Pusan National University Compute
r Science and Engineering Technical
Report 2025-09

신뢰 실행 환경을 활용한 스마트 컨트랙트 실행 보호 기술 개발

202155651 이준태

202055581 이준혁

202155648 위재준

지도교수 권동현

목 차

1. 서론

1.1. 연구 배경

블록체인 기술은 탈중앙화, 데이터 무결성, 투명성의 특성으로 인해 금융, 물류, 의료, 공공 행정 등 다양한 산업에서 폭넓게 활용되고 있다. 블록체인의 핵심 가치는 참여자 간의 신뢰할 수 있는 데이터 공유를 가능하게 하는 데 있으며, 이를 위해 합의 알고리즘과 스마트 컨트랙트가 중요한 역할을 한다. 특히 Hyperledger Fabric은 모듈화된 구조와 높은 확장성, 그리고 퍼미션드(permissioned) 네트워크를 지원하는 특성으로 인해 기업 환경에서 많이 채택되고 있다. Fabric은 사용자 정의 로직을 체인코드(chaincode)의 형태로 실행하여 애플리케이션 요구사항을 블록체인에 반영한다.

그러나 Fabric 체인코드는 일반적으로 운영체제의 사용자 영역(REE, Rich Execution Environment) 에서 실행된다. 이는 잠재적으로 신뢰할 수 없는 환경이며, 만약 공격자가 운영체제나 하이퍼바이저를 장악한다면 체인코드의 실행 과정과 결과는 쉽게 위조될 수 있다. 예를 들어, 체인코드의 상태 데이터가 변조되거나, 실행 결과가 공격자에 의해 조작된다면, 이는 블록체인의 근본적인 무결성과 신뢰성을 심각하게 훼손한다. 따라서 체인코드 실행의 보안성을 보장할 수 있는 새로운 실행 환경이 필요하다.

이러한 배경에서 신뢰 실행 환경(TEE, Trusted Execution Environment) 의 도입이 하나의 대안으로 제시되고 있다. TEE는 메인 프로세서 내의 격리된 보안 영역을 활용하여 코드와 데이터를 외부 공격으로부터 보호한다. Intel SGX와 ARM TrustZone은 대표적인 TEE 기술로, 각각 서버급 환경과 모바일·임베디드 환경에서 사용되고 있다. 특히 ARM TrustZone은 IoT 및 엣지 컴퓨팅 환경에서 폭넓게 지원되기 때문에, 자원 제약이 있는 환경에서도 체인코드 실행을 보호할 수 있는 가능성을 제시한다.

1.2. 기존 문제점

체인코드 실행 보호와 관련하여 대표적인 연구가 존재한다.

첫째, TZ4Fabric은 Hyperledger Fabric과 ARM TrustZone을 연동하여 체인코드를 보안 영역(Secure World)에서 실행하는 구조를 제안하였다. Peer 전체를 보안 영역에 포함시키지 않고, 체인코드 실행 부분만 Secure World에 오프로드함으로써 Trusted Computing Base(TCB) 를 최소화했다는 점에서 큰 의의가 있었다. 그러나 체인코드를 실행하기 위해서는 OP-TEE Trusted Application(TA) 형태로 직접 작성하고 빌드해야 했다. 즉, 체인코드가 원래 주로 사용되는

Go나 Java와 같은 고수준 언어로 작성된 경우에도 이를 그대로 활용할 수 없고, OP-TEE API에 맞게 포팅해야 하는 불편함이 있었다. 이는 결과적으로 언어 이식성과 확장성 부족이라는 문제로 이어졌다. 또한 world switching, 세션 생성과 같은 TEE 특유의 실행 과정에서 큰 오버헤드가 발생하여 성능 저하가 불가피했다.

둘째, WaTZ는 ARM TrustZone 내부에 WebAssembly(Wasm) 런타임을 구현함으로써 다양한 언어로 작성된 프로그램을 Wasm으로 변환해 실행할 수 있도록 하였다. 이를 통해 언어 이식성과 실행 격리를 동시에 달성했으며, SQLite, Genann과 같은 워크로드를 실행해 실제 환경에서의 성능과 보안성을 평가하였다. Wasm의 샌드박스 특성 덕분에 프로그램 간 격리 실행이 가능하고, AOT(Ahead-of-Time) 방식을 활용하여 비교적 낮은 오버헤드로 실행할 수 있었다. 그러나 WaTZ는 블록체인과의 연동 구조를 포함하지 않아, Hyperledger Fabric 체인코드 실행 환경으로 직접 활용하기에는 한계가 있었다.

결국, TZ4Fabric은 블록체인 연동은 가능했지만 언어 이식성이 부족했고, WaTZ는 언어 이식성은 확보했지만 블록체인 연동이 불가능했다. 두 연구는 상호 보완적인 장점을 가지고 있었지만, 각각 독립적으로는 체인코드 실행 보호를 위한 완전한 해결책이 되지는 못했다.

1.3. 연구 목표

본 연구는 이러한 기존 연구의 한계를 극복하기 위해, WaTZ와 TZ4Fabric의 장점을 통합하여 ARM TrustZone 기반 환경에서 체인코드를 안전하게 실행하고 Hyperledger Fabric과 연동하는 새로운 아키텍처를 제안한다.

1. 체인코드를 C, Go, Rust 등 다양한 언어로 작성할 수 있으며, 이를 WebAssembly (AOT) 모듈로 변환하여 Secure World에서 실행할 수 있는 실행 환경을 구축한다.
2. TZ4Fabric에서 제안된 Wrapper-Proxy 구조를 활용하여, 체인코드 실행 요청이 Proxy를 통해 Secure World로 전달되고, 실행 결과가 다시 Hyperledger Fabric 원장(Ledger)에 기록되도록 한다.
3. 실제 ARM 보드(i.MX8M-EVK)에서 체인코드 예제를 비롯한 다양한 벤치마크(SQLite, Genann)를 실행함으로써, 제안된 구조의 동작 가능성을 검증하고 성능 오버헤드를 평가한다

본 연구의 최종 목표는 체인코드 실행 환경의 보안성 강화, 언어 이식성 확보, 블록체인 연동

가능성을 동시에 달성하는 것이다. 이를 통해 기존 체인코드 실행의 취약점을 보완하고, 나아가 IoT 및 엣지 컴퓨팅 환경에서도 적용 가능한 확장성 있는 블록체인 보안 모델을 제시하고자 한다.

2. 연구 배경

2.1. 배경 지식

본 연구는 블록체인 체인코드 실행을 보호하기 위해 다양한 보안 및 실행 환경 관련 기술들을 활용하였다. 연구 내용을 이해하기 위해서는 먼저 Hyperledger Fabric의 동작 방식과 체인코드 실행 구조, 그리고 이를 보호하기 위해 사용된 ARM TrustZone 기반 실행 환경과 WebAssembly 기술에 대한 이해가 필요하다. 또한 체인코드 실행을 TEE와 연동하기 위해 OP-TEE, GlobalPlatform API, gRPC와 같은 핵심 소프트웨어 구성 요소가 사용되었으며, 실험 환경으로는 Docker 기반 Fabric 네트워크와 ARM i.MX8M-EVK 보드가 활용되었다. 본 절에서는 이러한 기술적 배경을 정리하고, 특히 TZ4Fabric과 WaTZ 연구를 중심으로 본 연구와의 연관성을 설명하고자 한다.

- **TZ4Fabric**
  - TZ4Fabric은 Hyperledger Fabric의 체인코드를 ARM TrustZone 기반 보안 영역에서 실행하기 위한 아키텍처이다. Peer 전체가 아닌 체인코드 실행 부분만 Secure World에 배치하여 Trusted Computing Base(TCB)를 줄이는 데 의의를 두었다. 이를 통해 운영체제 수준 공격으로부터 체인코드 실행을 보호할 수 있으나, 체인코드를 OP-TEE Trusted Application(TA) 형태로 빌드해야 하는 제약이 있어 언어 이식성이 부족하다는 한계가 있다.
- **WaTZ**
  - WaTZ는 ARM TrustZone 내부에 WebAssembly(WebAssembly Micro Runtime, WAMR 기반) 실행 환경을 구현한 아키텍처이다. 다양한 언어로 작성된 프로그램을 Wasm으로 변환해 실행할 수 있으며, 샌드박스 기반 격리를 통해 보안성을 확보할 수 있다. SQLite, Genann 등의 워크로드로 실험해 성능과 안정성을 입증한 바 있으며, 본 연구에서는 WaTZ의 실행 환경을 Fabric 체인코드 실행에 응용한다.
- **OP-TEE**
  - OP-TEE는 ARM TrustZone을 위한 오픈소스 TEE 구현체로, Linux Foundation의 Linaro 프로젝트에서 개발되었다. GlobalPlatform API를 지원하며, Normal World와 Secure World 간의 안전한 통신, 메모리 관리, Trusted Application 실행을 가능하게 한다. 본 연구에서는 체인코드 실행을 OP-TEE 상의 Secure World에서 수행한다.
- **WebAssembly(Wasm)**
  - WebAssembly는 웹과 독립적인 범용 바이트코드 포맷으로, 다양한 언어(C, Go, Rust 등)를 컴파일해 실행할 수 있다. Wasm은 샌드박스 환경에서 동작하기 때문에 보안성이 높으며, 플랫폼 간 이식성이 뛰어나다. 본 연구에서는 체인코드를 Wasm으로 변환하여 TrustZone 내부에서 실행함으로써 언어 이식성을 확보한다.

