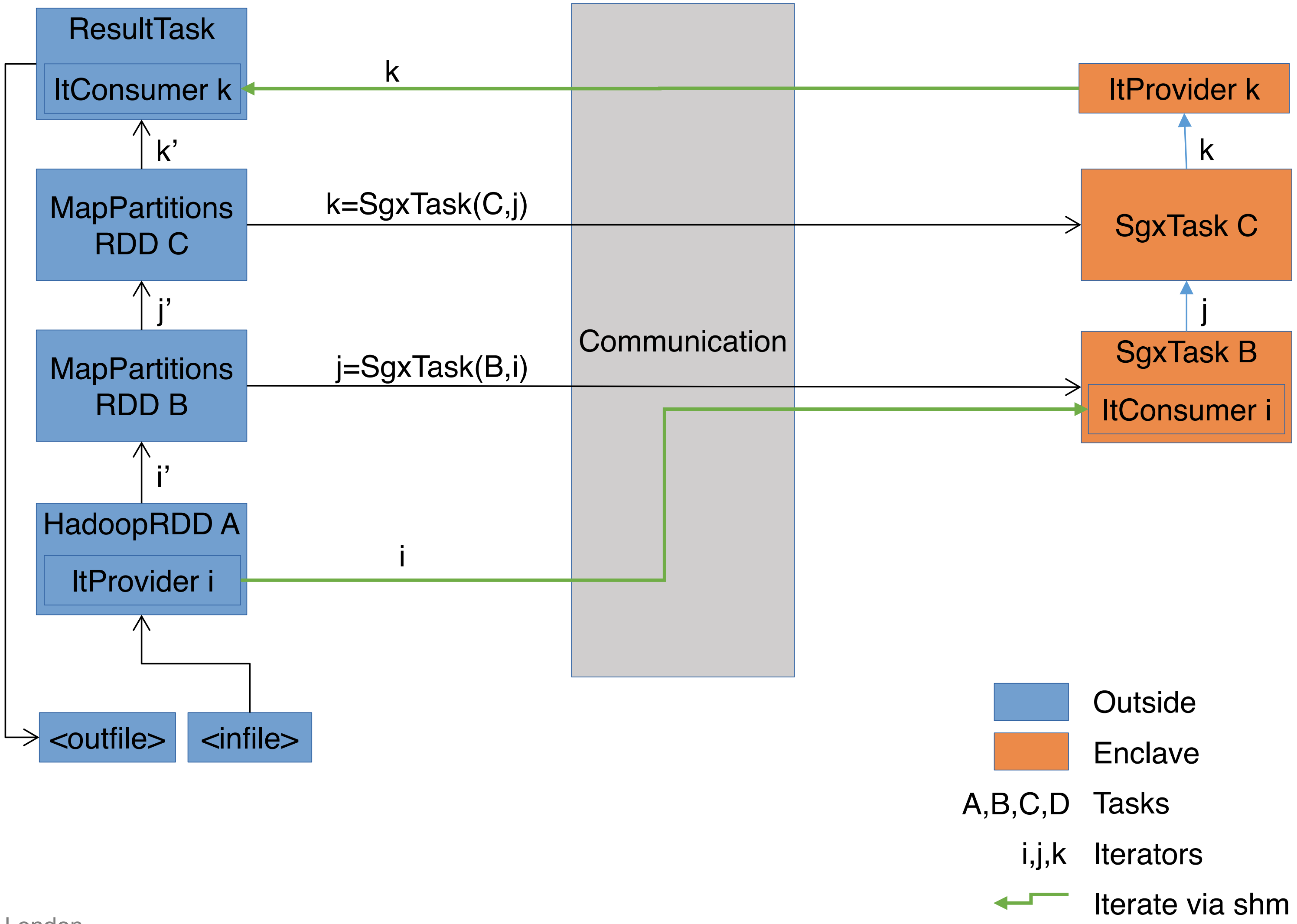
🟧 Enclave

A,B,C,D Tasks

i,j,k Iterators

↵ Iterate via shm

1. Partitioning Spark



