KIPS Tr. Comp. and Comm. Sys.
Vol.3, No.2 pp.31~42 pISSN: 2287-5891

http://dx.doi.org/10.3745/KTCCS.2014.3.2.31

An Implementation of JTAG API to Perform Dynamic Program Analysis for Embedded Systems

Hyung Chan Kim† · Il Hwan Park††

ABSTRACT

Debugger systems are necessary to apply dynamic program analysis when evaluating security properties of embedded system software. It may be possible to make the use of software-based debugger and/or DBI framework if target devices support general purpose operating systems, however, constraints on applicability as well as environmental transparency might be incurred thereby hindering overall analyzability. Analysis with JTAG (IEEE 1149.1) debugging devices can overcome these difficulties in that no change would be involved in terms of internal software environment. In that sense, JTAG API can facilitate to practically perform dynamic program analysis for evaluating security properties of target device software.

In this paper, we introduce an implementation of JTAG API to enable analysis of ARM core based embedded systems. The API function set includes the categories of debugger and target device controls: debugging environment and operation. To verify API applicability, we also provide example analysis tool implementations: our JTAG API could be used to build kernel function fuzzing and live memory forensics modules.

Keywords : Embedded Systems, JTAG, Program Analysis, Security Evaluation

임베디드 시스템 동적 프로그램 분석을 위한 JTAG API 구현

김 형 찬† · 박 일 환††

요 약

임베디드 시스템 소프트웨어의 보안성 분석을 위한 동적 프로그램 분석을 시도하기 위해서는 디버거 체계가 필요하다. 타겟 장비가 범용 운영체제와 비슷한 환경을 지원하는 경우에는 소프트웨어 기반의 디버거 혹은 DBI 프레임웍 등을 장비 내에 설치하여 분석할 수 있으나, 설치 가능성 제한이나 분석 환경의 투명성 문제 등의 어려움이 있을 수 있다. JTAG (IEEE 1149.1)디버거 장비를 이용하여 분석하는 경우에는 분석을 위해 타겟 장비 내의 소프트웨어적 환경을 변경하지 않아도 된다. 타겟 장비의 보안성 분석을 위한 프로그램 동적 분석 기법들을 용이하게 적용하기 위해서는 JTAG 디버거 장비를 제어하기 위한 API가 필요하다.

본 논문에서는 ARM 코어 기반 임베디드 시스템 분석을 위한 JTAG API를 소개한다. 구현된 API는 JTAG 디버거 하드웨어를 직접 제어하며 디버깅 환경 및 동작제어를 위한 함수 세트를 제공한다. API의 활용 용이성을 확인하기 위하여 커널 함수 퍼징과 라이브 메모리 포렌식 기법을 적용한 보안 분석 도구의 예제 구현을 제시한다.

키워드 : 임베디드시스템, JTAG, 프로그램 분석, 보안성 분석

1. 서 론

현재 우리는 임베디드 시스템의 홍수 속에서 살고 있다. 산업용 기기, 항공우주, 자동차, 무기체계, 의료 장비, 생활가전 등 광범위한 분야에 다양한 종류의 임베디드 시스템들이 각 분야에 맞게 진화되어 활용되고 있으며 일반 사용자들은 현재 스마트폰을 필두로 1인 1단말 이상의 개인 휴대 기기를 사용하고 있는 생활을 영위하고 있다. 과거 데스크탑이나 서버 시스템에서의 운영체제나 응용 어플리케이션이 그러했던 것처럼[1], 다수의 임베디드 시스템 장비들이 동종 혹은 비슷한 코드기반의 시스템 소프트웨어를 광범위하게 사용하는 만큼 공격자의 관심과 더불어 보안성 문제에 대한

† 정 회 원 : 한국전자통신연구원 부설연구소 선임연구원
†† 비 회 원 : 한국전자통신연구원 부설연구소 책임연구원
논문접수 : 2013년 11월 18일
심사완료 : 2014년 2월 10일
* Corresponding Author : Hyung Chan Kim(kimhc@ensec.re.kr)

우려도 증가하고 있다.

보안성 평가나 취약점 탐색 등의 목적으로 임베디드 시스템의 소프트웨어를 효율적으로 분석하기 위해서는 정적(static) 분석과 더불어 동적(dynamic) 분석이 필요하다. 정적 분석이 소스 혹은 기계어 수준의 코드를 기반으로 프로그램 흐름을 살펴보는 것과 달리 동적 분석은 분석 대상 소프트웨어를 직접 구동시켜 수행 중에 얻는 문맥 정보에 기반하여 분석을 수행할 수 있다. 보안 분석 작업에서는 두 가지 접근방법들이 모두 적극적으로 활용된다. 동적 분석은 정적 분석에 비하여 코드 커버리지(coverage) 문제나 환경 구축의 어려움이 있을 수 있지만, 실제적인 버그나 취약성을 찾는데 매우 중요한 역할을 한다. PC와 같은 범용 운용체제 기반에서는 소프트웨어 기반 디버거나 가상화 기술 기반 DBI(Dynamic Binary Instrumentation) 프레임웍을 이용하여 동적 분석 환경을 비교적 용이하게 구축하여 분석을 수행할 수 있다.

임베디드 시스템 소프트웨어의 동적 분석을 위하여 환경 구축을 하는 경우에는 일반적으로 소프트웨어 기반 혹은 하드웨어 기반 분석 환경을 구축할 수 있다. 타겟(분석 대상) 장비의 시스템 소프트웨어가 커스텀 디버거 모듈을 제공하는 경우에는 이 모듈에 디버거 클라이언트 등을 붙여서 분석을 시도해 볼 수 있다. 만약 분석 대상 시스템 소프트웨어가 Linux나 FreeBSD등의 범용 운영체제를 수정하여 제작된 경우에는 gdb[2]와 같은 범용 디버거 혹은 기타 동적 분석 프로그램을 빌드과정이나 패키지 설치과정 등을 통하여 적재하여 볼 수 있다. 하지만 해당 제품군의 개발사가 아닌 분석 측의 입장에서 이러한 소프트웨어 기반 디버거 적재가 항상 가능한 것은 아니며 코드 적재 용량이나 기타 내부 운영체제의 제한 등에 따라서도 제약 사항이 있을 수 있다. 또한 어떠한 경우에는 대상 장비의 시스템 소프트웨어 체계의 무결성을 해치게 되는 결과를 초래할 수도 있다.

분석 대상 장비의 코어 프로세스가 JTAG(IEEE 1149.1) [3] 디버그 로직 블록을 포함하는 경우에는 JTAG 디버깅 하드웨어를 이용하여 시스템의 내부 소프트웨어 체계를 변경하지 않고 타겟 장비 외부에서 디버깅 인터페이스를 연결하여 동적 분석을 시도해 볼 수 있다. 이 경우 내부 소프트웨어 체계의 무결성을 유지하며 외부에서 코어 프로세서나 버스만을 제어함으로써 보다 투명한 분석이 가능하다. 또한 내장형 소프트웨어 디버거나 DBI를 활용하는 경우에는 분석 대상 소프트웨어와 함께 디버거 프로세스를 위한 내부 코어 프로세스의 자원을 사용하고 대상 장비 운영체제의 스케줄링을 받는 등의 영향을 받을 수 있다. 하지만, JTAG 디버거 장비를 사용하는 경우에는 디버깅 소프트웨어 모듈이 타겟 장비 밖에 위치해 있기 때문에 이러한 영향에서 비교적 자유롭다.

JTAG 디버거 장비를 보유하게 되면 보통 GUI기반의 전용 디버거 소프트웨어가 제공된다. 기본적인 수동 디버깅 제어를 제외하고, 보안성 분석과정에서 사용되는 프로그램 동적 분석 기법들을 적용하기 위해서는 자동화 및 커스텀 디버깅 제어가 가능한 확장 기능이 지원되어야 하는데, 대부분 디버거 환경 내부에서만 사용할 수 있는 스크립트 형태로 지원되며 이는 효율적인 분석도구 구현에 충분하지 않다.

본 논문에서는 ARM 코어 기반 임베디드 시스템 소프트웨어 동적 분석 어플리케이션 구현에 활용될 수 있는 JTAG API 구현을 소개한다. JTAG API 구현 목적은 다음과 같다.

