ISSN 2383-630X(Print) / ISSN 2383-6296(Online)
Journal of KIISE, Vol. 44, No. 2, pp. 171-178, 2017. 2
https://doi.org/10.5626/JOK.2017.44.2.171

안드로이드 커널 모듈 취약점 탐지를 위한 자동화된 유닛 테스트 생성 기법

(Automated Unit-test Generation for Detecting Vulnerabilities of Android Kernel Modules)

김 윤 호 † 김 문 주 ††
(Yunho Kim) (Moonzoo Kim)

요 약 본 논문에서는 안드로이드 커널 모듈의 취약점을 탐지하기 위한 자동 유닛 테스트 생성 기법을 제안한다. 안드로이드 커널 모듈의 각 함수를 대상으로 테스트 드라이버/스텁 함수를 자동 생성하고 동적 기호 실행 기법을 사용하여 테스트 입력 값을 자동으로 생성한다. 또한 안드로이드 커널 모듈의 함수 포인터와 함수 선행 조건을 고려하지 않은 테스트 생성으로 인한 거짓 경보를 줄이기 위해 정적 분석을 통한 함수 포인터 매칭 기법과 def-use 분석을 사용한 함수 선행 조건 생성 기법을 개발하였다. 자동 유닛 테스트 생성 기법을 안드로이드 커널 3.4 버전의 세 모듈에 적용한 결과 기존에 존재하던 취약점을 모두 탐지할 수 있었으며 제안한 거짓 경보 감소 기법으로 평균 44.9%의 거짓 경보를 제거할 수 있었다.

키워드: 소프트웨어 테스팅, 자동 유닛 테스팅, 동적 기호 실행, 안드로이드 커널 모듈 테스팅

Abstract In this study, we propose an automated unit test generation technique for detecting vulnerabilities of Android kernel modules. The technique automatically generates unit test drivers/stubs and unit test inputs for each function of Android kernel modules by utilizing dynamic symbolic execution. To reduce false alarms caused by function pointers and missing pre-conditions of automated unit test generation technique, we develop false alarm reduction techniques that match function pointers by utilizing static analysis and generate pre-conditions by utilizing def-use analysis. We showed that the proposed technique could detect all existing vulnerabilities in the three modules of Android kernel 3.4. Also, the false alarm reduction techniques removed 44.9% of false alarms on average.

Keywords: software testing, automated unit testing, dynamic symbolic execution, Android kernel module testing

· 본 연구는 한국연구재단 한-아프리카 협력기반조성사업(NRF-2014K1A3A1A09063167) 및 중견연구자지원사업(2016R1A2B4008113), 미래창조과학부 및 정보통신기술진흥센터의 대학ICT연구센터육성 지원사업(IITP-2016-H8501161010120001002) 및 정보통신·방송 연구개발 사업[1711035350, 초소형·고신뢰(99.999%) OS와 고성능 멀티코어 OS를 동시 실행하는 듀얼 운영체제 원천기술 개발]의 지원으로 수행되었음
· 이 논문은 2016 한국컴퓨터종합학술대회에서 '안드로이드 커널 모듈 취약점 탐지를 위한 자동화된 유닛 테스트 생성 기법'의 제목으로 발표된 논문을 확장한 것임

† 학생회원 : KAIST 전산학부
yunho.kim03@gmail.com
†† 종신회원 : KAIST 전산학부 교수(KAIST)
moonzoo@cs.kaist.ac.kr
(Corresponding author임)

논문접수 : 2016년 9월 7일
(Received 7 September 2016)
논문수정 : 2016년 11월 12일
(Revised 12 November 2016)
심사완료 : 2016년 11월 20일
(Accepted 20 November 2016)

정보과학회논문지 제44권 제2호(2017. 2)

1. 서 론

안드로이드 시스템은 전 세계 스마트폰 시장의 80% 이상을 차지하고 있으며[1] 스마트 폰 이외에도 스마트 TV, 시계, 카메라, 자동차 등 다양한 전자 기기에 탑재되고 있다. 이렇게 다양한 시스템에 안드로이드 시스템이 탑재되어 영향력이 커짐에 따라 안드로이드 시스템의 보안 취약점으로 인해 발생할 수 있는 공격 위협 역시 크게 증가하고 있다. 따라서 안드로이드 시스템의 보안 취약점을 탐지하여 공격 위협을 줄이는 것이 중요하다.

안드로이드 커널 모듈의 취약점은 공격자가 전체 시스템을 장악할 수 있게 하기 때문에 가장 위험한 취약점이다. 일반 어플리케이션에 있는 보안 취약점은 공격에 성공하더라도 안드로이드 권한 시스템으로 인해 공격자가 할 수 있는 행동이 제약적이다. 하지만 안드로이드 커널 모듈의 취약점을 공격할 경우 공격자가 안드로이드 권한 시스템에서 가장 많은 권한을 갖고 있는 안드로이드 커널을 직접 조작할 수 있기 때문에 아무 제약 없이 기기를 조작하거나 정보를 유출할 수 있게 된다.

