Journal of The Korea Institute of Information Security & Cryptology ISSN 1598-3986(Print)
VOL.24, NO.3, Jun. 2014 ISSN 2288-2715(Online)
http://dx.doi.org/10.13089/JKIISC.2014.24.3.437

동적 기호 실행을 이용한 그래프 기반 바이너리 코드 실행 경로 탐색 플랫폼*

강 병 호,[1†] 임 을 규[2‡]
[1]한양대학교 컴퓨터·소프트웨어학과
[2]한양대학교 컴퓨터공학부

Graph based Binary Code Execution Path Exploration Platform for Dynamic Symbolic Execution*

Byeongho Kang,[1†] Eul Gyu Im[2‡]
[1]Department of Computer and Software, Hanyang University
[2]Division of Computer Science and Engineering, Hanyang University

요 약

본 논문에서는 그래프 기반의 바이너리 코드 동적 실행 경로 탐색 플랫폼을 제안한다. 바이너리 코드의 조건 분기 명령어를 노드(Node), 그 외의 명령어를 에지(Edge)로 구성된 그래프를 정의하며, 이 그래프를 기반으로 하여 실행 경로 탐색을 수행하는 방안을 제안한다. 실험을 통해 제안하는 그래프 기반 바이너리 코드 실행 경로 탐색 플랫폼의 프로토타입이 실행 경로 탐색을 올바르게 수행함을 확인하였으며, 본 논문에서 제안하는 방안을 통해 소프트웨어 테스팅을 보다 효과적으로 수행하여 소프트웨어 보증, 시큐어 프로그래밍 및 악성 프로그램 분석 등을 보다 효과적으로 수행할 수 있을 것으로 기대한다.

ABSTRACT

In this paper, we introduce a Graph based Binary Code Execution Path Exploration Platform. In the graph, a node is defined as a conditional branch instruction, and an edge is defined as the other instructions. We implemented prototype of the proposed method and works well on real binary code. Experimental results show proposed method correctly explores execution path of target binary code. We expect our method can help Software Assurance, Secure Programming, and Malware Analysis more correct and efficient.

Keywords: Execution Path Exploration, Code Coverage Improvement, Symbolic Execution, Taint Analysis

I. 서 론

소프트웨어는 다양한 개발사의 다양한 컴포넌트를 이용하여 개발된다. 즉 소프트웨어에서 사용되는 모든 컴포넌트에 대한 소스 코드를 구하기 어려우며, 만약 소스 코드를 구한다 하더라도 배포된 컴포넌트와 동일한 바이너리 코드를 가지도록 빌드하기 어렵다[1]. 따

접수일(2013년 11월 8일), 수정일(1차: 2014년 1월 21일, 2차: 2014년 4월 8일), 게재확정일(2014년 5월 19일)
* 본 연구는 미래창조과학부 및 정보통신산업진흥원의 대학 IT연구센터육성 지원사업/IT융합고급인력과정지원사업의 연구결과로 수행되었음(NIPA-2014-H0301-14- 1022)
† 주저자, a@hanyang.ac.kr
‡ 교신저자, imeg@hanyang.ac.kr(Corresponding author)

라서 소스 코드가 아닌 바이너리 코드를 대상으로 소프트웨어 테스팅을 수행해야 올바른 분석을 할 수 있다. 또한 바이너리 코드는 다양한 코드 최적화 기법으로 인해 정적 소프트웨어 테스팅으로는 올바른 결과를 도출해 내기 어려우므로[2], 최근의 소프트웨어 테스팅 연구는 동적 소프트웨어 테스팅에 초점이 맞추어져 있다[3][4][5][6][7][8][9][10][11][12][13][14][15]. 동적 소프트웨어 테스팅은 소프트웨어를 직접 실행시키며 분석을 수행하므로, 한 번의 실행에 하나의 실행 경로를 탐색하게 된다. 하지만 바이너리 코드는 많은 조건 분기 명령어를 포함하고 있기 때문에 이 중 단 하나의 경로만을 탐색한다면 소프트웨어 테스팅이 올바르게 수행되었다고 보기 어렵다. 즉 동적 소프트웨어 테스팅은 다중경로 분석이 필요하다고 볼 수 있다. 보안 취약점 분석을 위한 퍼징(fuzzing)[11][12], 콘콜릭 테스팅(Concolic testing)[8][10][13][14][15], 기호 실행(symbolic execution)[3][4][7] 등 최근에 수행된 많은 소프트웨어 테스팅 연구는 다중경로 분석을 기반으로 한다. 바이너리 코드의 실행 경로는 실행 과정에서 조건 분기 명령어를 만날 때 2개가 생성되므로, 바이너리 코드의 전체 실행 경로 탐색에 대한 경우의 수는 매우 많다고 볼 수 있다. 즉 바이너리 코드의 모든 실행 경로를 탐색하는 것은 비실용적이기 때문에, 기존의 연구에서는 전체 실행 경로 중 일부만을 탐색하였다[3][4][7].