① 분석 기능 확장: JTAG 디버거의 분석 인터페이스가 제공하는 기능 이외에 다른 동적 분석 기법을 커스텀하게 구현하여 구동시킬 수 있다. 또한 다른 분석 프로그램과의 연동이 가능하다.
② 자동화 제어: 반복적이거나 장시간의 분석 프로시져의 수행 혹은 일괄 배치 처리를 자동화하여 적용할 수 있다.
③ 리소스 및 성능 고려: 메모리를 많이 할당 하거나 획득된 시스템 컨텍스트를 기반으로 고성능의 분석계산이 필요할 때 충분한 리소스를 연계시킬 수 있다. 그리고 가급적 외적인 영향에 의한 성능 저하 요소 없이 빠르게 수행될 수 있어야 한다.

API소개와 더불어 JTAG API를 활용한 동적 분석 도구 제작의 용이함을 확인하기 위하여 ARM 기반 시스템에서 구동되는 안드로이드 시스템의 리눅스 커널을 대상으로한 예제 동적 분석 도구 구현들을 소개한다. 첫 번째는 커널 함수(시스템 호출)에 대하여 퍼징 기법을 적용하는 것과 두 번째로 커널 메모리 정보를 획득하여 분석하는 라이브 포렌식이다.

본 논문의 구성은 다음과 같다. 2장에서는 임베디드 시스템 동적 분석 관련 기술을 소개한다. 3장에서는 연구를 통해 구현된 JTAG API에 관한 설명을 하며, 4장에서 API를 활용한 예제 도구 구현 및 실험을 소개한다. 마지막 5장에서 본 논문의 결론을 맺는다.

2. 동적 분석 기반 기술

임베디드 시스템 소프트웨어를 분석하기 위해서는 기본적으로 정적 분석 및 동적 분석 두 가지 접근방법이 있다. 여기에서는 동적 분석 기반 기술에 대해서만 한정하여 살펴본다.

2.1 프로그램 동적 분석 기반 기술

프로그램을 실행시키는 상태에서 특정 순간 혹은 연속적인 순간의 집합에 대하여 코어 프로세서의 문맥(예, 현재의 레지스터 값)이나 메모리 상태 정보 등의 동적 상태 정보를 참조하여 분석에 적용하는 것을 동적 분석이라고 한다. 동적 분석은 프로그램 중단점(breakpoint)이나 관찰점(watchpoint)을 설정하여 프로그램 수행 흐름을 지켜보는 것부터 입력값에 변화를 주면서 버그를 유도해 내는 퍼징(fuzzing), 특정 코드 영역의 수행을 살펴보는 프로파일링(profiling), 코

드가 다루는 데이터 흐름을 추적하면서 입력 값에 대한 수행 흐름 영향을 살펴볼 수 있는 동적 오염 분석(dynamic taint analysis) 등의 다양한 기법들이 연구되고 있다. 이러한 동적 분석이 가능하려면 프로그램을 실행시키면서 수행 문맥을 관찰할 수 있는 기반 프레임웍이 있어야 한다. 임베디드 시스템 소프트웨어의 경우 다음과 같은 소프트웨어 및 하드웨어 기반 기술들을 고려해 볼 수 있다.

1) 소프트웨어 기반 기술

소프트웨어적으로 프로그램 수행 문맥을 관찰할 수 있는 가장 널리 사용하는 기반은 디버거(debugger)를 사용하는 것이다. 디버거는 중단점을 설정하는 명령어(예, X86의 INT 3, ARM의 BKPT, MIPS의 BREAK)를 중단점 주소에 있는 코드를 치환하면 프로세서가 해당 주소에서 설정된 명령어를 보고 적절한 예외 핸들러함수를 실행시킨다. 프로그램이 중단되면 레지스터나 메모리 값 등의 문맥 정보를 살펴볼 기회를 얻게 된다. 이러한 문맥 획득 행위는 디버거의 직접 명령 수행 혹은 펌웨어나 운영체제 커널 서비스를 경유하여 얻게 되는데 궁극적으로 디버깅되는 장비 코어 프로세스의 명령이 직접 사용되는 것이다.

문제는 타겟 장비에서의 디버거 활용 가능성이다. 크기가 비교적 큰 펌웨어 기반의 어떤 장비의 경우 제한적인 gdb 프로토콜을 지원하는 디버거 스텁을 내장하여 외부에서 gdb 클라이언트 세션을 연결하여 디버깅이 가능한 경우[4]가 있지만 많은 경우에서 이런 상황을 기대하기 힘들다. 소형 라우터 같은 펌웨어 크기가 작은 장비들은 개발자용의 숨겨진 디버그 쉘이 있어 간단한 메모리 읽기 쓰기가 지원되는 경우, gdb proxy를 구성하여 외부에서 디버거를 구동시키는 사례가 있다[5]. 만약 장비의 펌웨어를 구할 수 있고 해당 펌웨어를 정적 역공학 분석하여 gdb 스텁 기능을 이식할 수도 있지만 장비에 따라서는 쉬운 작업은 아니다. Linux나 BSD계열 범용 운영체제를 개조하여 만든 장비의 경우에는 라이브러리 환경을 분석하여 해당 프로세서에 맞는 gdb를 빌드하여 설치할 수 있다. 어플라이언스(appliances) 장비들은 일반 범용 컴퓨터와 비슷한 구성을 가지므로 gdb 패키지를 바로 설치하여 사용할 수 있는 경우도 있다. 이와 같이 임베디드 시스템 장비마다 제각각 구성 특성이 다르므로 기본적인 디버거를 명시적으로 지원하지 않는다면 펌웨어 분석 과정을 거쳐서 디버깅 기능을 구성해야 하는 어려움이 있으며, 펌웨어가 암호화되어 있는 경우라면 디버거를 구성하기 위한 분석 작업이 더욱 어려워지거나 불가능한 경우도 있다.

최근에는 소프트웨어 가상화(virtualization) 혹은 에뮬레이터 기술이 발전하여 가상화 프레임웍을 이용하여 임베디드 시스템 소프트웨어를 대상으로 동적 분석을 시도하는 연구도 활발하다. 임베디드 시스템 소프트웨어를 성공적으로 가상화 프레임웍에 이식(porting)을 하여 구동하게 만들면 가상화 프레임웍 자체가 제공하는 디버깅 기능이나 오픈소스 가상화 프레임웍을 개조하여 디버깅 기능과 유사하게 특정 시점에서 가상화 실행을 멈추어 실행 문맥을 살펴보는 기능을 구현할 수 있다.

어플리케이션 프로세스 단위의 가상화를 지원하는 PIN[6]이나 Valgrind[7]는 안드로이드 기반 임베디드 시스템용 코어들을 지원한다. Valgrind의 경우 X86 이외에 ARM, MIPS 등의 다양한 코어를 지원하며 다른 리눅스 계열 임베디드 시스템에 커스텀 빌드하여 적재할 수도 있다. 이 프레임웍들은 확장기능을 지원하는 플러그인 API를 제공하여 다양한 분석 루틴을 구현하여 별도의 동적 분석 도구를 구현할 수 있다. 하지만, 프로세스 단위 가상화 기술은 대상이 커널이 아닌 유저 프로세스를 대상으로 하기 때문에 커널레벨의 코드나 이벤트들에 대한 직접 분석이 불가능하다. 또한, 가상화 엔진 자체가 분석 대상 프로세스와 같은 주소 공간에 공존해야 함으로 메모리와 같은 시스템 자원을 충분하게 지원하는 최신 안드로이드 단말을 제외하고는 성능문제로 적용하기 힘든 단점이 있다.