- **WARM (WebAssembly Micro Runtime)**
  - WAMR은 리소스 제약 환경에서 동작하도록 설계된 경량 WebAssembly 실행 엔진이다. 작은 메모리 사용량과 빠른 초기화 속도를 특징으로 하며, AOT와 WASI를 지원해 성능과 이식성을 동시에 제공한다. 샌드박스 기반으로 안전한 실행 환경을 보장하며, C·Go·Rust 등 다양한 언어로 작성된 프로그램을 Wasm 모듈 형태로 실행할 수 있다. 본 연구에서는 WaTZ 런타임의 기반으로 활용되어, 체인코드를 TrustZone 내부에서 안전하게 실행하는 핵심 요소가 된다.

- **WASI(WebAssembly System Interface)**
  - WASI는 Wasm 모듈이 파일 시스템, 네트워크와 같은 운영체제 기능에 접근할 수 있도록 정의된 시스템 인터페이스이다. 보안성과 이식성을 동시에 확보할 수 있으며, 다양한 플랫폼에서 일관된 실행 환경을 제공한다. 본 연구의 WaTZ 실행 환경은 WASI를 기반으로 Wasm 체인코드를 실행한다.

- **Docker**
  - Docker는 컨테이너 기반 가상화 플랫폼으로, 경량화된 실행 환경을 제공한다. Hyperledger Fabric은 Peer, Orderer, Certificate Authority(CA) 등 구성 요소를 컨테이너로 실행할 수 있도록 지원한다. 본 연구에서는 Docker를 이용해 Fabric 네트워크를 구성하고, 체인코드 실행 환경을 관리한다.

- **Hyperledger Fabric**
  - Fabric은 Linux Foundation에서 개발한 오픈소스 프라이빗 블록체인 플랫폼이다. 모듈화된 구조와 퍼미션드 네트워크를 지원하며, 스마트 컨트랙트에 해당하는 체인코드가 트랜잭션 로직을 수행한다. 본 연구는 체인코드를 Peer에서 직접 실행하는 대신, TrustZone 내부의 Wasm 런타임에서 실행하여 보안성을 강화한 뒤 결과를 Fabric 원장(Ledger)에 기록하는 방식을 제안한다.

- **gRPC**
  - gRPC는 구글이 개발한 고성능 원격 프로시저 호출(Remote Procedure Call) 프레임워크로, HTTP/2와 프로토콜 버퍼를 기반으로 한다. 언어와 플랫폼 독립적으로 동작하며 양방향 스트리밍을 지원한다. 본 연구에서는 Wrapper와 Proxy 간 통신, 그리고 Proxy와 Fabric 간 상호작용에 gRPC를 활용한다.

- **Protocol Buffers (Protobuf)**
  - Protobuf는 구글이 개발한 직렬화(Serialization) 포맷으로, 구조화된 데이터를 효율적으로 인코딩·디코딩할 수 있도록 설계되었다. JSON이나 XML에 비해 크기가 작고 처리 속도가 빠르며, 언어와 플랫폼에 독립적으로 사용할 수 있다. 본 연구에서는 gRPC 통신 시 메시지 정의를 위해 Protobuf를 활용하여, Wrapper와 Proxy 간 체인코드 실행 요청과 응답을 효율적으로 교환한다.

- **Sandbox**

- 샌드박스는 실행 환경을 격리하여 프로그램이 외부 시스템 자원에 직접 접근하지 못하도록 하는 기술이다. 이를 통해 악성 코드 실행이나 외부 침입을 방지할 수 있다. Wasm은 본질적으로 샌드박스 기반으로 실행되며, 본 연구에서는 체인코드 실행의 무결성과 안정성을 보장하는 핵심 요소로 활용된다.

- **Ahead-of-Time (AOT) Compilation**
  - AOT 컴파일은 바이트코드(예: Wasm)를 사전에 네이티브 코드로 변환해 실행 성능을 향상시키는 방식이다. TrustZone과 같은 제한된 환경에서는 JIT(Just-in-Time)보다 AOT가 적합하다. WaTZ는 AOT 방식을 통해 실행 오버헤드를 줄였으며, 본 연구 역시 체인코드를 AOT 형태로 실행하여 성능과 안정성을 확보한다.

- **i.MX8M-EVK 보드**
  - i.MX8M-EVK는 NXP에서 개발한 ARM Cortex-A53 기반 개발 보드로, ARM TrustZone을 지원한다. Secure World와 Normal World를 분리하여 TEE 기능을 실험할 수 있는 하드웨어 환경을 제공한다. 본 연구에서는 i.MX8M-EVK 보드 위에 OP-TEE와 WaTZ 런타임을 배치하고, 체인코드 실행 실험을 수행하였다.

이와 같이 본 연구에서 다룬 위 지식들은 체인코드 실행 보호 아키텍처를 설계하고 구현하는 데 핵심적인 역할을 한다. 각각의 기술은 보안성, 이식성, 확장성, 실험 환경 구축 등 서로 다른 측면에서 기여하며, 이들이 통합됨으로써 신뢰할 수 있는 체인코드 실행 환경을 구성할 수 있다.

2.2. 관련 연구 동향

블록체인 환경에서 스마트 컨트랙트 실행의 보안성과 기밀성을 강화하기 위한 연구는 TEE(Trusted Execution Environment)의 활용을 중심으로 다양하게 진행되어 왔다.

먼저 Ekiden(2018)은 Intel SGX를 기반으로 스마트 컨트랙트를 오프체인에서 실행하고 결과만 블록체인에 기록하는 아키텍처를 제안하였다. 이를 통해 입력 데이터와 컨트랙트 상태를 enclave 내부에서만 처리하여 기밀성을 보장하고, 원격 증명을 통해 실행 무결성을 확보하였다. 또한 블록체인은 합의와 결과 저장에만 관여하도록 설계되어, 성능 확장성 측면에서도 기존 방식보다 우수함을 보였다.

이후에는 ARM TrustZone 환경을 활용한 연구들이 등장하였다. TZ4Fabric은 Hyperledger Fabric과 TrustZone을 연동하여 체인코드를 TEE 내부에서 실행하는 구조를 제안하였다. 이를 통해 블록체인 체인코드 실행 경로에 하드웨어 보안성을 부여할 수 있음을 보였으나, 체인코드가 네이티브 코드(C)로 작성되어야 한다는 점에서 언어 이식성과 확장성의 한계가 존재하였다.

한편, WaTZ(2022)는 ARM TrustZone 상에서 WebAssembly(Wasm) 모듈을 실행할 수 있는 경량 런타임을 제안하였다. Wasm을 TEE 내부에서 실행함으로써 다양한 언어로 작성된 코드를 안전하게 포팅할 수 있고, 샌드박스 기반 격리와 하드웨어 보안성을 동시에 확보할 수 있음을 보여주었다. 성능 평가 결과, 네이티브 실행 대비 1.6~2.4배의 오버헤드를 보였으나 실용적인 수준임을 입증하였다.

정리하면, Ekiden은 SGX 기반에서 기밀성과 성능을 동시에 달성할 수 있는 아키텍처를 제안한 연구이며, TZ4Fabric과 WaTZ는 ARM TrustZone 환경을 기반으로 각각 블록체인 연동과 Wasm 실행이라는 다른 접근을 통해 유사한 문제를 탐구하였다. 이러한 연구들은 블록체인 스마트 컨트랙트 실행에 있어 보안성과 이식성을 강화하려는 다양한 시도를 보여주며, 본 연구의 배경을 형성한다.

3. 연구 내용

3.1. WaTZ 환경 구성 및 성능 벤치 마크 실험

본 연구에서는 제안 아키텍처의 기반이 되는 WaTZ 환경을 실제 하드웨어에서 구축하고, 이를 통해 성능 특성을 검증하고자 한다. WaTZ는 ARM TrustZone 내부에 WebAssembly 런타임을 배치하여 애플리케이션을 안전하게 실행할 수 있도록 하는 핵심 요소이므로, 올바른 환경 구성과 성능 평가가 연구의 출발점이 된다. 이에 따라 먼저 i.MX8M-EVK 보드 위에 OP-TEE와 WaTZ 런타임을 설치하여 실행 환경을 마련하고(3.1.1), 이어서 SQLite 및 Genann과 같은 워크로드를 대상으로 성능 벤치마크 실험을 수행하여 WaTZ의 실행 효율성을 평가한다(3.1.2).