본 논문에서는 안드로이드 커널 모듈의 취약점을 효과적으로 탐지하기 위해 유닛 테스트를 자동으로 생성하는 기법을 제안한다. 안드로이드 커널 모듈의 각 함수를 대상으로 유닛 테스트를 수행하기 위한 테스트 드라이버/스텁 함수를 자동으로 생성하고 동적 기호 실행 기법[2]을 사용하여 테스트 입력 값을 자동으로 생성한다. 또한, 안드로이드 커널 모듈 함수를 테스트할 때 발생할 수 있는 거짓 경보를 제거하기 위한 정적 분석 기법을 개발하여 정확도를 높였다.

본 논문에서 제안하는 기법이 실제 효과적인지 확인하기 위해 안드로이드 커널 3.4 버전에서 발견된 3개 안드로이드 커널 모듈의 취약점을 대상으로 자동 유닛 테스트를 수행하였다. 그 결과 기존에 존재하던 취약점을 모두 탐지하여 취약점 탐지에 효과적임을 확인하였다. 또한 제안한 기법으로 44.9%의 거짓 경보를 제거하여 거짓 경보 감소 기술이 효과적임을 확인하였다.

논문 구성은 다음과 같다. 2장에서는 관련 연구를 설명한다. 3장에서는 동적 기호 실행 기반 자동화된 유닛 테스트 기법을 설명하고 4장에서는 안드로이드 커널 모듈 자동 유닛 테스팅의 거짓 경보를 줄이기 위한 기법을 설명한다. 5장에서는 실험 설정 및 실험 결과를 보여주고 6장에서는 결론 및 향후 연구 방향을 소개하며 논문을 맺는다.

2. 관련 연구

안드로이드 커널 모듈의 취약점을 탐지하는 관련 연구는 찾기 어려우며 대부분 리눅스 커널 모듈을 대상으로 테스트 및 검증을 수행하였다.

Linux Driver Verification 프로젝트[3], Avinux 프로젝트[4], DDVerify 프로젝트[5]에서는 모델 검증 도구를 사용하여 리눅스 디바이스 드라이버 소스 코드를 정형 검증하고 리눅스 커널 모듈의 버그를 검출하였다. Carburizer[6]는 정적 분석 기법을 활용하여 하드웨어 장애가 발생할 수 있는 지점을 탐지하여 하드웨어 장애로 인한 드라이버 오류를 방지하였다. BitBlaze[7]는 가상 머신 기반의 플랫폼으로 소스 코드 없이 바이너리 상태에서 리눅스 시스템을 분석할 수 있으며 바이너리 프로그램의 심볼릭 분석을 통해 보안 취약점을 발견하였다.

본 연구는 안드로이드 커널을 직접 타겟하고 있으며 유닛 테스트 기법을 사용하여 기존 정형 검증이나 전체 시스템을 대상으로 적용하는 기법과 달리 확장성(scalability)이 높아 다양한 종류의 커널 모듈에 대해 취약점을 탐지할 수 있는 장점이 있다.

유닛 테스트 자동 생성 기법은 크게 동적 기호 실행을 사용하여 테스트를 생성하는 기법과 랜덤 메쏘드 순열을 생성하여 유닛 테스트를 생성하는 기법으로 나뉜다. 먼저 동적 기호 실행을 사용하는 기법은 DART[2], CUTE[8], Pex[9], CONBOL[10], Symbolic JPF[11] 등이 있다. DART[2], CUTE[8]는 동적 기호 실행을 사용하여 C 프로그램의 API 함수에 대한 유닛 테스트 케이스를 자동으로 생성하고 Pex[9]는 사용자가 작성한 parameterized 유닛 테스트 드라이버를 사용하여 .Net 프로그램의 유닛 테스트 케이스를 자동 생성한다. CONBOL[10]은 C 프로그램의 유닛 테스트 드라이버/스텁 및 유닛 테스트 케이스를 자동 생성하지만 특정 타입의 타겟 프로그램에 특화되어 있다. Symbolic JPF[11]는 시스템 테스팅을 수행하다가 사용자가 지정한 메쏘드로부터 동적 기호 실행을 적용한 유닛 테스팅을 수행하여 자동으로 테스트 케이스를 생성한다.

랜덤 메쏘드 순열을 생성하는 기법은 Randoop[12], EvoSuite[13], TestFul[14] 등이 있다. Randoop은 Java 프로그램의 메쏘드 순열을 임의로 생성하면서 유닛 테스트 대상 함수의 파라메터 객체를 생성한다. Garg et al.[15]은 Randoop 기법에 동적 기호 실행 기법을 결합하여 분기 커버리지를 높이는 연구를 하였다. EvoSuite[13]은 임의로 생성한 메쏘드 순열에 유전 알고리즘[16]을 적용하여 커버리지를 높였다. TestFul[14]도 유전 알고리즘을 적용하여 테스트 커버리지를 높이는 방법을 사용하였다.