QEMU[8]와 같은 전체 시스템 에뮬레이터를 이용하는 가상화 기반 기술도 임베디드 시스템 분석에 적용되고 있다. Delugre의 경우 QEMU를 개조하여 MIPS기반의 네트워크 인터페이스 카드(NIC)의 펌웨어를 분석하였으며[9], Muniz와 Ortega의 경우, 상용 라우터 이미지를 가상화하여 구동시키는 Dynamips 에뮬레이터에 gdb 스텁을 구현하여 동적 분석을 시도하였다[10]. 이와 같이 QEMU와 같은 시스템 에뮬레이터를 활용하는 경우 가상 하드웨어 레벨의 시각에서 전체 시스템의 동작을 분석할 수 있다. 하지만, 에뮬레이터에 분석 대상 임베디드 시스템 소프트웨어 체계를 이식하는 것은 쉬운 일이 아니다. QEMU의 경우, 다양한 코어 프로세서들을 지원하지만, 디바이스의 경우 매우 제한적인 지원을 하기 때문에 지원하지 않는 디바이스가 동적 분석에 필수적인 경우 직접 하드웨어 스펙을 분석하여 가상 디바이스를 구현하여 QEMU에 포함시켜야 하는 어려움이 있다.

2) 하드웨어 기반 기술

임베디드 시스템 소프트웨어 체계에 의존하지 않고 외부 하드웨어적으로 동적 분석을 적용할 수 있는 대표적인 것은 현재 산업계에서 널리 사용하고 있는 JTAG 디버거 장비를 활용하는 것이다. JTAG 장비에 대해서는 다음 소절에서 서술한다.

JTAG 이외에 전용 하드웨어를 이용하여 시스템의 동적 정보를 모니터링 하는 것은 아직 기초 단계이지만 연구로써 진행되고 있다. 시스템에 설치된 커널 동작에서 루트킷 대응과 같은 무결성 검사를 하기 위한 목적으로 copilot[11]은 PCI 버스를 통해서 Ki-mon[12]은 시스템 버스와 DMA를 이용하여 물리 메모리의 변화를 감지하는 전용하드웨어를 시스템 외부에 구현하였다. 하지만 이들 연구는 아직 범용 운영체제를 대상으로 기초단계이며 획득되는 동적 정보도 메모리 읽기나 쓰기에 국한되어 제한적이다. Ki-mon의 경우 추후 ARM 코어 계열 기반 구현이 될 예정이다.

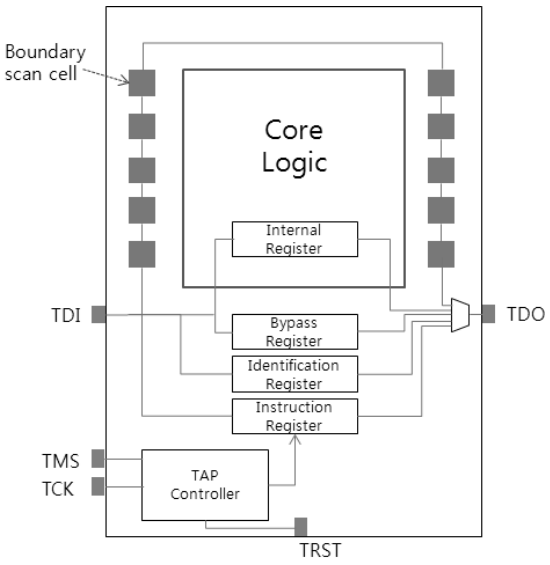

Fig. 1. JTAG (IEEE 1149.1) Device Architecture

2.2 JTAG 장비를 이용한 임베디드 소프트웨어 동적 분석

1) JTAG 장비 기반 동적 분석

JTAG(Joint Test Action Group)은 1980년대 중반부터 결성된 하드웨어 테스트 엔지니어들의 그룹으로 칩으로 구현되는 논리 회로 컴포넌트들을 테스트하기 위하여 경계 스캔(boundary scan) 아키텍처 Figure 1을 IEEE 1149.1[3]로 표준화였다.

JTAG 아키텍처를 지원하는 로직 컴포넌트는 코어 로직 경계에 경계 스캔 셀(boundary scan cell)들이 구성되어 있고 이들은 서로 연결되어 병렬적인 쉬프트 레지스터 구조를 이루어 코어 로직으로부터 시그널 값을 받거나(Capture) 입력(Update) 한다[13]. 또한 이 스캔 셀들은 직렬 쉬프트 구조를 이루어 외부 디버거 장비로부터 연결된 인터페이스의 TDI(Test Data Input) 핀으로부터 데이터를 받거나 TDO (Test Data Output) 핀으로 데이터를 출력한다. TDI 핀으로부터의 입력과 TDO 핀으로의 출력은 TCK(Test ClocK) 핀으로 제공되는 클럭에 동기화된다. 즉, 입출력 핀이 각각 하나씩이기 때문에 시리얼 구조로 디버거 장비와 통신을 하게 된다. TMS(Test Mode Select) 핀에 의한 시그널은 TAP 컨트롤러의 상태를 정하여 테스트 동작을 제어하게 된다. 타겟 장비에 프로세스가 여러 개인 경우, 한 코어의 TDI 핀이 다른 코어의 TDO로 연결되는 전체적으로 데이터 입출력 경로가 직렬 체인을 이루고, 제어를 위한 TCK와 TMS는 각 코어에 병렬적으로 연결되는 구조이다.

프로세서 제조사들은 이러한 JTAG 아키텍처를 지원하는 로직을 SoC(System-On-Chip) 형태로 프로세서에 포함하여 출시한다. 최근 출시되는 다수의 ARM 프로세서의 경우 CoreSight[14] 디버깅 아키텍처의 일부로 JTAG을 지원한다. 분석 대상 장비의 프로세서가 JTAG 아키텍처를 지원하면 JTAG 디버깅을 지원하는 장비[15-17]를 이용하여 동적 분석을 수행할 수 있다.

보통 개발자용 레퍼런스 임베디드 보드는 표준적인 14핀 혹은 20핀의 JTAG 인터페이스를 명시적으로 지원하는 경우가 많다. 하지만, 완제품 형태로 출시되는 임베디드 장비들의 경우에는 코어 프로세서의 JTAG 아키텍처가 포함된 상태임에도 불구하고 대부분 이러한 인터페이스용 핀들을 단순 제거하거나 아니면 섬유유리 테잎 등으로 물리적 봉쇄 처리를 하는 경우도 있다. 일부 엔지니어들은 프로세서로부터 나오는 회로 연결을 참조하여 경험적으로 해당 핀을 찾아내기도 한다. 역공학을 목적으로 임베디드 시스템 동적 분석을 시도하려는 엔지니어들은 이러한 JTAG 핀들을 찾는 방법들을 연구하고 있다. JTAGFinder[18]나 JTAGulator [19]는 TAP controller가 제어하는 JTAG 제어 상태 전이를 참조하여 여러 핀이나 회로상의 연결 지점에서 JTAG 시그널을 찾아내는 방법들을 제시하고 있다.

JTAG 디버깅 장비를 이용한 동적 분석은 분석 대상 장비에 소프트웨어 디버거나 DBI 프레임웍의 설치와 같이 소프트웨어 체계를 수정하지 않고, 분석이 가능하여 위에서 소개한 다른 분석 기반 접근보다 환경적 투명성이 높다. 또한 JTAG 아키텍처가 지원되는 코어 프로세스이면 전 소절에서 설명된 분석 기반 소프트웨어(디버거, 디바이스 드라이버 가상화 등) 포팅에 대한 문제가 없이 분석 환경을 구축할 수 있는 장점도 있다.

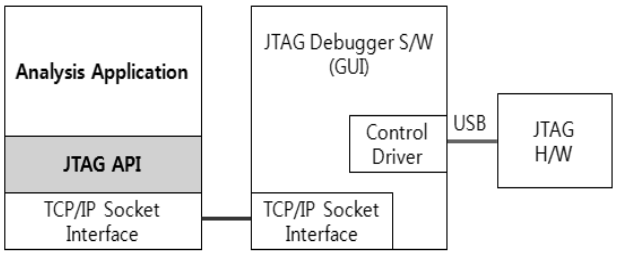

Fig. 2. The Composition of External Analysis Tool Based on Vendor Provided JTAG APIs (CodeViser, Trace32)

2) JTAG API

상용 JTAG 장비 벤더가 지원하는 분석(디버거) 소프트웨어의 기능을 확장하거나, 분석가가 고안한 동적 분석 기법들을 자동화 혹은 배치화하여 적용하기 위해서는 JTAG 디버거 장비를 통한 코어 프로세서 디버그 로직의 TAP Controller 동작을 제어할 수 있는 API를 이용하여 커스텀 분석 도구 어플리케이션을 작성할 수 있다.