본 논문에서는 그래프 기반의 바이너리 코드 동적 실행 경로 탐색을 제안한다. 바이너리 코드의 조건 분기 명령어를 노드(node), 그 외의 명령어를 에지(edge)로 표현하는 그래프가 정의되며, 이를 기반으로 실행 경로를 탐색한다. 제안하는 실행 경로 탐색 플랫폼을 통해 바이너리 코드 기반의 실행 경로를 보다 효율적으로 탐색할 수 있을 것으로 기대한다.

II. 관련 연구

관련 연구에서는 실행 경로 탐색에 관련된 몇 가지의 선행 연구에 대해 다룬다. 선행 연구 중에서는 본 논문의 제안 방안과 유사한 개념에서 연구된 것 또한 존재함을 미리 알려둔다.

2.1 BitBlaze

BitBlaze[3]는 바이너리 코드 분석 프레임워크의 일종으로, 분석을 바이너리 코드의 디스어셈블에서 직접 수행하지 않고 중간 언어(intermediate language)로 변환시킨 후에 분석을 수행한다는 점에서 특징적이다. 중간 언어를 사용하여 아키텍처의 독립성을 높였고, 암묵적 사이드이펙트(implicit side-effect)로 인한 프로그램 컨텍스트의 변화를 보다 정확하게 판단할 수 있는 장점을 가진다. 이 논문에서는 정적 분석과 동적 분석을 혼합하여 사용하였는데, 먼저 정적 분석기인 Vine으로 바이너리 코드의 분석 토대를 만든 후, TEMU로 입력 값에 영향을 받는 명령어를 테인트하여 도출한다. 이후 Rudder로 기호 실행을 수행하여 실행 경로를 탐색한다. 저자 Dawn Song 등[3]은 BitBlaze로 실행 경로를 탐색하는 과정에서 여러 가지의 취약점을 검색하도록 구성하여 보안 취약점을 발견하였다.

BitBlaze는 본 논문에서 제안하는 방안과 유사하나, 동작 구조가 정적 분석에 의존하는 동적 분석이기 때문에 실행 경로 탐색에 한계가 있으며, 실행 경로 탐색 방안에 대해 명확히 규정해 두지 않았다는 점에서 약점이 있다. 이에 비해, 본 논문에서 제안하는 방안은 동적 분석을 기반으로 분석이 진행되므로 정적 분석에서는 해결하기 어려운 실행 경로 확장이 가능하며, 그래프 기반 탐색을 통해 새로운 경로를 발견하는 방향으로 실행 경로를 탐색하기 때문에 경로의 폭발적 증가 현상에 보다 강인하다는 장점을 가진다.

2.2 S2E

S2E[4]는 앞서 설명한 BitBlaze와 유사한 바이너리 코드 분석 프레임워크이다. 하지만 BitBlaze가 입력 값에 영향을 받는 모든 명령어에 대해 실행 경로를 탐색하는 것과는 달리, S2E는 분석 시간의 단축을 위해 분석 대상 모듈과 분석 대상 코드 영역을 제한할 수 있다. 뿐만 아니라 S2E는 DMA와 하드웨어 인터럽트와 같이, 소프트웨어에 영향을 줄 수 있는 하드웨어 동작까지 고려하여 실행 경로 탐색을 수행한다는 점에서 특징적이다.

2.3 SAGE

Sage[11]는 취약점 분석을 위해 화이트박스 퍼징(white box fuzzing)을 이용하였다. 하지만 퍼징에 사용되는 입력 값의 집합을 줄이기 위해 이전에 실행된 여러 단계의 퍼징에서 실행 경로 확장을 위한 정보를 매 수행시 마다 획득하여, 퍼징 횟수 대비 탐색 실

행경로 수를 늘릴 수 있는 방안을 제안하였다.

