1.1. Pusan National University Computer Science and Engineering Technical Report 2025-09

마이크로 컨트롤러에서의 안전한 로봇 어플리케이션 수행을 위한 원격 증명 기술 개발

저자 1 : 강승민

저자 2 : 김의준

저자 3 : 박재선

지도교수 : 권동현

목 차

2. 서론

2.1. 연구 배경

로봇 시스템은 제조업, 유통, 의료 등 다양한 산업 분야에서 활용 범위를 확장하고 있다. 특히 IoT 기술과 결합한 임베디드 로봇의 중요성이 증가하고 있다. 스마트 팩토리, 자율 운송로봇 등 네트워크를 통해 외부와 데이터를 교환하며 고도의 서비스를 제공할 수 있기 때문이다.

그러나 IoT 로봇의 네트워크 연결성 증대는 보안 위협의 증가로 이어진다. 임베디드 장치가 공격 받을 경우 시스템 전체 마비가 발생하거나 잘못된 제어 명령이 전달될 수 있다. 이는 로봇의 안전 문제와 직결되기에 MCU (Microcontroller Unit)를 기반으로 한 로봇 시스템의 보안은 필수적이다[1]. 따라서 본 과제에서 로봇 시스템 속의 MCU의 변조를 감지하는 방법을 연구하고자 한다.

2.2. 기존 문제점

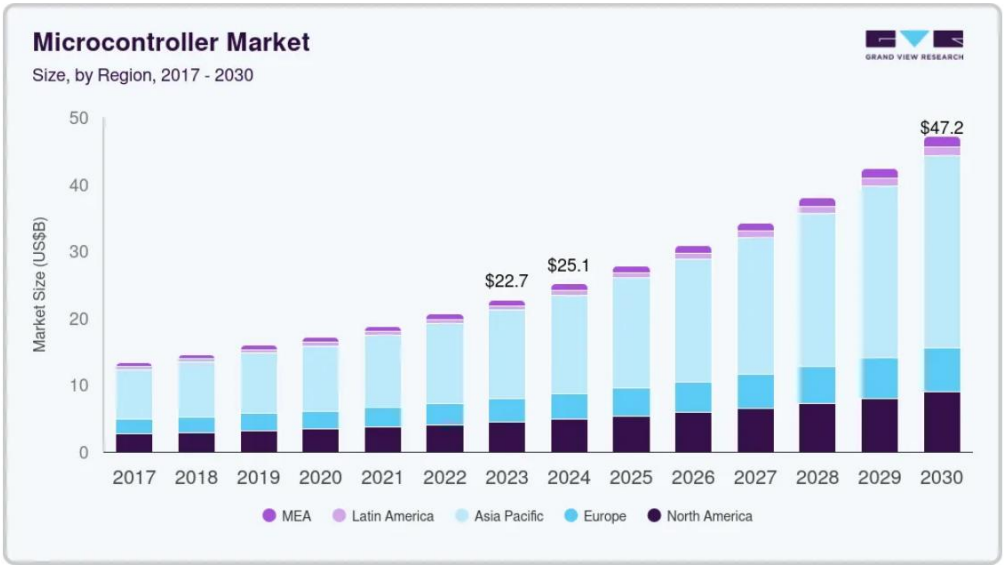
현재 사용되고 있는 ROS (Robot Operating System)의 경량화 버전인 Micro-ROS는 MCU에서 동작하는 로봇 어플리케이션 개발의 편의성을 높였으나 보안 측면에서의 한계가 존재한다. MCU는 자원이 제한되어 있어 높은 수준의 암호화, 인증 등의 보안 기능을 구현하기 어렵다. 또한 Micro-ROS는 보안 표준을 준수하여 개발되지 않아 보안성이 보장되어 있지 않다. 이러한 문제는 로봇 시스템이 안전이 요구되는 환경에 투입되었을 때 심각한 보안 취약점으로 작용할 수 있다.

초기 연구는 TrustedFirmware-M[2] 기반의 단일 Root-of-Trust와 Flash 영역 중심의 정적 증거 활용에 초점을 맞추었다. 이는 다양한 MCU 보드에 적용하기 어렵고 런타임 동안의 공격에 대한 대응이 미흡하다. 특히 TPM 부재 환경에서는 활용이 제한적이라는 문제점을 가지고 있다.

2.3. 연구 목표

이번 과제에서는 Micro XRCE-DDS[3] 통신 과정에 부팅 시점의 원격 증명을 통합하는 보안 구조를 구현하는 것을 목표로 한다. 세 가지 Root of Trust 기반 보안 구조(DICE 단독 구조, DICE와 MPU를 결합한 구조, TF-M과 TrustZone-M을 통합한 구조)를 설계하고 구현한다. 이후 각 구조에 대해 성능을 종합적으로 분석하고 평가하는 보고서를 작성한다. 이를 통해, 본 과제는 제한된 자원을 가진 MCU 기반 시스템에서 원격 증명을 포함한 보안 아키텍처를 제안하고 실효성을 입증하는데 중점을 두었다.

3. 연구 배경



[Figure 1] MCU의 시장 현황 및 추후 전망[4]

최근 IoT와 로봇 기술의 발전으로 MCU는 단순 제어를 넘어 자율주행, 스마트 팩토리 등 복잡하고 중요한 역할을 수행하게 되었으며 이러한 역할 확대에 힘입어 MCU 시장 역시 Figure 1과 같이 지속적인 성장이 전망됩니다. 이런 다양한 시스템에서 MCU는 센서 데이터 수집, 통신 등 핵심적인 기능을 담당하지만 크기와 전력 소비를 최소화하기 위해 연산 능력과 메모리가 제한적이라는 특징을 가진다. 이러한 특징으로 인해 PC나 서버 같은 일반 컴퓨팅 환경에서 사용되는 복잡한 보안 프로토콜을 적용시키기 어렵고, 악의적인 공격에 취약해질 위험이 높다. 만약 MCU의 소프트웨어가 변조될 경우 시스템 오작동을 넘어 물리적 파손이나 심각한 인명피해로 이어질 수 있다 따라서 자원 제약적인 MCU환경에 최적화된 경량 무결성 검증 기술을 확보하는 것이 필수적이다.

3.1. 기존 문제점

3.1.1. Micro-ROS 및 ROS2의 보안 기능 미흡

ROS2[5]는 DDS-Security 표준에 맞춰 설계되어 있다. 하지만 Micro-ROS[6]는 경량화에 집중해 설계되어 있어 인증, 암호화, 무결성 검증 등의 보안 기능이 충분히 구현되지 않았다. Micro-ROS에서 보안 기능을 활성화하기 위해서 추가적인 메모리와 연산 자원이 요구된다. 따라서 저사양 MCU에서는 보안 기능 구현 자체가 어렵다는 한계가 존재한다.

ROS2는 여러 노드가 데이터를 주고받는 분산 시스템 구조를 기반으로 한다. 이 구조에서는 각 노드 간에 기본적인 신뢰 관계가 형성되어 있어 하나의 기기가 탈취될 경우

연결된 다른 모든 기기까지 위험에 노출된다. 예를 들어 상대적으로 보안이 취약한 센서 노드 하나가 공격자에게 장악되면 해당 노드를 통해 시스템 전체에 잘못된 제어 명령을 전파하거나 시스템 마비를 유발할 수 있다. 이처럼 개별 노드의 신뢰성을 보장하지 않으면 가장 약한 지점 하나가 전체 로봇 시스템을 위협하는 심각한 보안 허점으로 작용하게 된다.

3.1.2. 기존 원격 증명 기술의 한계

TPM과 같은 전통적인 원격 증명은 고성능 CPU와 전용 보안 칩이 요구되어 연산과 메모리 자원이 제한적인 MCU, 임베디드 환경에 적용하기에 한계가 있다. 대표적인 문제점으로는 아래와 같다.

- **다양한 MCU 하드웨어의 미지원** : TF-M은 ARMv8-M 아키텍처의 TrustZone-M 기능을 활용하는데 최적화 되어 있다. 이는 ARMv7-M 기반의 MPU만 지원하는 대다수의 저사양 MCU에는 직접 적용하기 어렵다. 또한 TPM과 같은 전용 보안 칩이 없는 경우가 많아, 하드웨어 기반의 강력한 신뢰점 구축이 불가능하다는 문제가 있다. 이러한 하드웨어의 파편화는 솔루션의 범용성을 크게 저해한다.
- **자원 제약 환경에 부적합한 기존 보안기술 :** 기존의 IT 환경에서 사용되는 TLS/SSL과 같은 표준 보안 프로토콜이나 TPM을 활용한 원격 증명 방식은 높은 수준의 보안을 제공하지만 상당한 연산 능력과 메모리 공간을 요구한다. 이러한 기술들은 수 KB 수준의 메모리를 가진 자원 제약적인 MCU 환경에 그대로 적용하기에는 매우 무겁고 비효율적이다. 이로 인해 저사양 임베디드 시스템에서는 보안 기능의 구현 자체가 어렵거나 구현하더라도 시스템 본연의 실시간 성능을 심각하게 저해하는 문제가 있었다.
- **통신 보안의 부재 :** Micro-ROS의 핵심 통신 프로토콜인 DDS (Data Distribution Service)는 실시간 데이터 분배를 위하여 설계되었기 때문에 보안 기능이 기본적으로 미비하다. 그 결과 공격자가 중간자 (Man-in-the-Middle) 공격을 통해 데이터를 도청하거나 악의적인 노드를 위장하여 잘못된 명령을 주입하는 Replay 공격에 쉽게 노출된다. 예를 들어 외부 공격자가 로봇의 센서 데이터를 탈취하여 산업기밀을 유출하거나 허위 명령을 보내 로봇의 정상 동작을 방해할 수 있다.