기존 유닛 테스트 자동 생성 기법은 안드로이드 커널과 같은 시스템 프로그램보다 일반 어플리케이션을 대상으로 개발되었기 때문에 안드로이드 커널 모듈에 적

용할 경우 취약점 탐지 능력이 낮고 거짓 경보가 다수 발생할 수 있다. 본 논문에서는 안드로이드 커널 모듈의 취약점을 효과적으로 찾고 거짓 경보를 줄이는 기술을 개발하였다.

3. 동적 기호 실행 기반 자동화된 유닛 테스트 기법

본 장에서는 동적 기호 실행 기법과 동적 기호 실행 기반 자동화된 유닛 테스트 기법을 설명한다.

3.1 동적 기호 실행 기법

동적 기호 실행 기법은 기존 기호 실행 기법의 한계를 극복하기 위해 동적 분석과 결합하여 테스트 케이스를 자동으로 생성하는 기법이다. 먼저 주어진 테스트 대상 프로그램의 실행 과정에서 분기문, 대입문의 정보를 추출하기 위한 탐지 함수를 삽입하는 instrumentation 과정을 수행한 후, 주어진 테스트 케이스를 입력으로 테스트 대상 프로그램을 실행한다. 프로그램 수행 중 실행된 분기문에서 어떤 조건을 만족하였는지 나타내는 경로 제약 조건식을 생성하고 프로그램 실행 종료 후 생성된 경로 제약 조건식을 분석하여 기존에 실행하지 않은 새로운 실행 경로를 테스트할 수 있는 새로운 테스트 케이스를 생성한다. 이 과정을 모든 프로그램 실행 경로를 테스트하거나 사용자가 지정한 종료 조건(실행 시간 제한 등)을 만족할 때까지 반복한다.

그림 1의 예제 프로그램을 사용해 동적 기호 실행 기법의 실행 과정을 더 자세히 살펴보자. 그림 1의 C 프로그램은 세 정수 x, y, z를 입력으로 받아 가장 작은 값을 돌려주는 min() 함수이다. 초기 입력 값이 x=3, y=2, z=1로 주어졌다면 3번째 줄의 if 조건을 만족하지 않아 7번째 줄의 else 문을 실행하게 되고 7번째 줄의 if 조건을 만족하지 않아 9번째 줄의 else 구문을 실행하여 z 값을 리턴한다. 이 때 3, 7번째 줄의 if 구문을 만족하지 않았으므로 경로 제약 조건식 ¬(x<=y) ∧ ¬(y<=z)가 생성된다. 새로운 실행 경로를 실행하는 테스트 케이스를 생성하기 위해 마지막 경로 조건 ¬(y<=z)을 부정해서 새 경로 조건 ¬(x<=y) ∧(y<=z)

```
1 /* x, y, z: symbolic inputs
2 int min(int x, int y, int z){
3 if (x<=y){
4   if (x<=z)
5     return x;
6   else return z;
7 }else if (y<=z)
8   return y;
9  else return z;}
```

그림 1 동적 기호 실행 예제 프로그램

Fig. 1 An example program for dynamic symbolic execution

을 생성하고 제약 조건 해결기(constraint solver)를 사용해서 경로 조건을 만족하는 테스트 케이스 x=3, y=2, z=2를 생성한다. 이 과정을 동적 기호 실행이 종료될 때까지 반복하게 된다.

3.2 동적 기호 실행 기반 자동화된 유닛 테스트 기법

동적 기호 실행 기반 자동화된 유닛 테스팅 기법[10]은 유닛 테스트에 필요한 테스트 드라이버/스텁 함수와 유닛 테스트 케이스 값을 동적 기호 실행 기법을 사용해서 자동으로 생성하는 기법이다. 먼저 정적 분석을 사용하여 테스트 대상 함수가 사용하는 입력 값인 파라메터 변수와 전역 변수의 타입, 이름을 파악하고 해당 파라메터 변수와 전역 변수를 심볼릭 입력으로 생성하고 테스트 대상 함수를 호출하는 테스트 드라이버 함수를 생성한다. 그 후 테스트 대상 함수가 호출하는 함수들의 리턴 타입을 분석하여 해당 타입에 맞는 심볼릭 입력 값을 리턴하는 심볼릭 스텁 함수로 대체한다. 마지막으로 생성한 테스트 드라이버/스텁, 타겟 함수에 동적 기호 실행 기법을 적용하여 자동으로 유닛 테스트 케이스를 생성한다.

