공학석사 학위논문

ARM TrustZone을 위한 코드 자동 분리 기술

2017 년 2 월

서울대학교 대학원

전기정보공학부

서 기 수

국문초록

인터넷 뱅킹이나 사용자 인증과 같은 애플리케이션의 보안성을 강화하기 위하여 신뢰된 실행 환경(TEE)으로 민감한 데이터를 분리하는 방법이 제안되고 있다. TEE는 분리된 제한적인 실행환경을 제공하여 중요한 연산 과정을 공격받지 않도록 할 수 있다. 본 논문은 ARM TrustZone을 기반으로 구축된 TEE를 활용할 수 있도록 코드를 자동분리하기 위해 필요한 정적 분석기와 인터페이스 자동 삽입기의 구현에 대해 논한다.

실험결과 C언어로 작성된 코드에서 정적 분석기를 통해 민감한 데이터를 저장한 메모리 블록이 사용되는 함수들을 추출할 수 있었고, 이렇게 추출된 함수들에 대해 GlobalPlatform의 표준을 따르는 인터페이스를 자동으로 삽입하여 분리된 코드를 자동으로 작성할 수 있었다.

주요어 : TEE; ARM TrustZone; OPTEE; data isolation; partitioning
학 번 : 2015-20932

목 차

표 목 차

그 림 목 차

제 1 장 개요

최근 스마트폰과 같은 개인용 단말의 발달에 따라 금융 업무 등과 같이 인터넷 뱅킹 애플리케이션, 생체정보를 이용한 사용자 인증 애플리케이션과 같이 민감한 정보를 다루는 애플리케이션이 증가하고 있다. 이러한 애플리케이션은 데이터 암호화, 암호화된 통신 등을 사용해서 보안성을 보장한다. 그러나 암호화되기 이전의 민감한 데이터, 암호화 과정 등은 일반적인 실행환경에 노출되어 있으므로 공격에 대한 위험성을 지니고 있다.

이러한 민감한 데이터와 연산과정에 대한 공격을 막는 방법 중 하나로 신뢰된 실행 환경(Trusted Execution Environment, TEE)의 사용이 있다. TEE란 일반적인 실행환경과 독립된 실행환경으로, 프로세서와 운영체제 등의 지원을 통해서 프로그램의 안전한 수행을 보장하는 실행 환경을 의미한다[1]. TEE는 프로그램 수행에 대한 무결성, 기밀성 등을 보장함으로써 안전한 프로그램 수행을 제공한다. 따라서 민감한 데이터, 암호화 과정 등이 TEE에서 실행된다면 일반적인 실행환경에서 이루어지는 직접적인 접근을 방지하여 공격에 대비할 수 있다.

이러한 TEE 중 하나로 ARM사의 TrustZone 기술이 있다. ARM TrustZone은 하드웨어적으로 분리된 실행환경을 제공하여 TEE의 자원과 코드, 데이터를 보호한다. 현재 ARM TrustZone은 스마트폰과 같은 고성능 단말기의 애플리케이션 프로세서(Application Processor, AP)에서 지원되고 있으므로 이를 이용한다면 실생활에서 사용되는 애플리케이션의 보안성을 강화할 수 있다.

그러나 ARM TrustZone의 secure world는 보안성을 유지하기 위해 일반적인 실행환경에 비해 한정된 가용자원을 지니고 있다. 따라서 전체 애플리케이션이 secure world에서 동작하게 되면 성능적인 문제가 발생할 수 있고, 심지어는 애플리케이션이 동작하지 않을 수도 있다. 그러므로 ARM TrustZone에서 실행될 애플리케이션 개발자는 애플리케이션을

normal world에서 동작할 부분과 secure world에서 동작할 부분으로 나눠서 설계해야 한다.

이를 위해서는 개발자에게 알고리즘을 분석하여 민감한 데이터와 민감한 연산과정을 분리하는 것이 요구된다. 그러나 개발자가 애플리케이션 분리에 대한 최적화를 스스로 수행하는 것은 큰 부담이 된다. 또한 분리된 애플리케이션이 서로 실행되는 환경이 다르므로, 서로 통신하기 위한 인터페이스가 필요하다. 그리고 secure world에서 보안성을 유지하기 위해 신뢰된 저장 공간(Trusted Storage), 신뢰된 입출력 장치(Trusted I/O) 등의 가용 자원을 사용하는 방법에 대해서도 개발자가 숙지할 필요성이 있다.

위와 같은 부담을 덜기 위하여 컴파일러 기술을 이용한 코드 분석 및 애플리케이션 분리, 분리된 애플리케이션 간 인터페이스 구현과 같은 부분을 자동화를 제공한다면 개발자는 친숙한 기존 언어로 normal world에서 수행될 애플리케이션을 개발하는 것과 흡사한 과정으로 보안성이 강화된 애플리케이션을 개발할 수 있다. 여기서 개발자가 추가해야 하는 것은 민감한 데이터 또는 민감한 연산 과정에 대한 메타데이터로, 코드가 자동으로 분리될 수 있는 단서를 제공하는 것이다.

또한 코드 분리 자동화가 가능하다면 기존 코드에 대한 개발자의 최소한의 수정으로 코드 분리가 가능하므로 secure world를 이용할 수 있게 되어 기존 애플리케이션의 보안성 강화도 가능하다.

본 논문에서 제안하는 ARM TrustZone을 위한 코드 자동 분리 기술에는 개발자가 제공한 메타데이터를 이용한 민감한 데이터 및 연산 추출, 분리된 애플리케이션 간 통신에 필요한 데이터 직렬화, 공유 메모리 설정, 인터페이스 및 secure world 자원 활용 코드 삽입 등의 여러 기술이 필요하다. 본 논문에서는 그 중 메타데이터를 이용한 민감한 데이터 및 연산 추출과 인터페이스 코드 삽입 방법에 대해 서술한다.

제 2 장 배경

제 1 절 신뢰된 실행 환경

신뢰된 실행 환경(TEE)이란 모바일 단말기에서의 보안 관련 애플리케이션의 안전환 신뢰 기반 실행 환경을 제공하는 보안 하드웨어 및 소프트웨어 기능을 정의한 것이다. 스마트카드 관련 표준화 단체인 GlobalPlatform[2]에서 이와 관련된 아키텍처 및 애플리케이션 프로그래밍 인터페이스(Application Programming Interface, API) 표준을 제정하고 있다.

사용자 공간을 포함해 외부에 노출될 수 있는 단말기의 실행환경을 일반적인 실행 환경 또는 부유한 실행 환경(Rich Execution Environment, REE)라 하자. REE는 다양한 애플리케이션 실행과 관리를 위해 많은 기능을 제공하는 운영체제(Rich Operating System, Rich OS)를 제공한다. 하지만 그와 동시에 외부 또는 내부적으로 단말기의 기능과 메모리에 접근할 수 있는 경로가 다양해지므로 보안 취약성이 드러날 수 있다.

반면 TEE는 단말기의 보안 기능을 보호하고 신뢰된 코드만을 실행하는 공간으로, REE와 분리된 독립된 공간이다[3]. TEE는 다음과 같은 특성을 지니도록 설계 되어야 한다.