3.2. 관련 선행 연구 및 기반 지식

Tytan, DDS Security+, SMART 등 관련 선행 연구들을 분석하고, MCU 기반의 시스템 보안에 대한 기반 지식을 구축하였다. 이를 통해 DDS 통신 프로토콜의 취약점과 원격 증

명 기법의 중요성을 파악하였다.

3.2.1. TyTAN

TyTAN[7]은 임베디드 시스템에서 기존 TPM 기반 보안의 한계(비용, 복잡성, 실시간성 부족)를 극복하기 위해 제안된 보안 프레임워크이다. 핵심 구성요소인 "EA-MPU (Execution-Aware Memory Protection Unit)"는 태스크 단위의 메모리 접근을 강력히 제어하여 각 태스크의 코드·데이터를 독립적으로 보호하고 인터럽트 발생 시에도 안전한 상태 전환을 보장한다. 이를 기반으로 TyTAN은 태스크 간 강력한 격리, Secure Boot, Trusted Execution을 지원하며 동적 태스크 로딩, 안전한 IPC, Local 및 Remote Attestation을 가능하게 한다. FreeRTOS와 결합해 실시간성을 보장하면서도 경량성을 유지하여 자동차 전자제어장치 (ECU)나 IoT 기기 등 보안과 실시간 요구가 동시에 필요한 환경에 적합하다.

3.2.2. DDS Security+

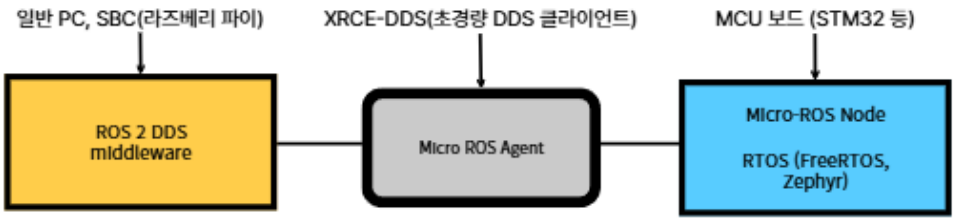
DDS Security+[8]는 기존 DDS 보안 아키텍처에 TPM (Trusted Platform Module) 기반 원격 증명 기능을 통합하여 통신 노드의 코드 무결성을 검증하는 방안을 제시한다. 이 연구는 TPM의 PCR (Platform Configuration Register) 값을 활용해 부팅 과정의 소프트웨어 스택 변조 여부를 확인하고 검증된 노드만 보안 채널을 설정하도록 한다. 본 보고서의 연구가 TPM이 없는 자원 제약적 MCU 환경을 대상으로 하는 반면 이 논문은 TPM 하드웨어에 의존한다는 점에서 차이가 있다. 결과적으로 이 연구는 신뢰할 수 있는 노드 간의 보안 통신 기반을 마련하는 선행 연구로서 의미가 있다.

3.2.3. SMART (Secure and Minimal Architecture for Root of Trust)

SMART[9]는 자원이 제한된 MCU 환경에서 원격 증명을 가능하게 하기 위해 제안된 경량 보안 아키텍처이다. SMART는 하드웨어와 소프트웨어의 공동 설계를 통해 각각 독립적으로 사용한 아키텍처의 단점을 보완한다. 원격 증명을 위한 key에 접근할 수 있는 유일한 영역과 key를 저장하는 영역을 하드웨어 변경을 통해 제공한다. 추가로 소프트웨어 구현으로 key에 접근을 제어하고 잘못된 동작이 발생하거나 정상적으로 종료되지 못한 경우 메모리를 초기화하는 기능을 제공한다. 이를 통해 저가형 MCU에서도 신뢰성을 보장할 수 있으며 본 보고서에서 구현하고자 하는 보안 기능과 유사한 방향성을 가진 선행 연구로 의의가 있다.

3.3. 핵심 기술 및 배경 지식

3.3.1. Micro-ROS



[Figure 2] Micro-ROS의 구조

Micro-ROS[6]는 ROS 2 (Robot Operating System 2)의 기능을 자원 제약형 임베디드 장치로 확장하기 위해 개발된 경량 런타임이다. ROS 2가 DDS (Data Distribution Service)를 기반으로 분산 시스템 간의 상호 운용성을 제공하는 반면 Micro-ROS는 이를 MCU 환경에 최적화하여 제공한다. 특히 Zephyr[10], FreeRTOS[11]와 같은 RTOS (Real-Time Operating System)와 통합되어 동작하며 초저전력·저용량 환경에서도 ROS 2 노드로서 기능할 수 있도록 한다. 그러나 Micro-ROS는 성능 및 메모리 최적화를 우선시하기 때문에 보안 기능의 지원은 제한적이며 이는 본 연구에서 보완하고자 하는 주요 대상이 된다. Micro-ROS는 ROS 2의 핵심 기능을 MCU로 코드 이식한 것으로 이를 통해 리소스가 제한적인 MCU에서도 ROS2 생태계의 실시간 통신 및 미들웨어 기능을 활용할 수 있었다.

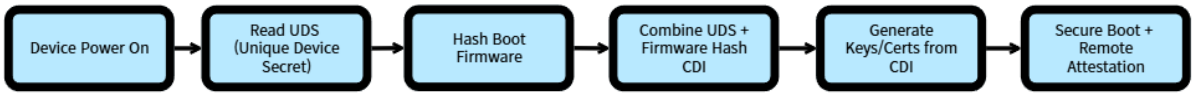
3.3.2. DICE (Device Identifier Composition Engine)

Device Hardware
UDS (Unique Device Secret) - 칩에 고유한 키
DICE Engine
1. 부팅 시 펌웨어 해시 계산 2. UDS + 펌웨어 해시 → CDI 생성 3. CDI로 Device Key 파생
Secure Bootloader
- 펌웨어 무결성 검증 - CDI 전달 및 신뢰 체인 생성
Firmware
- CDI 기반 서비스 키 및 인증서 - Remote Attestation 지원

[Table 1] DICE Architecture

DICE[12]는 TCG (Trusted Computing Group)에서 제안한 경량 신뢰 루트 (Root of Trust) 프레임워크로 자원 제약적 장치에서도 보안 부팅 및 신뢰 연속성 (Chain of Trust)을 제공하기 위해 설계되었다. DICE는 UDS (Unique Device Secret)를 기반으로 부팅 과정에서 각 소프트웨어 계층을 hashing하여 새로운 식별자와 키를 도출한다. 이를 통해 부팅 시점부터 이후 계층에 대한 무결성을 보장하며 원격 증명 (Remote Attestation)과 같은 상위 보안 기능의 기반을 제공한다. DICE의 가장 큰 장점은 별도의 보안 하드웨어 없이도 경량 MCU 환경에 적용 가능하다는 점이다. 위의 Table 1은 DICE의 기본 Architecture를 보여주고 있다.

TPM과 같은 전용 보안 칩이 없는 MCU 환경을 위한 경량화된 Roots-of-Trust 기술이다. DICE는 디바이스 부팅 시점의 펌웨어 해시값과 고유 식별자를 조합하여 신뢰 체인을 형성하며, 이를 통해 런타임 환경에 대한 신뢰를 증명한다. 아래 Figure 3은 DICE의 MainFlow를 간략화해서 나타낸 것이다.



[Figure 3] DICE main Flow

3.3.3. MPU (Memory Protection Unit)

임베디드 시스템에서 보안과 안정성 확보는 필수적이다. 특히 마이크로컨트롤러 (MCU)는 제한된 자원 환경에서 동작하기 때문에 하드웨어 수준의 메모리 접근 제어 기능과 실행 권한 관리가 중요하다. 이를 위해 ARM Cortex-M 계열을 포함한 최신 MCU는 MPU (Memory Protection Unit)를 제공하며 시스템 권한 레벨을 Privileged Mode와 Unprivileged Mode로 구분하여 안전한 소프트웨어 실행 환경을 제공한다[13].

MPU는 MCU 내부의 메모리 영역 접근 권한을 설정하고, 권한이 없는 접근 시 하드웨어적으로 예외 (Fault)를 발생시켜 시스템 안정성을 확보하는 보안 기능이다. MPU를 사용하면 펌웨어 코드, 민감한 데이터, 주변장치 레지스터 등에 대한 접근을 영역별로 제한할 수 있다. 이에 따른 소프트웨어 권한 레벨을 Privileged와 Unprivileged로 나누어 운영하는 것이 ARM Cortex-M 아키텍쳐이다.

Privileged Mode는 모든 시스템 자원에 접근 가능하며 커널 또는 부트로더와 같은 핵심 소프트웨어에서 사용된다. 반면 Unprivileged Mode는 제한된 자원만 접근할 수 있어 애플리케이션 코드 실행 시 보안을 강화한다.

시스템 부팅 시 MCU는 Privileged Mode로 시작된다. 부트로더는 MPU를 설정하여 메모리 접근 정책을 구성한 뒤 애플리케이션 코드를 Unprivileged Mode에서 실행하도록 전환한다. 애플리케이션이 하드웨어 접근 또는 권한 상승이 필요한 작업을 수행할 경우 SVC (Supervisor Call) 명령어를 통해 커널로 제어를 요청한다.

3.3.4. TF-M (TrustedFirmware-M)

TrustedFirmware-M[2] 은 Arm에서 제공하는 오픈소스 보안 펌웨어로, Armv8-M 아키텍처 기반의 Cortex-M 프로세서에서 실행된다. TF-M은 SPE (Secure Processing Environment)를 제공하여 민감 자원 및 보안 기능을 보호된 실행 영역에서 동작하도록 한다. 주요 기능으로는 보안 서비스(암호화, 키 관리, 초기 증명 등)의 제공, 보안 부팅 (Secure Boot) 지원, PSA (Platform Security Architecture)[14] API 구현 등이 있다. 이를 통해 MCU 환경에서도 범용 보안 표준에 부합하는 보안 체계를 구축할 수 있으며 본 연구에서는 초기 증명 서비스와의 연계를 통해 Remote Attestation을 구현하는 핵심 기반으로 활용한다.