국내에서 많이 사용되고 있는 JTAG 디버거 장비 벤더들도 JTAG API를 지원한다[20,21]. 대표적인 벤더들이 지원하는 API 지원 구조는 공통적으로 Figure 2와 같다. 기본적으

로 수동으로 분석을 수행하는 GUI 기반의 디버거 소프트웨어와 JTAG 하드웨어의 연결은 일반적인 활용 구성이다. 여기에, API를 이용하는 분석 어플리케이션은 TCP 소켓 통신을 통하여 GUI 기반의 JTAG 디버거 소프트웨어와 통신을 하는 구조이다. 즉, GUI 기반의 디버거가 디버깅 서버 역할을 하며 클라이언트인 어플리케이션의 API 호출에 대한 요구 처리를 하고 결과를 되돌려 준다.

이러한 지원 구조는 사용자 인터페이스나 API 기반 외부 분석 어플리케이션 세션들이 동시에 유효하여 상황에 따라 임의적으로 선택하여 사용가능한 장점이 있다. 예를 들어, 복잡한 디버깅 옵션 구성을 GUI 기반 디버거 소프트웨어 화면에서 설정하고, 분석 어플리케이션에서는 배치적인 디버깅 세션을 구동시킬 수 있다. 하지만 분석 어플리케이션이 완벽하게 독자적으로 JTAG 장비에 연결되지 못하고 API 호출이 부가적인 네트워크 채널을 경유하기 때문에 많은 중단점을 설정하는 디버깅 작업에서는 연속적인 네트워크 요청이 발생하게 된다. 또한 디버거 서버 프로세서에서 API 호출에 의한 제어 요청에 의한 결과 변화 요소를 먼저 GUI 화면에 갱신(예, 레지스터 값 변경)하고 API로 결과 값을 되돌려주는 방식은 분석 어플리케이션 측면에서 성능 부담적인 요소가 될 수 있다.

본 논문에서는 Figure 2의 API 지원 구조를 단순화하여 독립적인 구조의 분석 어플리케이션이 직접 JTAG 디버깅 하드웨어를 제어하는 API 구현을 소개한다.

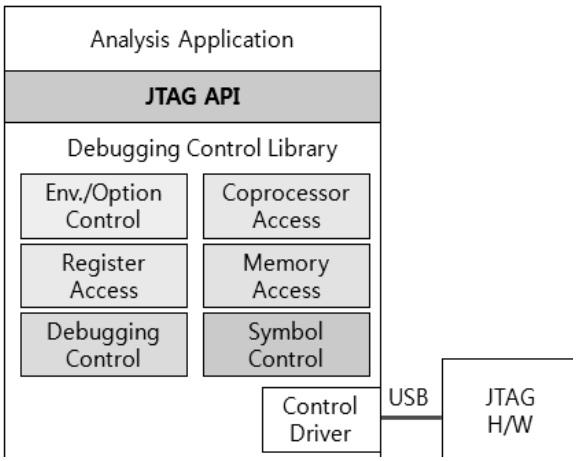

Fig. 3. The Proposed JTAG API Architecture

3. JTAG API 구현

이 장에서는 동적 프로그램 분석 어플리케이션을 구현하기 위한 기존 JTAG API의 구성 체계 Figure 2를 개선한 구조를 설명한다. 그리고 구현된 API 함수 분류에 관하여 설명한다. 설명 중 분석 대상 타겟 시스템과 관련된 내용들은 일반적인 ARM 코어 기반 시스템을 지칭한다.

3.1 API 구조

본 연구를 통하여 개발된 JTAG API의 모듈 및 분석 어플리케이션 체계 구조는 Figure 3과 같다. 분석 도구 어플리케이션은 JTAG API 및 하부의 디버깅 제어 라이브러리 체계 활용하여 구현되며 JTAG API 계층의 라이브러리와는 동적 바인딩 된다. 디버깅 제어 라이브러리는 JTAG 아키텍처가 지원하는 디버깅 동작을 제어하기 위한 모듈들로 구성되어 있으며 API로 익스포트(export)된 함수들의 분류 내용은 다음 소절에서 설명한다.

기존 상용 벤더들이 제공하는 API 구조 Figure 2와 가장 큰 차이점은 JTAG 디버깅 장비를 제어하기 위한 소프트웨어 구성에 있다. 기존 API는 벤더들이 제공하는 GUI 기반의 디버깅 소프트웨어는 JTAG 장비를 제어하는 기능을 네트워크 서버 형태로 제공한다. 분석 어플리케이션이 바인딩하는 API 계층은 이 서버에 접속하는 TCP 클라이언트 코드로써 디버깅 기능 제어에 대한 요청 명령들을 명령 포맷으로 만들어 전송하고, 요청에 의한 응답 메시지를 해석하여 상위 분석 어플리케이션에 전달하는 구조이다. 반면, 본 연구의 API 구조는 JTAG 제어 모듈들을 통합한 라이브러리 체계로 구성하여 분석 어플리케이션이 JTAG 디버깅 장비 제어를 독립적으로 수행할 수 있도록 제공한다.

JTAG 디버깅 제어 기능들을 구현한 디버깅 제어 라이브러리가 분석 어플리케이션과 직접 바인딩 되어 한 프로세스의 주소 공간에 존재하게 되며, JTAG API 계층은 기존 API처럼 네트워크 코드가 아닌 디버깅 제어 라이브러리에 대한 래퍼(wrapper)코드 형태이다. 따라서 디버깅 활동을 통하여 발생되는 시스템 이벤트들(예, 타겟 시스템 다운이나 중단점 걸림)은 분석 어플리케이션까지 직접적으로 통지가 되며, 분석 어플리케이션은 이에 대한 처리를 위해 핸들러 함수들을 직접 구현하여 처리할 수 있다.

컨트롤 드라이버는 시스템 드라이버로 USB 채널을 이용하여 JTAG 장비의 펌웨어를 제어하는 역할을 한다. 디버깅 제어 라이브러리는 입출력제어 인터페이스(IOCTL)를 이용하여 컨트롤 드라이버를 통해 JTAG 장비를 제어하게 된다.

3.2 API 함수 그룹

구현된 JTAG API는 하부의 디버깅 제어 라이브러리를 다음과 같은 기능별로 6개의 그룹으로 분류된 함수들을 제공한다.

1) 환경 및 옵션 제어

본 그룹의 함수들[Table 1]은 JTAG 장비가 분석 대상 타겟 시스템에 연결한 상태에서, 디버깅을 수행하기 위한 초기 환경 설정 및 시스템 제어 옵션들을 설정하거나 기본적인 환경 정보를 얻는 등의 기능들을 수행한다.

Table 1. Environment and Option Control Group API Functions

API Name	Description
GetAPIManager	Returns an API management instance.
Initialize	Initializes the internal resources of the given API instance.
Terminate	Releases the allocated internal resources of the specified API instance.
SetNotification Type	Enables the specified notification type.
SetNotification Callback	Sets the event handler for the specified notification type.
SetEmulatorType	Sets the connection H/W info (JTAG device or simulator).
DebugInitialize	Initializes the debug control resources.
DebugTerminate	Releases the debug control resources.
DownloadFirmware	Transfers the specified firmware to the attached JTAG H/W.
GetCoreList	Obtains the string list of supported core processes.
SetCoreCount	Sets the number of core processor.
SetBaseCoreIndex	Sets the base core index of SMP core processor.
SetCoreName	Sets the name of core processor.
SetCoreIndex	Sets the debugging target core index of SMP core.
SetJTagClock	Sets the JTAG clock.
SetSystemOption	Sets system options.
SetPrefix	Sets JTAG prefix.
SetAccessPort	Sets access port.
SetBaseAddress	Sets the debugging base address of the core processor.

