Technische Universität Braunschweig

Institute of Operating Systems and Computer Networks

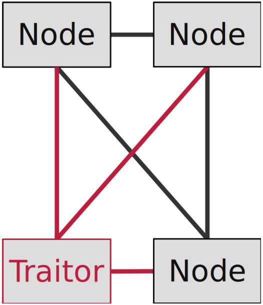
Resilient Byzantine Fault-Tolerance

Using Multiple Trusted Execution Environments

Markus Becker, October 14, 2021

Byzantine Agreement

- Goal: Reach consensus across multiple machines
- Application: State-Machine Replication (SMR)
  - Agree & Order requests
  - Execute deterministic operation



Byzantine Failures

- Malicious party exhibits arbitrary behaviour

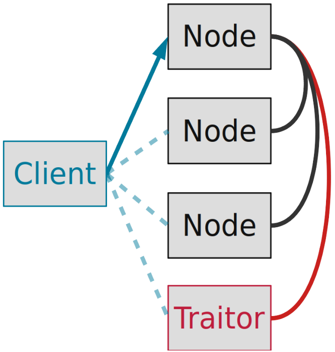
Fault Tolerant Systems

- Resilient against Byzantine Failures
- Operates correctly despite faults

Byzantine Fault Tolerant (BFT) Protocols

Example protocol: PBFT M. Castro and B. Liskov '99

1. Replicated nodes act as servers
2. Client sends request
3. Communication rounds between replicas
   - Ordering and agreement
   - Requires $n \geq 3f + 1$ for f faults
4. Client receives responses
5. Client performs voting on results



BFT Protocol Deployment

Usage in permissioned Blockchains

- BFT for ordering and agreement
- BaaS: Nodes & Infrastructure by cloud provider
  - Amazon Managed Blockchain
  - Azure Blockchain Service
- Agreement using specific protocol

But we do not want to have to trust the cloud provider!

Trusted Execution

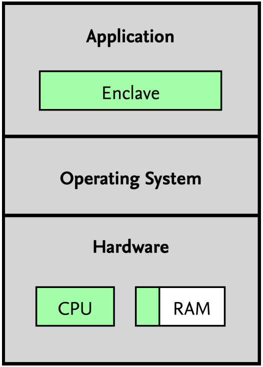
Features of most Trusted Execution Environments (TEEs):

- Execution of signed code on third parties machine
  - Local and remote attestation
- Confidentiality over host
  - Fully hardware-encrypted memory
- Reduced the chance of bugs
  - Small TCB

But we still have to assume Byzantine failures!

Intel Secure-Guard-Extensions (SGX)

- Trusted Execution Environment
- Extension of x86
- Exclusively on Intel CPUs
- Transparently encrypted memory
- Ring-3 only execution
- (Remote) Attestation



But enclaves can still contain bugs (or have other weaknesses) [1]

[1] Weichbrodt et al., AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves, (ESORICS'16)

Robustness

Reduce chance that practical BFT system enters irreparable state.

Ordinary

- Increase number of replicas
- Use safe programming strategies / frameworks
- Diversification (Lazarus'19)
  - Code & OS
  - Hardware
  - Passwords
  - Admins
- Rejuvenation

Invasive

- Trusted Execution
  - No golden bullet
  - Small *validated* TCB
  - Interface with OS needed
- Separation by functionality
  - Logically
  - Physically

BFT Partitioning

Naïve Solution:

- Entire protocol in TEE

Problem:

↯ Attacks on enclave

Solution:

- Separate agreement protocol

↯ Attacker in enclave $\Rightarrow$ liveness

↯ Attacker in f enclaves $\Rightarrow$ safety & liveness

- Depend on quorum decisions for safety

Thesis Goals

Combination of robustness features as SplitBFT

- Trusted Execution (SGX) in stronger fault model
- Separating into independent compartments

Goals

- Improve safety & resilience using TEEs
  - Tolerate up to f faults per compartment type
- Keep confidentiality as long as possible
  - Sensitive data in only one compartment type

Requirements for SplitBFT

Split PBFT into small protocol units for compartments:

Performance

- Efficient memory management
- Avoiding SGX overhead
- Efficient shim

Safety

- Independent compartments
- Security-sensitive functions isolated
- Eliminating shared state

Compartmentalization

Splitting State-Machine-Replication

- SMR is often physically split into Clients and Replicas
- Replicas are logically split into Agreement and Execution (SOSP'03)

We recognise further opportunity to split based on **quorum decisions**:

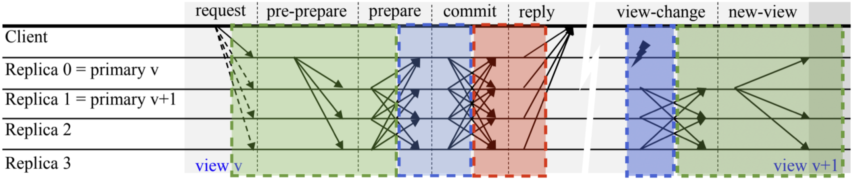
$$\text{BFT} \longrightarrow \text{Clients} + \text{Replicas}$$

$$\text{Replica} \longrightarrow \text{Agreement} + \textbf{Execution}$$

$$\text{Agreement} \longrightarrow \textbf{Preparation} + \textbf{Confirmation}$$

SplitBFT: Normal Operation

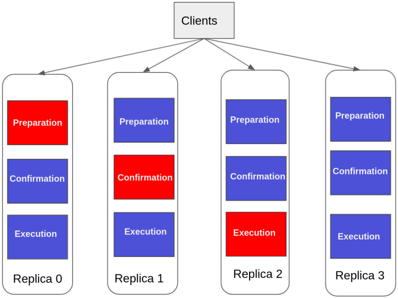
Preparation, Confirmation, Execution



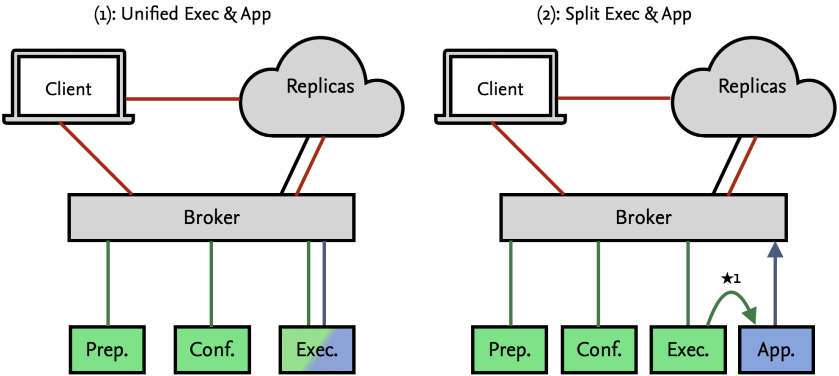
1. Collect symmetrically encrypted operations in untrusted memory
   - Only client and execution compartment can decrypt
2. Liveness decisions outside of compartments
3. Verify fine-grained asymmetric signatures in compartments
4. Follow PBFT-like operation

Compartment's responsibilities

1. Preparation:
   - $\Leftarrow$ Receive requests, pre-prepares
   - ☐ Order requests
   - $\Rightarrow$ Send pre-prepares, prepares
2. Confirmation:
   - $\Leftarrow$ Receive $\geq 2f$ prepares
   - $\Rightarrow$ Send commits
3. Execution:
   - $\Leftarrow$ Receive $\geq 2f + 1$ commits
   - ☐ Execute requests
   - $\Rightarrow$ Send replies



Option: Execution & Application split



Checkpointing and View-Changes

Design consideration:

- Independence
- Safety based on quorums

Garbage-collection & liveness:

- Checkpoints
  - Application state only in Execution compartments
  - Allows removing old messages
- View-Changes
  - View *required* in all compartments
  - Cannot trust a global variable or even local “View-Compartment”

$\Rightarrow$ New-View messages are broadcast to all compartments

Fault Model

- Faulty enclave escapes to replica
- Faulty replica cannot enter enclave
- Independent faults in enclaves
- Require quorum to advance protocol