TF-M은 시스템을 보안 영역 (Secure World)과 비보안영역 (Non-Secure World) 로 분리하여 중요자산을 보호한다. 특히 TF-M에서 제공하는 PSA (Platform Security Architecture) API를 활용하여 원격 증명 토큰을 생성하고 이를 통해 MCU의 무결성을 외부에 증명할 수 있다.

PSA Attestation은 PSA 사양에서 정의한 보안 서비스로 장치가 실제 하드웨어 기반 Root of Trust에 기반하고 있음을 증명하고 실행 중인 펌웨어가 변조되지 않았음을 증명하여 소프트웨어 무결성을 증명한다.

TF-M과 PSA Attestation은 리소스가 제한된 MCU 환경에서도 효과적으로 동작하며 경량 암호화와 압축 토큰으로 자원 절약하고 MPU를 통한 실행 환경 격리로 보안 경계 강화할 수 있다. 위에서 언급한 DICE 및 하드웨어 신뢰 앵커와 결합해 강력한 Root of Trust를 구현할 수 있다.

3.3.5. TZ-M (TrustZone-M)

TrustZone-M은 Arm Cortex-M 계열 프로세서에서 하드웨어 수준의 보안 분리를 제공하는 기술이다. 이는 메모리, 주변 장치, 실행 모드를 보안 영역 (Secure World)과 비보안 영역 (Non-Secure World)으로 구분하여 민감 자원에 대한 무단 접근을 방지한다.

TrustZone-M은 소프트웨어만으로는 달성하기 어려운 강력한 격리 (isolation)를 보장하며 TF-M과 결합될 경우 MCU 환경에서도 고신뢰도의 보안 아키텍처를 구현할 수 있다. 본 연구에서는 TrustZone-M을 이용하여 보안 부팅 및 실행 환경을 강화하고 원격 증명에 필요한 민감 데이터의 보호를 달성하고자 한다.

TZ-M은 프로세서 자체에서 Secure World와 Non-Secure World를 분리해, 한쪽 영역의 침해가 다른 쪽 영역에 영향을 미치지 않도록 강력한 격리를 제공한다. 이를 통해 TF-M과 같은 보안 펌웨어의 신뢰성을 더욱 강화할 수 있다.

4. 연구 내용

4.1. 연구 환경 구축

본 과제에서 사용한 하드웨어는 ST Nucleo-L552ZE-Q[15] 보드이다. 해당 보드는 Arm Cortex-M33 프로세서 기반으로 과제에서 설계한 MPU와 TrustZone-M 기능을 지원하며 적정 수준의 플래시 및 SRAM 자원을 제공한다. 따라서 DICE 기반의 보안 아키텍처나 TF-M, TrustZone-M 기반의 보안 아키텍처를 구축하기에 적합하다.

본 과제는 Micro-ROS가 지원하는 여러 RTOS 중 Zephyr를 선택하여 진행하였다. Zephyr는 오픈소스 RTOS로 보안 기능 확장을 고려하여 커널 설정 옵션과 메모리 관리 기능을 가지고 있어 설계한 보안 아키텍처 통합 실험을 진행하기에 적합하다. 또한, west 툴체인을 통한 모듈화 구조로 되어있고 의존성 관리와 모듈 업데이트가 용이하며 다양한 보드를 지원한다. 따라서 추후 다양한 보드로의 확장에 용이하다고 판단하여 선택했다.

4.1.1. Zephyr

Zephyr는 공식 문서에서 제공하는 방법을 기반으로 설치하였다. West를 이용하여 zephyr 프로젝트를 초기화하였고 nucleo-L552ZE-Q 보드 지원 패키지를 포함한 Zephyr SDK를 활용하였다. 빌드 환경은 Table 2와 같이 구성하였다. 펌웨어는 ARM-GCC 툴체인을 통해 빌드하고 디버깅하였다. 환경 설정은 이후 Micro-ROS 모듈과의 통합, 보안 기능 적용, 평가를 위한 일관된 개발 환경을 보장한다.

항목	버전
운영체제	Ubuntu 22.04 LTS
Python	Python 3.10.12
SDK	Zephyr-SDK 4.0

[Table 2] 환경 구성표

Zephyr의 Kconfig 및 CMake 기반 빌드 시스템은 실험 목적에 따라 커널 설정을 조정할 수 있도록 지원한다. 이를 이용하여 이후 각 실험에서 MPU 설정과 같은 보안 관련 옵션을 각 아키텍처에 맞게 구성하였다. 또한 Figure 4와 같이 stm32l5.dtsi 파일을 수정하여 Micro-ROS를 이용한 통신을 수행할 때 LPUART_1 하드웨어 자원을 소프트웨어적으로 식별할 수 있도록 하였다.

```
lpuart1: serial@40008000 {
    compatible = "st,stm32-lpuart", "st,stm32-uart";
    reg = <0x40008000 0x400>;
    clocks = <&rcc STM32_CLOCK(APB1_2, 0U)>;
    resets = <&rctl STM32_RESET(APB1H, 0U)>;
    interrupts = <66 0>;
    status = "disabled";
    label = "LPUART_1";
};
```

[Figure 4] stm32l5.dtsi 수정사항

4.1.2. Micro-ROS

Micro-ROS는 Zephyr 환경에 통합하기 위해 micro-ros-zephyr-module을 설치하여 사용하였다. 해당 모듈은 기본 zephyr의 firmware 부분을 대체하는 구조로 제공된다. 이를 통해 Micro-ROS 클라이언트 노드를 nucleo-L552ZE-Q 보드 상에서 실행할 수 있다.

Micro-ROS 에이전트는 Micro XRCE-DDS를 기반으로 하여 클라이언트와 DDS 네트워크 간의 중간자 역할을 수행한다. 본 과제에서는 micro-XRCE-DDS-Agent 소스 코드를 가져와서 필요에 맞게 수정하여 사용하였다. 특히 XRCE-DDS 세션 handshake 과정에 부팅 시점 원격 증명 절차를 통합하기 위한 변경을 진행하여 보안 기능의 실효성을 검증할 수 있도록 하였다.

실험에 사용하는 nucleo-L552ZE-Q 보드와의 연결을 위해 zephyr_transport_open 함수를 변경하였다. 기존 구현에서는 디바이스의 명칭으로 UART_1를 사용하였으나 본 과제에서는 보드의 메모리 맵을 참조하여 실제로 정의된 LPUART_1을 지정하도록 수정하였다. 이를 통해 통신 채널을 명확하게 설정하여 보드를 식별 가능하게 하였다.

4.2. DICE

4.2.1. Micro XRCE-DDS

Micro XRCE-DDS는 자원이 제한되어 있는 장치에서 ROS2와의 통신을 가능하게 하는 핵심 미들웨어다. 이들 통해 클라이언트인 보드와 에이전트가 세션을 통해 신뢰성 있는

데이터 교환을 수행할 수 있다. 해당 세션 관리 과정에 원격 증명 기능을 통합하기 위해 클라이언트 측 소스 코드를 일부 수정하였다.

클라이언트는 에이전트로부터 수신한 메시지를 파싱하는 과정에서 nonce 값을 추출하도록 변경하였다. 해당 과정은 session_info.c의 uxr_read_session_header 함수에서 수행되며 Figure 5과 같이 수신 메시지의 헤더를 파싱할 때 nonce를 읽어 info-> nonce에 저장한다. Nonce는 uint32_t 타입으로 정의되어 있다.

```
uint8_t session_id; uint8_t key[CLIENT_KEY_SIZE];
uxr_deserialize_message_header(ub, &session_id, stream_id_raw, seq_num, key);

////////////////////////// DICE START //////////////////////////
info->nonce = (uint32_t *)ub->init + 1;
////////////////////////// DICE END //////////////////////////

must_be_read = session_id == info->id;
```

[Figure 5] uxr_read_session_header 함수의 소스 코드 일부

클라이언트에서 에이전트로 메시지를 송신하는 과정은 부팅 시점 원격 증명이 통합되어 수행되도록 구현하였다. Figure 6와 같이 해당 과정은 전송 버퍼를 준비하는 중에 Session.c의 uxr_flash_output_streams_with_token 함수에서 수행된다. 4.2.2장의 내용에 기반하여 산출된 펌웨어 해시 값과 수신된 nonce에 XOR 연산을 수행해 새로운 입력 값을 생성하고 이를 4.2.3장에 작성된 DiceMainFlow 함수에 전달하여 DICE 기반의 CDI와 인증 토큰을 생성한다. 생성된 CDI 값을 전송 버퍼에 포함하여 에이전트로 전송하였다. 이 과정을 통해 클라이언트는 신뢰 가능한 부팅 과정을 거쳤음을 증명할 수 있다. 만약 DICE 연산이 정상적으로 완료되지 못할 경우 CDI 버퍼의 값이 전송되지 않도록 설계하여 보안성을 강화하였다.

```
while (uxr_prepare_next_reliable_buffer_to_send(stream, &buffer, &length, &seq_num))
{
    uxr_stamp_session_header(&session->info, id.raw, seq_num, buffer);

    uint8_t hash_result[DICE_CDI_SIZE];
    uint8_t nonce_buffer[DICE_CDI_SIZE];
    if (!hash_firmware(hash_result)) {
        printf("Firmware hash failed.\n");
    }
    for (int i = 0; i < DICE_CDI_SIZE; i++)
    {
        nonce_buffer[i] = ((uint8_t *)(session->info.nonce))[i%4] ^ hash_result[i];
    }
    uint8_t cdi_buffer[DICE_CDI_SIZE] = {0,};
    uint8_t cert_buffer[2048];
    size_t cert_size;
    DiceInputValues input_values = {};
    memcpy(input_values.code_hash, nonce_buffer, DICE_CDI_SIZE);
    DiceResult result;
    result = DiceMainFlow(NULL, cdi_buffer, cdi_buffer,
            &input_values, sizeof(cert_buffer), cert_buffer,
            &cert_size, cdi_buffer, cdi_buffer);

    if (result == kDiceResultOk) {
        for (int i = 0; i < DICE_CDI_SIZE; ++i) {
            buffer[length + i] = cdi_buffer[i];
        }
    }
    send_message(session, buffer, length + DICE_CDI_SIZE);
}
```