2.4 CUTE

CUTE[8]에서는 기호 실행을 효율적으로 수행하는 콘콜릭(Concolic)을 최초로 제안하였다. Concolic은 Concrete과 Symbolic의 합성어로, 필요한 부분에서만 선택적으로 기호 실행을 하여 효율성을 높인 방식이다. 하지만 CUTE가 제안하는 콘콜릭 방법은 소스 코드가 있는 소프트웨어에서 수행되므로 소스 코드가 없는 바이너리 코드의 분석에는 한계가 있다.

III. 바이너리 코드 실행 경로 탐색의 어려움

여기서는 바이너리 코드 실행 경로 탐색의 어려움에 대해 설명한다. 보다 상세하게, 바이너리 코드 실행 경로 탐색에서 발생하는 일반적인 문제점에 대해 설명하며, 문제를 해결하기 위해 기존의 연구에서는 어떤 방법을 사용하였는지를 설명한다. 또한 본 논문에서 제안하는 방법이 기존의 연구와 비교하여 어떻게 향상되었는지를 기술한다.

3.1 경로의 폭발적 증가(Path Explosion)

바이너리 코드의 조건 분기 명령어를 노드, 그 외의 명령어를 에지라 하면, 실행 경로는 시작 노드와 종료 노드를 연결하는 사이클이 있는 방향 그래프로 표현할 수 있다. Stefan Bucur 등[19]은 기호 실행의 병렬화를 주장하는 근거 중 하나로, 실행 경로의 전체 수는 바이너리 크기에 지수적으로 비례하여 증가한다고 설명하였다. 예를 들어 바이너리 코드를 한 번 실행할 때 n개의 조건 분기 명령어가 실행되었다면, 그 바이너리 코드의 실행경로 수는 최서 2^n개가 있을 것으로 추정할 수 있다. 하지만 대부분의 바이너리 코드는 사이클을 포함하기 때문에 실제로는 더 많은 수의 실행 경로가 존재할 것으로 추정할 수 있으며, 바이너리 코드의 모든 실행경로를 탐색한다면 경로의 폭발적 증가(path explosion) 현상, 즉 확장성(scalability) 문제가 발생한다.

기존의 연구에서는 탐색 대상 조건 분기 명령어를 제한하는 방법으로 경로의 폭발적 증가 현상을 완화시켰다[3][4]. BitBlaze[3]에서는 입력 값에 영향을 받는 조건 분기 명령어만을 탐색하였고, S2E[4]에서는 특정 컴포넌트 및 특정 코드 영역에 존재하는 조건 분기 명령어만을 탐색하여 이 문제를 완화시키려 하였다. 본 연구에서는 입력 값에 영향을 받는 조건 분기 명령어만을 탐색하여 경로의 폭발적 증가 현상을 완화시키려 하였다는 점에서는 기존의 연구와 유사하나, 얕은 경로 탐색 문제를 최소화시키기 위해 매 분석 과정에서 새로운 실행 경로를 탐색하도록 바이너리 코드의 그래프를 기반으로 하여 탐색을 수행하였다.

3.2 얕은 경로 탐색(Shallow Path Exploration)

소프트웨어 테스팅에서 널리 사용되는 퍼징(fuzzing)은 소프트웨어의 입력 값으로 실행 경로를 변경시킬 수 있을 것으로 예상되는 값을 선별하여 입력시키는 방법으로 실행 경로 탐색을 수행한다[7][8][11][[13][14][15]. 하지만 퍼징은 실행 경로를 바이너리 코드의 실행 시간에 변경시키는 것이 아닌, 실행 이전에 수집한 자료를 토대로 하여 실행 경로를 변경시킬 수 있을 것으로 예상되는 값을 입력 값으로 주어 실행 경로가 기존과 다르게 실행되도록 유도하는 방법이다. 따라서 입력 값에 따라 실행 경로가 변화가 없을 수도 있으며, 만약 변화가 있다 하더라도 비슷한 실행경로에 계속 맴도는 얕은 경로 탐색(shallow path exploration) 현상이 있다[11]. 기존의 연구는 주로 콘콜릭 테스팅을 이용하여 얕은 경로 탐색 현상을 완화시키려 하였다[8][10][13]. 하지만 콘콜릭 테스팅 또한 바이너리 코드를 실행시키면서 실행경로를 탐색하는 방법이 아닌, 실행경로를 변경시킬 것으로 예상되는 값을 선별하여 입력 값으로 사용하는 퍼징이기 때문에 얕은 경로 탐색 현상을 구조적으로 해결했다고 보기 어렵다.