API를 사용하기 위해서는 분석 어플리케이션은 기본적으로 API 인스턴스를 만들어(GetAPIManager) 디버깅 제어와 관련된 내부 동적 변수에 대한 메모리 할당 및 초기화를 시켜야한다. 그리고 타겟 장비 디버깅 도중 특정한 이벤트들이 발생하였을 때 처리할 수 있는 조건 설정 및 핸들러 함수에 대한 연결을 수행한다. API가 처리하는 이벤트들은 디버깅 도중 중단점에 프로그램 수행이 도달하거나 시스템 호출에 의한 소프트웨어 인터럽트 발생(SWI)되는 소프트웨어적 이벤트들 및 타겟이 다운되거나 코어가 리셋 되는 등의 하드웨어적 현상에 대한 이벤트들을 통지 받을 수 있다.

JTAG 장비에 의한 디버깅을 수행하기 위해서는 타겟 시스템의 프로세스 코어에 대한 설정이 필요하다. JTAG 장비가 지원하는 코어를 사용한다면 기본적으로 코어 이름과 디버깅 베이스 주소를 지정하는 것으로 간단하게 기본 설정을 할 수 있다. 일반적이지 않은 ARM 디버깅 로직을 사용할 때는 프로세서 매뉴얼 등에 나와 있는 상세(Specification) 정보를 참조하여 Prefix 및 접근 포트 등을 설정하여 JTAG 체인이 구성되도록 환경 설정을 수행하여야한다.

SMP 코어를 사용하는 경우 인덱스 번호로 디버깅 하고자 하는 코어를 지정하여 디버깅 한다. 코어 프로세스 리셋(SysReset)이나 Tap Controller 리셋(TRST) 등의 JTAG에서 일반적으로 지원하는 시스템 옵션들도 필요에 따라 SetSystemOption 함수를 이용하여 설정할 수 있다.

Table 2. Coprocessor Access Group API Functions

API Name	Description
ReadCoRegister	Reads from the specified coprocessor register.
WriteCoRegister	Write into the specified coprocessor register.

2) Coprocessor 접근

ARM 코어 기반 시스템에서는 Coprocessor를 통하여 시스템의 환경 설정을 하거나 설정 정보를 확인할 수 있다. 예를 들어, 코어의 부팅 초기화 과정에서 Memory Management Unit(MMU)를 활성화하거나 Endian을 설정할 수 있다. 이러한 제어 설정을 위한 접근은 Coprocessor의 레지스터에 값을 읽거나 쓰는 형태로 지원된다. 코드상으로는 MRC, MCR 명령으로 가능하며, 분석 어플리케이션에서는 Table 2의 함수들을 활용하여 제어할 수 있다.

Table 3. Register Access Group API Functions

API Name	Desciprtion
ReadRegister	Reads the value of the specified general register.
WriteRegister	Writes a value into the specified general register.
ReadRegisterAll	Reads from all the general registers of the current mode.
ReadRegisterAllMode	Reads from all the general registers of the all mode.
ReadFPUSRegister	Reads the value of the specified single precision FPU register.
WriteFPUSRegister	Writes a value into the specified single precision FPU register.
ReadFPUSRegisterAll	Reads from all the single precision FPU register of the current mode.
ReadFPUDRegister	Reads the value of the specified double precision FPU register.
WriteFPUDRegister	Writes a value into the specified double precision FPU register.
ReadFPUDRegisterAll	Reads from all the double precision FPU register of the current mode.

3) 레지스터 접근

ARM 코어에는 기본적으로 R0부터 R15까지의 범용 레지스터에 대한 접근을 할 수 있으며 본 그룹의 API 함수들

[Table 3]을 이용하여 레지스터 값에 대한 접근(읽기 및 쓰기)이 가능하다. ARM의 아키텍처에 의해 R8 이상의 레지스터들은 코어 프로세스의 동작 모드(User, System, FIQ, Supervisor, Abort, IRQ, Undefined)에 따라서 다른 값을 가질 수 있으며, 이에 따라 디버깅 세션 시 수행되고 있는 프로그램 문맥상 동작 모드의 정의(CPSR 레지스터 값을 통하여 확인)에 따라서 ReadRegister/WriteRegister API를 통한 값의 읽고 쓰는 물리적 레지스터 대상이 달라질 수 있다. 현재 모드의 레지스터 값을 벡터 형태로 얻으려면 ReadRegisterAll을 사용하고, 모든 모드의 모든 범용 레지스터 값을 확인 하려면 ReadRegisterAllMode 함수를 사용한다.

실수형 연산을 지원하는 아키텍처(Floating Point Architecture, VFP)가 포함된 ARM 코어를 사용하는 경우 FPU 레지스터들이 별도로 존재하며 coprocessor 레지스터(CP10, 0x8)를 이용하여 활성화된다. 레지스터 접근 그룹 API는 이들에 대한 읽고 쓰기를 지원한다.

Table 4. Memory Access Group API Functions

API Name	Description
ReadBytes	Reads from memory with one-byte boundary.
ReadWords	Reads from memory with two-byte boundary.
ReadLongs	Reads from memory with four-byte boundary.
ReadQuads	Reads from memory with eight-byte boundary.
WriteBytes	Writes into memory with one-byte boundary.
WriteWords	Writes into memory with two-byte boundary.
WriteLongs	Writes into memory with four-byte boundary.
WriteQuads	Writes into memory with eight-byte boundary.
Upload	Uploads to the specified address range with the given file.
Download	Downloads the specified address range data into the given file.

4) 메모리 접근

메모리 접근 그룹 함수들[Table 4]은 기본적으로 1, 2, 4, 8 바이트 단위로 메모리 읽기 및 쓰기를 지원한다. 메모리를 접근할 때는 옵션에 따라서 물리 메모리 주소를 그대로 사용하거나 MMU에 의하여 변환된 논리 주소를 사용할 수도 있다. Cortex-A 계열의 타겟 시스템인 경우 코어를 거치지 않고 AHB(Advanced High Performance Bus) 및 APB (Advanced Peripheral Bus)를 통하여 직접 메모리에 접근하는 것도 가능하다.

메모리 업로드 및 다운로드 기능은 타겟 시스템의 메모리 특정영역으로부터 데이터를 가져와서 파일로 저장하거나 반대로 파일로 저장된 데이터를 타겟의 메모리에 적재한다. 현재 본 접근 그룹 함수는 플래쉬(flash) 영역에 대한 접근은 지원하지 않는다.

5) 디버깅 제어

본 그룹의 함수들[Table 5]은 타겟 시스템과 JTAG 장비의 연결을 관리하거나 타겟의 동작 상태를 제어하는 등의 제어를 지원한다. JTAG 장비와 타겟 시스템의 연결은 환경 및 옵션 제어 그룹에서 설정된 시스템 옵션을 적용하여 연결(Up)할 수 있고 현재 타겟의 시스템 상태를 그대로 유지하며 연결(Attach)을 할 수도 있다. 타겟이 연결이 되면 Step 동작을 통하여 한 개의 명령어씩만을 수행시켜가며 메모리나 레지스터 값의 동적 변화 상태를 확인할 수 있다.

수행 코드가 있는 특정 주소에 중단점(Breakpoint)을 설정하여 타겟을 동작(Run) 상태로 전환하면 프로그램 카운터(pc)가 해당 주소에 도달하여 명령이 수행되기 직전 수행이 자동 중단하여 중단된 상태의 실행 문맥 정보를 확인할 수 있다. 본 API의 중단점 설정함수(InstallBreakPoint)는 SW 및 HW기반 중단점을 지원한다. SW기반 중단점의 경우 RAM영역에 임의 개수의 중단점을 설정할 수 있으며, HW 기반 중단점을 사용하는 경우 코어의 내부 중단점 레지스터를 사용하므로 해당 레지스터 개수만큼 중단점을 설정할 수 있다. 중단점은 해당 코드의 수행뿐만 아니라 읽기 및 쓰기 행위에 대한 타입으로도 지정하여 관찰점(Watchpoint) 형태로도 사용할 수 있다.

Table 5. Debugging Control Group API Functions

API Name	Description
SystemUp	Connects to the target after applying system options.
SystemDown	Disconnects with the target.
SystemAttach	Connects to the target without applying system options.
SystemDetach	Disconnects with the target without any reset operations.
SystemGo	Connects to the target and changes to RUN state.
GetDebugState	Returns the current debugging state.
IsARMMode	Returns true if the specified address code is in ARM mode.
IsThumbEEMode	Returns true if the specified address code is in ThumbEE mode.
GetCurrentPC	Returns the current PC value.
SetCurrentPC	Replaces the current PC value with the specified one.
GetCurrentCPSR	Returns the current CPSR value.
Step	Performs a single step operation.
Run	Changes to RUN state.
Stop	Changes to Stop state.
ResetBreakPoint	Deletes the specified breakpoint.
InstallBreakPoint	Installs a new breakpoint.
DeleteBreakPoint	Deletes all the breakpoints.