[Figure 6] uxr_flash_output_streams_with_token 함수의 소스 코드 일부

에이전트 측 소스 코드도 Figure 7와 같이 일부 수정하였다. 에이전트는 클라이언트에 에이전트가 존재함을 알리기 위해 process_create_client_submessage 함수에서 메시지를 송신한다. 이 과정에서 rand 함수를 통해 생성된 uint32_t 타입의 난수인 nonce 값을 메시지 헤더에 포함하도록 변경하였다.

```
/*
nonce 값 전송 패킷 생성
*/

nonce = std::rand();

const size_t message_size = status_header.getCdrSerializedSize() +
                            status_subheader.getCdrSerializedSize() +
                            status_agent.getCdrSerializedSize() + 4;

OutputPacket<EndPoint> output_packet;
output_packet.destination = input_packet.source;
output_packet.message = std::shared_ptr<OutputMessage>(new OutputMessage(status_header, message_size));
output_packet.message->append_raw_uint32(nonce);
output_packet.message->append_submessage(dds::xrce::STATUS_AGENT, status_agent);

server_.push_output_packet(std::move(output_packet));
```

[Figure 7] process_create_client_submessage 함수의 소스 코드 일부

다음으로 클라이언트가 생성한 토큰을 에이전트 측에서 검증 과정에 활용할 수 있도록 보관하는 과정을 구현하였다. 기존 recv_message 함수 내부에서는 시리얼 통신을 통해

클라이언트로부터 메시지를 수신하여 상위 로직에서 사용할 수 있도록 InputPacket 구조로 변환한다. Figure 8과 같이 클라이언트로부터 수신된 DICE CDI 값을 device_token에 저장하는 절차를 추가하였다.

```
input_packet.message.reset(new InputMessage(buffer_, static_cast<size_t>(bytes_read)));
input_packet.source = SerialEndPoint(remote_addr);

rv = true;
memcpy(device_token, input_packet.message->get_buf() + input_packet.message->get_len()-DICE_CDI_SIZE, DICE_CDI_SIZE);
uint32_t raw_client_key;
```

[Figure 8] recv_message 함수의 소스 코드 일부

마지막으로 저장한 CDI 값을 검증하는 기능에 대한 구현 부분을 추가하였다. 이는 create 함수에서 진행된다. 해당 함수는 클라이언트 측 노드를 시스템 내 Participant 객체로 등록하는 기능을 수행한다. 에이전트는 사전에 등록된 정상 펌웨어의 해시 값을 기반으로 DiceMainFlow 함수를 실행하여 기준 CDI 값을 산출한다. 해당 값과 클라이언트가 전송한 device_token과 비교 검증을 수행하여 일치하지 않을 경우 null 포인터를 반환하여 노드 생성을 거부한다. 반대로 검증에 성공한 경우에는 Participant 객체를 생성하여 시스템에 등록한다. 이 과정은 Figure 9와 같으며 클라이언트가 실제로 신뢰 가능한 펌웨어를 실행 중임을 증명하는 역할을 수행한다.

```
result = DiceMainFlow(/*context=*/NULL, cdi_buffer, cdi_buffer,
                      &input_values, sizeof(cert_buffer), cert_buffer,
                      &cert_size, cdi_buffer, cdi_buffer);
printf("cdi_attest: ");
for (int i = 0; i < DICE_CDI_SIZE; i++)
{
    printf("%02X ", cdi_buffer[i]);
}
printf("\n");

if (result == kDiceResultOk)
{
    printf("DiceMainFlow successfully executed.\n");
    // You can now use cdi_attest and cdi_seal for attestation and sealing.
}
else
{
    printf("DiceMainFlow failed with error code: %d\n", result);
}

printf("device_token: ");
for (int i = 0; i < DICE_CDI_SIZE; i++)
{
    printf("%02X ", device_token[i]);
}
printf("\n");

for (int i = 0; i < DICE_CDI_SIZE; i++)
{
    if(device_token[i] != cdi_buffer[i])
    {
        printf("DICE Remote Attestation Failed\n");
        return (nullptr);
    }
}
```

[Figure 9] create 함수의 소스 코드 일부

이와 같이 Micro XRCE-DDS 계층에서 세션 관리 코드를 확장함으로 단순한 통신 연결을 넘어 부팅 시점의 Remote Attestation 기능을 통합하였다. 이는 기존 Micro XRCE-DDS 프로토콜의 중점인 경량화와 실시간 시스템을 유지하면서 클라이언트와 에이전트 간 통신에 신뢰 기반을 추가한다.

4.2.2. Hashing Firmware

보드에 업로드된 펌웨어 전체를 해시하는 과정을 구현하여 증명 토큰 생성 시 추가적인 무결성 검증 요소로 활용하였다. 이 과정은 펌웨어가 의도하지 않은 변조를 겪지 않았음을 보장하기 위한 단계이다. 이후 DICE 기반 증명 절차에서 CDI 생성의 입력 값으로 활용된다.

펌웨어 해시는 zephyr에 내장된 TinyCrypt 라이브러리에 구현된 SHA-256를 이용해 수행된다. 해시 연산을 위해 hash_firmware 함수를 구현하였고 해당 함수는 Figure 10과 같다. 지정된 시작 주소 (FIRMWARE_START_ADDR) 로부터 펌웨어의 크기 (FIRMWARE_SIZE) 만큼을 순차적으로 읽어 SHA-256 다이제스트를 계산한다. 해시 범위는 hash.h 파일 내 상수 값을 변경하여 조정할 수 있고 특정 영역에 대해서만 선택적으로 무결성 검증을 수행할 수 있다. 본 과제에서는 메모리의 코드 (flash) 영역을 해시하여 사용하였다. 아래의 과정을 통해 생성된 32바이트 크기의 해시 값은 펌웨어 전체의 고유한 무결성 지표로 활용된다.

```
int hash_firmware(uint8_t *output_hash)
{
    if (output_hash == NULL) {
        return 0;
    }

    const uint8_t *firmware_ptr = (const uint8_t *)FIRMWARE_START_ADDR;

    struct tc_sha256_state_struct sha_ctx;
    if (!tc_sha256_init(&sha_ctx)) {
        return 0;
    }

    for (uint32_t offset = 0; offset < FIRMWARE_SIZE; offset += CHUNK_SIZE) {
        size_t len = (FIRMWARE_SIZE - offset > CHUNK_SIZE) ? CHUNK_SIZE : (FIRMWARE_SIZE - offset);
        if (!tc_sha256_update(&sha_ctx, firmware_ptr + offset, len)) {
            return 0;
        }
    }

    if (!tc_sha256_final(output_hash, &sha_ctx)) {
        return 0;
    }
    return 1;
}
```

[Figure 10] hash_firmware 함수의 소스 코드

구현한 hash_firmware 함수는 다음과 같은 단계로 동작한다. 먼저, SHA-256 컨텍스트를 초기화한다. 이후, 지정된 펌웨어 영역을 일정한 블럭 크기 단위로 분할하여 순차적으로 업데이트한다. 펌웨어 영역에 대한 업데이트가 모두 완료되면 최종적으로 32비트 크

기의 다이제스트를 출력한다.

4.2.3. DICE

본 과제에서는 원격 증명 실험을 위해 google이 오픈소스로 공개한 Open-DICE[16] 라이브러리의 핵심 함수들을 zephyr 환경에 통합하였다. Open-dice 아키텍처에서 중심이 되는 함수는 DiceMainFlow로 현재 CDI와 입력값을 이용해 새로운 CDI를 계산하는 역할을 수행한다. 함수 내부에서 주요 구현부는 DiceHash와 DiceKdf 함수를 호출하여 최종 CDI를 산출한다.

Open-dice에서는 DICE[12] 아키텍처를 구현하기 위해 필요한 DiceHash, DiceKdf 등의 함수를 구현하기 위한 템플릿을 제공한다. 또한, mbedTLS, boringSSL과 같이 기존에 존재하는 보안 라이브러리를 활용해 구현된 파일도 함께 제공한다. 이에 본 과제에서는 실험 과정에서 구현의 안전성을 보장하기 위해 zephyr에 모듈로 존재하는 mbedtls 라이브러리를 사용하여 dice 아키텍처를 구현하였다.

Open-dice의 mbedtls_ops..c를 추가하여 연동했고 해당 파일에서 DiceHash 함수는 mbedTLS의 SHA-512 해시 함수를 기반으로 입력값을 해싱하는 과정으로 구현되어 있다.

DiceKdf는 mbedTLS의 HKDF 함수를 이용하여 최종 CDI를 도출하는 과정이다. 이때 HKDF의 Salt 값으로는 DiceHash 함수에서 산출된 해시 결과를 사용하였다. 해당 함수에 동일한 입력이 주어질 경우 동일한 CDI가 생성되며 입력값이 달라질 경우 서로 다른 CDI가 생성된다. 이를 통해 세션마다 유일한 토큰 값을 확보할 수. 있다.