본 연구에서는 바이너리 코드를 실행하면서 그에 대한 실행 경로 그래프를 생성한다. 실행 경로 탐색은 그래프 기반으로 수행되며, 탐색되지 않은 경로를 우선적으로 탐색한다. 따라서 모든 실행 경로가 탐색되기 전까지는 항상 새로운 실행 경로를 탐색하기 때문에 제안하는 방안은 기존의 방안에 비해 얕은 경로 탐색 현상에 대해 보다 견고하다고 볼 수 있다.

IV. 실행 경로 탐색 플랫폼

제안하는 실행 경로 탐색 플랫폼은 바이너리 코드를 대상으로 하는 그래프 기반의 실행 경로 탐색을 동적 분석을 이용한 기호 실행을 통해 수행하는 특징을 가진다.

4.1 바이너리 코드 그래프

바이너리 코드 그래프는 조건 분기 명령어를 노드로, 그 외의 명령어를 에지로 가지며, 각 에지는 조건 분기 명령어의 참 조건에 해당하는 참 경로(true path)와 거짓 경로(false path)로 구성되어 있는 사이클이 있는 방향 그래프(cyclic directed graph)로 정의한다. 각 노드는 참 경로 한 개와 거짓 경로 한 개의 에지를 가지지만, 종료 노드에서는 에지를 가지지 않을 수 있다. 실행 경로는 시작 노드와 종료 노드를 연결하는 사이클이 있는 방향 부분그래프(cyclic directed sub-graph)로 정의한다.

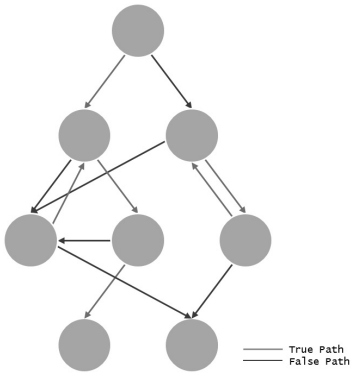

Fig. 1. Binary Code Graph

4.2 그래프 기반 바이너리 코드 실행 경로 탐색

실행 경로 탐색은 바이너리 코드 그래프 중에서 사용자 입력 데이터로부터 영향을 받는 조건 분기 명령어를 선택하여 수행한다. 예를 들어서 입력 값을 사용자 입력 파일 데이터로 가정할 때, 이 데이터에 영향을 받는 조건 분기 명령어를 테인트 분석(taint analysis) 등의 기술을 이용하여 선택하고, 이에 대해서만 실행 경로를 탐색한다. 이와 같이 입력 데이터에 영향을 받는 조건 분기 명령어만을 탐색하는 하는 것은 바이너리 코드의 실행 과정에서 외부로부터 영향을 받는 조건 분기 명령어에 대해서만 실행 경로 탐색을 수행하기 위함이다. 만약 모든 조건 분기 명령어에 대해 실행 경로를 탐색한다면 실행 과정에서 수행될 수 없는 데드코드(dead code)까지도 탐색하기 때문에 불필요한 탐색을 야기할 수 있다.탐색을 수행하여 이러한 문제점들을 완화시킨다.

```
if (false)
    deadcode area
else
    do some work here...
```

Fig. 2. Deadcode Example

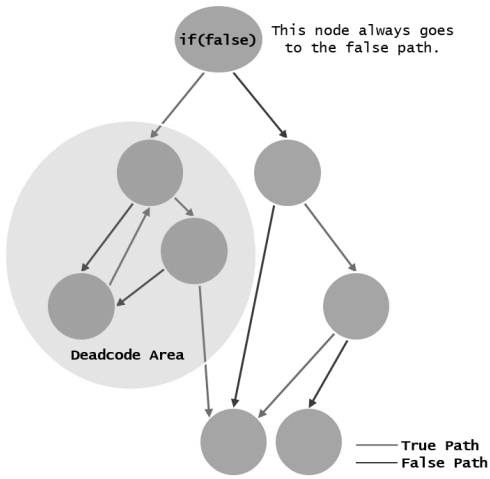

Fig. 3. Deadcode Area

Fig. 2와 Fig. 3에서와 같이 if문의 조건이 항상 거짓이라면 참 조건에서 실행되는 참 경로는 실행이 불가능하며 거짓 조건에서 실행되는 거짓 경로만 실행된다. 즉 참 경로에서만 실행되는 코드는 데드코드로 볼 수 있다. 만약 이러한 데드코드까지 실행 경로를 탐색하고자 할 경우에는 탐색이 되는 대상 조건 분기 명령어가 증가함에 따라 경로의 폭발적 증가 현상 또한 더욱 심하게 발생할 수 있다. 따라서 입력 값에 영향을 받는 조건 분기 명령어, 즉 바이너리 코드의 실행 과정에서 외부로부터 영향을 받는 조건 분기 명령어에 대해서만 실행 경로 탐색을 수행하여 이러한 문제점들을 완화시킨다.