6) 심볼 제어

심볼 제어 그룹 함수들[Table 6]은 동적 분석 중에 특정 주소 위치에 대한 스트링 형태의 심볼을 연관 시켜준다. 심볼 정보는 주로 함수나 변수 이름들을 참조할 때 사용한다. 이러한 심볼 정보를 활용하려면 디버깅 정보가 포함된 이미지가 필요하다. 현재 본 API에서는 DWARF Version 2[22]에 준하는 디버깅 정보가 포함된 이미지를 지원하며 gcc에서는 -gdwarf-2옵션으로 이를 지원한다. 운영체제 분석과 같이 커널 및 동적 라이브러리 등 여러 개의 이미지들에 포함된 심볼 정보를 로드하여 분석하는 경우 각 이미지를 ID로 구분하여 개별적으로 심볼 정보를 적재하거나 제거할 수 있다.

분석 환경에서 소스코드의 활용이 가능하고 심볼 이미지에 소스코드 정보가 포함되어 빌드된 경우 특정 주소에 소스코드 라인 및 소스코드 경로까지 연관시켜 참조할 수 있다.

Table 6. Symbol control group API functions

API Name	Description
LoadImage	Loads the specified image file which includes ELF, DWARF symbols.
UnloadImage	Deletes the symbols of the specified image.
UnloadImageAll	Deletes all the symbols.
GetLastProgramID	Returns the ID of the recently loaded image.
GetSymbolAddress	Returns the address of the specified symbol.
GetSymbolName	Returns the name string of the specified address.
GetLineNumber	Returns the source code line of the specified address.
GetSourceFile	Returns the source file name string of the specified address.

3.3 구현환경

JTAG API를 구현하는 것은 사용하는 JTAG 장비에 종속된다. 본 연구에서는 ARM계열의 Cortex A/R/M 프로세스 코어들을 지원하는 CodeViser 장비[15]를 대상으로 구현하였다. API 라이브러리 체계 및 다음 장에서 소개될 분석 도구 어플리케이션은 윈도우즈 XP 및 7에서 Visual C++ 2005/2008 환경에서 구현되었다. 실험 환경 구성은 Figure 4와 같다.

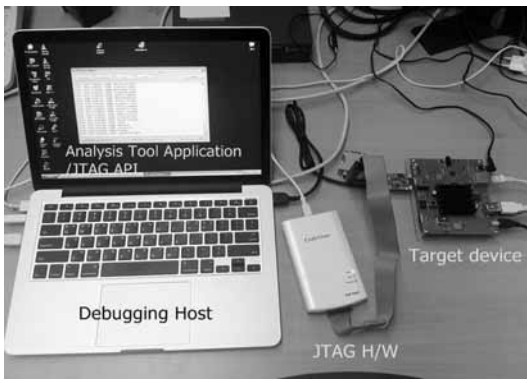

Fig. 4. Experiment Environment

4. 예제 분석도구 구현 실험

이 장에서는 전 장에서 설명한 JTAG API를 바탕으로 임베디드 시스템 보안성 분석을 위한 동적 분석 도구 어플리케이션 작성에 활용할 수 있음을 확인하기 위한 예제 도구 구현을 소개한다. 예제 도구로써 동적 수행 정보가 필요한 함수 퍼징 기능과 메모리 포렌식 기능을 각각 구현하였다. 실험 대상 타겟 장비는 Cortex-A9 (Exynos 4210)코어 기반의 ARM 시스템[23]에서 수행하였으며 해당 시스템에는 리눅스 커널 3.0.15버전이 탑재된 안드로이드 시스템(버전 4.0.4)이 설치되어 있다.

4.1 커널 함수 퍼징

퍼징(Fuzzing)은 소프트웨어의 버그를 찾아내기 위하여 입력부분에 정상적이지 않은 값을 포함하여 반복적으로 여러 가지 입력을 시도해 보고 나타나는 결과를 분석하는 방법이다. 본 예제 분석 도구는 안드로이드 시스템이 설치된 시스템에서 리눅스 커널의 sys_perf_event_open 시스템 호출 함수를 대상으로 인메모리 퍼징(In-memory fuzzing) 기법[24]을 적용하는 예제를 구현하였다. 해당 함수는 리눅스 커널 3.10이전 버전의 ARM 코어 기반 시스템에서 특정 입력 대해 크래쉬 되는 버그를 가지고 있다[25].

일반적인 퍼징 과정에서 퍼징 데이터의 입력이 주로 외부적인 파일 입력, 네트워크 데이터 전송, 혹은 프로그램 수행 시 인자 값 입력 등의 방법으로 이루어진다. 반복적인 테스트 세션 구성을 위해서는 복구 과정이 필요한데 퍼징 대상 어플리케이션 혹은 시스템 자체를 리셋하는 경우도 있다. 이와 달리, 인메모리 퍼징 과정에서는 수행 중인 프로그램 메모리상에 퍼징 데이터(여기에서는 랜덤 값)를 직접 주입한다. 테스트의 반복성을 부여하기 위해서 메모리상에서 함수 호출이나 베이직 블록(basic block) 연결 구조 등의 프로그램 제어 흐름을 변경하여 테스트 루프(loop)를 구성하기도 한다. 이 과정을 실현하려면 특정 코드 수행 위치에서의 중단점을 설정하거나 레지스터 및 메모리에 접근할 수 있어야 하며 수행 흐름을 임의적으로 변경하기 위해 프로그램 카운터도 제어할 수 있어야 한다.

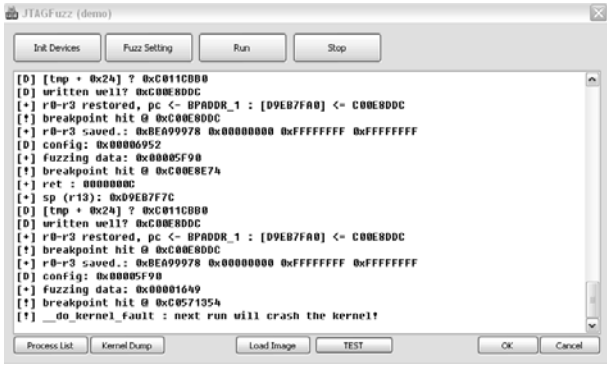

Fig. 5. In-memory fuzzing session driven by the example analysis tool

예제 분석 도구 Figure 5는 본 연구의 JTAG API를 사용하여 다음과 같이 퍼징 과정을 프로그램으로 구현하였다.

① 퍼징 대상 함수(sys_perf_event_open)의 처음과 마지막 명령(리턴) 주소 위치 중단점을 설정한다. 타겟 시스템 구동 중에 프로그램 카운터(pc)가 해당 중단점 위치들에 도달하면 함수 포인터에 의해 설정된 핸들러 함수가 실행된다.

② 함수 처음 명령 주소에 대한 중단점 핸들러 처리 루틴에서는 퍼징에 의한 복구 스냅샷(snapshot)을 구성하기 위하여 함수 인자로 사용되는 레지스터들(r0-r3)의 값들과 함수 인자가 가리키는 스택 내용을 저장한다. 그리고 퍼징 대상 주소 위치에 랜덤 값을 기록한다. 본 실험에서는 sys_ perf_event_open 함수의 첫 번째 인자인 perf_event_attr구조체 중에 config필드에 정수형 랜덤 값을 주입한다.

③ 함수 마지막 명령 주소(리턴위치)의 중단점 핸들러 함수에서는 이전에 복구 스냅샷 정보로써 저장된 레지스터 값들과 스택내용을 복구하고 프로그램 카운터 레지스터 값을 퍼징 대상 함수 처음 위치 주소 값으로 대치한다. 그러면 프로그램 흐름은 다시 반복적으로 함수 호출을 하게 구성 된다.