테스트 드라이버 생성 시 테스트 대상 함수 f()의 파라메터 및 전역 변수의 타입에 따라 심볼릭 입력 설정 방법이 달라진다. 표 1은 각 타입 별로 심볼릭 입력 설정 방법을 나타낸다. Primitive 타입 변수의 경우 해당 변수의 타입에 맞는 심볼릭 입력 값을 생성하여 할당하고 배열의 경우 각 원소 타입에 맞는 심볼릭 입력 값을 생성하여 배열의 모든 원소에 각각 할당한다. 구조체의 경우 각각의 필드의 타입에 따라 심볼릭 입력 값을 설정한다. 포인터 타입의 경우 먼저 해당 포인터 타입의 심볼릭 입력 값을 처음 설정하는 경우 해당 포인터가 가리키는 타입의 크기만큼 메모리를 할당하고 가리키는 타입에 해당하는 심볼릭 입력을 생성하여 포인터가 가리키는 메모리에 할당한다. 이미 이전에 해당 포인터 타입의 심볼릭 입력을 설정한 경우 새로 심볼릭 입력을 설정하지 않고 기존에 생성한 심볼릭 입력 값을 가리키도록 포인터를 설정하여 무한히 많은 심볼릭 입력 값을 생성하는 것을 방지한다.

보안 취약점을 효과적으로 탐지하기 위해선 취약점을 탐지하기 위한 단언문이 필요하다. 유닛 테스트 수행 과정에서 취약점을 효과적으로 탐지하기 위해 포인터 참조 구문, 배열 참조 구문 및 나누기 연산 구문을 실행하기 직전에 널 포인터 참조, 잘못된 배열 범위 참조, 0으로 나누기를 탐지하기 위한 단언문을 삽입하였다.

4. 안드로이드 커널 모듈 자동 유닛 테스팅의 거짓 경보 감소 기법

본 장에서는 동적 기호 실행 기반 자동화된 유닛 테

표 1 동적 기호 실행 입력 설정
Table 1 Symbolic input setting rule

Input type	Symbolic input setting
Primitive type	A unit test driver specifies a symbolic input to an input variable according to the input variable type
Array type	A unit test driver specifies symbolic inputs to all elements according to the element's type
Struct type	A unit test driver specifies symbolic inputs to all fields of the struct type according to each field's type
Pointer type	1) If a symbolic input of a pointer type T is not yet generated, a unit test driver allocates memory space whose size is equal to sizeof(*T) and specifies a symbolic input according to the pointee type. 2) If a symbolic input of a pointer type T is already generated, the unit test driver assigns the existing symbolic input variable to an input variable.

스트 기법을 안드로이드 커널 모듈에 적용할 때 발생할 수 있는 거짓 경보 감소 기법을 설명한다.

4.1 정적 분석을 사용한 함수 포인터 매칭 기법

안드로이드 커널 모듈은 객체 지향 기법을 C로 구현하기 위해서 커널 모듈의 함수를 구조체의 함수 포인터 필드로 선언하여 사용한다. 그림 2는 net/ceph 모듈의 함수 포인터 초기화 코드이다. 1039 번째 줄에서 ceph_connection_operations 구조체 mon_con_ops를 선언하면서 각각의 함수 포인터 필드에 실제 어떤 함수가 대입되는지 정의한다. 그림 2의 1040번째 줄에서 get 함수 포인터는 ceph_con_get 함수를 가리키고 있으며 1041번째 줄에서 put 함수 포인터는 ceph_con_put 함수를 가리킨다. ceph_mon_init이라는 ceph 모듈의 초기화 함수가 호출되면 768번째 줄에서 모듈이 사용할 함수 포인터 구조체를 대입하는 역할을 수행하여 커널 모듈의 함수 포인터를 초기화한다.

테스트 드라이버 함수가 함수 포인터를 성공적으로 초기화하려면 안드로이드 커널 모듈에서 함수 포인터를 어떻게 초기화하는지 분석하고 그에 따라 테스트 드라이버 함수에서 초기화를 수행해야 한다. 먼저 테스트 대상 커널 모듈 소스 코드를 분석하여 함수 포인터를 포함하고 있는 구조체 변수를 정의하는 소스 코드를 찾는다. 그림 2의 1039번째 줄에서 ceph_connection_operations 구조체 타입의 변수 mon_con_ops를 정의하고 있고 해당 구조체는 함수 포인터를 포함하고 있기 때문에 net/ceph 모듈의 소스 코드를 분석하면 그림 2의 1039번째 줄의 구조체 정의를 찾게 된다. 그 후 테스트 드라이버

```
net/ceph/mon_client.c
 748 int ceph_monc_init(struct ceph_mon_client *monc,
                                  struct ceph_client *cl)
…
 768     monc->con->ops = &mon_con_ops;
…
1039 struct ceph_connection_operations mon_con_ops = {
1040     .get = ceph_con_get,
1041     .put = ceph_con_put,
…
1045 };
```

그림 2 안드로이드 커널 net/ceph 모듈의 함수 포인터
Fig. 2 Function pointers of Android kernel net/ceph module

에서 ceph_connection_operations 구조체 변수를 초기화 할 때 1039번째 줄에서 정의한 mon_con_ops 변수를 대입함으로써 테스트 드라이버가 함수 포인터 코드를 정상적으로 초기화할 수 있게 한다.