실행 경로의 탐색 과정은 사용자 입력 데이터에 영향을 받는 모든 명령어에 대하여 명령어가 실행되기 전에 기호 실행(symbolic execution)을 수행하며, 조건 분기 명령어에 대하여 명령어가 실행되기 전에 탐색 경로에 대한 프로그램 컨텍스트를 도출하여 반영하는 과정으로 구성되어 있다. 기호 실행은 주어진 바이너리 코드의 수식을 형성하는 과정이다. 따라서 형성된 수식을 이용하여 조건 분기 명령어의 조건 수식

의 참 또는 거짓을 만족하는 해를 도출할 수 있으며, 그 값이 해당 경로, 즉 참 경로 또는 거짓 경로에 해당하는 프로그램 컨텍스트이다. 이 컨텍스트는 해당 경로가 실제로 실행될 때의 값을 가지며, 실제 입력 값과는 다른 분기를 가진다 하더라도 올바른 프로그램 컨텍스트를 유지하고 있기 때문에 입력 값을 변경하여 분기를 변경시킨 것과 같은 결과를 가진다고 볼 수 있다. 실행 경로의 탐색은 위의 과정을 반복하여 더 이상 새로운 경로가 존재하지 않을 때 까지 반복하여 수행한다.

V. 프로토타입

그래프 기반 바이너리 코드 실행 경로 탐색 플랫폼은 x86의 IA-32(Intel Architecture 32-bit) 바이너리 코드를 대상으로 실행시간(run-time)에 실행 경로 탐색을 수행하도록 구성하였다.

5.1 시스템 디자인

그래프 기반 바이너리 코드 실행 경로 탐색 플랫폼의 프로토타입은 Intel Pin[16]의 인스트루멘테이션(instrumentation) 기능을 이용하여 바이너리 코드의 실행시간에 동적 분석을 통해 수행되도록 개발되었다. 사용자 입력 데이터에 영향을 받는 명령어를 선택하는 테인트 분석은 시스템 콜을 후킹하여 테인트 데이터 소스를 탐지하도록 설계하였다. 기호 실행은 IA-32 명령어를 수식으로 변경하며, 각 명령어에 대응하는 수식은 미리 정의하여 사용하였다. 기호 실행 과정에서 탐색 경로를 만족하는 프로그램 컨텍스트를 도출하기 위해 기호 실행 결과로 얻은 수식의 해를 도출하는 과정에서 제약조건 해결기(constraint solver)를 사용하였다. 제약조건 해결기란, 주어진 수식과 조건에서의 만족 여부를 계산할 수 있으며, 방정식의 해를 찾을 수 있는 프로그램이다. 본 논문에서는 널리 사용되고 있는 제약조건 해결기 중 하나로, YICES SMT Solver[17]를 사용하였다. 실행 경로의 정보는 SQLite의 인메모리 데이터베이스(in-memory database)를 이용하여 오버헤드를 줄였다.

프로토타입은 Windows 환경에서 동작하도록 구성되어 있으나, 확장성을 염두에 두어 시스템 콜 기반으로 동작하도록 설계하였기 때문에 Linux, Android, 및 OS X과 같이 다양한 플랫폼에서 동작할 수 있도록 구성되어 있다. 또한 향후 ARM을 비롯한 다양한 아키텍처에 대응할 수 있도록 중간 언어 삽입이 가능한 구조로 설계하였다.