3.1.1. WaTZ 환경 구성

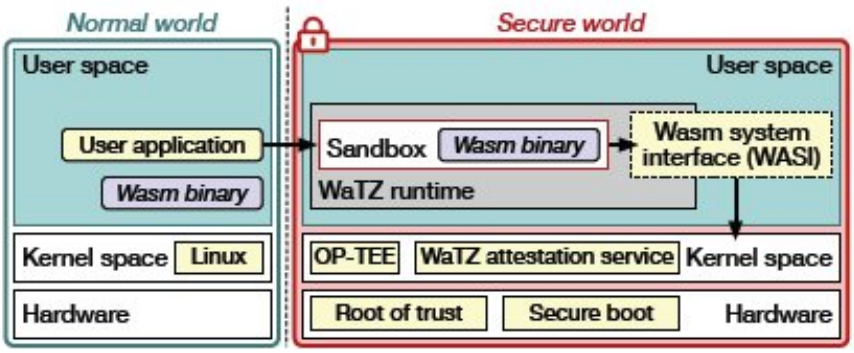

그림 1. WaTZ 구조

WaTZ 실행을 위한 환경으로는, WaTZ GitHub에서 제공되는 OP-TEE OS와 buildroot를 활용하여 운영체제를 빌드하였다. 해당 리눅스 OS는 i.MX8M-EVK 보드 내에서 실행되고, 해당 OS는 WaTZ의 런타임과 TEE 드라이버, WaTZ를 사용하여 AOT 파일을 실행할 수 있는 attester와 verifier도 포함되어 있다. WaTZ 벤치마크 실행을 위해서 Ubuntu 20.04 버전의 인스턴스에서 WASM 바이너리를 AOT 파일로 빌드하고, 해당 파일을 i.MX8M-EVK 보드 내로 전송하여 실행할 수 있도록 우분투 인스턴스와 보드는 같은 내부망에 위치하도록 두었다.

3.1.2. WaTZ 성능 벤치 마크 실험

WaTZ 환경이 올바르게 구축되었음을 확인한 이후, 본 연구에서는 성능 특성을 검증하기 위한 일련의 벤치마크 실험을 수행하였다. 해당 실험은 WaTZ 논문에서 제시된 평가 방법을 참고하여 동일한 환경과 워크로드를 재현하는 것을 목표로 하였으며, i.MX8M-EVK 보드 기반 OP-TEE 환경에서 측정 데이터를 수집하였다. 특히 Time retrieval 및 TrustZone 전환 지연, Wasm 프로그램의 Startup 단계별 오버헤드, PolyBench/C 및 SQLite와 같은 마이크로·매크로 벤치마크, 암복호화 연산, 그리고 Genann 신경망 학습 시나리오까지 포함하여 다양한 관점에서 성능을 측정하였다. 이를 통해 WaTZ 실행 환경이 기존 보고된 수치와 일치하는지 확인하고, TEE 환경에서의 오버헤드가 어느 정도 수준인지를 정량적으로 분석하였다.

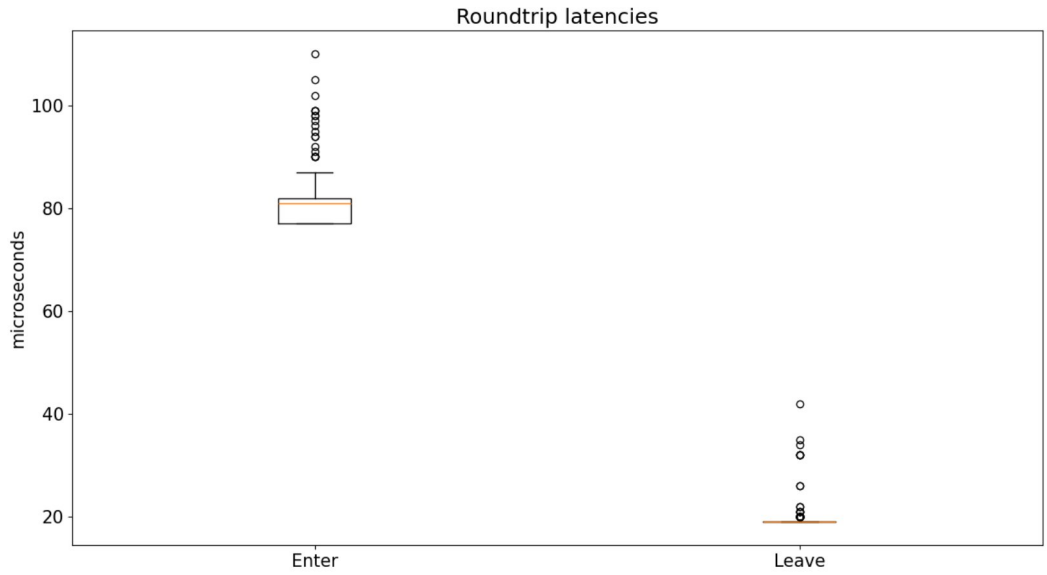

그림 2. Time retrieval 및 TrustZone 전환 지연 측정 결과

TEE 진입/이탈 시의 roundtrip latency를 boxplot으로 시각화하였다. 실험 결과, Enter 평균은 약 80µs, Leave 평균은 약 18µs 수준으로 나타났으며, 이는 논문 보고치인 86µs / 20µs와 근접한 수치이다. 측정값 분산은 enter 동작에서 상대적으로 컸으며, outlier가 다수 발생하였다. 이는 보드 환경의 I/O 변동성과 관련 있을 수 있다.

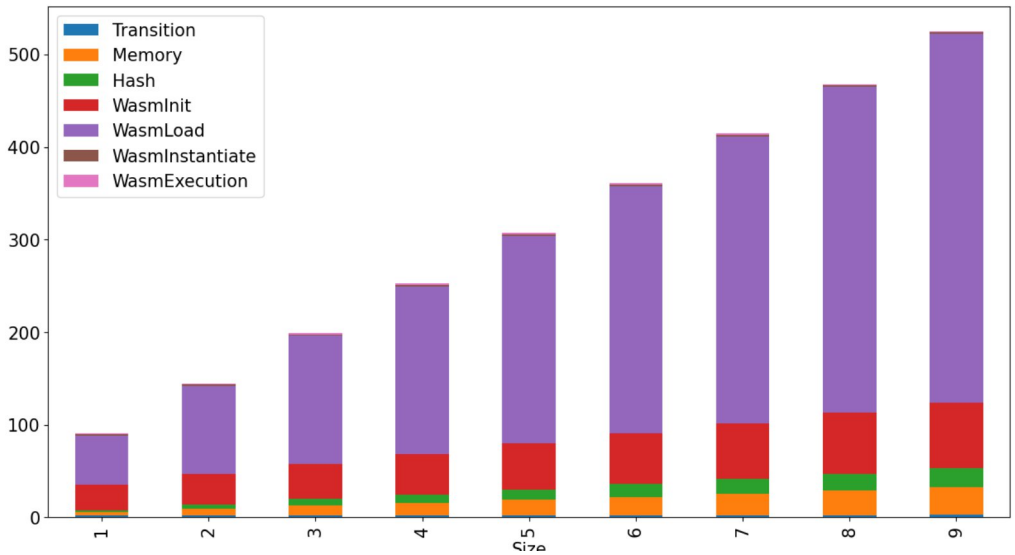

그림 3. Wasm 프로그램 Startup 단계별 시간 분석

애플리케이션 크기(1MB - 9MB)에 따른 startup 단계별 시간(Transition, Memory, Hash, Init, Load, Instantiate, Execution)을 스택 막대그래프로 표현하였다. 실험 결과, WasmLoad가 전체 시간의 대부분을 차지하며, 파일 크기에 정비례 증가하였다 (예: 1MB에서 약 55ms, 9MB에서는 약 400ms 이상). 해싱, 인스턴스 생성 등의 오버헤드는 상대적으로 작았으며, 전체 startup 지연은 최대 약 530ms 수준으로 확인되었다.

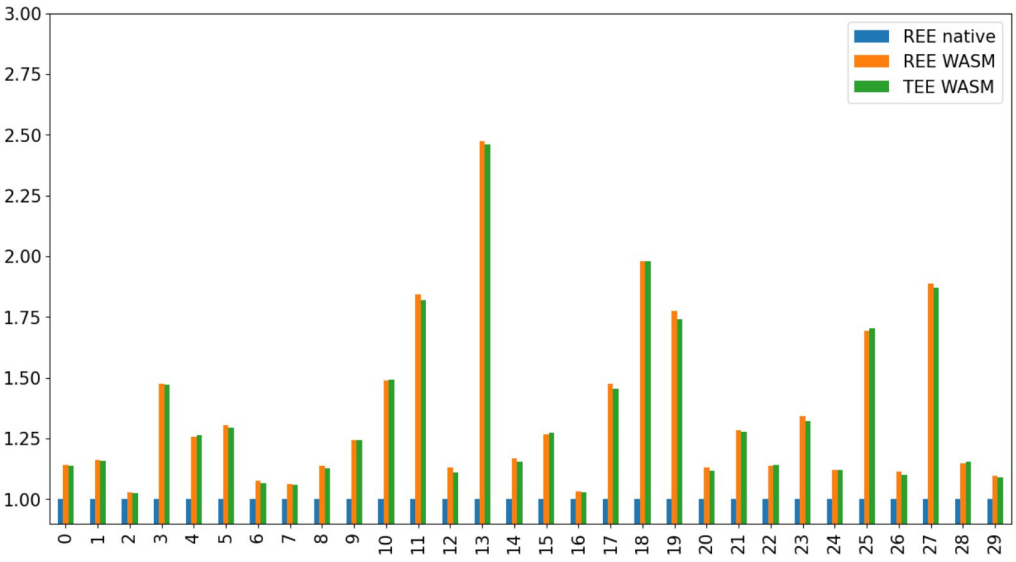

그림 4. PolyBench/C 워크로드 실행 성능 비교

30개 워크로드에 대해 REE Native (=1) 기준으로 REE WASM, TEE WASM 성능을 정규화하여 비교하였다. 실험 결과, 대부분의 워크로드에서 TEE WASM은 평균 약 1.2~1.8배 정도의

실행 시간 오버헤드를 보였으며, 가장 큰 경우는 약 2.47배 (index 13)로 나타났다. 이는 전반적으로 논문 수치(1.3~2.4배)와 유사하며, 수치 계산 중심 워크로드에 대해 WATZ의 AOT 실행 성능이 실용적임을 보여준다.