4.2 Def-use 분석을 사용한 함수 선행 조건 생성 기법

유닛 테스트 드라이버를 자동으로 생성할 때 테스트 대상 함수 f의 선행 조건을 만족시키지 않고 테스트 대상 함수를 호출함으로써 거짓 경보가 발생할 수 있다. 그림 3은 함수 선행 조건을 만족하지 않아 발생하는 거짓 경보 예제이다. 유닛 테스트 드라이버를 자동 생성하여 유닛 테스트를 수행할 경우 net/ceph/osd_client.c의 2123번째 줄에서 널 포인터 참조가 발생할 수 있다는 경보가 발생한다. 테스트 대상 함수인 get_authorizer()가 2123번째 줄에서 con 포인터를 참조하기 전에 NULL 체크를 수행하지 않기 때문에 get_authorizer()의 첫번째 파라메터로 NULL 포인터를 넘긴 테스트 케이스에서 널 포인터 참조를 발생시킨 것이다. 하지만 실제 안드로이드 커널 모듈 실행에서 get_authorizer()는 항상 net/ceph/messenger.c 파일의 get_connect_authorizer() 함수에서만 호출되고 get_connect_authorizer() 함수에서 get_authorizer()함수를 호출할 때(net/ceph/messenger.c의 849라인) 항상 NULL이 아닌 포인터를 파라메터로 넘겨주기 때문에 해당 경보는 거짓 경보가 된다. 이는 테스트 드라이버가 "get_authorizer() 함수를 호출할 때

```
net/ceph/osd_client.c
2120 static struct ceph_auth_handshake *get_authorizer(
2121   struct ceph_connection *con, ……)
2122 {   // False alarm on assert(con!=NULL);
2123     struct ceph_osd *o = con->private;

net/ceph/messenger.c
 836 static struct ceph_auth_handshake
 837 *get_connect_authorizer(struct ceph_connection
*con,
 838                         int *auth_proto) {
 839     struct ceph_auth_handshake *auth;
…
 849     auth = con->ops->get_authorizer(con, ……);
```

그림 3 함수 선행 조건을 만족하지 않아 발생하는 거짓 경보 예제
Fig. 3 An example of false alarm caused by missing pre-conditions

첫번째 파라메터는 NULL이 아니다"라는 선행 조건을 만족시키지 않은 상태에서 get_authorizer()를 호출했기 때문에 발생하는 거짓 경보이다.

자동화된 유닛 테스트의 거짓 경보를 줄이기 위해 def-use 분석을 활용한 함수 선행 조건 생성 기법을 개발하였다. 자동화된 유닛 테스트 수행 시 테스트 대상 함수 f를 테스트 드라이버가 직접 호출하는 것이 아니라 f를 호출하는 f의 caller 함수를 호출함으로써 f의 선행 조건을 생성할 수 있게 하였다. 함수 f를 호출할 때 함수 f의 선행 조건을 만족하는 입력 값을 생성하거나 선행 조건 만족 여부를 체크하는 것은 주로 함수 f를 호출하기 이전에 실행되는 f의 caller 함수들이기 때문에 f의 caller 함수들을 사용해 f를 호출함으로써 선행 조건을 만족한 상태에서 f를 실행할 수 있다. 함수 f의 모든 caller 함수를 호출할 경우 각 caller 함수마다 유닛 테스트를 수행해야 해서 테스트 수행 시간이 길어지고, 선정할 상위 caller 함수의 단계를 제한하지 않으면 main 함수와 같은 프로그램 시작 함수까지 호출하여 테스트 범위가 커지고 유닛 테스트의 효과가 떨어진다. 따라서 def-use 분석을 사용하여 caller 함수와 테스트 대상 함수 f의 의존도를 계산하고 의존도가 큰 caller 함수만 호출하여 테스트 대상 함수 f의 선행 조건 생성에 큰 영향을 줄 수 있는 함수만 선정하였다.

함수 f가 다른 함수 g에 의존하는 정도를 나타내는 의존도 Df(g)는 함수 g에서 정의하고 함수 f에서 사용하는 변수의 def-use 관계를 분석하여 계산하였다. 자세한 의존도 계산 방법은 다음과 같다.

- f에서 사용하는 변수가 primitive 타입인 경우: 함수 g에서 정의하고 f에서 사용한 def-use 쌍의 수를 합산한다.
- f에서 사용하는 변수가 포인터 자체인 경우: primitive 타입과 동일하게 함수 계산한다.
- f에서 사용하는 변수가 포인터 참조인 경우: 먼저 포인터 분석을 통해 p와 동일한 메모리를 가리키는 포인터 q, r 등을 찾고 *q, *r을 def(혹은 use)로 하는 모든 def-use의 수를 계산하여 그 합을 def-use의 수로 계산하였다.

배열의 경우 각각의 배열 원소를 해당 타입에 맞는 개별 변수로 간주하였으며 구조체의 경우 구조체의 각 빌드를 개별 변수로 간주하여 계산하였다.