5.2 프로토타입 실험

논문에서 제안하는 방안의 프로토타입을 이용하여 사용자의 입력 값에 따라 서로 다른 결과를 출력하는 작은 소프트웨어에 대해 실험을 진행하였다. 실험에 사용된 소프트웨어의 의사코드(pseudo-code)는 Fig. 4에 표현되어 있다.

```
void foo(int x, int y) {
    printf("%d", x * y);
}

void bar(int x, int y) {
    printf("%d", x / y);
}

int main (void) {
    int x = read();
    int y = read();

    if (x + y > 100)
        foo(x, y);

    else
        bar(x, y);


    return 0;
}
```

Fig. 4. FooBar Program

실험은 BitBlaze와의 비교로 이루어졌으며, 실행 경로의 탐색 정도는 코드 커버리지 측정을 통해 판단하였다. 코드 커버리지는 명령어 커버리지를 사용하였으며, 중복을 제거한 실행 명령어 수로 판단하였다. 동적 분석에서는 프로그램의 전체 명령어 수를 알 수 없기 때문에 상대적인 코드 커버리지 우위 여부 판단을 위해 중복을 제거한 실행 명령어 수를 사용한 것임을 알려 둔다.

Fig. 5의 실험 결과와 같이, 제안하는 방안은 BitBlaze보다 적은 분석 횟수에서 보다 많은 코드 커버리지를 향상시키는 것을 확인할 수 있었다. BitBlaze는 14회의 분석을 통해 실험 프로그램의 코드 커버리지를 향상 시켰으나, 제안하는 방안은 11회

의 분석을 통해 코드 커버리지 향상을 완료하였다.

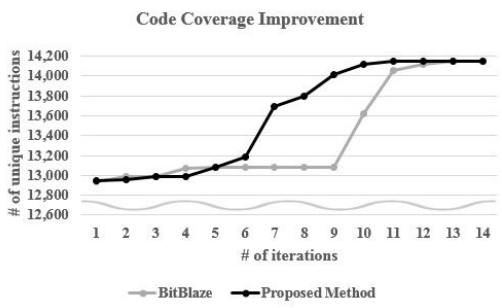

Fig. 5. Experiment Results

5.3 논의

그래프 기반 바이너리 코드 실행 경로 탐색 플랫폼의 프로토타입은 사용자 입력 파일 또는 설정 파일과 같은 입력 파일에 영향을 받는 조건 분기 명령어를 탐색한다. 즉 입력 파일에 영향을 받지 않고, 통신 패킷, 시스템 시간 등의 다른 입력 값에 영향을 받는 조건 분기 명령어는 확장하지 않는다. 하지만 소프트웨어 테스팅에서는 입력 파일만을 입력 값으로 가정하고 우선적으로 고려하는 방안[11]이 일반적이며, 필요시에는 네트워크 플로우와 시스템 시간과 같은 다른 입력 값에 대하여 버퍼 주소를 테인트하거나 해당 API 또는 시스템 콜을 후킹하여 추적이 가능하므로 큰 문제라 판단하기는 어렵다.

하지만 제안하는 방안의 프로토타입은 Intel Pin의 플러그인으로 개발되었기 때문에 Pin VM(Pin Virtual Machine)과 호환되지 않는 소프트웨어에 대해서는 실행 경로 확장이 불가능한 문제가 있으며, Intel Pin이 유저 모드(user mode) 프로그램이기 때문에 분석 대상 소프트웨어의 멀티스레드를 올바르게 제어하지 못하므로 멀티스레드 소프트웨어의 실행 경로 탐색은 제한적으로 가능하다. 이 문제를 해결하기 위해서는 커널 모드(kernel mode)에서 동작하는 분석 또는 QEMU와 같은 가상 머신를 이용하여 실행 경로를 탐색하는 연구가 필요할 것으로 보인다.

VI. 결론

본 논문에서는 그래프 기반의 바이너리 코드 동적 실행 경로 탐색 방안을 제안하였으며, 제안하는 방안의 프로토타입 구현을 보였다. 제안하는 방안은 동적 분석을 통해 수행되며, 실행 경로의 탐색은 테인트 분석을 통해 탐색 대상 명령어를 선택, 기호 실행을 이용하여 대상 명령어에 대한 수식 도출 및 조건 분기문의 탐색 대상 경로에 대한 프로그램 컨텍스트 도출 및 반영을 통해 이루어진다. 실행 경로의 탐색은 모든 노드가 탐색될 때 까지 수행된다. 제안하는 방안의 실험을 통해, 소스 코드가 없는 프로그램을 바이너리 코드만을 이용하여 실행 경로가 확장되는 것을 확인하였다. 본 논문에서 제안하는 방안을 통해 바이너리 코드의 취약점 분석에 도움이 될 것으로 기대한다.