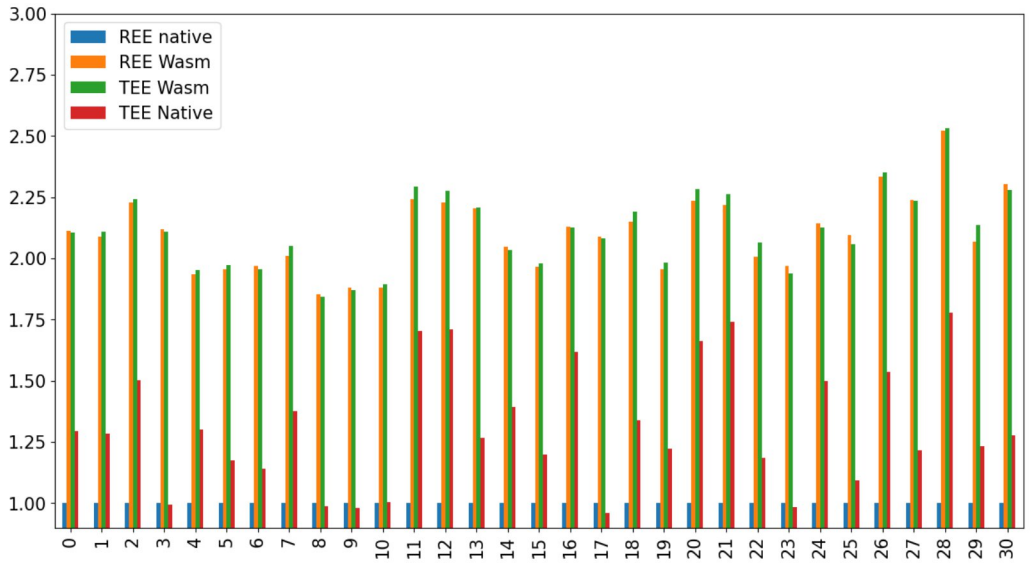

그림 5. SQLite Speedtest1 성능 비교

30개 이상 쿼리에 대해 REE Native 기준으로 REE Wasm, TEE Wasm, TEE Native까지 비교한 정규화된 실행 시간을 막대그래프로 시각화하였다. 실험 결과, TEE WASM은 평균 약 2.1~2.4배의 성능 저하를 보였으며, 특정 쿼리에서는 최대 2.6배 수준까지 관찰되었다. 이는 WASI 호출, secure memory 접근 및 RA hash 계산에 따른 비용으로 판단된다.

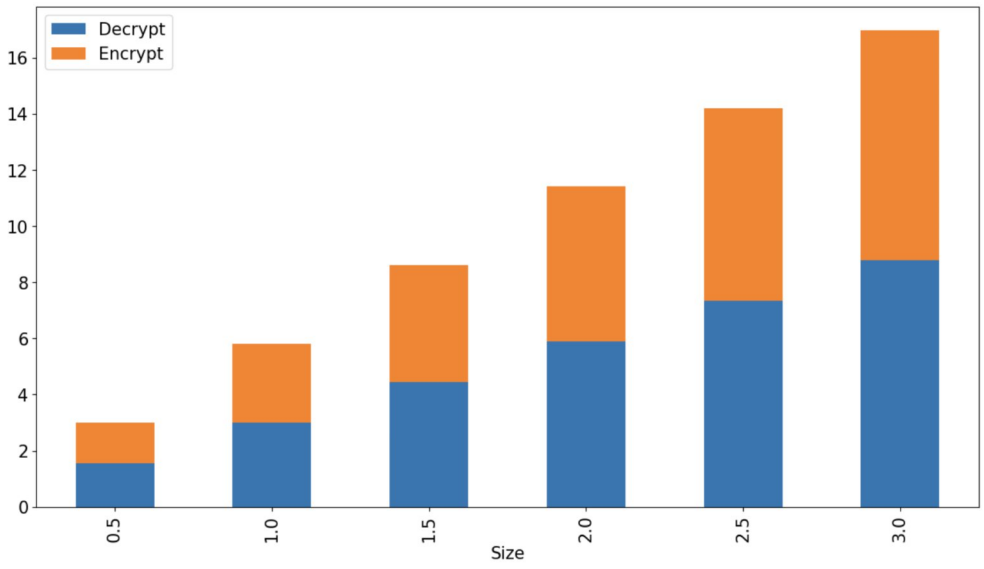

그림 6. msg3 단계에서 AES-GCM 암복호화 시간

RA 프로토콜의 마지막 단계에서 전송되는 기밀 데이터(Secret Blob)에 대한 AES-GCM 기반 암복호화 성능을 측정하였다. 데이터 크기는 0.5MB에서 3MB까지 증가시키며, 암호화(Encrypt)와 복호화(Decrypt) 각각의 실행 시간을 비교하였다. 실험 결과, 암복호화 시간은 데이터 크기에 비례하여 선형적으로 증가하며, 3MB 데이터 처리 시 총 17ms 내외로 나타났다. 이는 논문에서 보고된 결과와 거의 동일한 값이다. 따라서, 기밀 데이터 전송 시 암복호화 연산은 전체 RA 비용에서 차지하는 비중이 매우 작아, 실용적 성능을 보장함을 확인할 수 있다.

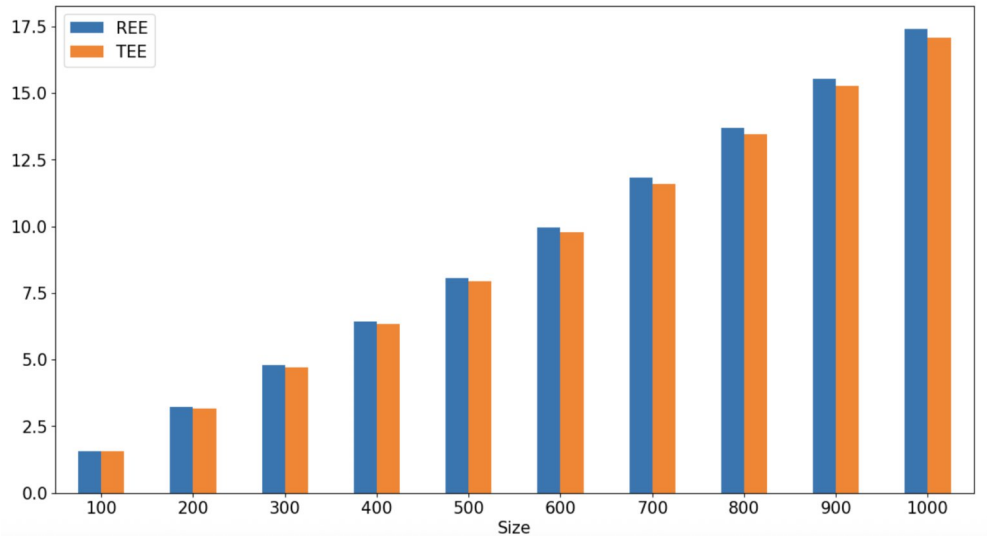

그림 7. Genann 학습 워크로드 성능 비교 (WAMR vs WaTZ)

경량 신경망 라이브러리인 Genann을 사용하여, RA 이후 데이터셋을 안전 채널을 통해 수신하고 학습을 수행하는 end-to-end 시나리오를 평가하였다. 실험 구성은 논문과 동일하게 Iris 데이터셋을 기반으로, 입력크기를 100KB에서 1MB까지 증가시키며 학습시간을 측정하였다. 결과는WAMR(REE)와 WaTZ(TEE) 간의 성능 차이가 거의 없으며, 오히려 WaTZ에서 약간의 성능 향상(평균 1.4%)이 관찰되었다. 이는 논문에서 보고된 결과와 동일한 경향을 보이며, RA 초기 단계(msg0, msg1, msg2)에서 발생하는 오버헤드는 학습 단계에 미미한 영향을 주는 것으로 확인되었다. 따라서, TEE 환경에서도 머신러닝 워크로드 실행이 실용적임을 확인할 수 있다.