현재 구현된 실험에서는 DiceMainFlow 함수에 UDS를 직접 입력값으로 사용하지 않았으나 필요시 반영할 수 있게 설계하였다. 따라서 향후 실험에서는 UDS를 결합하여 강한 신뢰 증명이 가능하다.

4.2.4. MPU

Micro-ROS 실행 환경에서 펌웨어 업로드 이후 DICE 아키텍처 실행 시 공격을 예방하기 위해 MPU를 활용해 메모리 접근을 제어하도록 하였다.

MPU 설정에는 다음과 같은 레지스터들이 사용된다. MPU_CTRL은 MPU의 전역 활성화 여부를 제어한다. MPU_MAIR은 메모리 영역의 속성을 지정하기 위해 사용된다. MPU_RNR은 현재 설정할 영역의 인덱스를 선택하는데 사용되어 해당 값에 따라 이후 레지스터에서 설정한 값이 해당 영역에 반영된다. MPU_RBAR은 설정할 영역의 시작 주소와 함께 메모리 영역의 권한을 지정하며, MPU_RLAR은 설정할 영역의 마지막 주소와

활성화 여부를 정의한다. 위 레지스터들을 조합하여 각 영역의 주소 범위, 접근 권한, 캐시 속성 등을 설정할 수 있다.

MPU 설정은 MPU_MAIR, MPU_RNR, MPU_RBAR,, MPU_RLAR에 순차적으로 값을 기록하는 방식으로 진행하였다. 해당 과정 전후로 MPU_CTRL를 이용해 전체 MPU 기능을 끄고 키는 것이 필요하다. 위 과정을 거쳐 MPU로 보호하는 메모리 영역은 펌웨어 코드 영역인 0x08000000~0x08080000 이다. 해당 영역을 Privileged Read-Only로 설정하고 Execute를 허용하였다.

MPU 제어를 위해 구현된 라이브러리가 존재하지만 라이브러리르 사용하는 오버헤드를 줄이고자 직접 필요한 레지스터에 접근하여 MPU를 제어하는 방법을 사용하였다. 헤더 파일과 소스 코드를 작성하여 MPU 레지스터의 주소를 지정하여 직접 값을 작성하였다. Figure 11과 같이 MPU 레지스터를 설정하기 위해 각 레지스터의 주소와 비트마스킹을 위한 값, 비트 연산을 위한 함수를 헤더 파일에 매크로로 정의하여 사용하였다. 이후 Figure 12와 같이 해당 매크로를 이용해 위의 내용처럼 레지스터를 순서대로 설정하여 펌웨어 코드 영역에 대해 MPU로 설정하였다. 본 과제에서는 위 과정을 통해 펌웨어의 무결성을 보장하고 런타임 변조를 방지하였다.

```
#define MPU_BASE    0xE000ED90UL

#define MPU_TYPE    (*(volatile uint32_t *)(MPU_BASE + 0x00))
#define MPU_CTRL    (*(volatile uint32_t *)(MPU_BASE + 0x04))
#define MPU_RNR     (*(volatile uint32_t *)(MPU_BASE + 0x08))
#define MPU_RBAR    (*(volatile uint32_t *)(MPU_BASE + 0x0C))
#define MPU_RLAR    (*(volatile uint32_t *)(MPU_BASE + 0x10))
#define MPU_MAIR0   (*(volatile uint32_t *)(MPU_BASE + 0x30))
#define MPU_MAIR1   (*(volatile uint32_t *)(MPU_BASE + 0x34))

// Set Bit
#define SET_BIT(REG, BIT)       ((REG) |= (BIT))
#define CLEAR_BIT(REG, BIT)     ((REG) &= ~(BIT))
#define READ_BIT(REG, BIT)      ((REG) & (BIT))
#define CLEAR_REG(REG)          ((REG) = 0x0)
#define WRITE_REG(REG, VAL)     ((REG) = (VAL))
#define READ_REG(REG)           ((REG))
#define MODIFY_REG(REG, CLEARMASK, SETMASK)  \
    WRITE_REG((REG), (((READ_REG(REG)) & (~(CLEARMASK))) | (SETMASK)))

// CTRL Register
#define MPU_CTRL_ENABLE_Pos     0U
#define MPU_CTRL_ENABLE_Msk     (0x1UL << MPU_CTRL_ENABLE_Pos)
#define MPU_CTRL_HFNMIENA_Pos   1U
#define MPU_CTRL_HFNMIENA_Msk   (0x1UL << MPU_CTRL_HFNMIENA_Pos)
#define MPU_CTRL_PRIVDEFENA_Pos 2U
#define MPU_CTRL_PRIVDEFENA_Msk (0x1UL << MPU_CTRL_PRIVDEFENA_Pos)
```

[Figure 11] mpu_config.h 파일의 코드 일부

```
// Memory Region 0
WRITE_REG(MPU_RNR, 0);

// Flash Base Address
uint32_t rbar_value = 0;
rbar_value |= (0x08000000UL & MPU_RBAR_BASE_Msk);  // 베이스 주소
rbar_value |= (0x0UL << MPU_RBAR_SH_Pos);          // Non-shareable
rbar_value |= (0x1UL << MPU_RBAR_AP_Pos);          // 읽기/쓰기 권한
rbar_value &= ~MPU_RBAR_XN_Msk;                    // 실행 가능
WRITE_REG(MPU_RBAR, rbar_value);
```

[Figure 12] MPU_Config 함수의 소스 코드

4.3. TF-M

4.3.1. Micro XRCE-DDS

클라이언트 측 Micro XRCE-DDS에서 TF-M 기반 원격 증명을 지원하도록 일부 함수를 수정하였다. uxr_read_session_header 함수는 4.2.1장의 DICE 기반 구현과 동일하게 에이전트로부터 전달된 nonce 값을 수신하여 세션 정보 info->nonce에 저장한다.

클라이언트 측에서 4.2.1장과 차이점은 uxr_flash_output_streams_with_token 함수에 있다. 기존 DICE 기반 구현에서는 펌웨어 해시와 nonce를 결합하여 CDI를 산출하고 이를 토큰으로 전송하였다. TF-M 통합 환경에서는 동일한 함수 내에서 Figure 13과 같이 4.3.2장에 작성한 PSA IAT (Initial Attestation Token) 요청 API를 호출하도록 수정하였다. 클라이언트는 TF-M의 보안 기능인 PSA IAT 서비스를 호출하여 증명 토큰을 생성하고 결과를 송신 버퍼에 삽입한 뒤 크기 정보를 반영하여 에이전트로 전송한다.

이를 통해 클라이언트는 TF-M이 보장하는 보안 서비스에 기반하여 원격 증명 토큰을 생성할 수 있다. 해당 과정을 통해 생성된 토큰에는 nonce가 포함되어 통신의 신선성과 부팅 시점의 무결성이 동시에 보장된다.

```
while (uxr_prepare_next_reliable_buffer_to_send(stream, &buffer, &length, &seq_num))
        {
            uxr_stamp_session_header(&session->info, id.raw, seq_num, buffer);
            uint32_t iat_sz = ATT_MAX_TOKEN_SIZE;
            uint8_t iat_buf[ATT_MAX_TOKEN_SIZE] = { 0 };

            /* String format output config. */
            struct sf_hex_tbl_fmt fmt = {
                .ascii = true,
                .addr_label = true,
                .addr = 0
            };
            psa_status_t err = PSA_SUCCESS;
            /* Request the IAT from the initial attestation service. */
            err = att_get_iat(session->info.nonce, 64, iat_buf, &iat_sz);
            if (err == PSA_SUCCESS) {
                for (int i = 0; i < iat_sz; ++i) {
                    buffer[length + i] = iat_buf[i];
                }
                buffer[length + iat_sz] = iat_sz;
                buffer[length + iat_sz + 1] = iat_sz >> 8;
                buffer[length + iat_sz + 2] = iat_sz >> 16;
                buffer[length + iat_sz + 3] = iat_sz >> 24;
            }
            send_message(session, buffer, length + iat_sz + sizeof(uint32_t));
        }
```

[Figure 13] uxr_flash_output_streams_with_token 함수의 소스 코드 일부

에이전트 측 코드 역시 클라이언트와 마찬가지로 nonce 교환 및 토큰 수신 과정은 4.2장의 DICE 기반 구현과 유사하게 진행하였다. Process_create_client_submessage 함수는 메시지 헤더에 무작위 nonce를 포함하도록 수정하였다. 또한, recv_message 함수는 클라이언트가 전송한 PSA IAT를 수신하여 상위 계층에서 처리할 수 있는 InputPacket으로 변환하였다.