테스트 대상 함수 f와의 의존도를 사용하여 실제 호출할 함수를 찾는 과정은 다음과 같다. 먼저 유닛 테스트 드라이버가 호출할 시작 함수를 찾는다. 테스트 대상 함수 f에서 시작하여 f의 caller 함수 중 함수 f와 caller 함수의 의존도가 평균 의존도보다 낮은 caller 함수를 만날 때까지 함수 호출 그래프를 탐색한다. 평균

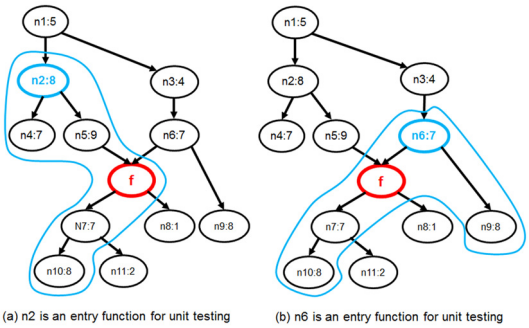


그림 4 서로 다른 2개의 테스트 드라이버에서 실제 호출될 함수 그룹

Fig. 4 Two groups of functions whose code is used in two different test drivers/stubs

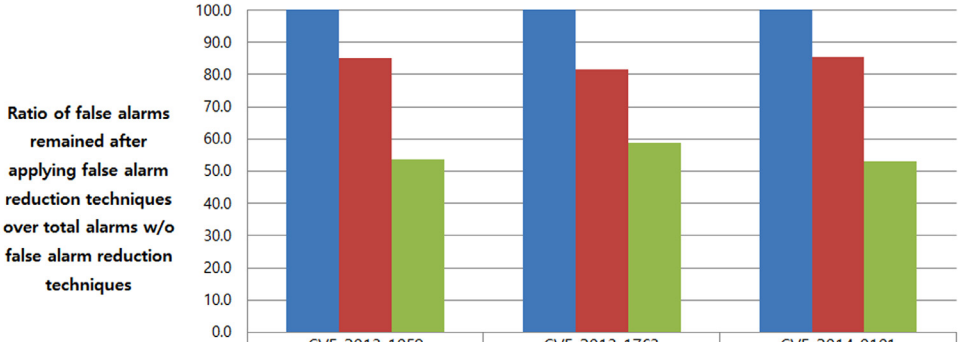
의존도보다 의존도가 큰 가장 상위의 caller 함수 g를 시작 함수로 하고 시작 함수 g에서 출발하여 모든 호출 가능한 함수 정점을 탐색하면서 테스트 대상 함수 f와의 의존도가 평균보다 높은 함수를 찾아 해당 함수를 실제 호출한다.

그림 4의 예제를 살펴보자. 각 정점은 <함수명:테스트 대상 함수 f의 의존도>이고 평균 의존도는 5.7이다. 먼저 (a)의 경우 함수 f에서 왼쪽 caller 함수 n5로 탐색을 시작하는 경우이다. n5와 n5의 caller n2는 평균 의존도보다 의존도가 더 크지만 n2의 caller n1은 의존도가 평균보다 작기 때문에 n2가 시작 함수가 된다. n2가 호출 가능한 함수 중 평균보다 큰 의존도를 갖는 함수는 파란색 실선에 포함되는 함수들이다. (b)의 경우는 오른쪽 caller n6으로 탐색을 시작하는 경우이다. n6의 caller n3는 평균보다 의존도가 작기 때문에 n6이 시작 노드가 되며 n6이 호출 가능한 함수 중 평균보다 의존도가 높은 함수가 파란 실선 안에 포함된다. (a)에서는 n2를 시작함수로 하여 n2, n4, n5, f, n7, n10이 실제 호출되는 함수이며 (b)에서는 n6를 시작 함수로 하여 n6, f, n7, n9, n10이 실제 호출되는 함수이다. 그 외의 함수는 심볼릭 값을 리턴하는 심볼릭 스텁 함수로 대체된다.

5. 실험 설정 및 결과

안드로이드 커널 드라이버 유닛 테스트 자동 생성 도구는 Clang/LLVM 3.4[17] 버전을 사용하여 구현하였다. 동적 기호 실행은 CREST-BV[18]를 사용하여 수행하였다.

본 도구를 사용하여 안드로이드 커널 3.4 버전의 3개 네트워크 모듈 net/ceph(CVE-2013-1059[19]), net/core (CVE-2013-1763[20]), net/sctp(CVE-2014-0101[21]) 모듈에 있는 3개 취약점을 대상으로 테스트를 수행하였다.