1. 모든 알려진 원격공격과 소프트웨어 공격에 대해 저항성을 지닌다.
2. 외부의 하드웨어 공격에 대해 저항성을 지닌다.
3. TEE 내부에서 실행되는 코드는 신뢰된 코드만 실행된다.
4. TEE의 자원과 TEE 내부에서 실행되는 코드는 허가되지 않은 추적과 디버깅을 통한 허가되지 않은 제어에서 보호된다.

REE와 TEE에 대한 아키텍처는 그림 1과 같은 형태로 표현될 수 있다.

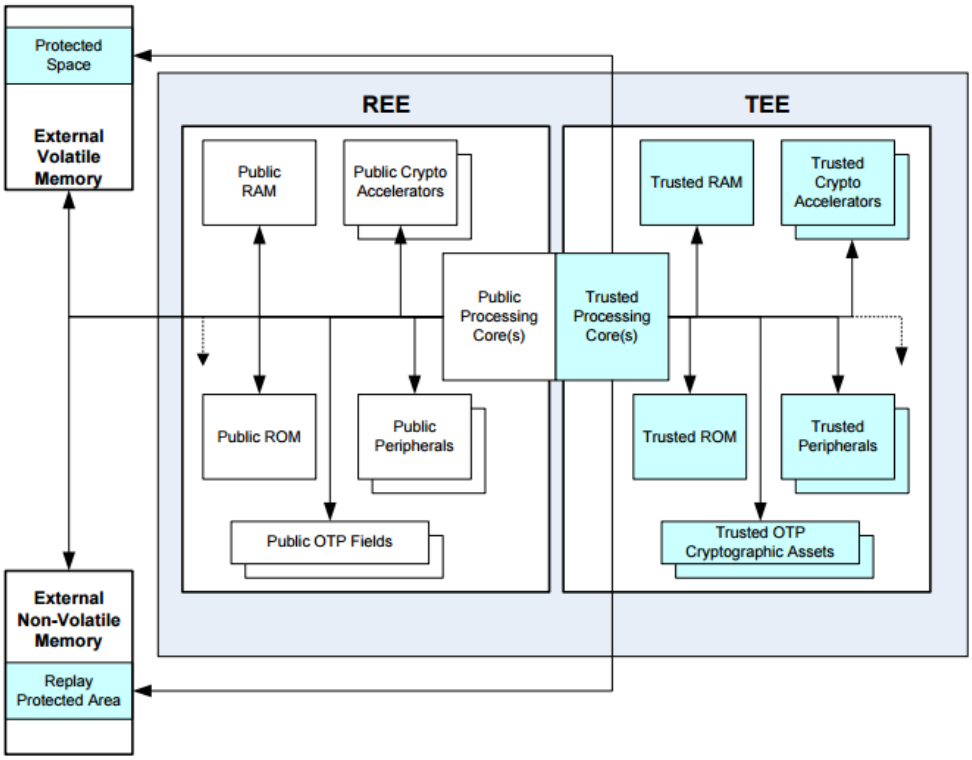

그림 1 TEE system architecture

실제 구현은 완전히 분리된 칩으로 이루어진 REE와 TEE로 될 수도 있고, 하나의 칩에 구현된 시스템의 일부로 TEE를 구현 할 수도 있다. 중요한 것은 앞에 서술된 TEE의 특성을 유지하여 TEE 내부의 코드와 데이터를 보호하도록 설계하는 것이다.

제 2 절 ARM TrustZone

ARM 사의 TrustZone이란 시스템 온 칩(System-on-Chip, SoC)과 중앙처리장치를 통해 보안성을 확보하기 위한 하드웨어 기반의 보안 해결책이다. TrustZone 기술이 적용된 ARM 사의 프로세서 코어로는 Cortex-A 군과 Cortex-M23, Cortex-M33 등이 있다. 특히 Cortex-A 군은 스마트폰과 같은 고성능 단말기에 사용된 것을 목표로 설계된 프로세서로, 일상적으로 자주 사용하는 단말기의 보안성을 증대시킬 수 있다.

TrustZone은 하드웨어적으로 분리된 REE와 TEE로 구성되어 있다. 각 실행환경은 normal world와 secure world로 지칭된다. Normal world에서 실행되는 코드 및 normal world의 자원은 secure world의 자원에 직접적으로 접근할 수 없다. 프로세서 내부에서, 모든 애플리케이션은 normal world 또는 secure world 둘 중 하나에서만 실행되도록 되어 있다. 즉 한 애플리케이션이 normal world에서 실행되다 모든 프로그램 환경이 보존된 채로 secure world에서 실행될 수 없다. 프로세서의 normal world와 secure world 교차는 별도의 보안 감시자를 통해서만 이루어질 수 있다.

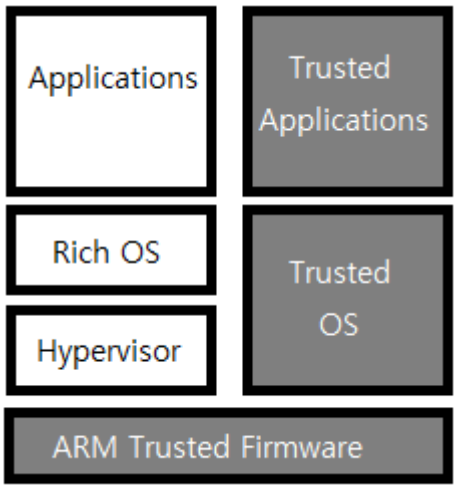

그림 2 ARM TrustZone concept

TrustZone에서 REE와 TEE를 구성하기 위해 ARM 사에서는 부트로더를 포함한 ARM Trusted Firmware를 제공한다. ARM Trust Firmware는 normal world와 secure world의 교차를 수행하도록 보안 감시자 코드를 실행한다.

제 3 절 OP-TEE

OP-TEE(Open Portable Trusted Execution Environment)는 ARM 기반의 리눅스 오픈 소스를 지향하는 비영리 단체인 Linaro에서 배포하는 ARM TrustZone 기술이 적용된 오픈 소스 TEE이다[5]. OP-TEE는 GlobalPlatform의 TEE 표준 및 REE와 TEE간 통신에 필요한 API 표준, TEE의 자원 이용을 위한 API 표준을 따르도록 설계되어 있다.

OP-TEE는 단말기 부팅 과정에서 trusted OS를 포함한 TEE를 먼저 구성하고 다양한 애플리케이션 실행을 지원하도록 rich OS를 포함한 REE를 구성하도록 되어 있다. 따라서 REE가 TEE 구성에 영향을 끼치지 못하도록 설계되어 있다.

OP-TEE의 구성은 OP-TEE Client, OP-TEE Linux Kernel device driver, OP-TEE Trusted OS로 이루어져 있다. OP-TEE Client는 normal world의 사용자 공간에서 실행되는 API로, normal world에서 실행되는 애플리케이션의 코드에 포함되어 secure world에서 실행되는 애플리케이션과 연결해 통신할 수 있도록 한다. OP-TEE Linux Kernel device driver는 normal world의 사용자 공간과 secure world의 통신을 제어하는 디바이스 드라이버다. 마지막으로 OP-TEE Trusted OS는 secure world에서 실행되는 운영체제로, secure world의 애플리케이션 관리 및 internal API를 통한 TEE 자원 사용에 관여한다.

본 논문은 ARM TrustZone 기술이 적용된 OP-TEE를 대상으로 한 코드 자동 분리 기술에 대해 서술한다.