2. Data Movement between JVMs

Goal: **Shared memory**

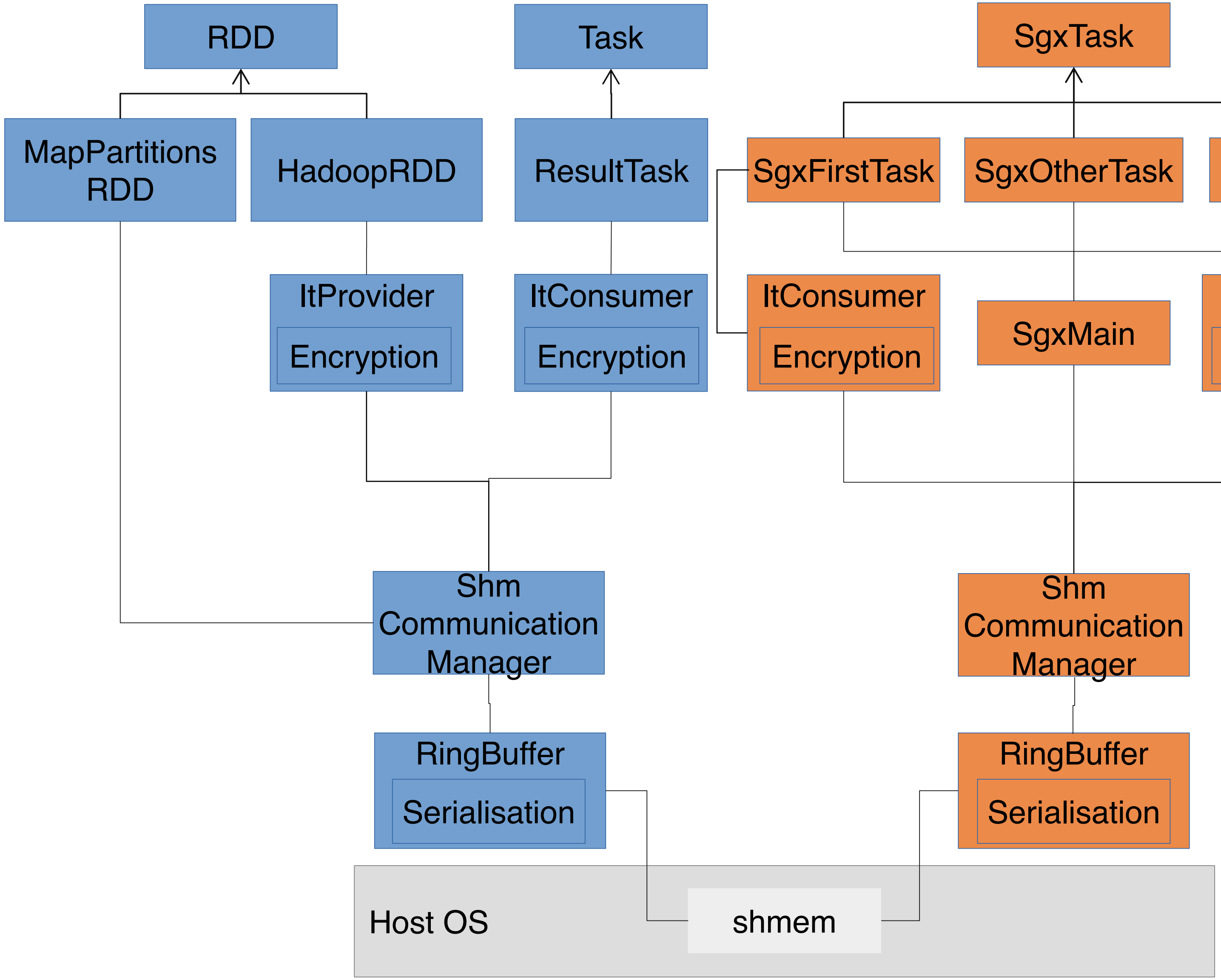
Use use host OS shared memory between two JVMs