Figure 14와 같이 TF-M 통합 환경에서 create 함수는 DICE 구현 시와는 다르게 run_check_iat 함수를 호출하여 PSA IAT의 유효성을 검증하도록 변경하였다. 이를 위해 에이전트는 python 검증 스크립트 실행 인자를 설정하고 해당 인자를 run_check_iat 함수에 전달한다. 이후 내부적으로 python 코드를 호출하여 검증을 진행한다. 검증 결과가 실패할 경우 null 포인터를 반환하여 participant 생성을 거부하며 성공 시에 participant 객체를 반환하여 클라이언트를 시스템에 등록한다.

```
int iat_attest_return = 0;

char *custom_argv[] = {
    "main_executable",
    "-t",
    "PSA-IoT-Profile1-token",
    "-k",
    "iat/tfm_initial_attestation_key.pem",
    "iat/attestation_token.dat"
};
int custom_argc = sizeof(custom_argv) / sizeof(char *);

iat_attest_return = run_check_iat(custom_argc, custom_argv);
```

[Figure 14] create 함수의 소스 코드 일부

4.3.2. TF-M (TrustedFirmware-M)

본 과제에서는 DICE 아키텍처와 더불어 TF-M에서 제공하는 PSA Initial Attestation 서비스[14]를 활용하여 원격 증명 토큰을 생성하였다. 이를 위해 클라이언트 측에는 Figure 15와 같이 att_get_iat 함수를 구현하였다. 해당 함수는 입력으로 전달된 챌린지 (nonce)를 기반으로 TF-M의 attestation API인 psa_initial_attest_get_token 함수를 호출한다. 그 결과 IAT를 생성하고 결과 토큰을 출력 버퍼에 저장한 후 에이전트로 전송한다.

```
/* Request the initial attestation token w/the challenge data. */
// LOG_INF("att: Requesting IAT with %u byte challenge.", ch_sz);
err = psa_initial_attest_get_token(
    ch_buffer,      /* Challenge/nonce input buffer. */
    ch_sz,          /* Challenge size (32, 48 or 64). */
    token_buffer,   /* Token output buffer. */
    token_buf_size,
    token_sz        /* Post exec output token size. */
    );
// LOG_INF("att: IAT data received: %u bytes.", *token_sz);
```

[Figure 15] att_get_iat 함수의 소스 코드 일부

psa_initial_attest_get_token 함수는 TF-M의 Secure world에서 동작하는 API로 attestation 서비스를 호출하는 과정을 포함한다. 함수의 입력은 클라이언트 측에서 수신한 nonce를 포함한 psa_invec 구조체 배열로 전달된다. 출력은 psa_outvec 구조체 배열 통해 토큰 버퍼에 기록된다. 정상적으로 함수가 실행되면 생성된 IAT의 크기가 반환된다. 내부적으로는 psa_call 함수를 이용하여 TFM_ATTESTATION_SERVICE_HANDLE에 TFM_ATTEST_GET_TOKEN 요청을 전달하는 과정을 통해 secure world에서 attestation 프로세스를 수행한다. 해당 함수의 구현은 Figure 16과 같다.

```
psa_status_t
psa_initial_attest_get_token(const uint8_t *auth_challenge,
                             size_t         challenge_size,
                             uint8_t       *token_buf,
                             size_t         token_buf_size,
                             size_t        *token_size)
{
    psa_status_t status;

    psa_invec in_vec[] = {
        {auth_challenge, challenge_size}
    };
    psa_outvec out_vec[] = {
        {token_buf, token_buf_size}
    };

    status = psa_call(TFM_ATTESTATION_SERVICE_HANDLE, TFM_ATTEST_GET_TOKEN,
                      in_vec, IOVEC_LEN(in_vec),
                      out_vec, IOVEC_LEN(out_vec));

    if (status == PSA_SUCCESS) {
        *token_size = out_vec[0].len;
    }

    return status;
}
```

[Figure 16] psa_initial_attest_get_token 함수의 소스 코드

PSA의 Initial Attestation 서비스를 활용해 클라이언트는 4.3.3장에서 설정한대로 TrustZone-M 기반 Secure World에서 실행되는 TF-M의 보안 서비스를 직접 호출할 수 있다. 이를 통해 Non-Secure World에서 동작하는 Micro-ROS 클라이언트는 독립적으로 증명 토큰을 생성하는 대신 보호된 영역에서 신뢰할 수 있는 IAT를 구할 수 있다.

4.3.3. TZ-M (TrustZone-M)

Nucleo-L552ZE-Q 보드는 Armv8-M 아키텍처 기반 Cortex-M33 프로세서를 탑재하고 있으며 TZ-M을 지원한다. 그러나 보드의 기본 zephyr 환경에서 제공되는 그대로 사용하게 되면 TF-M을 통합하기에 메모리 자원이 부족하여 실행이 불가능하다. 이에 따라 TF-M을 정상적으로 구동할 수 있도록 보드의 메모리 맵을 재구성하였다.

```
partitions {
    compatible = "fixed-partitions";
    #address-cells = <1>;
    #size-cells = <1>;

    /*
     * Following flash partition is compatible with requirements
     * given in TFM configuration given for current board:
     * multiple image boot, no tests.
     * It might require adjustment depending on evolutions on TFM.
     */

    boot_partition: partition@0 {
        label = "mcuboot";
        reg = <0x00000000 DT_SIZE_K(80)>;
        read-only;
    };
    /* Secure image primary slot */
    slot0_partition: partition@14000 {
        label = "image-0";
        reg = <0x00014000 DT_SIZE_K(180)>;
    };
    /* Non-secure image primary slot */
    slot0_ns_partition: partition@41000 {
        label = "image-0-nonsecure";
        reg = <0x00041000 DT_SIZE_K(128)>;
    };
    /* Secure image secondary slot */
    slot1_partition: partition@4a000 {
        label = "image-1";
        reg = <0x0004a000 DT_SIZE_K(88)>;
    };
    /* Non-secure image secondary slot */
    slot1_ns_partition: partition@77000 {
        label = "image-1-nonsecure";
        reg = <0x00077000 DT_SIZE_K(36)>;
    };
    /* Applicative Non Volatile Storage */
    /* Not available in this config, use secure storage */
};
```

[Figure 17] nucleo_l552ze_q_stm32l552xx_ns.dts 파일의 일부

zephyr/boards/st/nucleo_l552ze_q 경로에 위치한 nucleo_l552ze_q_stm32l552xx_ns.dts 파일을 Figure 17과 같이 수정하였으며 그에 맞추어 Figure 18과 같이 nucleo_l552ze_q_stm32l552xx_ns.yaml 파일을 부분적으로 변경하여 TF-M 사용 시 Non-

Secure world의 메모리 크기를 재조정하였다.

```
supported:
  - gpio
  - dac
ram: 192
# flash: 36
flash: 128
vendor: st
```

[Figure 18] nucleo_l552ze_q_stm32l552xx_ns.yaml 파일의 일부

또한, Non-Secure world에서 사용할 flash 메모리 크기를 변경하기 위해 modules/tee/tf-m/trusted-firmware-m/platform/ext/target/stm/nucleo_l552ze_q/partition 경로에 있는 flash_layout.h 파일을 Figure 19와 같이 수정하였다. 이때 펌웨어의 크기에 따라 Non-Secure world의 크기를 유동적으로 조절하면서 빌드할 수 있다.

```
#define FLASH_S_PARTITION_SIZE          (0x38000) /* S partition */
#define FLASH_NS_PARTITION_SIZE         (0x2A000) /* NS partition */
#define FLASH_PARTITION_SIZE (FLASH_S_PARTITION_SIZE+FLASH_NS_PARTITION_SIZE)
```

[Figure 19] flash_layout.h 파일의 코드 일부

이와 같은 과정을 통해 Secure 영역과 Non-Secure 영역 간의 경계를 보드 특성에 맞게 조정함으로써 TF-M을 정상적으로 실행할 수 있는 환경을 준비하였다.

TZ-M 기반 실행 흐름은 다음과 같다. Non-Secure world에서 동작하는 애플리케이션은 특정 서비스 호출 시 Config 파일에 정의된 핸들러 함수를 통해 psa_call 함수를 실행한다. psa_call 함수는 Non-Secure world에서 호출되지만 전달된 핸들을 기반으로 Secure world에 존재하는 대응 함수가 실행된다. 실행 결과는 다시 psa_call 함수를 통해 Non-Secure world로 반환된다. 이 과정에서 4.3.2장의 PSA IAT와 같은 보안 서비스는 Secure world에서만 동작하므로 Non-Secure 영역의 애플리케이션은 직접 민감한 자원에 접근할 수 없으며 TZ-M이 제공하는 하드웨어 수준의 격리를 통해 보안이 보장된다. 실제로 사용된 psa_call 함수의 정의는 Figure 20와 같다.

```
psa_status_t psa_call(psa_handle_t handle,
                      int32_t type,
                      const psa_invec *in_vec,
                      size_t in_len,
                      psa_outvec *out_vec,
                      size_t out_len)
{
    if ((type    > PSA_CALL_TYPE_MAX) ||
        (type    < PSA_CALL_TYPE_MIN) ||
        (in_len  > PSA_MAX_IOVEC)     ||
        (out_len > PSA_MAX_IOVEC)) {
        psa_panic();
    }

    return tfm_psa_call_pack(handle, PARAM_PACK(type, in_len, out_len),
                             in_vec, out_vec);
}
```

[Figure 20] psa_call 함수의 소스 코드 일부

5. 연구 결과 분석 및 평가

5.1. 실험 환경

본 연구의 성능 평가를 위한 실험 환경에서 하드웨어는 MPU, TF-M (TrustedFirmware-M), TZ-M (TrustZone-M) 기능을 모두 지원하는 Arm Cortex-M33 프로세서 기반의 ST Nucleo-L552ZE-Q 보드를 사용했으며 보드의 운영체제는 Zephyr RTOS를 기반으로 하며 통신 환경은 Micro-ROS와 XRCE-DDS를 통합하여 구축했다. 에이전트 측 환경은 Intel Core i7-14700K CPU와 16GB 메모리를 탑재한 PC에서 Ubuntu 22.04 운영체제와 ROS 2 Humble 기반의 Micro-ROS 및 XRCE-DDS를 사용했다. 통신은 보드와 PC 간 직렬 UART (115200 bps) 연결을 통해 이루어졌다.

성능 평가는 에이전트 측에서 다음의 세 단계로 나누어 시간을 측정했으며 캐시(cache)의 영향을 최소화하기 위해 각 단계는 30회 반복 측정하여 평균값을 산출했다.

1. Nonce 생성 단계: 에이전트가 Nonce를 생성하여 보드로 전송하기 직전까지
2. Token 수신 단계: Nonce 전송 후 보드에서 생성된 Token을 에이전트가 수신 완료하기까지
3. 검증 및 세션 생성 단계: 수신된 Token을 에이전트가 검증하고 세션을 생성하기까지

MCU 보드 측에서는 캐시가 없으므로 전체 증명 과정에 소요되는 시간을 사이클(cycle) 단위로 10회 측정하였다. 추가적으로 각 보안 아키텍처를 적용한 펌웨어의 전체

크기 (Flash 사용량) 도 함께 비교했다.