제 4 절 ARM TrustZone의 소프트웨어 아키텍처

ARM TrustZone 기술이 적용된 OP-TEE는 GlobalPlatform에서 제안한 소프트웨어 아키텍처를 따른다. 애플리케이션은 normal world에서 동작하는 애플리케이션(Client Application, CA)와 secure world에서 동작하는 애플리케이션(Trusted Application, TA)로 구분된다[6].

CA와 TA는 서로 독립된 실행환경에서 실행되므로 서로 직접적인 연결이 이루어지지 않는다. 각 TA는 ID를 갖고 있기 때문에, 이 ID에 대한 정보를 갖고 있는 CA 혹은 TA만 client/internal API를 통해 아래 그림처럼 특정 TA에 서비스를 요청할 수 있다.

CA와 TA간의 연결은 각 해당 client API를 호출함으로써 이루어진다.

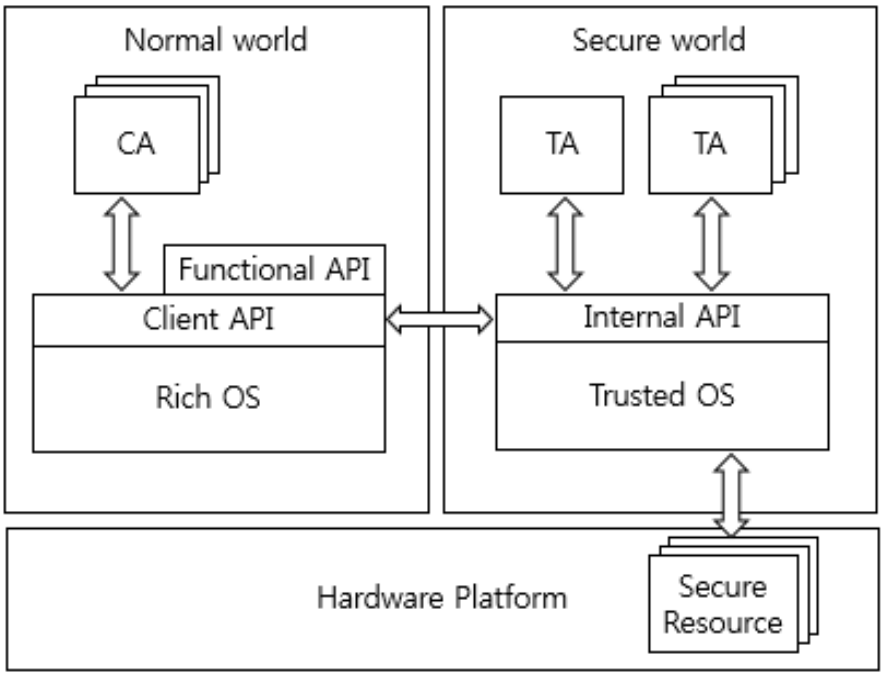

그림 3 ARM TrustZone software architecture

CA는 TA와 연결되기 전에 먼저 secure world와 연결되어야 한다. 이 CA와 secure world 간의 논리적인 연결을 context라 한다. CA는 TEEC_InitializeContext API 호출함으로써 context를 생성하고 TEEC_FInalizeContext API를 호출함으로써 context를 소멸시킨다. 만약 TEE 또는 secure world가 여럿이라면 한 CA가 서로 다른 context를 지니는 것이 가능하다.

CA와 secure world 간 context가 생성되면 CA와 TA가 연결되어야 한다. 이 연결을 session이라 한다. 이 때 요구되는 TA의 ID 정보를 UUID(Universally Unique Resource Identifier)라 한다. Session은 CA측에서 TEEC_OpenSession API를 호출하여 생성되고 TEEC_CloseSession API를 호출하여 소멸된다. Context가 생성되어 있다면 한 CA가 하나 혹은 다수의 TA에 대하여 여러 session을 생성하는 것이 가능하다.

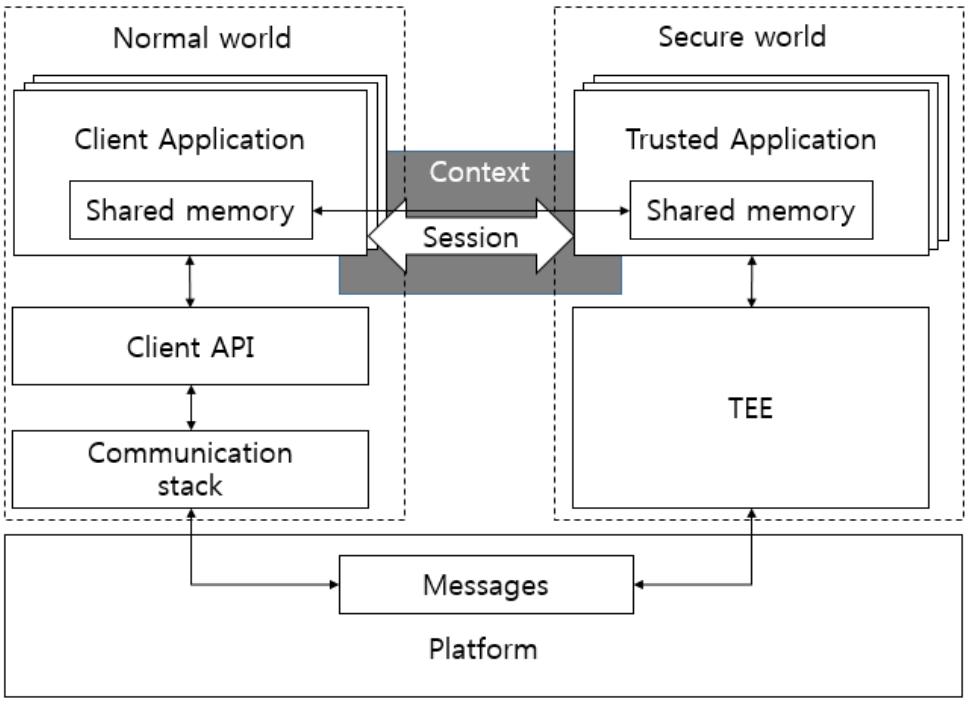

그림 4 Connection between CA and TA

그러나 각 context와 session간 공유 메모리와 같은 실행 환경의 공유는 불가능하다.

CA가 TA로 서비스를 요청할 때 수반되는 데이터 전송은 두 애플리케이션이 공유하는 공유 메모리를 사용한다. 공유 메모리는 메모리 블록에 대한 포인터와 메모리 블록 크기 또는 변수로 지정될 수 있다. 이 때 공유 메모리에 대한 데이터 전달 방향성을 지정함으로써 CA가 TA에게 혹은 TA가 CA에게 영향을 끼치는 것을 제어할 수 있다.

CA는 TEEC_InvokeCommand라는 API를 호출함으로써 TA에게 특정 서비스를 요청할 수 있다. 이 때 operation 이라는 데이터 구조를 통해 공유 메모리로 전달될 데이터를 관리할 수 있다. 또 CA는 TA의 특정 서비스를 요청하기 위하여 TA의 각 함수에 대한 ID 또는 TA가 제공하는 각 서비스에 대한 ID를 알고 있어야 한다.

표 1은 client API의 CA-TA간 통신에 필요한 데이터 타입을 정리한 것이다. 각 데이터는 client API의 매개 변수로 필요하며, CA와 TA를 연결하고 TA 서비스 요청에 필요한 데이터를 전달하는데 사용된다. 표 2는 CA와 TA의 통신에 필수적으로 사용되어야 하는 client API 함수를 정리한 것이다.