References

[1] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A Few Billion Lines of Code Later: using Static Analysis to Find Bugs in the Real World," Communications of the ACM, vol. 53, no. 2, pp. 66-75, 2010.

[2] A. Moser, C. Kruegel, and E. Kirda, "Limits of Static Analysis for Malware Detection," Proceedings of the 23th Computer Security Applications Conference, pp. 421-430, 2007.

[3] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M.G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "BitBlaze: A New Approach to Computer Security via Binary Analysis," Proceedings of the 4th International Conference on Information Systems Security. Keynote invited paper, 2008.

[4] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A Platform for In-Vivo Multi-Path Analysis of Software Systems," ACM SIGARCH Computer Architecture News, vol. 39, no. 1, pp. 265-278, 2011.

[5] M. Maurer, and D. Brumley, "TACHYON: Tandem Execution for Efficient Live Patch Testing," Proceedings of the 21st USENIX Security Symposium, pp. 617,

2012.

[6] T. Wang, T. Wei, G. Gu, and W. Zou, "TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection," Proceedings of the 31st IEEE Symposium on Security and Privacy, pp. 497-512, 2010.

[7] C. Cadar, D. Dunbar, and D.R. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," Proceedings of OSDI, vol. 8, pp. 209-224, 2008.

[8] K. Sen, D. Marinov, and G. Agha, "CUTE: A Concolic Unit Testing Engine for C," Proceedings of the 5th Joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering, 2005.

[9] C. Cadar, V. Ganesh, P. M. Pawlowski, David L. Dill, and Dawson R. Engler, "EXE: Automatically Generating Inputs of Death," ACM Transactions on Information and System Security, vol. 12, no. 2, pp. 10, 2008.

[10] R. Majumdar, and K. Sen, "Hybrid Concolic Testing," Proceedings of IEEE 29th International Conference on Software Engineering, pp. 416-426, 2007.

[11] P. Godefroid, M.Y. Levin, and D. Molnar, "Sage: Whitebox Fuzzing for Security Testing," Queue, vol. 10, no. 1, pp. 20, 2012.

[12] V. Ganesh, T. Leek, and M. Rinard, "Taint-Based Directed Whitebox Fuzzing," Proceedings of IEEE 31st International Conference on Software Engineering, pp. 474-484, 2009.

[13] J. Bau, E. Bursztein, D. Gupta, and J. Mitchell, "State of the Art: Automated Black-Box Web Application Vulnerability Testing," IEEE Symposium on Security and Privacy, pp. 332-345, 2010.

[14] K. Jayaraman, D. Harvison, V. Ganesh, and A. Kiezun, "jFuzz: A Concolic Whitebox Fuzzer for Java," NASA Formal Methods, pp. 121-125, 2009.

[15] C.Y. Cho, D. Babic, P. Poosankam, K.Z. Chen, E.X. Wu, and D. Song, "MACE: Model-Inference-Assisted Concolic Exploration for Protocol and Vulnerability Discovery," USENIX Security Symposium, 2011.

[16] C.K. 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," ACM SIGPLAN Notices, vol. 40, no. 6, pp. 190-200, 2005.

[17] B. Dutertre, and L.D. Moura, "The Yices Smt Solver," http://yices.csl.sri.com/tool-paper.pdf, pp. 2, 2006.

[18] F. Bellard, "QEMU, A Fast and Portable Dynamic Translator," USENIX Annual Technical Conference, pp. 41-46, 2005.

[19] S. Bucur, V. Ureche, C. Zamfir, and G. Candea. "Parallel Symbolic Execution for Automated Real-World Software Testing." EuroSys 2011. pp. 183-198, 2011.

〈저 자 소 개〉

강 병 호 (Byeongho Kang) 정회원
2013년 2월: 한양대학교 컴퓨터공학부 학사
2013년 3월~현재: 한양대학교 컴퓨터·소프트웨어학과 석사과정
〈관심분야〉 취약점 검사, 소프트웨어 보증, 정보보호

임 을 규 (Eul Gyu Im) 정회원
1992년: 서울대학교 컴퓨터공학과 학사
1994년: 서울대학교 컴퓨터공학과 석사
2002년: University of Southern California, Computer Science Ph.D.
2005년~현재: 한양대학교 컴퓨터공학부 부교수
〈관심분야〉 제어시스템 보안, 악성코드, 정보 보호, 소프트웨어 취약 점검