추가로 재사용 공격에 대비한 정확성 테스트 시나리오를 설계하여 이전 세션의 토큰을 재전송하거나 RA 토큰 없이 세션을 요청하거나 위조된 해시로 서명된 토큰을 제출하는 경우 Agent가 모두 세션 거부 처리하는지를 확인했다.

5.2. 성능 실험 결과

5.2.1. Nonce 생성 단계

Baseline	DICE	DICE + MPU	TF-M + TZ-M
101.47	119.46	121.88	139.23

[Table 3] Nonce 생성까지 측정 결과 (μs)

Baseline은 별도의 추가 연산이 없어 소요 시간이 가장 짧았다. DICE와 DICE+MPU는 Nonce 생성 로직이 동일하여 유사한 결과가 나타났다. 반면 TF-M+TZ-M 구현은 64Byte 길이의 Nonce를 사용하므로 다른 아키텍처에 비해 Nonce를 생성하는 데 상대적으로 더 많은 시간이 소요되었다.

5.2.2. Token 수신 단계

Baseline	DICE	DICE + MPU	TF-M + TZ-M
9194.67	373031.15	441789.76	551961.94

[Table 4] Token 생성 후 Agent 수신까지 (μs)

Baseline은 토큰 생성 및 검증 절차가 없어 통신 시간이 가장 짧았다. 보안 아키텍처 중에서는 연산이 가장 단순한 DICE가 약 373,031μs(0.37초)로 가장 빨랐다. DICE+MPU의 경우, MPU 관련 SVC (Supervisor Call) 호출로 인한 오버헤드가 추가되어 약 441,789μs(0.44초)가 소요되었다. TF-M+TZ-M 구현은 Secure World와 Non-Secure World 간의 전환, 추가적인 암호화 연산, 그리고 더 긴 토큰 데이터 길이로 인해 약 551,961μs(0.55초)로 가장 긴 시간이 소모되었다.

5.2.3. 검증 및 세션 생성 단계

Baseline	DICE	DICE + MPU	TF-M + TZ-M
5589.86	9500.15	10303.67	242926.20

[Table 5] Token 검증 후 Session 생성까지

검증 및 세션 생성 단계는 Agent에서 검증과 세션을 생성하는 단계이므로 Agent에서 동작하는 부분으로써 5.2.1장, 5.2.2장과 같은 추세로 아무 검증이 존재하지 않는 Baseline

이 5,589μs(0.005초)로 가장 짧았고 DICE와 DICE + MPU는 Agent측은 MPU가 존재하지 않아 같은 연산이므로 9,500μs(0.0095초)와 10,300(0.01초)로 유사한 결과를 얻을 수 있었다. TF-M + TZ-M은 Agent측에서 PSA IAT 검증 코드를 python을 호출하여 검증하도록 구현이 되어있기에 242,926μs(0.24초)로 가장 긴 시간이 소모된 모습을 볼 수 있었다.

5.2.4. Board cycle

	Baseline	DICE	DICE + MPU	TF-M + TZ-M
Cycles	2538704	41791713	45679547	61123655
Overhead	100%	1646%	1799%	2407%

[Table 6] Board cycle 측정 및 Overhead

Table 6은 펌웨어 hashing, Agent와 통신, 검증 Token 발급하는 Board에서 세션 생성 시점까지 cycle을 측정한 것이다. 3개의 비교군중에서 가장 빠른 DICE 구현에서도 검증이 없는 Baseline보다 약 16배의 cycle이 소모되었다. DICE + MPU에서는 약 18배의 cycle 소모, TF-M + TZ-M 구현에서는 약 24배의 cycle이 소모되었다.

5.2.5. Flash size

	Baseline	DICE	DICE + MPU	TF-M + TZ-M
Byte	126596	220296	277924	298372
Overhead	100%	174%	219%	235%

[Table 7] 펌웨어 Flash size 측정 및 Overhead

Table 7은 Board에 flash되는 펌웨어의 크기를 측정한 것이다. DICE는 DICE module 추가로 인해 Baseline 대비 174% 크기로 측정되었고 DICE + MPU는 MPU를 조작하기 위한 라이브러리가 추가되기 때문에 Baseline 대비 219% 크기로 측정되었다. TF-M + TZ-M 구현은 보안 부팅을 위한 bootloader 이미지와 Secure world, Non-Secure world 이미지가 따로 생성되어 flash되기에 가장 큰 235%로 측정되었다.

5.3. 성능 평가

	Baseline	DICE	DICE + MPU	TF-M + TZ-M
측정값 (μs)	14886.00	382650.76	452215.32	795027.38
Overhead	100%	2570%	3037%	5340%

[Table 8] Agent 종합 성능

5.2.1장, 5.2.2장, 5.2.3장의 결과를 통합하면 Table 8과 같다. 종합 성능치를 살펴보면 세 가지 보안 아키텍처 모두 검증 과정으로 인해 Baseline 대비 세션 생성까지 더 많은 시

간이 소요되었다. 하지만 이러한 시간 증가는 매번 일관되게 나타나며 가장 오래 걸리는 TF-M+TZ-M 아키텍처의 경우에도 총 소요 시간이 약 0.79초로 사용자가 인지하기 어려운 수준이며 연결할 때 한번만 실행된다.

5.4. 보안 성능 평가

본 평가는 제안된 세가지 구현이 초기 연구 목표인 'MCU 펌웨어 변조 탐지'와 '재사용 공격 탐지', '위조된 토큰 탐지'에서 얼마나 효과적으로 달성했는지에 초점을 맞춘다. 핵심적인 평가 기준은 펌웨어가 변조되었을 때 원격 증명을 통한 탐지 여부이다.

5.4.1. DICE

DICE 기반 구현은 부팅 과정에서 펌웨어 hash와 고유 식별자 (UDS)를 사용하여 증명 토큰 (CDI)을 생성한다. 해당 과정에서 Agent에서 생성한 Nonce값을 포함하기에 재사용 공격을 방지한다. CDI를 Agent에서 검증하기에 정상 펌웨어에서는 세션 연결을 허용하지만 hash값이 달라지는 변조된 펌웨어에 대해서는 연결을 거부한다.

DICE 기반 구현은 세션 연결 초기에 무결성 검증이므로 메모리 변조와 같은 런타임 공격에 대해서는 방어가 불가능하다는 한계가 존재한다.

5.4.2. DICE + MPU

DICE + MPU는 5.4.1장에서 언급한 메모리 변조와 같은 공격을 MPU의 기능으로 보호하는 기능을 DICE 기반 구현에서 추가한 것이다. MPU를 통해 펌웨어 코드 (Flash)영역에서 write 권한을 제거하기에 코드 변경이 불가능하고 어플리케이션이 unprivileged 권한으로 동작하기에 허용되지 않은 메모리에 접근을 한다면 Fault가 발생하기에 공격에 대한 내성이 생겨 보안성이 한층 강화된다. 이는 TF-M이 탑재되지 않은 MCU 환경에서 선택할 수 있는 방안이다.

5.4.3. TF-M + TZ-M

TF-M + TZ-M 기반 구현은 ARM에서 제공하는 하드웨어 수준의 실행 환경 격리를 통해 세개의 구현중 가장 강력한 보안을 제공한다. 증명 토큰 (PSA IAT) 생성과 관련된 모든 민감한 작업을 어플리케이션이 실행되는 Non-Secure world와 분리하여 Secure world에서 동작한다. 또한 Secure world는 psa_call 함수와 사전에 정의된 handle을 통해서만 접근이 가능하기에 Non-Secure world의 코드가 악성코드에 의해 감염되더라도 증명과정에서 사용되는 증명 토큰과 키와 같은 민감한 자원에는 접근을 할 수 없다. 또한 Secure world에 존재하는 키와 ECDSA 알고리즘을 사용하여 증명 토큰을 생성하기에 중간자 공

격을 통한 토큰 탈취에 대한 내성도 존재한다.

5.4.4. 종합 평가

테스트 결과 재전송 공격, 원격 증명 토큰이 없는 접근, 펌웨어 hash를 위조하여 생성한 토큰 제출 세가지 시나리오에서 세가지 구현 모두 비정상적인 요청을 탐지하여 세션 생성 요청을 거부했다. 하지만 시나리오에서 제시하지 않은 런타임 공격에 대한 탐지는 DICE 기반 구현은 탐지하지 못한다. DICE + MPU는 부분적으로 공격을 탐지하여 Fault를 발생시키고 TF-M+TZ-M은 증명 토큰과 관련된 모든 함수에 접근이 불가능하기에 공격할 수 있는 방법 자체가 존재하지 않는다.

결론적으로 세가지 구현 모두 펌웨어 변조 탐지 목표는 달성했지만 보안수준에서는 차이가 존재한다. Table 9와 같이 DICE 기반 구현은 메모리 보호와 프로세스 격리가 불가능하고 DICE + MPU 기반 구현은 메모리 보호는 가능하지만 프로세스 자체에는 접근이 가능하여 문제가 발생할 잠재적인 가능성이 존재한다. 반면 TF-M+TZ-M 기반 구현은 하드웨어 기반 프로세스 격리까지 지원하기에 Root-Of-Trust라는 목표를 달성하여 공격이 불가능한 안전한 환경이라는 최종적인 목표를 달성한다.