실험 결과, WaTZ 환경에서의 성능 특성은 기존 논문에서 보고된 수치와 대체로 일치하였으며, TEE 도입에 따른 오버헤드는 평균 1.6~2.1배 수준으로 확인되었다. 이는 보안성을 확보하기 위해 요구되는 성능 비용으로 충분히 수용 가능한 범위임을 보여준다. 또한 SQLite, Genann과 같은 매크로 벤치마크를 통해 실제 애플리케이션 수준에서도 WaTZ 실행 환경이 실용적인 성능을 유지함을 검증하였다.

3.2. WaTZ-TZ4Fabric 연동 구조

앞서 WaTZ 환경 구성 및 성능 검증을 통해 언어 이식성과 실행 효율성을 확인하였고, TZ4Fabric을 통해 Hyperledger Fabric과 TrustZone 연동 구조를 살펴보았다. 본 절에서는 이러한 두 접근법을 통합하여, 체인코드를 Wasm 모듈 형태로 TrustZone 내에서 실행하면서도 Fabric 원장과 연동할 수 있는 실행 구조를 제안한다. 이를 위해 Wrapper, Proxy, Wrapper TA, WaTZ 런타임 간의 상호작용 과정을 정의하고, 체인코드 실행 요청이 전달·처리·기록되는 전체 경로를 설명한다. 이러한 연동 구조는 기존 연구의 한계를 극복하고, 체인코드 실행의 보안성·이식성·연동성을 동시에 달성하는 기반이 된다.

3.2.1. 기존 TZ4Fabric 구조 요약

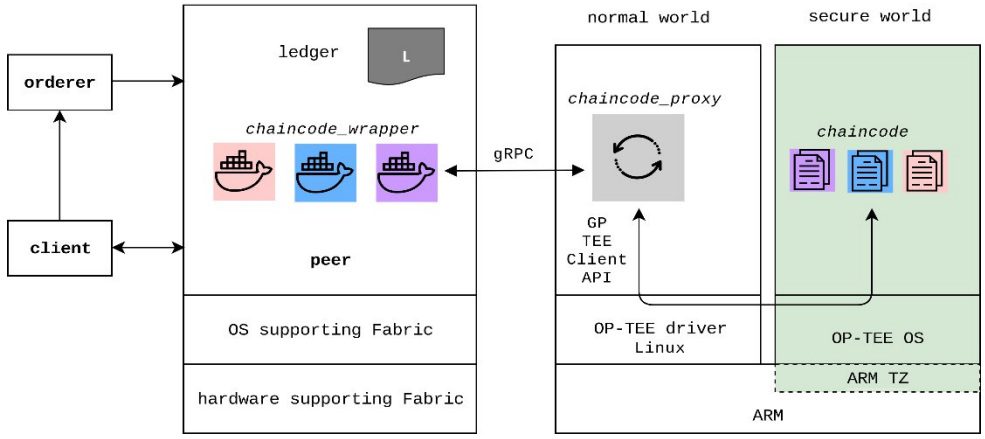

그림 8. 기존 TZ4Fabric 구조

그림 7은 TZ4Fabric에서 제안된 기본 아키텍처를 나타낸다. Hyperledger Fabric의 클라이언트(Client)가 트랜잭션 요청을 생성하면, 이를 오더러(Orderer)를 거쳐 피어(Peer) 노드로 전달한다. 피어 내부에서는 체인코드 래퍼(Chaincode Wrapper)가 요청을 받아 체인코드 실행을 담당하는 프록시(Chaincode Proxy)로 gRPC를 통해 전달한다. 프록시는 Normal World에서 동작하며, OP-TEE 클라이언트 API를 호출하여 Secure World 내부의 체인코드(Trusted Application)를 실행한다.

이 구조의 핵심은 피어 전체를 Secure World에 포함시키지 않고, 체인코드 실행 부분만 TEE 내부로 오프로드했다는 점이다. 이를 통해 Trusted Computing Base(TCB)를 줄이고 운영체제 수준 공격으로부터 체인코드 실행을 보호할 수 있다. 그러나 체인코드를 OP-TEE Trusted Application(TA)로 작성·빌드해야 하므로 언어 이식성이 부족하고, world switching 과정에서 성능 저하가 발생하는 한계가 존재한다.

3.2.2. 통합 구조 개요

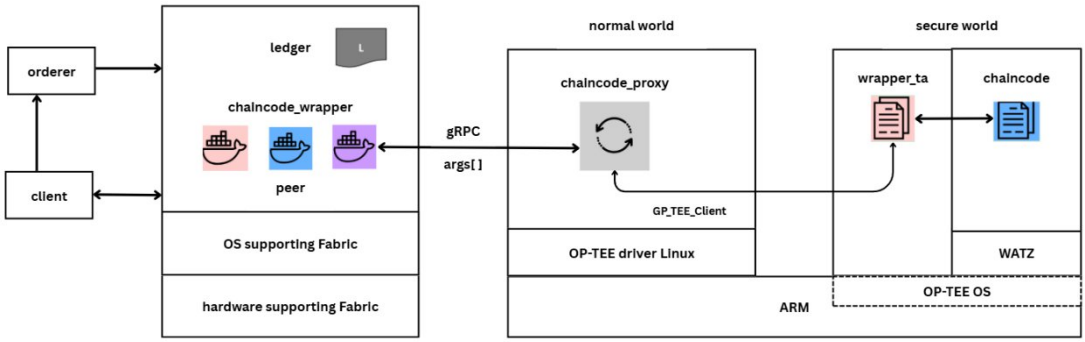

그림 9. WaTZ-TZ4Fabric 통합 구조

그림 8은 본 연구에서 제안하는 WaTZ-TZ4Fabric 통합 구조를 나타낸다. 기존 TZ4Fabric 구조가 Wrapper → Proxy → TA 형태였다면, 통합된 구조에서는 Proxy가 체인코드 실행 요청과 함께 Wasm AOT 모듈을 전달하고, 이를 Secure World 내부의 Wrapper_TA에서 처리하도록 확장하였다. Wrapper_TA는 WaTZ 런타임과 연동하여 체인코드를 Wasm 형태로 실행하며, 실행 중 발생하는 get_state, put_state와 같은 원장 접근 요청은 Native Function을 통해 Proxy로 전달된다. Proxy는 gRPC를 이용해 Hyperledger Fabric 원장에 접근한 뒤 결과를 다시 Secure World로 반환하고, 최종적으로 Wrapper를 통해 Fabric Ledger에 기록된다.

이 구조의 특징은 기존 Fabric 연동 방식을 그대로 유지하면서도, 체인코드를 Wasm 모듈로 실행할 수 있도록 하여 언어 이식성을 확보하였다는 점이다. 또한 Wrapper_TA와 WaTZ 런타임을 도입함으로써 Secure World 내부에서의 실행 환경을 확장하고, 다양한 언어 기반 체인코드를 동일한 방식으로 보호 실행할 수 있는 기반을 마련하였다.

3.2.3. Proxy 수정 사항

1) **설계 전환 개요**

기존 TZ4Fabric에서는 Proxy가 체인코드별로 OP-TEE TA를 직접 호출하는 구조(Wrapper → Proxy → TA)를 사용하였다. 즉, 체인코드마다 별도의 TA가 필요하며, Proxy는 해당 TA의 UUID를 기반으로 세션을 열고, TA_CHAINCODE_CMD_INIT_INVOKE 및 RESUME_INVOKE 명령을 통해 함수 호출과 재개를 수행하였다.

반면, 본 연구에서 제안하는 WaTZ-TZ4Fabric 통합 구조에서는 단일 Wrapper_TA 내부의 WaTZ 런타임이 Wasm(AOT) 모듈을 로드하여 실행한다(Wrapper → Proxy → Wrapper_TA → WaTZ). 이에 따라 Proxy의 역할은 기존의 TA 호출기 에서 Wasm 모듈 전달·실행 오케스트레이터로 확장되었다.

2) **인터페이스 및 gRPC 메시지 변화**

gRPC 요청 확장: 기존에는 chaincode_uuid, function_name, arguments[]만 포함되었으나, 현재 구조에서는 aot_file 필드가 추가되어 Proxy가 해당 파일(./chaincode/<aot_file>)을 읽어 바이트코드 형태로 전달한다.

GlobalPlatform API 파라미터 재구성: 기존에는 공유 메모리(MEMREF_WHOLE)를 통해 함수명, 인자, 키/값, 응답을 단일 버퍼로 관리하였다. 반면, 현재 구조는 MEMREF_TEMP_INPUT(바이트코드), MEMREF_TEMP_INOUT(arguments/shared_buf)
, MEMREF_TEMP_INOUT(output_buffer)로 명확히 분리하여 전달한다. 함수명은 arguments[0]에 포함시켜 관리하며 ABI 단순성이 향상되었다.