CA에서 secure world와 통신하기 위한 client API를 호출하게 되면, 이와 쌍을 이루는 secure world의 internal API들이 호출된다. Internal API는 CA와 TA의 통신에만 관여하는 것이 아니라 TA와 TA의 통신, TA와 secure resource의 통신에도 관여한다. 표 3과 표 4는 CA와 TA의 통신에 관여하는 데이터 타입 및 API를 정리한 것이다.

TEEC_UUID
RFC4122[7]에 따른 UUID 특정 TA 확인에 필요한 정보.
TEEC_Context
CA와 TEE의 논리적 연결 정보 관리
TEEC_Session
CA와 TA의 연결 정보 관리
TEEC_SharedMemory
공유 메모리를 관리하기 위한 데이터 구조로 CA의 메모리 블록을 등록하여 재사용성을 높임
TEEC_TempMemoryReference
TA 서비스 요청에 따라 일시적으로 생성되는 공유 메모리
TEEC_RegisteredMemoryReference
TEEC_SharedMemory 중 일부를 TA 서비스 요청에 사용하기 위한 데이터 구조
TEEC_Value
TA 서비스 요청에 사용할 수 있는 32-bit unsigned integer
TEEC_Parameter
공유 메모리로 TA에 전달하는 매개변수를 관리하기 위한 데이터 구조
TEEC_Operation
TEEC_Parameter와 데이터 전달 방향성을 관리하기 위한 데이터 구조
TEEC_Result
Client API의 실행 결과에 대한 에러 코드

표 1 Client API data types

TEEC_Result TEEC_InitializeContext
TEE와 CA의 논리적인 연결인 context를 생성
void TEEC_FinalizeContext
Context에 저장된 논리적 연결을 해지
TEEC_Result TEEC_RegisterSharedMemory
해당 context의 scope 안에서 CA의 메모리 블록을 공유 메모리에 등록 등록된 메모리 블록이 TA 서비스 요청에 사용될 경우 TempSharedMemory 생성을 생략
TEEC_Result TEEC_AllocateSharedMemory
해당 context의 scope 안에서 새로운 메모리 블록을 공유 메모리에 할당 할당된 메모리 블록이 TA 서비스 요청에 사용될 경우 TempSharedMemory 생성을 생략
void TEEC_ReleaseSharedMemory
해당 메모리 블록을 공유 메모리에서 배제
TEEC_Result TEEC_OpenSession
UUID로 특정된 TA와 CA를 연결하여 session 생성
void TEEC_CloseSession
Session에 저장된 CA와 TA의 연결을 해지
TEEC_Result TEEC_InvokeCommand
Session으로 연결된 TA의 함수 또는 서비스 ID로 서비스 요청
void TEEC_RequestCancellation
Session 생성 또는 TA 서비스 중지 요청
uint32_t TEEC_PARAM_TYPES
TEEC_Operation의 매개변수 방향성 설정

표 2 Client API functions

TEE_UUID
TEEC_UUID와 상동
TEE_Param
공유 메모리로 TA에 전달된 매개변수를 관리하기 위한 데이터 구조 공유 메모리의 블록에 대한 포인터와 블록 크기 또는 32-bit unsigned integer로 구성됨
TEE_Result
TEEC_Result와 상동

표 3 Internal API data types

TEE_Result TA_CreateEntryPoint
CA와 TA의 연결이 최초로 생성될 때 실행됨
void TA_DestroyEntryPoint
CA와 TA의 연결이 완전히 종료될 때 실행됨
TEE_Result TA_OpenSessionEntryPoint
CA와 TA의 연결을 위해 TEEC_OpenSession과 짝을 이루어 실행됨
void TA_CloseSessionEntryPoint
CA와 TA의 연결 해지를 위해 TEEC_CloseSession과 짝을 이루어 실행됨
TEE_Result TA_InvokeCommandEntryPoint
TA의 함수 또는 서비스 ID에 따라 해당 서비스를 제공하기 위해 TEEC_InvokeCommand와 짝을 이루어 실행됨

표 4 TA interface functions of internal API

제 5 절 Data Structure Analysis

본 논문에서 구현한 정적 분석기의 alias analysis는 data structure analysis를 바탕으로 하고 있다. Data structure analysis는 프로그램 상의 포인터가 어떤 메모리 블록을 가리키는지 분석하는 points-to analysis이다. 이 data structure analysis의 결과물은 코드의 각 함수별로 생성된 DSGraph다.

그림 5는 DSGraph의 일부를 간략하게 나타낸 것이다. 현재 두 메모리 블록이 각각 stack과 heap 영역에 있다. 이 때 stack 영역의 배열을 나타내는 노드가 그림의 왼쪽 사각형이고 heap 영역의 메모리 블록을 나타내는 노드가 오른쪽 사각형이다. 타원형 노드들은 LLVM IR(Intermediate Representation)에 나타난 변수들이다. 그림의 %buff1과 %buff2가 같은 stack 영역의 메모리 블록을 가리키고 있다. 또한 load 명령어가 %buff1의 주소에서 포인터 값을 읽어오기 때문에 이 load 명령어의 결과물은 %buff1과 같은 메모리 블록을 가리키게 된다. 이 세 변수들은 같은 메모리 블록을 가리키기 때문에 alias 관계이다. 반면 %buff3는 heap 영역의 메모리 블록을 가리키기 때문에 앞의 세 변수와 alias가 아니다.

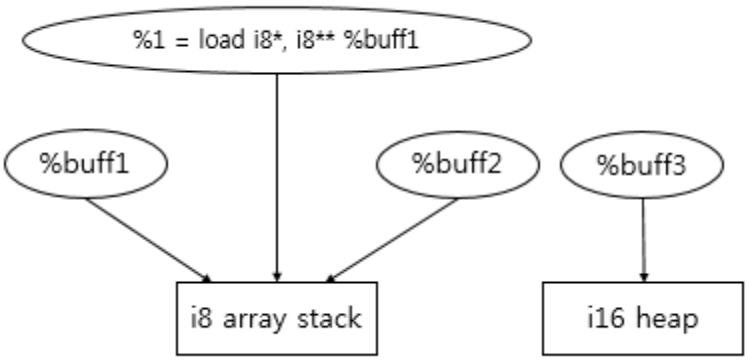

그림 5 DSGraph

DSGraph는 변수를 가리키는 노드와 메모리 블록을 의미하는 노드 외에도 call 노드와 return 노드를 가질 수 있다. Call 노드는 분석하는 함수 내에서 다른 함수를 호출할 때, 매개변수로 포인터 타입 변수를 사용하는 경우 그 포인터 타입 변수가 가리키는 메모리 블록을 나타낸다. Return 노드는 분석하는 함수가 포인터 타입 변수를 반환할 때 그 포인터 타입 변수가 가리키는 메모리 블록을 나타낸다. 이 두 노드는 interprocedural 분석을 할 때 사용될 수 있다.