④ 반복적으로 프로그램이 수행되는 중간에 커널 크래쉬가 발생되는 시점을 인지하기 위해서 본 실험에서는 리눅스 커널의 __do_kernel_fault 함수에 중단점을 설정하여 동적 상태를 관찰하였다.

⑤ 타겟 시스템에서는 sys_perf_event_open 시스템 호출을 1회 호출하는 간단한 사용자 프로그램을 올려 한 번 구동시킨다. 그러면 테스트 루프 구조에 의하여 해당 시스템 호출이 반복적으로 수행되는 인메모리 퍼징 세션이 구동된다.

위의 구성 과정으로 인메모리 퍼징을 시도한 결과, 기 발표된 버그 내용과 관련되는 위치(PMU 관련 코드)에서 커널이 크래쉬(crash)되는 현상을 유도해낼 수 있었다[Figure 6]. 크래쉬를 유발하는 퍼징 값에 대한 유효성을 확인하기 위해

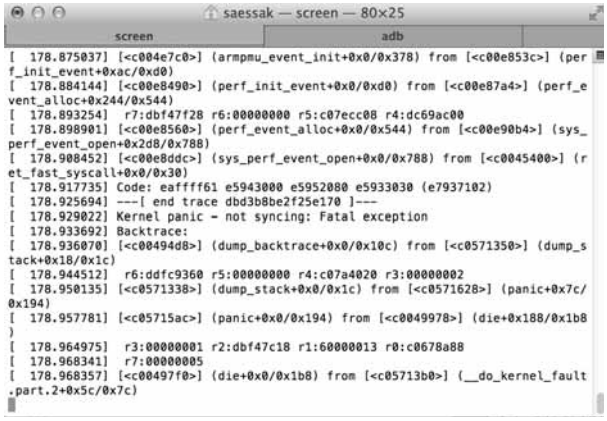

Fig. 6. Kernel crash log (serial terminal)

해당 퍼징 값을 인자 값으로 하여 시스템 콜을 호출하는 프로그램을 작성하였다. 이를 직접 타겟 시스템에서 실행시킨 결과 동일한 크래쉬 결과를 확인할 수 있었다.

4.2 라이브 메모리 포렌식

JTAG API는 타겟 시스템이 구동중인 상태에서의 프로세스 코어의 레지스터 및 메모리 접근이 가능하기 때문에 시스템 중단 없이 수행하는 라이브 메모리 포렌식 도구 구현에 용이하게 활용될 수 있다. 본 연구의 두 번째 예제 동적 분석 도구로 커널 메모리에 대한 라이브 포렌식 기능을 구현하였다.

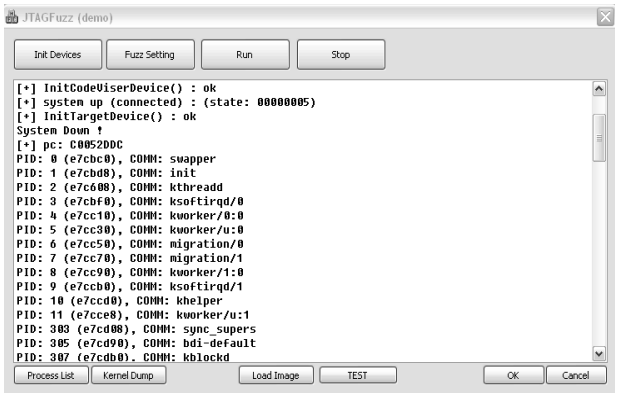

Fig. 7. Kernel data structure information (process list) obtained during system runtime

1) 커널 메모리 자료 구조 분석

포렌식의 첫 번째 테스트는 API를 활용하여 커널 영역의 프로세스 리스트 자료구조를 추출해내는 것이다. 리눅스 커널은 task_struct구조체로 프로세스 정보를 나타내고 이를 링크드리스트 형태로 유지한다. 구조체의 필드 값 분석을 위해 메모리 접근 API 함수로 포인터가 가리키는 메모리상의 데이터를 읽으려면 먼저 포인터 값을 읽고, 이 포인터가 가리키는 데이터를 읽으면 해당 필드 값의 획득이 가능하다. task_struct 링크드리스트 자료구조를 분석하는 프로그

램은 다음 과정과 같이 작성할 수 있다.

① RUN상태의 시스템을 PAUSE 상태로 전환하고, 커널 영역 주소 공간에서 init_task 심볼에 대한 주소를 확인한다.
② init_task가 가리키는 주소 위치에서 task_struct 구조체를 참고하여 필요한 필드 정보를 추출한다. 본 실험에서는 프로세스 ID(pid)와 커맨드 이름(comm)을 추출하여 화면에 출력한다.
③ task_struct의 next필드는 링크드리스트 구조상에서 다음 프로세스에 대한 정보를 가리킨다. 이는 task_struct 상의 처음 주소가 아닌 next필드의 주소 위치를 가리키므로 해당 오프셋만큼 주소를 감해주면 다음 task_struct의 처음 위치 주소를 가리킨다. 그러면 ②와 같이 필요한 정보를 추출한다.
④ 링크드리스트 구조 상 다음 프로세스가 init_task가 될 때까지 ③의 과정을 반복한다. 분석 과정이 종료되면 시스템을 다시 RUN상태로 전환하여 타겟 시스템을 계속 동작시킨다.

위의 과정을 API로 구현한 결과[Figure 7] 현재 구동중인 타겟 장비의 프로세스 정보를 성공적으로 획득할 수 있었다.

2) 외부 분석도구와의 연동

현재 포렌식 용도로 커뮤니티에서 널리 활용되고 있는 오픈소스 기반의 Volatility 프레임웍[26]은 최근 버전에서 ARM 코어 기반 시스템에 대한 지원이 추가되었다. 2.3 베타 버전에서 현재 ARM 기반 리눅스 시스템 포렌식 명령으로 시스템 콜 무결성 검증(linux_check_syscall_arm)과 예외 벡터 테이블(Exception Vector Table)에 대한 변조 가능성 검증(linux_check_evt_arm) 기능을 제공한다. 이러한 커뮤니티 기반 도구와 연동이 되면 앞으로 지원되는 포렌식 기능을 JTAG 기반 분석 도구와 연동시켜 활용할 수 있는 장점이 있다.

두 번째 포렌식 실험에서는 메모리 접근 API를 활용하여 시스템 메모리 덤프를 획득하고 Volatility 포렌식 도구에 적용을 시도하였다. 커널 메모리 영역인 0xC0000000-0xFFFFFFFF 구간 주소영역에 대한 메모리 덤프를 획득하고, 이를 Volatility가 지원하는 lime 덤프 포맷으로 변환하는 루틴을 작성하였다. 포맷 변환 루틴에 의해 처리된 덤프 파일은 Volatility에서 성공적으로 인식되어 해당 도구가 제공하는 포렌식 명령을 활용할 수 있었다[Figure 8].

Fig. 8. The result of applying dump image obtained by JTAG API to Volatility Forensics Framework

4.3 API 활용성 논의

위에서 소개한 예제 분석 도구 구현을 통해서 본 연구에서 구현된 JTAG API를 활용하여 보안성 분석 관련 동적 분석 도구를 용이하게 구현할 수 있음을 확인하였다. 기존 상용 벤더들이 제공하는 API 지원 아키텍처를 개선하고 분석 호스트에서 직접 JTAG 디버깅 하드웨어를 제어하는 C++기반 라이브러리 형태의 API를 제공함으로써 서론에서 언급한 구현 목적에도 다음과 같이 잘 부합됨을 확인할 수 있었다.