3) **세션 생명 주기 및 런타임 설정**

기존에는 요청마다 InitializeContext → OpenSession(uuid=체인코드TA) → Invoke → CloseSession → FinalizeContext 과정을 반복하였다. 반면 통합 구조에서는 Proxy 프로세스 수준에서 Wrapper_TA 세션을 상시 유지하며, 요청 종료 시 세션을 재시작하여 메모리를 초기화한다.

초기화 단계: allocate_buffers()로 stdout/benchmark 버퍼를 준비하고, prepare_tee_session()으로 Wrapper_TA 세션을 오픈한다. 이후 configure_heap_size(10MB)를 호출하여 WaTZ 힙 크기를 설정한다.

종료 단계: restart_session()을 통해 세션을 종료·재초기화하여 상태 잔류를 제거한다.

4) **실행 흐름 비교**

공통점: gRPC 스트리밍 기반 양방향 통신 구조는 동일하다. 체인코드에서 get_state/put_state 요청이 발생하면 Proxy가 이를 Wrapper로 위임하여 원장을 접근하고, 응답을 다시 Secure World로 전달한다. 결과는 op.params[1].value.a의 enum 값(INVOCATION_RESPONSE, GET_STATE_REQUEST, PUT_STATE_REQUEST, ERROR)을 통해 구분된다.

차이점: 기존 구조에서는 함수명을 전달하고 공유 메모리를 통해 실행 상태를 관리하였으나, 통합 구조에서는 AOT 바이트코드와 arguments를 명시적으로 전달하며, 별도의 output_buffer를 통해 Wasm stdout을 캡처할 수 있다. 실행 명령 역시

TA_CHAINCODE_CMD_INIT_INVOKE/RESUME_INVOKE에서 COMMAND_RUN_WASM/COMMAND_RESUME_WASM으로 대체되었다.

3.2.4. 체인코드 변환 및 Native Function 사용

1) **변환 목표와 핵심 아이디어**

기존 TZ4Fabric 체인코드는 OP-TEE TA(C 언어)로 작성되어 write_get_state, read_get_state, write_put_state, read_put_state, write_response와 같은 내부 함수를 통해 원장 접근 및 응답 반환을 수행하였다. 본 연구에서는 동일한 체인코드 로직을 WebAssembly(AOT) 기반으로 이식하고, 원장 접근과 응답 반환을 네이티브 임포트(hostcall) 방식으로 Wrapper_TA-Proxy-Wrapper-Ledger 경로를 거쳐 처리하도록 설계하였다. 이를 통해 체인코드 실행은 언어 이식성(Wasm 타깃)과 실행 격리(Wasm 샌드박스 + TrustZone TEE)의 장점을 동시에 확보할 수 있다.

2) **네이티브 임포트 인터페이스 (env 모듈)**

이식된 Wasm 체인코드는 표준 라이브러리를 배제(-nostdlib)하고, 다음과 같은 env 모듈의 네이티브 임포트를 통해 모든 I/O와 런타임 상호작용을 수행한다.

메타데이터 및 인자 취득:

- cc_get_function(char *out, int out_len): 호출된 함수명(예: "create", "add", "query") 획득
- cc_get_arg(int idx, char *out, int out_len): 지정 인덱스의 인자 값 획득

원장 접근:

- cc_get_state(const char *key, int key_len, char *out, int out_len): 키 조회 (값은 재개 단계에서 out에 반영)
- cc_put_state(const char *key, int key_len, const char *val, int val_len): 키/값 저장 (ACK는 재개 단계에서 확인)

응답 및 로그 처리:

- cc_return_response(const char *msg, int msg_len): 최종 응답 문자열 반환
- cc_log(const char *msg, int msg_len), debug_log(int step_num): 디버그 로깅 지원

이들 호출은 Wrapper_TA의 네이티브 심볼 테이블(chaincode_native_symbols)에 매핑되어 yield-resume 흐름을 유발한다. 즉, 호출 시 Wrapper_TA는 pending_type과 key/value를 설정하고 즉시 반환한다. 이후 Proxy가 gRPC를 통해 원장과 상호

작용한 뒤, COMMAND_RESUME_WASM을 호출하여 실행을 재개(resume)하면 Wrapper_TA가 응답 데이터를 Wasm 메모리에 복사한다.

3) **상태 기계(FSM) 기반 단계 전환**

기존 TA 코드가 ctx->chaincode_fct_state로 단계 전환을 관리했던 것처럼, Wasm 체인코드도 전역 FSM을 이용해 로직을 보존한다.

전역변수:

- current_op ∈ {OP_NONE, OP_CREATE, OP_ADD, OP_QUERY}
- fsm_state ∈ {0(Idle), 11(CREATE_AFTER_GET), 12(CREATE_AFTER_PUT), 21(AD
D_AFTER_GET), 22(ADD_AFTER_PUT), 31(QUERY_AFTER_GET)}

실행 단계:

- step_init(): 초기화 함수
- step_resume(): 실행 진입점. 최초 호출 시 함수명/인자 취득 및 분기, 이후 호출마다 FSM 상태에 따라 단계별 처리 수행

체인코드 FSM 동작

- Create:
  - cc_do_create_init(): 인자 취득 후 cc_get_state(person) 호출, fsm_state=11로 전환 후 종료
  - cc_do_create_resume():
    - state 11 → 기존 값 존재 시 "EXIST", 없으면 cc_put_state() 후 fsm_state=12
    - state 12 → "OK" 반환 후 FSM 리셋
- Add:
  - cc_do_add_init(): 인자 취득 후 cc_get_state(), fsm_state=21
  - cc_do_add_resume():
    - state 21 → 값이 없으면 "EMPTY", 있으면 정수 파싱 후 합산 → cc_put_state() → fsm_state=22
    - state 22 → "OK" 반환 후 FSM 리셋

- Query:
    - cc_do_query_init(): 인자 취득 후 cc_get_state(), fsm_state=31
    - cc_do_query_resume(): 값이 없으면 "NOTFOUND", 있으면 해당 값 반환 후 FSM 리셋

4) **메모리/버퍼 정책 및 경량 유틸리티**

버퍼 크기 상수: KEY_MAX=64, ARG_MAX=64, VAL_MAX=256 (Wrapper_TA/Proxy의 상수와 경계 일치)

전역 버퍼: g_function, g_arg0, g_arg1, g_person, g_cur_val, g_tmp 등은 재개 간 상태 유지를 위해 전역 변수로 유지

경량 유틸리티: s_strlen, s_memset, s_memcpy, s_streq, s_atoul, s_ultoa 등 자체 정의 함수로 표준 라이브러리 대체

3.2.5. Wrapper_TA 구현

1) **역할과 설계 의도**

Wrapper_TA는 Secure World(OP-TEE) 내부에서 WaTZ(WAMR 기반) 런타임을 구동하고, Proxy가 전달한 Wasm AOT 모듈과 체인코드 인자를 로드·실행하는 브리지이다. 기존 TZ4Fabric의 “체인코드=TA” 모델을 대체하여, 체인코드를 Wasm 모듈로 가져와 실행하며, 실행 중 발생하는 원장 접근(get_state, put_state)은 Native Function(호스트콜) 로 Proxy에 위임한다. 결과적으로 언어 이식성과 실행 격리(Wasm 샌드박스+TEE)를 동시에 달성한다.

2) **명령 집합과 인터페이스 (GlobalPlatform/TA API)**

Wrapper_TA는 세 가지 커맨드를 제공한다.

COMMAND_CONFIGURE_HEAP:

입력: VALUE_INPUT(size)

기능: WaTZ 런타임 힙 크기 사전 설정(정적 변수 heap_size 저장).

COMMAND_RUN_WASM:

입력: MEMREF_INPUT(AOT) | VALUE_INOUT(type) | MEMREF_IN

OUT(shared) | MEMREF_INOUT(stdout)

기능:

① Proxy로부터 전달받은 AOT 바이트코드를 신뢰 메모리로 복사

② struct arguments를 수신하여 함수명/인자 준비

③ WAMR 런타임 초기화 및 모듈 인스턴스 생성

④ Wasm 측 초기 진입점 step_init 호출

⑤ 이후 호스트콜 처리 루프(process_hostcall_flow)로 진입

COMMAND_RESUME_WASM:

입력: VALUE_INOUT(type) | MEMREF_INOUT(shared) | MEMR EF_INOUT(stdout)

기능: Proxy가 전달한 호스트 응답(원장 값 또는 ACK)을 Wasm 메모리로 되돌려주고, step_resume를 호출해 체인코드 실행을 재개

3) **세션 및 메모리 관리**

세션 컨텍스트: chaincode_session_ctx 전역 포인터(g_chaincode_sess)를 세션 단위로 유지. 최초 세션 오픈 시 메모리 할당 및 0 초기화

힙/모듈 버퍼:

- 힙 버퍼: heap_size 만큼 TEE_Malloc → WAMR 힙으로 전달
- 바이트코드 버퍼: AOT 모듈을 TEE_MemMove로 신뢰 메모리에 복사

표준 출력 버퍼: TA_SetOutputBuffer()로 Proxy가 준 stdout 수집 버퍼를 등록(디버깅/로그 수집)