Data structure analysis의 interprocedural 분석은 각 함수에 대해 생성된 DSGraph들을 call graph를 bottom-up으로 방문하며 callee의 DSGraph 정보를 caller의 DSGraph에 통합하고, 다시 top-down으로 call graph를 방문하며 caller의 DSGraph 정보를 callee의 DSGraph에 통합한다. 이 과정에서 DSGraph의 메모리 블록을 의미하는 노드간 병합이 일어날 수 있으며, 병합이 일어난 노드들은 해당 메모리 블록을 가리키는 포인터 변수들이 runtime에 alias가 될 수 있음을 나타낸다.

제 3 장 코드 자동 분리 기술

본 장에서는 C언어로 작성된 코드의 자동 분리 기술을 위해 필요한 정적 분석인 alias analysis와 taint analysis의 구현과 CA와 TA에 각각 인터페이스를 자동으로 삽입하기 위한 구현에 대해 서술한다. 본 논문에서 제안하는 코드 자동 분리 기술 중 정적 분석을 담당하는 부분은 LLVM[8]을 기반으로, 인터페이스 자동 삽입은 Clang[9]을 기반으로 구현되었다.

제 1 절 코드 분리 과정

본 절에서는 ARM TrustZone을 위한 코드 자동 분리 기술 과정의 전체적인 순서에 대해 개략적으로 서술한다. OpenSSL과 같은 보안성이 요구되는 코드에서, 일반적으로 민감한 데이터는 프로그램 외부의 네트워크나 파일 등을 통해 현재 프로그램이 실행되고 있는 메모리에 로드된다. 따라서 개발자는 민감한 데이터를 저장해야 하는 메모리 블록에 대한 포인터 또는 메모리 블록 자체에 메타데이터를 제공하여 민감한 데이터가 secure world로 격리될 수 있는 단서를 제공해야 한다.

그림 5의 코드에서 func1은 fgets 함수를 통하여 민감한 정보를 buff가 가리키는 메모리 블록에 저장한다. 이 메모리 블록에 대한 포인터 값은 func3에 의하여 func4에 반환되고, 이후 func2에 의하여 민감한 데이터를 저장한 메모리 블록이 dereference 된다.

이 소스 코드는 애플리케이션을 구성하는 다른 소스 코드와 함께 IR으로 변환되어 object file linking과 흡사하게 IR linking 과정을 거쳐 runtime 분석과 흡사한 분석을 할 수 있도록 그림 6처럼 변환된다.

```
1 void func1( int* buff, int buffsize,
              int p1, int p2 ) {
   // get a template and put it into buff
   // through fgets()
2   ..... }

3 int func2( int* inbuff1, int* inbuff2,
     int inbuff1size, int inbuff2size) {
  // match the template stored in inbuff1
  // and the input stored in inbuff2
  // if they are matched, return 1, else, return 0
4   ..... }

5 int* func3( int* inbuff, int inbuffsize ) {
6   int* buff;
7   int buffsize;
8   .....
9   buff = (int*)malloc(sizeof(int)*buffsize));
10   if( buff ) {
       func1( buff, buffsize, p1, p2 )
11     return buff; }
14   else return NULL;
15  return NULL; }

16 int func4( ... ) {
17  .....
18  int* tmplt = func3( inbuff, inbuffsize );
19  .....
20  result = func2(tmplt, sth, tmpltsize, sthsize);
21  ..... }
```

그림 6 Example code for auto partitioning

IR linking이 완료되면 fgets의 첫 번째 매개변수로 사용되는 포인터에 대해 정적분석을 하여 민감한 데이터가 저장되는 메모리 블록이 dereference 되는 함수들을 찾아 secure function을 구별할 수 있는 단서를 추출한다. 이때 해당 메모리 블록에 담긴 데이터의 propagation에 대한 추적과 포인터의 alias 분석이 필요하다. 즉 포인터에 의해 민감한 데이터를 담고 있는 메모리 블록이 dereference 되는 함수들을 모두 secure function으로 구별함으로써 직접적인 접근을 막도록 한다. 그림 6에서 %buff의 propagation에 의해 %buff가 가리키는 메모리 블록이 dereference 되는 func1, func2, func3이 secure function이 된다.

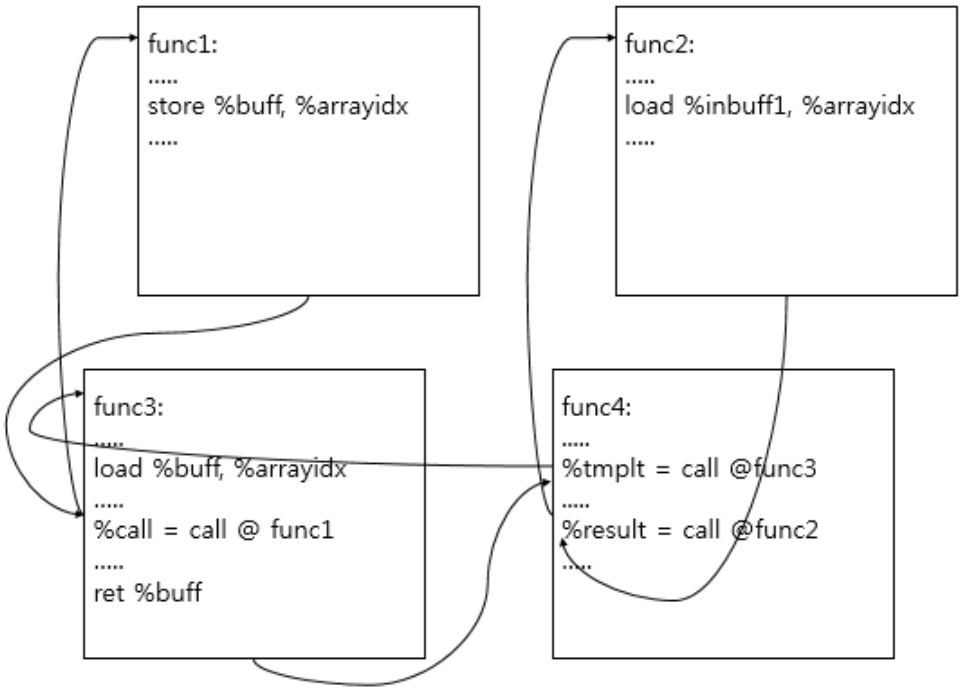

그림 7 Linked IRs

분류된 secure function 중 CA에 의해서 호출되어야 하는 함수들을 secure function entry라 하자. 그림 5의 함수 중 func2와 func3가 secure function entry가 되므로 이 두 함수의 caller인 func4가 CA에서 TA로 서비스를 요청하는 코드가 되어야 한다. 따라서 TA는 func1, func2, func3을 포함해야 하고 CA는 이외의 코드를 포함해야 한다.

Secure function entry 추출과 코드 분리가 끝나면 인터페이스 삽입을 위해 CA와 TA의 코드를 재구성한다. CA의 코드 재구성은 secure function entry caller 코드 재구성은 context 및 session 관련 코드 삽입, secure function entry의 매개변수 분석 및 wrapping,, secure function entry 호출 코드 치환으로 이루어진다.

TA 코드 재구성은 각 secure function entry를 호출하여 CA에게 서비스를 제공하기 위한 InvokeCommandEntryPoint API와 session 생성/소멸

에 관여하는 API 삽입으로 이루어진다. 이 때 secure function entry 수정을 최소화하기 위하여 InvokeCommandEntryPoint API의 코드를 생성할 때 secure function entry의 매개변수를 공유 메모리로부터 분배하며 secure function entry를 호출하는 방법으로 코드를 작성한다.