- Outside access by enclave JVM

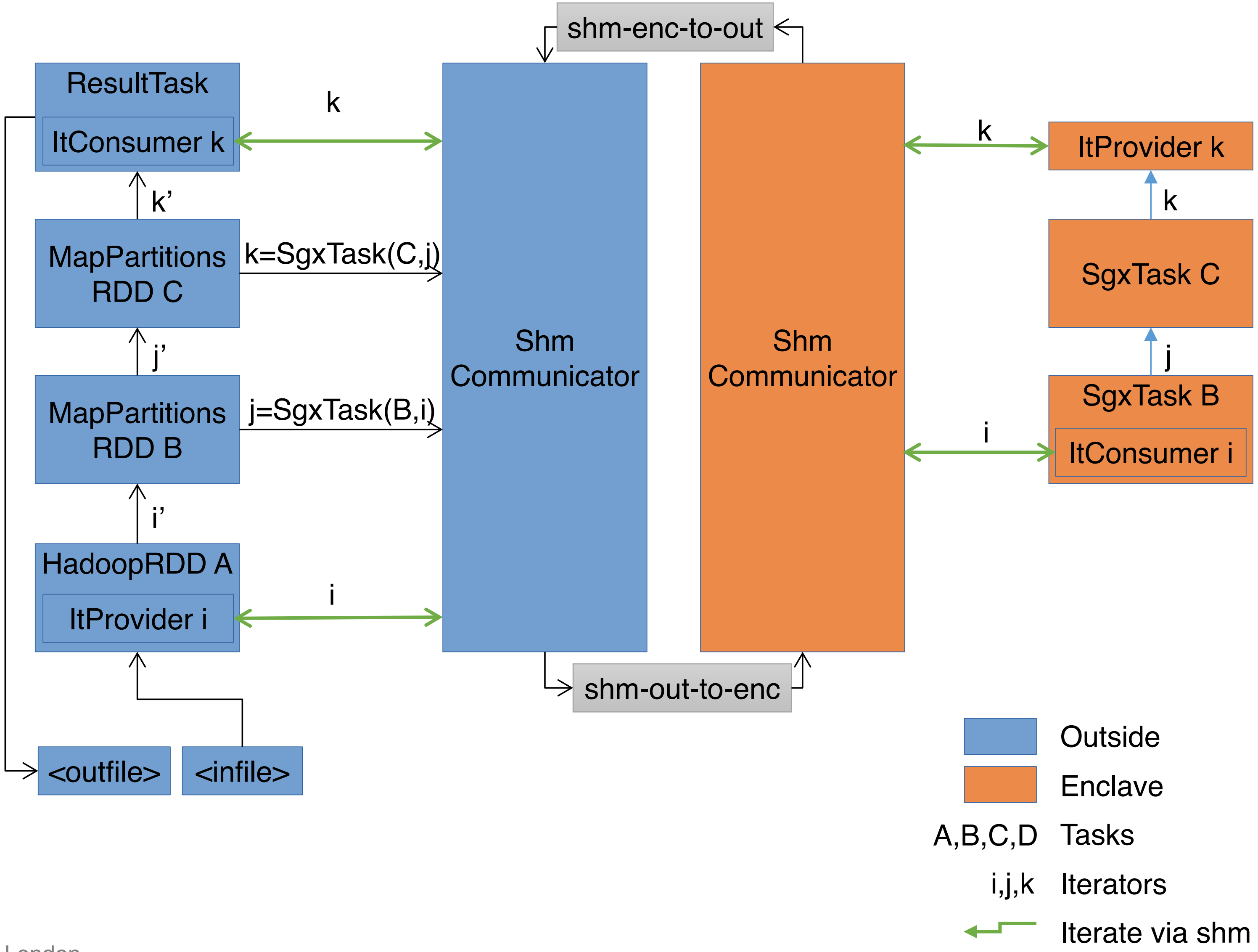
Manage shared memory between outside and enclave