공유 버퍼: Proxy와의 MEMREF_INOUT는 매 호출 시 struct key_value, struct acknowledgement, struct invocation_response, struct arguments 등 서로 다른 구조체를 교대로 사용(최대 크기 기준)

4) **WAMR 연동과 실행 스텝**

런타임 초기화:

- wamr_context에 힙 버퍼, 네이티브 심볼 테이블(chaincode_native_symbols), 바이트코드 포인터/크기를 설정
- TA_InitializeWamrRuntime()로 모듈 로드·인스턴스 생성

진입점 호출:

- call_step(runtime, "step_init")로 Wasm 내 초기화 루틴 수행
- 이후 체인코드의 흐름 제어는 step_resume 반복 호출을 통해 진행

함수 호출 보조:

- call_step()는 wasm_runtime_lookup_function으로 심볼을 조회하고, 필요 시 exec_env를 온디맨드 생성 후 wasm_runtime_call_wasm 호출
- 예외(wasm_runtime_get_exception)를 사전/사후로 검사·클리어하여 런타임 안정성을 높임
- 호출 성공 시 exec_env를 정리(메모리 누수 방지)

5) **호스트콜 처리(원장 접근) - process_hostcall_flow()**
체인코드 Wasm 모듈은 네이티브 임포트를 통해 원장 접근을 요청하면, Wrapper_TA는 그 사실을 pending_type, key, value 등으로 세션 컨텍스트에 저장한다. 이후 흐름은 다음과 같다.

① **재개 스텝 실행**: step_resume 호출 → Wasm 측 네이티브 임포트가 pending_type을 설정

② **요청 판정 및 전달**:

- GET_STATE_REQUEST → params[1].value.a = GET_STATE_REQUEST 설정, params[2]에 key 기록 후 즉시 반환 (Normal World로)
- PUT_STATE_REQUEST → params[1].value.a = PUT_STATE_REQUE
  ST 설정, params[2]에 key/value 기록 후 즉시 반환

③ **호스트 응답 수신**(COMMAND_RESUME_WASM 경로):

- GET: params[2]의 value를 Wasm 메모리 버퍼 (wasm_out_offset/l
  en)로 검증 후 안전 복사(주소검증 wasm_runtime_validate_app_addr, 변환 addr_app_to_native)
- PUT: ACK만 확인하고 다음 스텝 진행

④ **최종 응답 생성**: 추가 요청이 없을 경우 params[1].value.a = INVOCATION_RESPONSE 설정, invocation_response.execution_response에 결과 문자열을 채워 반환

3.2.6. 실행 흐름 요약

WaTZ-TZ4Fabric 통합 구조에서 체인코드가 실행되는 전체 흐름의 요약은 다음과 같다.

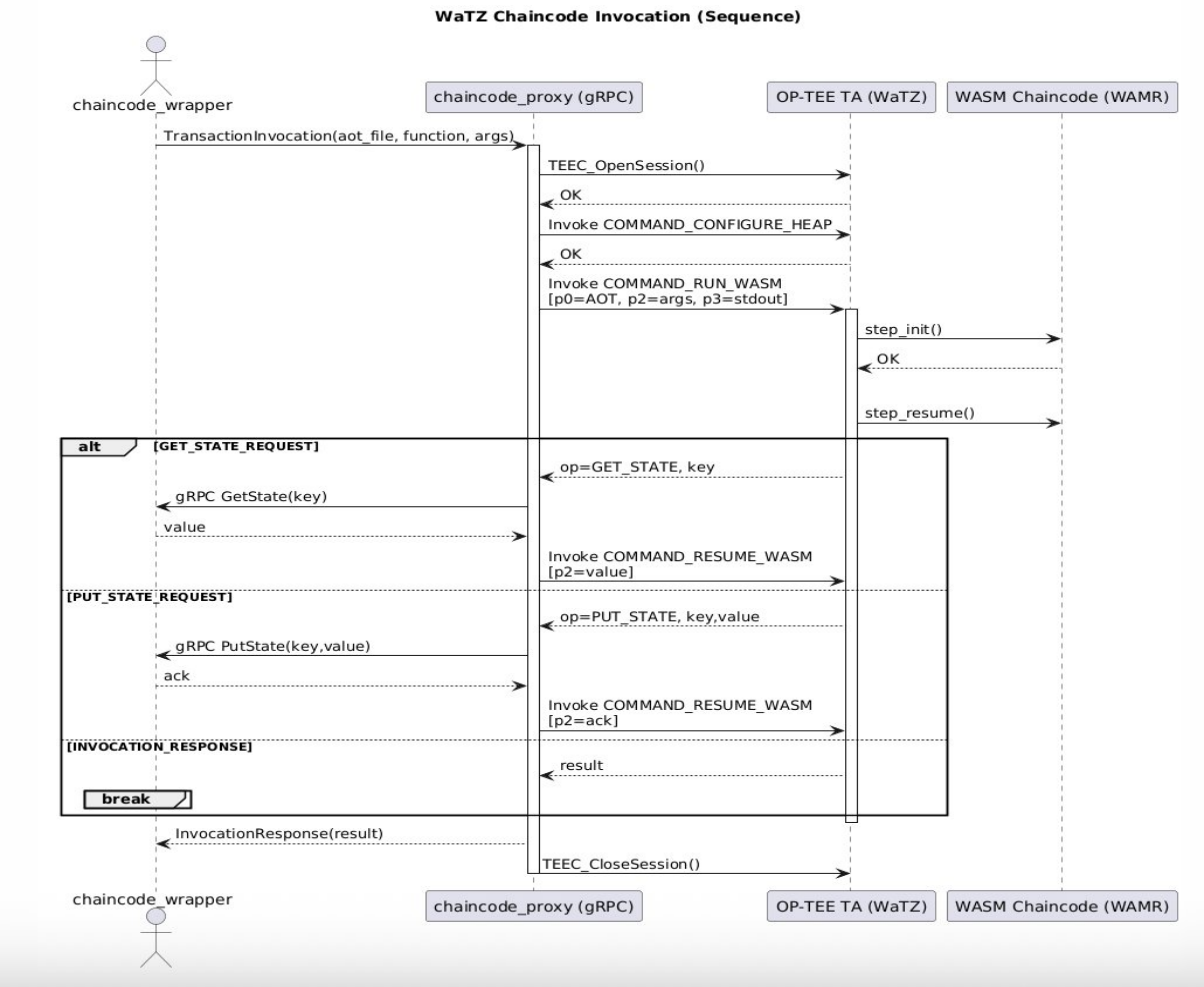

그림 10. 체인코드 실행 시퀀스 다이어그램

① **Invoke 요청**
Hyperledger Fabric 피어 노드가 Proxy에 체인코드 Invoke 요청을 전송한다.

② **세션 생성**
Proxy는 요청을 수신한 후 TEE 내부에서 세션을 열고, Wasm 체인코드 실행 환경을 초기화한다.

③ **체인코드 실행 시작**
Wrapper_TA는 WaTZ 런타임을 호출하여 Wasm 체인코드를 Secure World에서 실행한다.

④ **체인코드 로직 분리**
체인코드 로직이 실행되며, 실행 중 원장 접근이 필요한 경우 get_state, put_state 요청이 발생하고, 이때 Wrapper_TA 내 Native Function이 호출된다.

⑤ **상태 접근 요청**
Native Function 호출 시 현재 실행 컨텍스트를 보존한 뒤, Proxy로 get_state 또는 put_state 요청을 전달한다.

⑥ **원장 접근**
Proxy는 gRPC를 통해 Hyperledger Fabric 원장에 접근하여 요청된 데이터를 조회하거나 기록한다.

⑦ **결과 반환**
Proxy는 원장에서 얻은 결과를 다시 Wrapper_TA로 전달하고, TA는 해당 값을 체인코드 실행에 반영한다.

⑧ **최종 결과 전송**
체인코드 실행이 완료되면 최종 결과가 Proxy에 반환되며, Proxy는 이를 gRPC를 통해 Fabric 네트워크로 다시 전달한다.

4. 연구 결과 분석 및 평가

AOT로 컴파일한 WASM 체인코드를 WaTZ 라이브러리를 통해 Secure World에서 실행하였고, Native Function을 이용한 GET_STATE/PUT_STATE 연동에 성공하였다. 단발 호출뿐 아니라 컨텍스트를 유지한 상태에서의 반복 호출도 정상 동작했으며, 검증에는 (i) GET_STATE 1회 호출 함수와 (ii) GET_STATE 후 조건에 따라 PUT_STATE를 1회 수행하는 함수를 사용했다.