C언어로 작성된 함수 호출 방식은 call-by-reference와 call-by-value로 나뉜다. 이 두 경우에 전달되는 매개변수는 메모리 블록을 가리키는 포인터 타입 변수와 변수에 저장된 값 또는 상수 값으로 나뉜다. 따라서 CA와 TA간에 매개변수 전달이 이루어지려면 포인터 타입이나 배열 타입 매개변수가 가리키는 메모리 블록이 공유 메모리를 통해 전달이 되어야 한다. 본 논문에서는 공유 메모리 관리를 단순화하기 위하여 CA-TA 간 통신이 이루어질 때 임시로 생성되는 공유 메모리를 통해서만 매개변수가 가리키는 메모리 블록이 전달되도록 한다.

본 논문에서 제안하는 전체적인 코드 자동 분리 과정은 위와 같으나, 현재 구현한 것은 그 중 fgets의 매개변수를 활용한 secure function 추출과 인터페이스 자동 삽입기의 일부다.

제 2 절 정적 분석기

정적 분석기의 분석 대상은 애플리케이션이 네트워크, 파일 등 외부로부터 받아온 민감한 데이터가 저장되는 메모리 블록이다. 정적 분석기가 수행하는 정적 분석은 민감한 데이터가 저장된 메모리 블록을 가리키는 포인터 변수에 대한 alias analysis와, 민감한 데이터가 저장된 메모리 블록에 대한 taint analysis다. Taint analysis는 alias analysis의 결과를 보완하도록 되어 있다. 즉 fgets 함수의 첫 번째 매개변수인 민감한 데이터를 저장할 메모리 블록을 가리키는 포인터 변수에 대한 alias analysis가 수행되고, memcopy 또는 직접적인 메모리 블록 접근에 의한 데이터 propagation으로 인해 taint 된 메모리 블록에 대한 복제가 이루어지면 이 복제된 메모리 블록을 가리킬 수 있는 포인터 변수들에 대한 alias analysis가 이루어진다.

Alias analysis는 2장 5절에서 설명한 data structure analysis를 기반으로 구현되어 소스 코드의 각 함수들에 대한 DSGraph를 생성한다. Taint analysis는 소스 코드의 IR을 분석하여 먼저 fgets의 첫 번째 매개변수가 가리키는 메모리 블록을 의미하는 DSNode를 TaintedDSNodes 집합에 넣는다. 그리고 memcopy 등의 연산을 통해 메모리 블록의 복제가 이루어졌을 때 이 복제된 메모리 블록을 나타내는 DSNode도 TaintedDSNode 집합에 넣어 복제된 메모리 블록을 가리키는 포인터 변수들이 원본 메모리 블록을 가리키는 포인터 변수들과 같은 의미를 지닐 수 있음을 표시한다. 그리고 TaintedDSNode 집합의 DSNode들을 가리키는 포인터 변수가 dereference 되는 함수들의 ID를 추출하여 secure function list를 작성한다.

일련의 과정이 끝나고 나면 caller의 DSNode 정보와 callee의 DSNode 정보를 통합하여 TaintedDSNode 집합을 갱신한다. Taint analysis는 taint된 DSNode의 집합이 변하지 않을 때까지 반복적으로 수행되어 민감한 정보를 갖고 있는 메모리 블록들을 가리키는 모든 포인터들을 추적한다.

Pseudo Algorithm: Taint Analysis

```
Module: IR of input source code
DSA: result of interprocedural data structure analysis
1 SFuncList, TaintList //empty lists
2 do
3  | Taint_Analysis ( Module, DSA )
4  | | foreach( Function in Module )
5  | | | DSG = DSA.getDSGraph( Function )
6  | | | foreach( BasicBlock in Function )
7  | | | | foreach( Instruction in BasicBlock )
8  | | | | | switch( Instruction->opcode )
9  | | | | | case: BinaryOperator
10 | | | | | | if( one of Instruction->operands is tained )
11 | | | | | | | Instruction->result->state = tainted
12 | | | | | | break
13 | | | | | case: Store or Load
14 | | | | | | if( Instruction->source is tainted )
15 | | | | | | | TaintList<-( DSG.getDSNode( Instruction->result ))
16 | | | | | | | SFuncList<-( Instruction->getFunction )
17 | | | | | | break
18 | | | | | case: Call
19 | | | | | | if( is fgets )
20 | | | | | | | TaintList<-(DSG.getDSNode(Instruction->argument(0))
21 | | | | | | | break
22 | | | | | | if( Instruction->Callee->Return is tainted )
23 | | | | | | | TaintList<-( DSG.getDSNode( Instruction->result ))
24 | | | | | | break
25 | | | foreach( DSNode in TaintList )
26 | | | | TaintList<-( DSNode->getEquivalentDSNode )
27 while( TaintList is changed )
```

표 5 Pseudo algorithm of taint analysis

제 3 절 인터페이스 자동 삽입기

본 논문의 인터페이스 자동 삽입기는 CA의 secure function entry가 호출되는 코드를 고쳐 쓰고, TA에 통신에 필요한 인터페이스를 작성하여 삽입하는 역할을 수행한다. 따라서 소스 코드를 수정하기 용이하도록 LLVM의 front-end인 clang을 기반으로 구현하였다.

Clang은 텍스트로 이루어진 소스 코드를 parsing하여 AST(Abstract Syntax Tree)를 생성한다. 그림7 에 표현된 것처럼 AST의 각 노드는 소스 코드의 일부로서 텍스트로 표현된 symbol의 정보와 statement의 정보를 담는다.

CA에 인터페이스를 자동 삽입하기 위한 알고리즘은 표와 같다. 입력은 Clang library를 사용하여 CA로 분리된 소스 코드 AST로 만든 것과 secure function들에 대한 정보다. 먼저 알고리즘의 3번째 줄에서 CA 코드의 제일 처음 부분에 각 secure function entry 실행을 TA에 요청하기 위한 commandID를 secure function들에 대한 정보를 활용하여 삽입한다. 그 후 CA 인터페이스 자동 삽입기는 알고리즘의 4번째 줄부터 8번째 줄처럼 AST를 처음부터 끝까지 방문하며 secure function들을 호출하는 call expression 노드들을 수집한다.

```
FunctionDecl func1 int (int)
 ├ParmVarDecl a int
 ├ParmVarDecl x int
 ├CompoundStmt
 | ├DeclStmt
 | | └VarDecl x int
 | ├BinaryOperator =
 | | ├DeclRef lvalue Var x int
 | | └CallExpr
 | |   ├DeclRef Function foo int (int, int)
 | |   ├DeclRef lvalue Var x
 | |   ├DeclRef lvalue Var a
```

그림 8 Simplified AST

알고리즘의 9번째 줄부터 24번째 줄은 앞서 수집한 call expression을 단서로 삼아 client API를 삽입하는 부분이다. Call expression 노드 수집이 완료되면 각 call expression 노드의 부모 노드를 처음으로 function declaration을 만날 때까지 계속 방문한다. 본 논문에서 대상으로 삼은 언어는 C언어로, 함수를 중첩해서 선언하는 것은 허가되지 않았으므로 처음으로 만나는 함수 선언이 call expression 노드가 소속된 부모 함수가 된다. 만약 이 부모함수가 아직 수정되지 않은 함수이고 secure function의 callee가 아니라면 이 부모 함수에 대한 AST 노드를 PFuncList에 넣고 부모 함수의 local definition list의 마지막 부분과 statement list의 시작 부분에 각각 TEEC_Context와 같은 client API의 데이터 선언과 TEEC_InitializeContext 같은 API 함수를 넣는다. 그리고 부모 함수의 return statement 전에 TEEC_FinalizeContext 같은 API 함수를 넣는다.