Implement high-level read/write primitives

1. Partitioning Spark



2. Data Movement between JVMs



3. Memory Efficiency

Only ~80 MB available

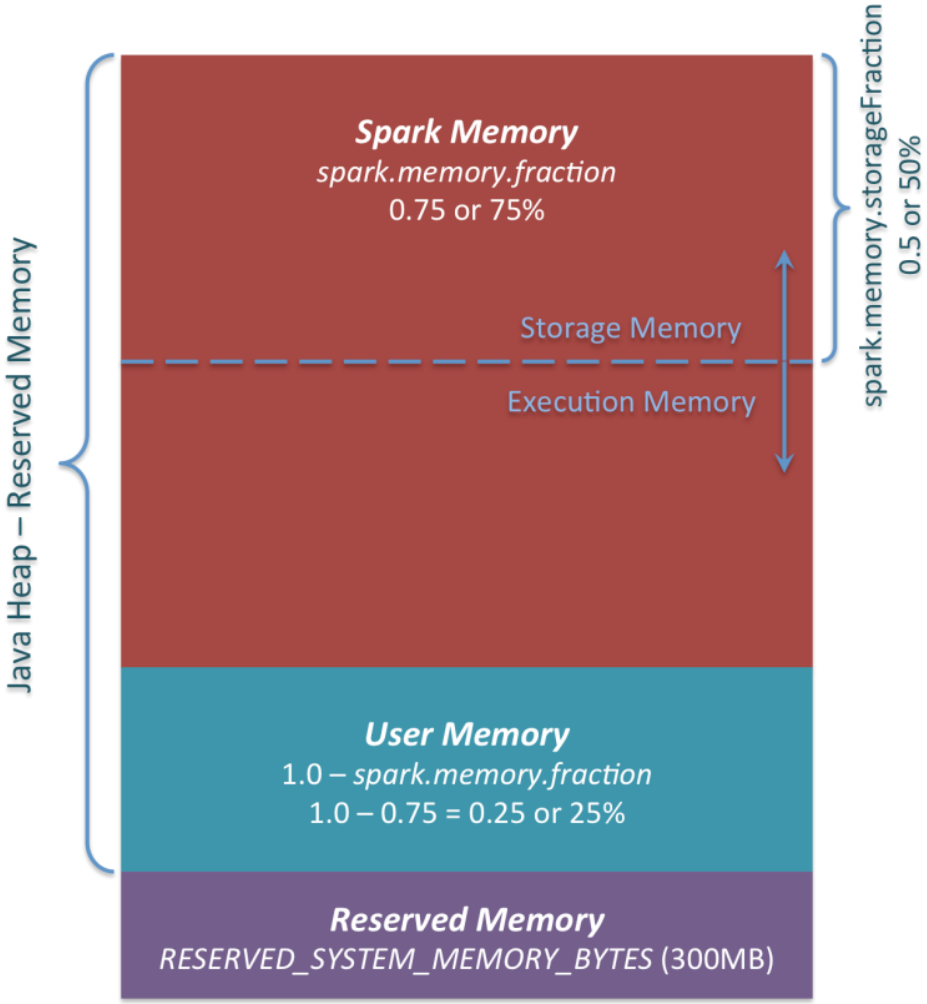
Native Spark Worker: ~320 MB 😢

Our Spark in-enclave code: 94 MB

- With Java flags (-XX:InitialCodeCacheSize=2m
  -XX:ReservedCodeCacheSize=2m -Xms2m -Xmx3m
  -XX:CompressedClassSpaceSize=2m
  -XX:MaxMetaspaceSize=8m
  -XX:+UseCompressedClassPointers) → 50 MB

SGX-LKL:
8 MB Kernel + 18 MB other → 26 MB

- Working on memory efficiency
- Eg thread stack size, kernel size, deactivating features



Maru Research Directions

1. **Security model for shielded data science jobs**
   - How to harden shielded jobs? How to deal with vulnerabilities, bugs?
   - What about external dependencies/libraries?
2. **Integration of language runtimes with secure enclaves**
   - How to integrate SGX support for the JVM?
   - What is the right programming model for SGX enclaves?
3. **Unikernel support for secure enclaves**
   - How to support existing legacy binaries?
   - How to build type-safe minimal secure enclaves for data science jobs?
4. **Prototype platform implementation and evaluation**
   - Integration with Apache Flink or other dataflow frameworks
5. **Dataflow attacks and mitigations strategies**
   - What attacks are possible by observing encrypted dataflows?
   - Can we apply techniques for unobservable communication?