	CVE-2013-1059	CVE-2013-1763	CVE-2014-0101
W/O False Alarm Reduction	100.0	100.0	100.0
Function Pointer Matching	85.0	81.6	85.4
+ Pre-condition Generation	53.7	58.7	53.0

그림 5 거짓 경보 감소 기술 적용 후 남은 거짓 경보 비율

Fig. 5 Ratio of residual false alarms after applying false alarm reduction techniques

표 2 타겟 취약점 및 모듈 정보

Table 2 Target vulnerabilities and module information

Vuln.	CVE-2013-1059	CVE-2013-1763	CVE-2014-0101
Module	net/ceph	net/core	net/sctp
Type of Vuln.	NULL pointer dereference	Array out-of-bound access	NULL pointer dereference
# of source files	25	34	28
# of functions	381	487	435
# of branches	1312	4876	3780
Total LOC	11904	33972	34853

CVE-2013-1059 취약점은 net/ceph 네트워크 모듈 초기화 과정에서 잘못된 초기화로 인해 함수 포인터가 NULL로 설정되는 NULL 포인터 참조 취약점이고 CVE-2013-1763은 배열 크기보다 더 큰 값을 참조하는 취약점이다. CVE-2014-0101 취약점은 net/sctp 모듈에서 SCTP 통신을 수행할 때 발생하는 NULL 포인터 참조 취약점이다.

표 2는 테스트 대상 취약점과 모듈의 크기를 보여준다. 모든 실험은 쿼드코어 Core I5-3570K@3.8GHz, 16GB 메모리를 갖는 서버 50대 규모의 클러스터에서 수행하였으며 각 서버당 4개의 유닛 테스트를 동시에 수행하였다. 동적 기호 실행 탐색 기법은 DFS 탐색 기법을 사용하고 각 유닛 테스트마다 5분의 시간 제한을 설정하였다.

테스팅에서 보고된 경보가 참인지 거짓인지는 다음과 같은 기준으로 결정하였다. 해당 취약점을 고치기 위해 다음 패치에서 수정될 프로그램 구문(buggy statement)을 실행하고 단언문이 실패한 테스트의 경우 실제 버그를 찾은 것으로 간주하고 수정된 구문을 실행하지 않고 단언문을 발생시킨 테스트는 거짓 경보로 간주하였다. CVE 취약점 보고를 탐지하는 단언문이 아닌 다른 단언문 실패가 탐지된 경우 아직 보고되지 않은 실제 취약점일 가능성도 있지만 본 실험에서는 거짓 경보로 간주하였다. 실제 취약점인지 판단하기 위해선 각각의 단언문 실패를 전부 분석해야 하나 복잡한 안드로이드 커널 코드에서 발생한 단언문 실패를 전부 분석하는 것은 현실적으로 어렵고 안드로이드 커널과 같이 완성도가 높은 코드의 경우 취약점 발생 가능성이 낮기 때문에 단언문 실패가 보고되지 않은 취약점일 가능성은 낮다.

적용 결과 약 56.6분의 시간이 소요되었으며 세 모듈에서 기존에 보고된 취약점을 모두 발견할 수 있었다. 거짓 경보 제거 결과는 그림 5와 같이 평균 44.9%의 거짓 경보를 제거할 수 있었다. 그림 5의 가로 축은 테스트 대상 취약점을 나타내며 세로축은 거짓 경보 감소 기술 적용 후 남은 거짓 경보 비율을 나타낸다. 파란색은 거짓 경보 감소 기술을 적용하지 않았을 때를 나타내며 빨간색은 함수 포인터 매칭 기술만 적용했을 때, 녹색은 함수 포인터 적용 기술과 선행 조건 생성 기술을 적용했을 때 남은 거짓 경보를 나타낸다. 예를 들어 CVE-2013-1059의 경우 함수 포인터 매칭 기술만 적용 후 15.0%의 거짓 경보가 제거되어 기존 거짓 경보 대비 85.0%의 거짓 경보만 보고되었으며 선행 조건 생성 기술까지 적용하면 46.3%의 거짓 경보가 제거되어 거짓 경보 감소 기술 적용 전 대비 53.7%의 거짓 경보만 보고되었다. 세 모듈 평균 44.9%의 거짓 경보가 제거되고

감소 기술 적용 전 대비 55.1%의 거짓 경보만 보고되었다. 거짓 경보 감소 기술 적용 전에 세 모듈 총합 9024개의 거짓 경보가 발생했으며 기술 적용 후 평균 5014개의 거짓 경보가 발생하였다.

6. 결론 및 향후 연구

본 논문에서는 안드로이드 커널 모듈의 취약점을 찾기 위한 자동 유닛 테스트 기법을 제안하고 커널 모듈 유닛 테스트의 거짓 경보를 제거하기 위한 기법을 제안하였다. 실제 안드로이드 커널 취약점을 대상으로 적용한 결과 효과적으로 취약점을 탐지할 수 있었으며 제안한 거짓 경보 제거 기술은 44.9%의 거짓 경보를 효과적으로 제거할 수 있었다.

향후 연구는 개발 기술을 적용하여 신규 취약점을 발견하여 효과성을 입증하는 방향과 거짓 경보를 추가적으로 제거하여 개발자 분석 시간을 줄이고 사용성을 높이는 방향으로 진행하고자 한다. 특히 안드로이드 메인 커널 모듈이 아닌 각 기기 제조사별로 탑재되는 하드웨어 디바이스 드라이버 모듈의 경우 안드로이드 커널 메인 모듈보다 테스트 및 검증이 부족하여 취약점이 존재할 가능성이 상대적으로 높아 각 제조사별 디바이스 드라이버를 대상으로 하여 추가 연구를 진행할 예정이다. 또한 그 과정에서 발견되는 거짓 경보를 분석하여 거짓 경보를 제거할 수 있는 기술을 추가 개발할 것이다.