CA에서 TA에 서비스 요청을 할 때, 필요한 매개변수를 공유 메모리를 통해 전달한다. 이 때 CA에서 공유 메모리에 각 매개변수를 넣기 위하여 TEEC_Operation 타입의 필드 중 하나인 TEEC_Param 타입 크기 4짜리 배열인 param[4]을 이용하도록 되어 있다. 그런데 secure function entry가 반환하는 값이 TA에서 CA로 전해지기 위해서는 CA에서 TA로 매개변수를 전달할 때처럼 공유 메모리를 사용할 수밖에 없다. 따라서 secure function entry의 필요 매개변수가 4개 이상인 경우에는 본 논문에서 구현한 삽입기에서 warning을 출력하여 개발자에게 데이터 직렬화(serialization)을 요구하도록 하였다. Secure function entry 호출에 사용될 각 매개변수는 secure function entry의 형식 인자가 나열된 순서대로 데이터 타입을 분석하여 포인터 타입 또는 배열 타입인 경우 param[0-2]의 tmpref 필드에 메모리 블록을 가리키는 포인터를 넣고, 이외의 경우 param[0-2]의 value 필드를 사용한다. 이 때 param[3]은 secure function entry의 반환값을 param[3].value로 전달받기 위하여 매개변수를 전달하는데 사용하지 않는다.

Pseudo Algorithm: CA Interface Inserter

```
AST: AST constructed from input source code
SFuncList: secure function list
InsertCode( Location, text ): insert text into Location of source file
ReplaceCode( Location, text ): replace code indicated by Location to text
1 CA_Interface_Inserter( AST,SFuncList )
2  | CallList, PFuncList //empty lists
3  | InsertCode( AST->begin->location->begin, FunctionIDs )
4  | foreach( Node in AST )
5  |  | if( Node is CallExpr )
6  |  |  | Callee = Node's callee
7  |  |  | if( Callee->functionID is in SFuncList )
8  |  |  |  | CallList<-Node
9  | foreach( Item in CallList )
10 |  | PNode = Item
11 |  | while( Node != root of AST )
12 |  |  | if( Node == function declaration && has body )
13 |  |  |  | if( Node is not in PFuncList && Node is not in SFuncList )
14 |  |  |  |  | PFuncList<-Node
15 |  |  |  |  | InsertCode( Node->LocalDef->location->end,
   |  |  |  |  |          ClientAPI data declarations)
16 |  |  |  |  | InsertCode( Node->getBody->begin->location->start,
   |  |  |  |  |          TEEC_Context/Session init codes)
17 |  |  |  |  | foreach( return in Node )
18 |  |  |  |  |  | InsertCode( return->location->start,
   |  |  |  |  |  |          TEEC_Context/Session finit codes )
19 |  |  | Node = Node->parent
20 | Node = Item
21 |  | InsertCode( Node->location->start, parameter setting codes )
22 |  | if( Node->parent == BinaryAssign )
23 |  |  | InsertCode( Node->location->end, return value assign codes )
24 | ReplcaceCode( Node->location, InvokeCommand codes )
```

표 6 Pseudo algorithm of CA interface inserter

TA 인터페이스 삽입기는 CA 인터페이스 삽입기와 비교했을 때 더 단순하다. Clang library를 사용하여 만들어진 TA 소스 코드의 AST를 활용하여 소스 코드의 시작 부분에 secure function entry의 commandID를 삽입하고 소스 코드의 마지막 부분에 통신에 필요한 internal API를 삽입한다. 이 때 CA의 TEEC_InvokeCommand API 함수에 반응하여 실행될 TA_InvokeCommandEntryPoint를 작성한다.

TA_InvokeCommandEntryPoint 함수에서 반드시 사용되어야 하는 형식인자는 TEE_Param 타입 배열 params[4]와 각 secure function entry를 가리키는 32비트 정수 타입 cmd_id가 있다. 본 논문에서 구현한 TA 인터페이스 삽입기는 cmd_id를 인자로 받는 switch 구문을 이용하여 secure function entry에 해당하는 commandID가 전달된 경우 해당 secure function entry를 호출하도록 코드를 자동 생성한다. 이때 secure function entry의 코드 수정을 피하기 위하여 공유 메모리로 받은 param[4]에서 각 secure function entry 호출에 필요한 매개변수를 분배한다. 매개변수 분배 과정은 CA 인터페이스 삽입기와 동일하게 secure function entry의 형식 인자들의 타입을 순서대로 분석하여 param[0-2]의 memref 또는 value 필드의 값을 분배한다.

제 4 장 실험

제 1 절 실험 환경 및 실험 코드

실험에 사용한 환경은 표 7과 같다.

현재 정적 분석기와 인터페이스 삽입기 사이의 정보 연동과 인터페이스 삽입기의 internal API 치환이 완전히 해결되지 않았으므로 각각 다른 코드를 사용하여 실험하였다. 정적 분석기 실험은 OpenSSL-1.0.1f[10]의 코드 일부를 사용하였으며 인터페이스 삽입기 실험에 사용한 코드는 OPTEE에서 제공하는 xtest[11] 중 crypto 관련 코드 일부를 normal world 코드로 바꾼 코드에 대해 실험하였다.

Experimental environment	
CPU	Intel(R) Core(TM) i7-3770 3.40GHz
RAM	up to 16GB
OS	Ubuntu 14.04.4 LTS
Target language	C

표 7 Experimental environment

제 2 절 정적 분석기 실험

정적 분석기를 실험하기 위하여 사용한 코드는 OpenSSL-1.0.1f의 코드 일부로, 총 세 함수로 구성되어 있다. 첫 번째 함수 PEM_read_bio 함수로, fgets를 사용하여 파일로부터 민감한 정보를 읽고, 매개변수로 받은 포인터가 가리키는 메모리 블록에 민감한 정보를 복사하는 함수다. 두 번째 함수는 dataaccess 함수로, 첫 번째 함수로 읽은 민감한 정보에 직접 접근하는 함수다. 세 번째 함수는 첫 번째 함수와 두 번째 함수를 호출하는 main 함수다. 이 때 민감한 정보를 저장한 메모리 블록을 가리키는 포인터 변수들이 PEM_read_bio 함수로부터 main 함수의 포인터 변수들로 전달된다. 이 포인터 변수들은 다시 main 함수로부터 dataaccess 함수로 전달된다. 따라서 민감한 정보에 직접적으로 접근하는 함수는 PEM_read_bio 함수와 dataaccess 함수가 된다.

실험결과 PEM_read_bio 함수와 dataacess 함수만 secure function으로 판별되었다.