SGX-Spark: **System Goals**

Primary System Goals:

1. Ensure **integrity & confidentiality** for tasks, input data & output results

2. Support **arbitrary** workloads and tasks

3. **Low performance overhead** (throughput and latency)

SGX-Spark: **Summary of V1**

Version 1 (V1):

1. **Prototype** of Apache Spark using TEEs (Intel SGX)
   - Shows the **feasibility** of the approach

2. Each **worker partitioned** into trusted and untrusted JVM:
   - Minimize the trusted computing base (**TCB**)
   - Provides **confidentiality** for input data & result

3. Use "**pull**" model for cross JVM communication:
   - Trusted JVM "asks" for data, objects, context
   - Blocking message requests → **high overhead**

SGX-Spark: **Goals of V2**

Version 2 (V2):

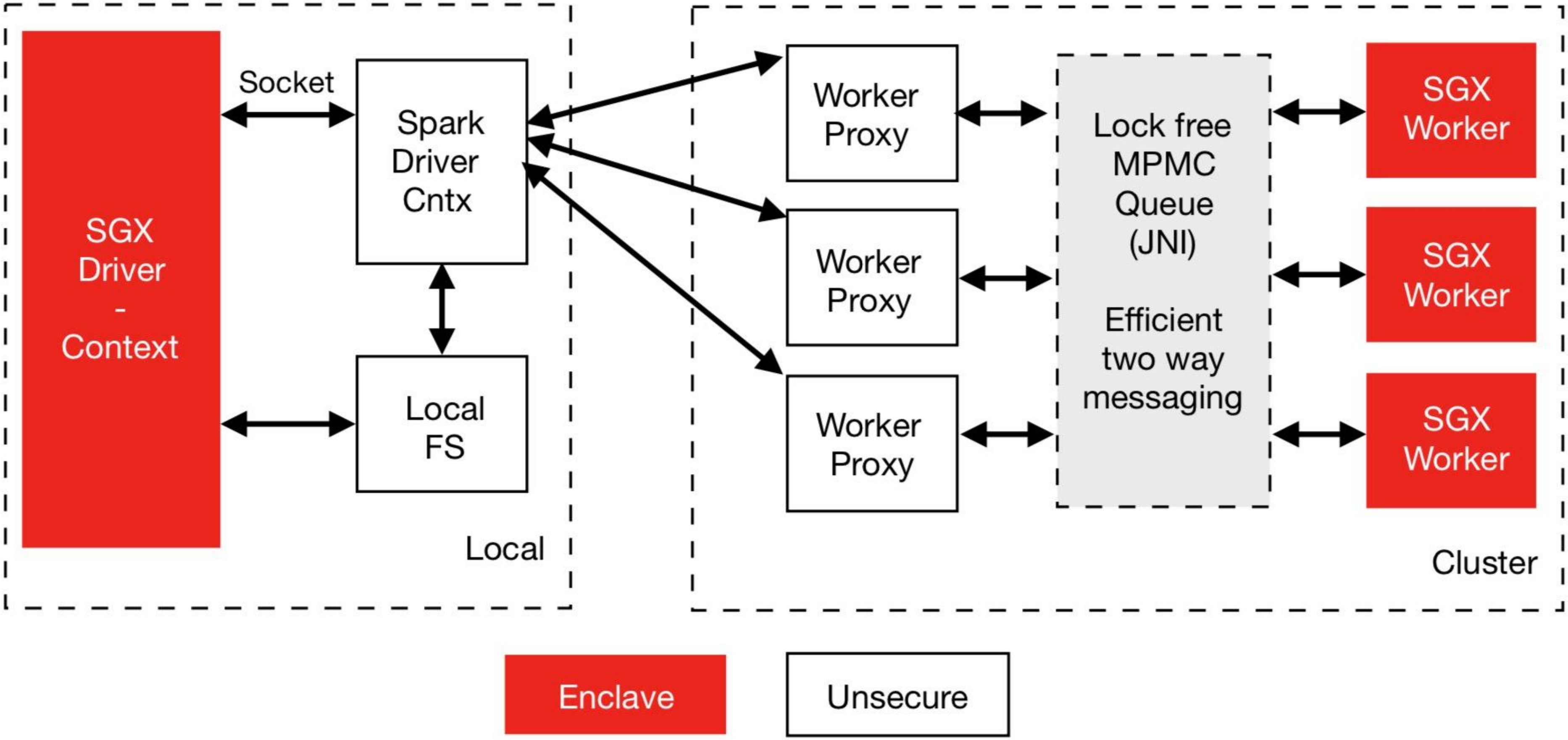
1. Provide **integrity** as well as confidentiality
   - Protect driver with TEE to enforce **job integrity**

2. Improve performance with a “**push**” model:
   - Push data into trusted JVM to avoid blocking requests (e.g. use pre-fetching and leverage EPC memory)
   - Batch messages (reduce communication costs)
   - Use lock free and highly parallelized comm. channels