	DICE	DICE+MPU	TF-M+TZ-M
연결 시점 무결성 검증	○	○	○
토큰 생성 메모리 변조 방지	X	○	○
토큰 생성 프로세스 격리	X	X	○

[Table 9] 세가지 구현 보안성 비교

6. 결론 및 향후 연구 방향

6.1. 결론

본 연구는 자원이 제한된 마이크로컨트롤러 (MCU) 환경에서 동작하는 Micro-ROS 기반 로봇 어플리케이션의 보안 취약점을 해결하고자 시작되었다. 이를 위해 Micro XRCE-DDS 통신 프로토콜의 세션 생성 과정에 원격 증명 기술을 통합하는 세 가지 보안 아키텍처 (DICE 단독, DICE+MPU, TF-M+TZ-M)를 성공적으로 설계 및 구현하였다.

성능 평가 결과, 보안 기능 추가로 인해 Baseline 대비 모든 아키텍처에서 시간 및 연산 오버헤드가 발생했다. 특히 하드웨어 격리 환경을 사용하는 TF-M+TZ-M 구조에서 가장 큰 오버헤드 (약 0.79초)가 측정되었으나 이는 TPM을 기반으로 구현한 선행 연구인 DDS-Security+[8] 에서 발생한 지연인 520-710ms 와 유사한 수준으로 강화된 보안 기능

을 고려할 때 합리적인 트레이드오프 (trade-off)이다.

보안성 평가에서는 세 아키텍처 모두 펌웨어 변조, 토큰 재사용 및 위조 공격 시나리오에서 성공적으로 비정상적인 세션 요청을 탐지하고 거부함을 확인했다. 각 아키텍처는 DICE의 부팅 시점 무결성 검증, DICE+MPU의 런타임 코드 변조 방어, TF-M+TZ-M의 하드웨어 기반 프로세스 격리 등 명확히 구분되는 보안 수준을 제공한다.

결론적으로 본 연구는 경량 MCU 환경에서도 원격 증명 기술을 효과적으로 적용할 수 있음을 입증했으며 시스템의 보안 요구사항과 자원 제약에 따라 선택할 수 있는 다각적인 보안 솔루션을 제시한다는 점에서 의의를 가진다.

6.2. 향후 연구 방향

6.2.1. 포괄적인 런타임 보안 강화

본 연구에서 MPU는 펌웨어의 코드 영역을 보호하는 데 중점을 두었으나 시스템 부팅 이후 발생하는 ROP, DOP와 같은 정교한 메모리 공격을 완벽히 방어하기에는 한계가 있다. 향후 연구에서는 MCU 환경에 적용 가능한 경량 제어 흐름 무결성 (Control-Flow Integrity) 기술을 도입하거나 Stack이나 Heap 등 데이터 영역까지 MPU의 보호 범위를 동적으로 확장하여 런타임 보안을 더욱 강화하는 방향으로 나아갈 수 있다.

6.2.2. 다양한 하드웨어 및 RTOS 환경으로의 이식성 검증

본 연구는 Zephyr RTOS와 ST Nucleo-L552ZE-Q 보드를 기반으로 진행되었다. 제안된 보안 아키텍처들의 범용성을 확보하기 위해 FreeRTOS와 같이 로봇 분야에서 널리 사용되는 다른 RTOS나 ARMv7-M 기반의 저사양 MCU 등 다양한 하드웨어 플랫폼으로 이식하고 그 성능을 검증하는 연구가 필요하다. 이를 통해 하드웨어 파편성으로 인한 적용의 어려움을 해소할 수 있을 것이다.

6.2.3. 지속적인 무결성 검증 및 회복 메커니즘 연구

현재 구현은 세션 생성 시점에 시스템의 무결성을 검증한다. 향후에는 시스템 동작 중에도 주기적으로 상태를 검증하는 동적 원격 증명 (Dynamic Remote Attestation) 기술을 도입하여 지속적인 보안 모니터링 체계를 구축할 수 있다. 더 나아가 무결성 훼손이 탐지되었을 때 시스템을 안전한 상태로 복구하거나 관리자에게 경고하는 자동화된 회복 메커니즘에 대한 연구도 필요하다.

6.2.4. 실시간 성능 최적화

TF-M 아키텍처의 에이전트 측 검증 과정은 Python 스크립트 호출 방식으로 구현되어 상대적으로 긴 시간이 소요되었다. 로봇의 실시간성이 중요한 응용 분야를 고려하여 C++ 등으로 검증 로직을 재구현하여 세션 생성 시간을 단축하는 최적화 연구를 진행할 수 있다. 또한 보드 측의 암호화 연산에 하드웨어 가속기를 활용하여 사이클 소모를 줄이는 방안도 고려해볼 수 있다.

7. 구성원별 역할 및 개발 일정

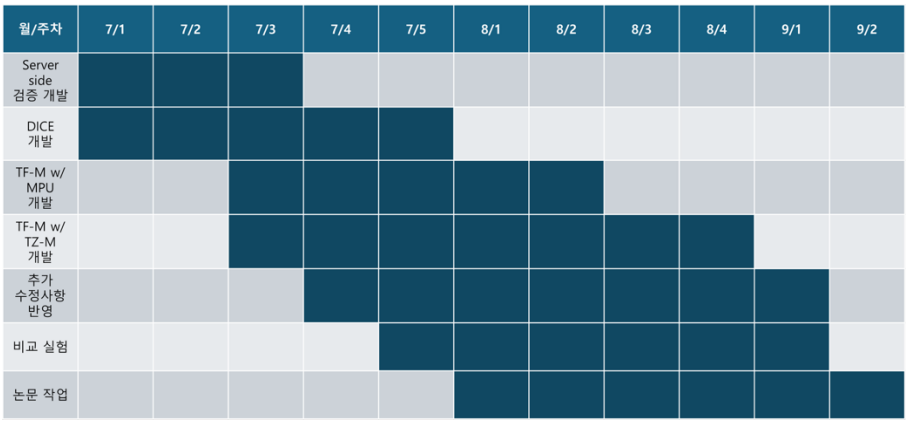
7.1. 구성원별 역할

이름	역할
강승민	- Agent 검증 관련 개발 - Board 동작 검증 - TF-M 기반 구현 개발 - TZ-M 기능 관련 개발 - 성능 실험 구현 및 개발 - 보고서 작성 - 논문 작성 작업
김의준	- 개발 환경 구성 - DICE 코드 이식 - 성능 실험 결과 분석 - 보고서 작성
박재선	- 개발 환경 구성 - DICE 기반 구현 개발 - MPU 기능 관련 개발 - 성능 실험 측정 - 보고서 작성

[Table 10] 구성원별 역할

7.2. 개발 일정

개발 일정은 Figure 21과 같다.



[Figure 21] 개발 일정

8. 참고 문헌

[1] B. Dieber, B. Breiling, S. Taurer, S. Kacianka, S. Rass, and P. Schartner, "Security for the robot operating system," Robotics and Autonomous Systems, Vol. 98, pp. 192-203, Dec. 2017.

[2] TrustedFirmware.org. (2024). TF-M (Trusted Firmware-M) [Online]. Available: https://www.trustedfirmware.org/projects/tf-m/ (downloaded 2025, Sep. 17)

[3] eProsima. (2024). eProsima Micro XRCE-DDS Documentation [Online]. Available: https://micro-xrce-dds.docs.eprosima.com/ (downloaded 2025, Sep. 17)

[4] Grand View Research. (2024, Mar.). Microcontroller Market Size, Share & Trends Analysis Report By Product (8-bit, 16-bit, 32-bit), By Application (Automotive, Consumer Electronics, Industrial), By Region, And Segment Forecasts, 2024 - 2030 [Online]. Available: https://www.grandviewresearch.com/industry-analysis/microcontroller-market (downloaded 2025, Sep. 17)

[5] Open Robotics. (2025). ROS 2 Documentation [Online]. Available: https://docs.ros.org/ (downloaded 2025, Sep. 17)

[6] The micro-ROS Consortium. (2024). micro-ROS [Online]. Available: https://micro.ros.org/ (downloaded 2025, Sep. 17)

[7] BRASSER, Ferdinand, et al. TyTAN: Tiny trust anchor for tiny devices. In: Proceedings of the 52nd annual design automation conference. 2015. p. 1-6.

[8] WAGNER, Paul Georg; BIRNSTILL, Pascal; BEYERER, Jürgen. Dds security+:

Enhancing the data distribution service with tpm-based remote attestation. In: Proceedings of the 19th International Conference on Availability, Reliability and Security. 2024. p. 1-11.

[9] ELDEFRAWY, Karim, et al. Smart: secure and minimal architecture for (establishing dynamic) root of trust. In: Ndss. 2012. p. 1-15.

[10] The Zephyr Project. (2025). The Zephyr Project [Online]. Available: https://www.zephyrproject.org/ (downloaded 2025, Sep. 17)

[11] Amazon Web Services, Inc. (2025). FreeRTOS Real Time Kernel (RTOS) [Online]. Available: https://www.freertos.org/ (downloaded 2025, Sep. 17)

[12] Trusted Computing Group (TCG), "Device Identifier Composition Engine (DICE) Architectures," Revision 1.0, Aug. 2017.

[13] Arm Limited. (2022, Feb. 24). Cortex-M33 Processor Technical Reference Manual [Online]. Available: https://developer.arm.com/documentation/100230/ (downloaded 2025, Sep. 17)

[14] ARM Ltd., "Platform Security Architecture (PSA) – Initial Attestation API 1.0," ARM Developer Resources, 2019.

[15] STMicroelectronics. (2025). NUCLEO-L552ZE-Q, STM32 Nucleo-144 board with STM32L552ZE-Q MCU [Online]. Available: https://www.st.com/en/evaluation-tools/nucleo-l552ze-q.html (downloaded 2025, Sep. 17)

[16] Google. (2021). OpenDICE GitHub Repository [Online]. Available: https://github.com/google/opendice (downloaded 2025, Sep. 17)