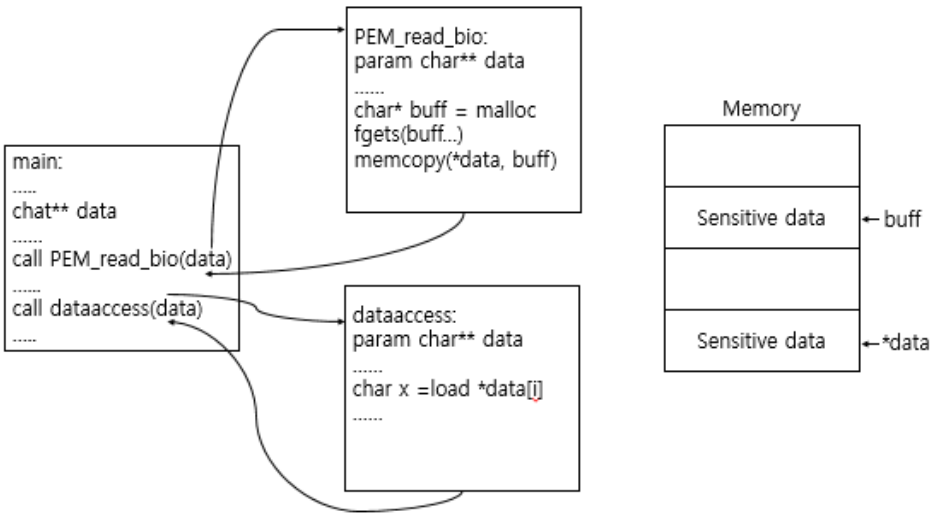

그림 9 Taint analysis example

제 3 절 인터페이스 삽입기 실험

실험에 사용한 코드는 xtest의 xtest_tee_test_1004 코드로, 고급 암호화 표준을 따르는 암호화 코드다. 현재 정적 분석기에서 secure function list는 생성 가능하나 코드를 분리하는 것은 구현하지 않았으므로 코드가 분리되었음을 가정하고 실험하였다.

CA와 TA의 인터페이스를 삽입하기 이전의 코드 크기와 인터페이스 삽입 후 코드 크기는 다음과 같다.

	Size of code		
	before	after	diff
CA	15	38	153%
TA	617	646	4.7%
Total	632	684	8.2%

표 8 Interface inserter experimental result

CA의 코드 증가량이 큰 이유는 실험에 사용한 코드의 대부분이 TA의 코드이고, CA는 TA에 서비스를 요청하는 부분만 있는 작은 코드이기 때문이다. 애플리케이션을 구성하는 모든 코드의 증가량은 약 8.2%로, TA의 secure function 중 실제 CA에서 호출되는 secure function에 대해서만 인터페이스가 삽입되었다.

CA와 TA의 인터페이스 삽입 결과의 일부는 그림 9와 그림 10과 같다. Secure function인 secure_aes256ecb_encrypt 함수를 호출하는 코드가 TEEC_InvokeCommand API 함수로 바뀌었으며, 매개변수를 전달하기 위한 공유 메모리 생성 코드가 삽입되었다. 그 외에 TEEC_Context와 TEEC_Session을 구성하기 위한 API 함수 및 API 데이터, secure function에 대한 commandID가 삽입되었다.

인터페이스 삽입된 코드와 원래 코드의 수행 시간 비교는 다음과 같다.

	Total execution time (ms)
Original code	2
Divided code	53

표 9 Total execution time

인터페이스가 삽입되어 secure world에서 일부가 수행되는 코드는 원래 코드에 비해 25배 이상의 수행 시간을 소요했다. 이 인터페이스가 삽입된 코드의 수행 시간을 좀 더 자세하게 보면 다음과 같다.

Parts of divided code	Execution time (ms)	Portion to total exec. time
Context initialization	3	5.66%
Context finalization	1	1.88%
Session open	33	62.26%
Session close	4	7.55%
Secure function call	12	22.64%

표 10 Divided code execution time in detail

표 10은 인터페이스가 삽입된 코드의 normal world에서 해당 동작을 수행하는 API 함수를 호출했을 때 걸리는 시간을 나열한 것이다. CA와 특정 TA를 연결하기 위해서는 normal world의 user space에서 TEE driver와 tee-supplicant, TEE core를 거쳐 secure world의 TA를 찾아 session을 생성한다. 이 과정에서 tee-supplicant와 TEE core 간의 통신이 수차례 이루어지기 때문에 시간이 오래 걸린다. 하지만 한번 CA와 TA간의 연결인 session이 생성되고 나면 TA의 secure function을 호출하기 위한 TA_InvokeCommandEntryPoint API 함수 실행과 secure function 호출이 이루어지는 과정에서 TEE driver와 TEE core를 거쳐 TA와 통신하므로 상대적으로 통신에 필요한 시간이 크게 단축된다.

제 5 장 결론

본 논문에서 구현한 정적 분석기는 fgets를 통해 민감한 데이터를 저장하는 메모리 블록에 대한 taint analysis와 메모리 블록을 가리키는 포인터 변수들에 대한 alias analysis를 수행하여 민감한 데이터에 직접적으로 접근하는 함수들을 선별할 수 있다. 또한 인터페이스 자동 삽입기는 정적 분석기로 선별한 함수들에 대하여 각각 CA와 TA로 기능할 수 있는 API를 삽입할 수 있다.

그러나 아직 민감한 데이터를 읽는 fgets와 민감하지 않은 데이터를 읽는 fgets의 구별이 되지 않는다. 또한 fopen과 fgets 등 secure world에서 지원하지 않는 라이브러리 함수에 대해 trusted storage를 사용하는 internal API로 치환이 이루어지지 않는다. 따라서 완전한 코드 자동 분리 기술을 구현하기 위해서는 정적 분석기와 인터페이스 자동 삽입기의 보완이 필요하다.

본 논문의 의미는 코드 자동 분리를 위한 개략적인 그림을 제시하고, 가장 기초라 할 수 있는 정적 분석기와 인터페이스 자동 삽입기의 기초 구현을 위한 알고리즘을 제시한 것이다.

참 고 문 헌

[1] Platform, Global. "The trusted execution environment: delivering enhanced security at a lower cost to the mobile market." Global Platform white paper (2011): 1-26.

[2] https://www.globalplatform.org/

[3] GlobalPlatform TEE System Architecture ver. 1.0

[4] https://www.arm.com/products/security-on-arm/trustzone

[5] https://github.com/OP-TEE/optee_os

[6] GlobalPlatform TEE Client API Specification ver. 1.0

[7] Leach, P., M. Mealling, and R. Salz. "RFC 4122: A Universally Unique IDentifier (UUID) URN Namespace, 2005." Online: http://www. ietf. org/rfc/rfc4122. txt. Accessed 4 (2013).

[8] http://llvm.org//

[9] http://clang.llvm.org/

[10] http://www.openssl.org/source/openssl-1.0.1f.tar.gz/

[11] https://github.com/OP-TEE/optee_test

Abstract

Gisoo Seo

Electrical and Computer Engineering

The Graduate School

Seoul National University

To reinforce securities of applications such as user athentication and internet banking, data isolation to the trusted execution environment(TEE) is suggested. TEE provides the isolation execution environment to prevent attacks to important functions which access sensitive data. In this thesis, static code analyzer and interface inserter are described, which are needed to divide codes to make applications to be executed on the TEE based on the ARM TrustZone.

As an experimental result, the static analyzer can extract secure functions which access to sensitive data. Also, the interface inserter can insert GlobalPlatform client/internal APIs into seperated codes to make client applications and trusted application.

keywords : TEE; ARM TrustZone; OPTEE; data isolation; partitioning

Student Number : 2015-20932