3. Support generic RDDs / avoid assumptions about data layout (e.g. avoid DataFrames and DataSets)

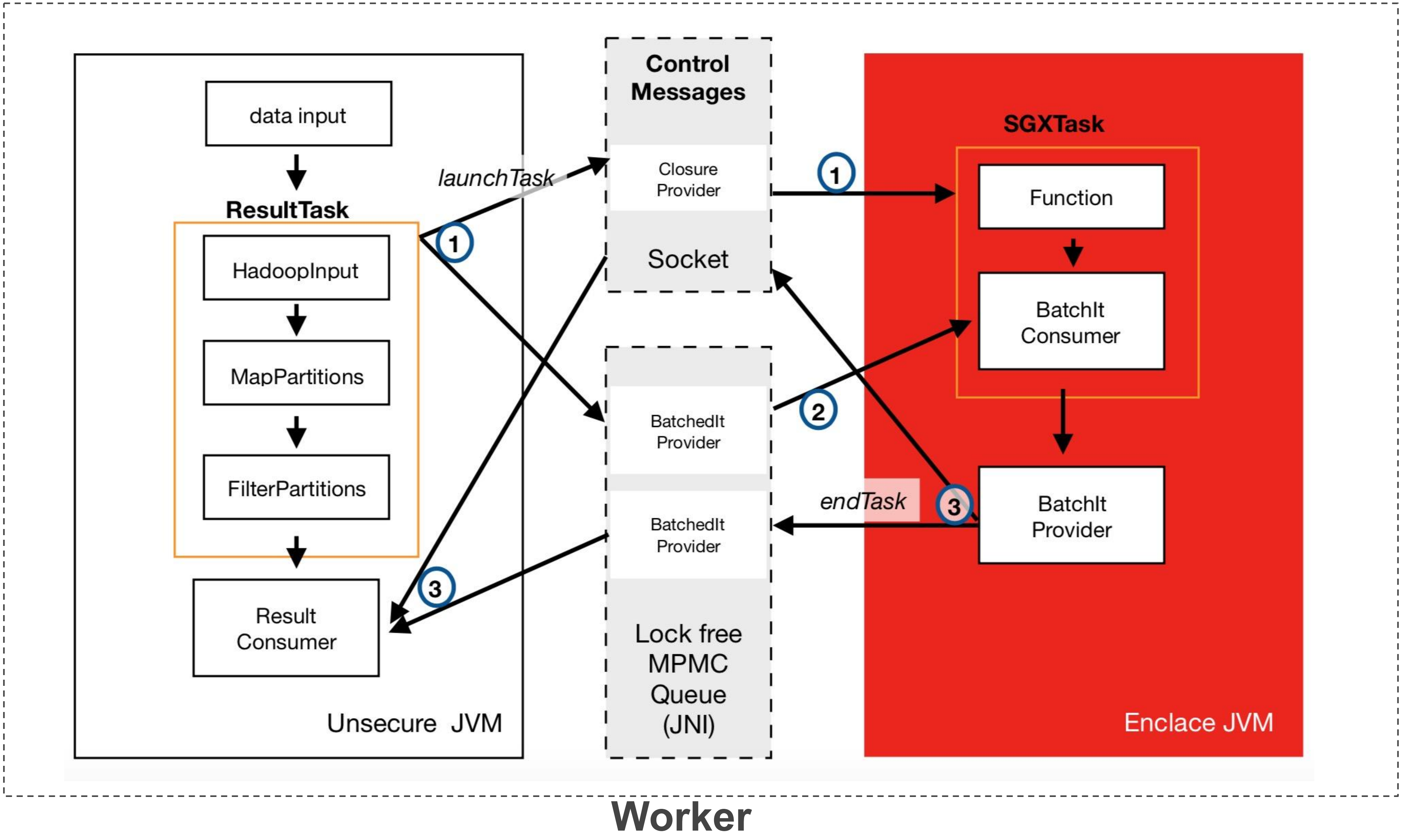
SGX-Spark: **Design of V2**

High-level Overview:



SGX-Spark: **Worker Execution**

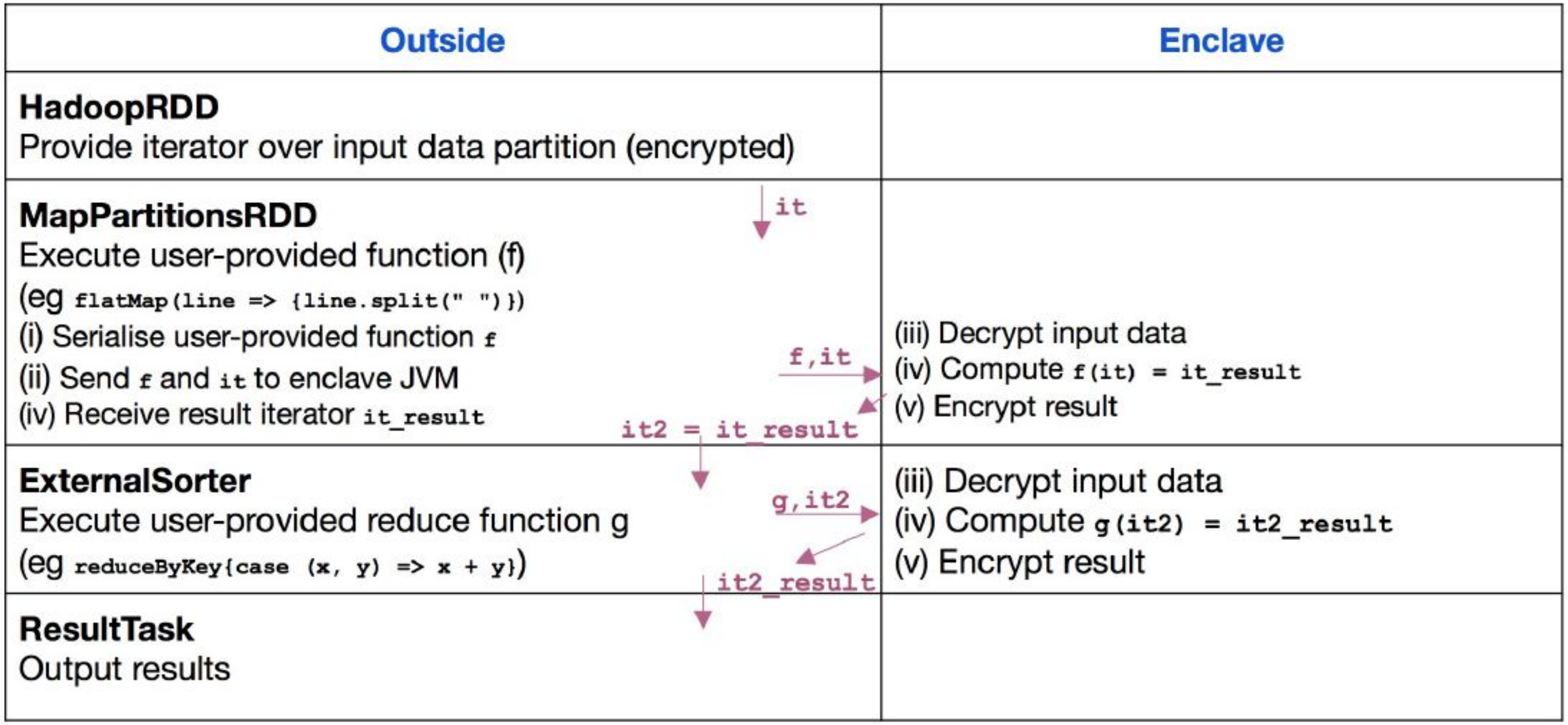
Imperial College London



SGX-Spark: **Design of V2**

Imperial College London

Spark Worker Execution:



Outside	Enclave
HadoopRDD Provide iterator over input data partition (encrypted)	
MapPartitionsRDD Execute user-provided function (f) (eg `flatMap(line => {line.split(" ")})` (i) Serialise user-provided function `f` (ii) Send `f` and `it` to enclave JVM (iv) Receive result iterator `it_result`	(iii) Decrypt input data (iv) Compute `f(it) = it_result` (v) Encrypt result
ExternalSorter Execute user-provided reduce function g (eg `reduceByKey{case (x, y) => x + y}`)	(iii) Decrypt input data (iv) Compute `g(it2) = it2_result` (v) Encrypt result
ResultTask Output results	