① 분석 기능의 확장: 기존 GUI 기반 JTAG 디버거 소프트웨어에서 제공하지 않는 보안성 분석을 위한 동적 분석 기법인 인메모리 퍼징이나 메모리 포렌식을 지원하는 도구를 구현할 수 있었다. 또한 기존 분석 소프트웨어인 Volatility 프레임웍을 활용할 수 있도록 데이터 연동도 가능하였다. JTAG API의 활용은 이외에도 임베디드 소프트웨어의 동적 수행 정보에 기반한 다양한 보안 분석 기법의 적용이나 다른 분석 도구와의 연동을 가능하게 한다.
② 자동화 제어: GUI 기반 디버거 소프트웨어를 이용하는 것보다 API기반의 분석 프로그램을 작성하는 것이 필연적으로 자동화 제어에 유리하다. 퍼징 기법의 경우도 일종의 배치 프로세스에 의한 장시간의 수행이 필요한 경우가 발생한다. API의 활용으로 이러한 자동화 배치 프로세스 과정을 프로그램화 하여 적용할 수 있었다.
③ 리소스 및 성능 고려: 바이너리로 빌드된 분석 도구의 수행은 스크립트 기반 분석 모듈보다 분석 호스트의 리소스 동원에 대한 자유도가 높고 실행 성능 또한 비교적 빠르다. 예를 들어, 동적 오염 기법 분석과 같이 타겟 시스템의 메모리 전체를 쉐도우(shadow) 기법 처리 하는 경우 기가 단위의 메모리 할당이 필요할 수 있는데, C++ 기반 프로그램은 스크립트 기반 확장 모듈보다 이를 수월하게 지원할 수 있다. 또한 본 연구에서 구현된 API 구조[Figure 3]는 상용 벤더들이 제공하는 API 구조[Figure 2]를 단순화 개선하여 성능 향상을 도모하였다. 실제 두 아키텍처의 API 명령 채널 오버헤드를 비교 하기위하여, 연속해서 많은 제어 명령이 내려질 수 있는 step명령을 대상으로 비교하였다. 구현된 API와 실험에 사용된 JTAG 장비의 벤더가 지원하는 API의 step 함수를 각각 1000회 연속하여 수행시켰을 때, 2배 정도의 성능향상이 있음을 확인하였다.

5. 결 론

JTAG 디버거 장비를 활용한 임베디드 시스템 분석은 시스템 소프트웨어 체계의 무결성을 유지한 상태로 분석이 가능한 장점이 있다. 본 논문에서는 이러한 JTAG 기반 분석

의 장점을 보안성 분석에 활용하기 위하여, ARM 코어 기반 임베디드 시스템 프로그램 동적 분석을 위한 JTAG API 구현을 소개하였다. 구현된 API는 JTAG 디버거 하드웨어와 연결된 타겟 시스템을 분석하는데 있어서 JTAG 디버깅 아키텍처에 의한 디버깅 동작을 제어할 수 있는 환경 및 옵션 제어, Coprocessor 접근, 레지스터 접근, 메모리 접근, 디버깅 제어, 및 심볼 제어 등의 제어 기능별 함수 세트를 제공한다.

JTAG 디버거 벤더들이 제공하는 API 아키텍처는 네트워크 채널에 의한 제어 명령을 수행하며 GUI 기반의 디버거 소프트웨어와 연동이 되어야 한다. 본 연구의 JTAG API 완전 독립적으로 분석 호스트에 연결된 JTAG 하드웨어 장비를 직접 제어하는데 있어서 분석 어플리케이션 구현의 유연성과 분석 API의 실행 성능 면에서 보다 유리하다.

구현된 API를 임베디드 시스템 분석에 활용하는데 있어서의 용이성을 확인하기 위하여, 보안성 분석과 관련한 동적 프로그램 분석 도구의 예제 구현들을 제시하였다. 첫 번째 예제 도구 구현에서는 커널 함수에 대한 인메모리 퍼징 기법을 적용하여 배치적 자동화가 필요한 동적 기법이 용이하게 구현될 수 있음을 확인하였다. 또한, 두 번째 예제 구현에서는 커널 시스템 동작 중에 운영체제 커널의 자료구조를 분석하는 라이브 포렌식 기능과 커뮤니티 포렌식 도구인 Volatility와의 연동을 시도하였다. 예제 구현 결과, JTAG API를 활용하여 보안성 분석과 관련된 동적 프로그램 분석 도구를 작성하는데 유용하게 활용될 수 있음을 확인하였다.

6. 감사의 글

본 연구를 수행하는데 있어 기술적 도움을 주신 ㈜제이앤디테크 부설연구소 연구원분들에게 감사의 말씀을 전합니다.

참 고 문 헌

[1] G. Goth, “Addressing the monoculture,” IEEE Security & Privacy, Vol.1, No.6, pp.8-10, 2003.

[2] GDB: The GNU Project Debugger [Internet], http://www.gnu.org/software/gdb/.

[3] IEEE Computer Society, “IEEE Standard Test Access Port and Boundary-Scan Architecture,” IEEE Std 1149.1-2013, 2013.

[4] G. Chawdhary and V. Uppal, “Cisco IOS Shellcode,” BlackHat US, 2008.

[5] J. Zaddach, “Embedded devices’ firmware reversing,” MOCA 2012, 2012.

[6] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi and K. Hazelwood, “Pin: building customized program analysis tools with dynamic instrumentation,” Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI), pp.190-200, 2005.

[7] N. Nethercote and J. Seward, “Valgrind: a framework for heavyweight dynamic binary instrumentation,” SIGPLAN Not., Vol.42, No.2, pp.89-100, 2007.

[8] F. Bellard, “QEMU, a fast and portable dynamic translator,” Proc. of the USENIX 2005 Annual Technical Conference, FREENIX Track, pp.41-46, 2005.

[9] G. Delugre, “Closer to metal: Reverse engineering the Broadcom NetExtreme’s firmware,” HACK.LU 2010, 2010.

[10] S. Muniz and A. Ortega, “Fuzzing and Debugging Cisco IOS,” BlackHat EU, 2011.

[11] N. L. Petroni, Jr., T. Fraser, T. Fraser, and W. A. Arbaug, “Copilot - a coprocessor-based kernel runtime integrity monitor,” Proc. of the 13th conference on USENIX Security Symposium, p.13, 2004.

[12] H. Lee, H. Moon, D. Jang, K. Kim, J. Lee, Y. Paek, and B.B. Kang, “KI-Mon: A Hardware-assisted Event-triggered Monitoring Platform for Mutable Kernel Object,” Proc. of the 22nd USENIX Security Symposium, pp.511-526, 2013.

[13] R. G. Bennetts, “Boundary-Scan Tutorial,” ASSET InterTech, Inc., Version 2.1, 2002.

[14] ARM, “CoreSight Architecture Specification,” ARM IHI0029B, 2005.

[15] JndTech, CodeViser [Internet], http://www.jndtech.com/.

[16] Lauterbach, Trace32 [Internet], http://www.lauterbach.com.

[17] Arium [Internet], http://www.arium.com/.

[18] JTAG Finder [Internet], http://elinux.org/JTAG_Finder.

[19] Joe Gran, “JTAGulator: Assisted discovery of on-chip debug interfaces,” BlackHat US, 2013.

[20] JndTech, “CVD API Guide,” Rev 1.0, 2011.

[21] Lauterbach GmbH, “API for Remote Control and JTAG Access,” 2013.

[22] M. J. Eager, Introduction to the DWARF Debugging Format Introduction to the DWARF Debugging Format, http://dwarfstd.org/.

[23] Hardkernel, ODROID-PC [Internet], http://www.hardkernel.com/.

[24] X. Roussel, “In-Memory Fuzzing with Java,” High-Tech Bridge, 2012.

[25] CVE-2013-4254 [Internet], http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-4254.

[26] The Volatility Framework [Internet], http://code.google.com/p/volatility/.

김 형 찬

e-mail : kimhc@ensec.re.kr
2001년 경북대학교 컴퓨터과학과(학사)
2003년 광주과학기술원 정보통신공학과(석사)
2007년 광주과학기술원 정보통신공학과(박사)
2007년~2008년 Columbia University, 전산과 NSL, Postdoc.
2009년~2010년 일본 정보통신연구기구, Expert Researcher
2010년~현 재 한국전자통신연구원 부설연구소 선임연구원
관심분야 : 운영체제, 가상머신, 임베디드시스템 보안

박 일 환

e-mail : ihpark@ensec.re.kr
1988년 고려대학교 수학과(학사)
1990년 고려대학교 수학과(석사)
1996년 고려대학교 수학과(박사)
1996년~현 재 한국전자통신연구원 부설연구소 책임연구원
관심분야 : 정보보호, 네트워크 보안