$\Rightarrow$ Integrity as long as at most f faults per enclave type

$\Rightarrow$ Confidential as long as execution compartments non-faulty

Themis

- IBR's own BFT Framework!
- Written in Rust: memory-safe, systems language
- Protocols:
  - PBFT
  - Railchain
- Implemented applications:
  - Benchmark-Counter
  - YCSB-KVS

Implementation

1. Setup for Teaclave SGX SDK
2. Integration into Themis
   - Dependencies, dependencies, dependencies
3. Structured communication between TCB and Themis
4. Applications

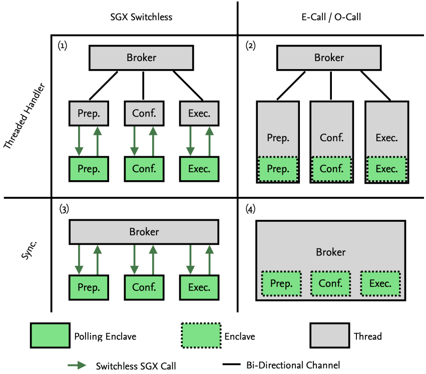
Setup

- Rust `nightly-2021-02-17`
- Themis `@master`
- Teaclave SGX SDK >`1.1.3`
  - `1.1.3` incompatible with Themis
  - `master` incompatible with `SGX World`

⇒ Own `SGX World` forks & docs + wiki
  - `bytes-sgx`, `ring-sgx`, `serde-sgx`, `msgpack-rust-sgx`, ...

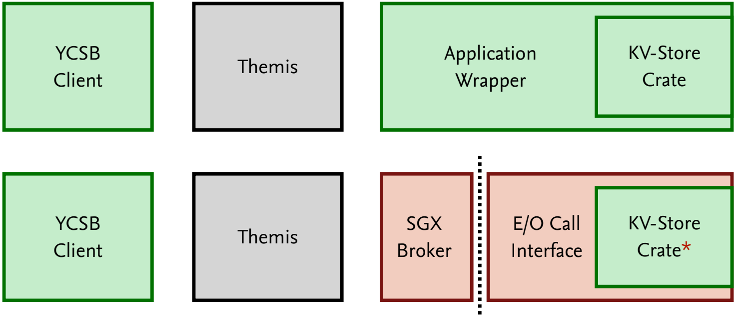
SGX Broker Layer

- Compatible management structure for enclaves
- Efficient memory management
- Minimize SGX overhead
- Translation layer between Themis and TCB
- Deciding asynchronous/synchronous operation



Adaptation of KV-Store implementation

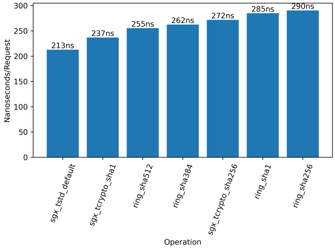
Writing TCB to allow swapping of non-SGX and SGX SMR without changing the client:



Wrapping for transparent interation with Themis is non-trivial

Evaluation

- Benchmarking `#[no_std]` crates in SGX
- Deploying and measuring integration tests
- Measure with benchmark application
- Measure with YCSB-KVS application



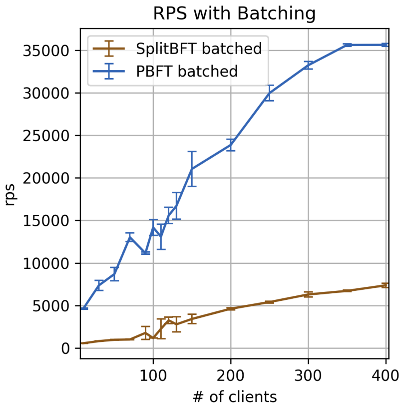
Example benchmark of hashing inside enclaves.