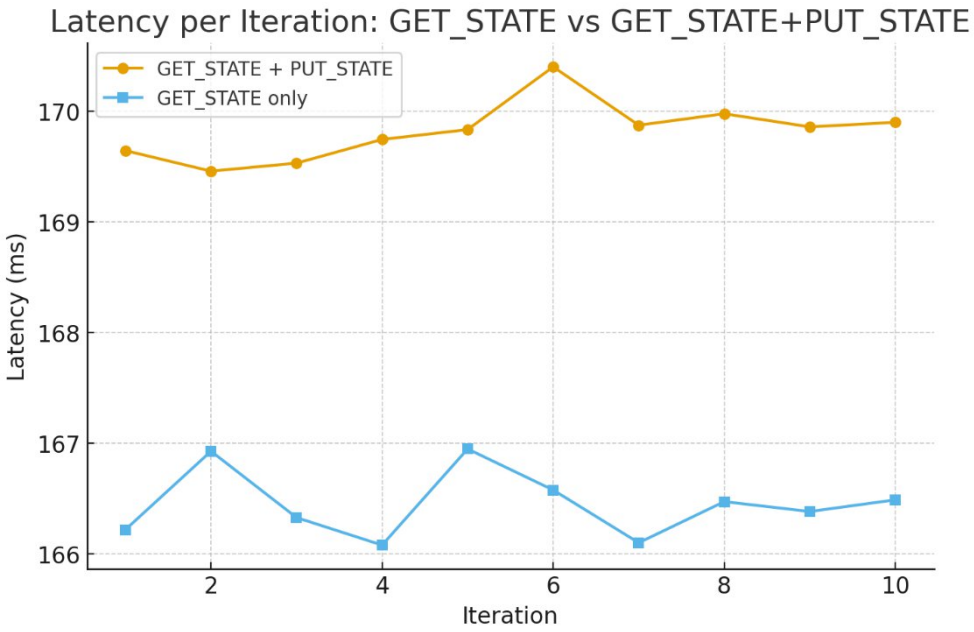

그림 11. 반복별 지연 시간 그래프

성능 관측 결과, GET_STATE만 수행할 때에 비해 GET_STATE 후 PUT_STATE까지 수행하는 경우 평균 지연이 약 **3ms 증가**하였다. 이는 현재 스택에서 **Native Function 1회 호출당 약 3ms의 오버헤드**가 발생함을 시사한다. 단순·단발 체인코드에서는 영향이 경미하지만, Native Function 호출이 다수 누적되는 복잡한 체인코드의 경우 총 지연이 의미 있게 커질 수 있다.

또한, 기존 TZ4Fabric 경로에서 **체인코드 1회 실행당 약 2-3ms**가 소요되던 것과 비교하면, 현재 Wrapper TA 기반 WASM 실행 경로에서는 **약 166ms**의 지연이 관측되어 **WAMR 런타임 생성·초기화 오버헤드**가 지배적인 병목으로 해석된다. 이 수준의 지연은 실사용 관점에서 수용하기 어렵기 때문에, **런타임 생애주기 관리의 최적화**가 필요해 보인다.

종합하면, 본 연구는 기능적 타당성(보안 영역 내 WASM 체인코드 실행 및 원장 연동)을 입증하면서도, **지연 시간 프로파일**에서는 (1) 호출당 ≈3ms의 Native Function 경계 비용과 (2) 1회 실행당 ≈166ms의 초기화 비용이 핵심 제약임을 확인하였다. 따라서 실용화를 위해서는

다음과 같은 개선이 요구된다.

- **런타임/모듈 재사용**: WAMR 런타임과 체인코드 모듈/인스턴스를 호출 간에 **상주시켜 재사용**하여 초기화 비용을 상쇄.
- **컨텍스트 캐싱**: 안전성을 해치지 않는 범위에서 **키-값 캐시**와 **컨텍스트 재사용**으로 반복 접근 지연을 완화.

결론적으로, 본 프로토타입은 **보안 실행 모델의 가능성**을 보여주었으나, **지연 시간의 절대값**이 실사용 임계치를 초과한다. 상기 최적화 항목을 적용해 초기화 오버헤드를 한 자릿수 ms대로, 호출당 오버헤드를 1ms 내외로 축소하는 것이 차기 단계의 핵심 목표가 될 것이다.

5. 결론 및 향후 연구 방향

본 연구는 AOT로 컴파일된 WASM 체인코드를 WaTZ 기반으로 Secure World(OP-TEE)에서 실행하고, Native Function을 통해 GET_STATE/PUT_STATE를 안정적으로 수행할 수 있음을 실증하였다. 컨텍스트를 유지한 반복 호출도 기능적으로 문제없이 동작하였다.
다만 성능 측면에서는 (i) **WAMR 런타임/모듈 초기화로 인한 1회 실행당 약 166 ms의 초기화 오버헤드**와 (ii) **Native Function 경계 통과 시 호출당 약 3 ms의 지연**이 확인되었다. 이는 단순 체인코드에는 수용 가능하나, 호출이 누적되는 복잡 시나리오에서는 실사용 레이턴시를 저해할 수 있다. 결과적으로, 본 프로토타입은 **기능적 타당성**은 입증했으나 **실용적 성능을 확보하기 위한 체계적 최적화**가 필요하다는 결론에 이른다.

향후에는 런타임 생애주기를 최적화하는 방향으로 연구를 진행하여야 할 것 같다. WAMR 런타임과 체인코드 모듈을 **상주시켜 재사용**하는 구조가 도입되어야 할 것이다. 모듈의 로드·검증·링크를 사전에 완료하고, 호출 시에는 준비된 인스턴스를 즉시 사용하는 방식이 채택되어야 한다. 이를 통해 대부분의 실행 경로를 **웜 스타트**로 전환함으로써 초기화 지연을 근본적으로 억제할 수 있을 것이다.

6. 구성원별 역할 및 개발 일정

6.1.1. 구성원별 역할

이름	역할
이준태	- 체인코드 연구 및 시도 - 발표 자료 준비 및 발표 진행 - 최종보고서 작성
이준혁	- 전체 디자인 설계 - 전반적인 코드 작성 및 연동 - 최종보고서 작성
위재준	- 프록시 연구 및 시도 - 포스터 제작 - 최종보고서 작성

6.1.2. 개발 일정

월	계획
5월: 논문 분석 및 기초 설계	- 관련 논문 및 기술 문서 분석 - TrustZone, Wasm, OP-TEE 개념 학습 - 착수보고서 및 지도확인서 제출
6월: 실험 환경 구축 및 구조 이해	- MCIMX8M-EVK 보드를 활용한 OP-TEE 환경 구축 - WATZ 빌드 및 AOT 실행 실습 - SQLite, Genann 벤치마크 준비
7월: 성능 측정	- AOT 기반 성능 실험 및 실행 시간 측정 - WATZ 내 프로토콜 분석 - evidence 생성 및 해시 측정 테스트 - 중간보고서 및 중간평가표 제출
8월: 하이퍼레저 패브릭 연동 구현	- TZ4Fabric 구조 분석 - chaincode proxy ↔ WATZ 연동 흐름 설계 - 체인코드 결과 전달 및 wrapper 통합 구현
9월: 최종 발표 준비 및 마무리	- 최종보고서 및 최종평가표 작성 - 발표자료 제작 및 시연 영상 정리 - 졸업 과제 발표 심사 진행

7. 참고 문헌

[1] Rossi, M., Platzer, C., and Kargl, F., "WaTZ: A Trusted WebAssembly Runtime Environment with Remote Attestation for TrustZone," *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2023.

[2] Linaro, "OP-TEE: Open Portable Trusted Execution Environment," Available: https://www.op-tee.org/, Accessed: 2025.

[3] Haas, A., Rossberg, A., Schuff, D. L., Titzer, B. L., Holman, M., Gohman, D., Wagner, L., Zakai, A., and Bastien, J., "Bringing the Web up to Speed with WebAssembly," *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 185-200, 2017.

[4] Hyperledger Foundation, "Hyperledger Fabric Documentation," Available: https://hyperledger-fabric.readthedocs.io/, Accessed: 2025.

[5] Ménétrey, J., Pasin, M., Felber, P., and Schiavoni, V., "WaTZ: A Trusted WebAssembly Runtime Environment with Remote Attestation for TrustZone," *Proceedings of the 42nd IEEE International Conference on Distributed Computing Systems (ICDCS)*, pp. 00116, 2022.

[6] Müller, C., Brandenburger, M., Cachin, C., Felber, P., Göttel, C., and Schiavoni, V., " TZ4Fabric: Executing Smart Contracts with ARM TrustZone (Practical Experience Report)," *Proceedings of the 39th IEEE International Symposium on Reliable Distributed Systems (SRDS)*, pp. 00011, 2020.

[7] R. Cheng, S. Dani, J. Zhang, N. Hynes, A. Johnson, D. Song, I. Abraham, A. Goyal, E. Shi, M. Maffei, M. Mohassel. (2018). Ekiden: A Platform for Confidentiality-Preserving, Trustworthy, and Performant Smart Contract Execution.

[8] GitHub. (2022, Jun. 21). *unine-watz* [Online]. Available: https://github.com/jaejunwi/unine-watz (Accessed: 2025, Sep. 16).

[9] GitHub. (2019, Aug. 28). *open-source-fabric-optee-chaincode* [Online]. Available: https://github.com/jaejunwi/open-source-fabric-optee-chaincode (Accessed: 2025, Sep. 16).