References

[1] IDC Smartphone OS Market Share, 2015 Q2, http://www.idc.com/prodserv/smartphone-os-market-share.jsp

[2] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed Automated Random Testing," *Proc. of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation*, pp. 213-223, Jun. 2005.

[3] I. Zakharov, M. Mandrykin, V. Mutilin, E. Novikov, A. Petrenko, and A. Khoroshilov, "Configurable Toolset for Static Verification of Operating Systems Kernel Modules," *Programming and Computer Software*, Vol. 41, No. 1, pp. 49-64, Jan. 2015.

[4] H. Post, C. Sinz, and W. Kuchlin, "Towards Automatic Software Model Checking of Thousands of Linux Modules-A Case Study with Avinux," Vol. 19, No. 2, pp. 155-172, Jun. 2009.

[5] T. Witkowski, N. Blanc, D. Kroening, and G. Weissenbacher, "Model Checking Concurrent Linux Device Drivers," *Proc. of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, pp. 501-504, Nov. 2007.

[6] A. Kadav, M. Renzelmann, and M. Swift, "Tolerating Hardware Device Failures in Software," *Proc. of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pp. 59-82, Oct. 2009.

[7] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "BitBlaze: A New Approach to Computer Security via Binary Analysis," *Proc. of the 4th International Conference on Information Systems Security*, pp. 1-25, Dec. 2008.

[8] K. Sen, D. Marinov, and G. Agha, "CUTE: A Concolic Unit Testing Engine for C," *Proc. of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 263-272, Sep. 2005.

[9] N. Tillmann, and J. De Halleux, "Pex: White Box Test Generation for .NET," *Proc. of the 2nd International Conference on Tests and Proofs*, pp. 134-153, Apr. 2008.

[10] Y. Kim, Y. Kim, T. Kim, G. Lee, Y. Jang, and M. Kim, "Automated Unit Testing of Large Industrial Embedded Software using Concolic Testing," *Proc. of the 28th IEEE/ACM International Conference on Automated Software Engineering*, pp. 519-528, Nov. 2013.

[11] C. S. Pasareanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape, "Combining Unit-level Symbolic Execution and System-level Concrete Execution for Testing NASA Software," *Proc. of the 2008 International Symposium on Software Testing and Analysis*, pp. 15-26, Jul. 2008.

[12] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed Random Test Generation," *Proc. of the 2007 International Conference on Software Engineering*, pp. 75-84, May. 2007.

[13] G. Fraser and A. Arcuri, "EvoSuite: Automatic Test Suite Generation for Object-oriented Software," *Proc. of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, pp. 416-419, Sep. 2011.

[14] L. Baresi, P. L. Lanzi, and M. Miraz, "TestFul: An Evolutionary Test Approach for Java," *Proc. of the 2010 3rd International Conference on Software Testing, Verification and Validation*, pp. 185-194, Apr. 2010.

[15] P. Garg, F. Ivancic, G. Balakrishnan, N. Maeda, and A. Gupta, "Feedback-directed Unit Test Generation for C/C++ Using Concolic Execution," *Proc. of the 2013 International Conference on Software Engineering*, pp. 132-141, May. 2013.

[16] P. McMinn, "Search-based Software Test Data Generation: A Survey," *Software Testing, Verification and Reliability*, Vol. 14, No. 2, pp. 105-156, Jun. 2004.

[17] Clang/LLVM. http://llvm.org

[18] Y. Kim, M. Kim, and Y. Jang, "CREST-BV: An

improved concolic testing technique supporting bitwise operations for embedded software," *Journal of KIISE: Software and Applications*, Vol. 40, No. 2, pp. 90-98, Feb. 2013. (in Korean)
[19] CVE-2013-1059: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-1059
[20] CVE-2013-1763: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-1763
[21] CVE-2014-0101: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0101

김 윤 호

2007년 KAIST 전산학과 학사. 2007년~2009년 KAIST 전산학과 석사. 2009년~2017년 KAIST 전산학부 박사. 관심분야는 자동화된 Concolic 유닛 테스팅, 변이 테스팅, 자동 오류 위치 추정, 정형검증

김 문 주

1995년 KAIST 전산학과 학사. 2001년 Univ. of Pennsylvania 박사 2002년~2004년 SECUi.COM 차장. 2004년~2006년 POSTECH 연구원. 2006년~2012년 KAIST 전산학과 조교수. 2012년~현재 KAIST 전산학부 부교수. 관심분야는 Concolic 테스팅, 자동 오류 위치 추정, Concurrency 테스팅, 변이 테스팅, 정형검증, 내장형 소프트웨어