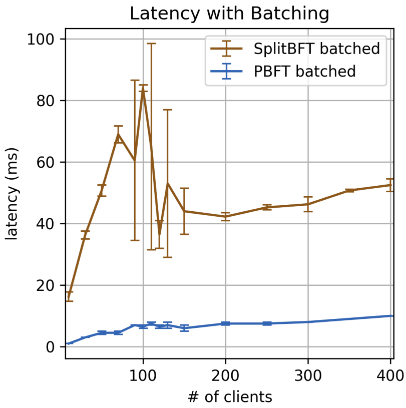
Benchmark Application

Compare PBFT against SplitBFT

- Replicated on ssgx machines
  - 4× Intel Xeon E3-1230 v5
  - 1 Gbps
  - 32 GB RAM
  - 94.5 MiB EPC Size
- Threaded clients on dsgx machines
  - 4× Intel Core i7-6700
  - 1 Gbps
  - 24 GB RAM

Benchmark Application





Benchmark Application Evaluation

- Cost of ECalls and OCalls spread across requests when batching
  - Performance bottleneck: Sending replies

PBFT replying to batch

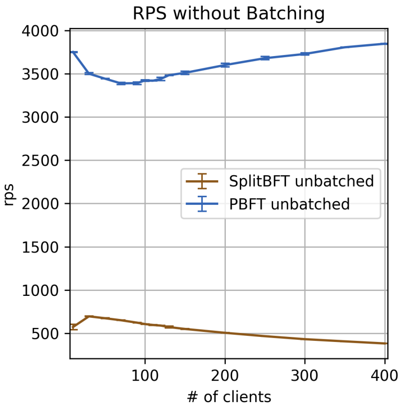
1. Create reply & sign
2. Network send (serialize)

SplitBFT replying to batch

1. Create reply & sign
2. *Batch with other replies*
3. *Serialize batch*
4. *Perform OCall*
5. *Deserialize OCall batch*
6. Network send (serialize)

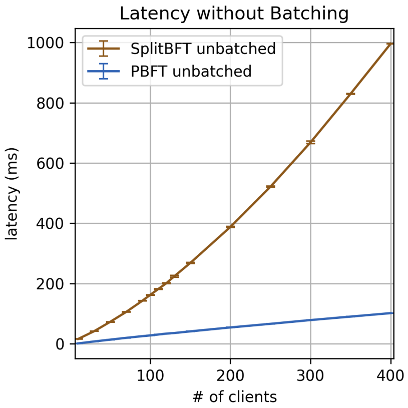
Benchmark Application Evaluation

Comparison without Batching
ECalls and OCalls of ordering each request individually dominate performance.



Benchmark Application Evaluation

Comparison without Batching
ECalls and OCalls of ordering each request individually dominate performance.



YCSB-KVS Application

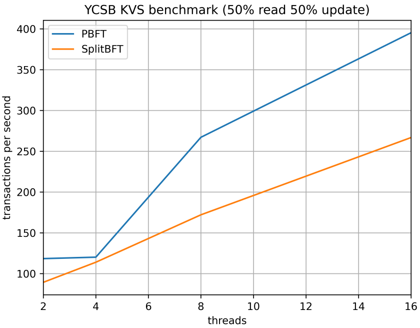
Distributed benchmark

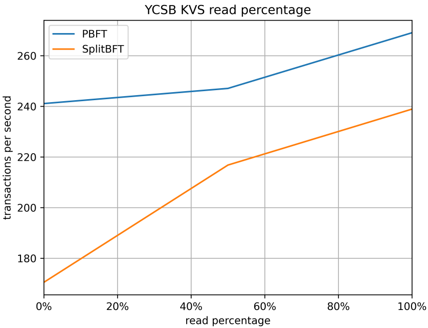
- Data store benchmark
  - Realistic access patters (SoCC '10)
- SplitBFT vs. PBFT in Themis
- Application is in-memory KVS

Workload

1. Load phase
   - Insert data
2. Run phase
   - Reads & Updates

YCSB-KVS Evaluation





Conclusion

- Design and implementation of SplitBFT in Themis
- SGX broker and enclave maintainable and exchangeable
- Distributed evaluations in hardware mode against PBFT

Future Work

- More applications in SplitBFT
- Further optimizations and fast-tracks