H-Fuzz: 스냅샷 기반의 실용적인 하이브리드 퍼징

정재영 [1], 이병영 [2]
[1] 서울대학교 전기정보공학부 석박통합과정
[2] 서울대학교 전기정보공학부 교수

jjy600901@snu.ac.kr, byoungyoung@snu.ac.kr

H-Fuzz: A Snapshot-Based Practical Hybrid Fuzzing

Jae-young Chung[1], Byoung-young Lee[2]
[1]Dept. of Electrical and Computer Engineering, SeoulNational University
[2]Dept. of Electrical and Computer Engineering, SeoulNational University

요 약

프로그램의 버그는 해커에 의해 악용될 수 있기 때문에, 이를 사전에 발견하는 것이 매우 중요하다. 최근에는 프로그램의 취약점을 자동으로 찾기 위해 하이브리드 퍼징 기술이 연구되고 있다. 우리는 기존 하이브리드 퍼저들의 한계점인 부족한 확장성을 해결하고자, 스냅샷 기반 하이브리드 퍼저인 H-Fuzz 를 제안한다. H-Fuzz 는 스냅샷 기반 퍼징을 도입하여 하이브리드 퍼징의 확장성 부족 문제를 해결하였다. 그리고 기존 커버리지 기반 퍼저에 비해 H-Fuzz 가 버그를 발견하는데 효과적임을 실험을 통해 확인하였다.

1. 서론

하이브리드 퍼징은 기존의 커버리지 기반 퍼징과 동적 심볼릭 테스팅의 장점을 모두 활용하여, 프로그램 상의 버그를 더 효과적으로 찾아나가는 기술이다. 하지만 하이브리드 퍼징은 확장성이 부족하다는 한계가 있어, 리얼월드 프로그램에 직접적으로 적용하기엔 어려움이 있다. 이러한 문제를 해결하려는 여러 연구들이 있지만, 여전히 실용성 측면에서는 개선이 필요하다.

본 연구에서는 실용적으로 활용 가능한 하이브리드 퍼저인 H-Fuzz 를 제안한다. H-Fuzz 는 프로그램의 특정 지점에서 스냅샷을 찍은 뒤, 해당 스냅샷으로부터 국소적인 범위에 대해서만 하이브리드 퍼징을 적용한다. 우리는 실험을 통해 기존 커버리지 기반 퍼저에 비해 H-Fuzz 가 프로그램의 버그를 찾는데 효과적임을 확인하였다.

2. 배경지식

2.1 퍼징

퍼징은 프로그램에서 버그를 자동으로 찾기 위한 방법론이다. 대상 프로그램에 무작위로 생성한 입력값을 주어서 실행했을 때, 실행 중에 에러가 발생하는지 확인한다. 이 과정을 반복하여 프로그램에 내재되어있는 버그를 자동으로 찾을 수 있다.

2.2 커버리지 기반 퍼징

커버리지는 프로그램이 실행 중에 도달한 코드를 뜻하며, 커버리지 기반 퍼징은 프로그램의 커버리지를 점차적으로 늘리는 것을 목표로 수행하는 퍼징이다. 커버리지 기반 퍼징은 실제로 프로그램의 알려지지 않은 취약점을 찾는데 효과적임이 입증되어있다.[1]

커버리지 기반 퍼징은 프로그램 내에서 서로 다른 실행 경로를 가지는 다양한 입력 값들을 빠르게 찾아나가는데는 매우 효율적이다. 하지만, 복잡한 조건을 만족해야 실행되는 코드에 도달하는 입력 값을 잘 만들어내지 못한다는 한계점이 있다. 예를 들어, 프로그램의 실행 흐름이 *if x == 0xdeadbeef* 구문 내부로 진입하는 입력 값 x 를 만들어내는데 어려움을 겪는다.

2.3 동적 심볼릭 테스팅

기존의 심볼릭 실행은 프로그램의 입력 값을 기호화한 뒤, 특정 지점에 도달하기 위해 만족되어야하는 제약 조건을 풀어내는 방식으로 프로그램의 입력 값을 생성한다. 심볼릭 실행의 변종으로 동적 심볼릭 테스팅(Concolic Testing)이 등장하였다.[2] 동적 심볼릭 테스팅은 실제로 프로그램이 실행되는 경로에 대해서만 심볼릭 실행을 제한적으로 적용하여, 심볼릭 실행의 고질적인 '경로 폭발'문제를 겪지 않으면서도 프로그램이 특정 지점에 도달하기 위한 입력 값을 효과적으로 만들어낼 수 있다.

동적 심볼릭 테스팅에서는 커버리지 기반 퍼징에 비해 복잡한 조건을 만족하는 입력 값을 빠르게 만들어낼 수 있지만, 확장성이 낮다는 점에서 크고 복잡

한 프로그램에 적용하는데 어려움을 겪는다.

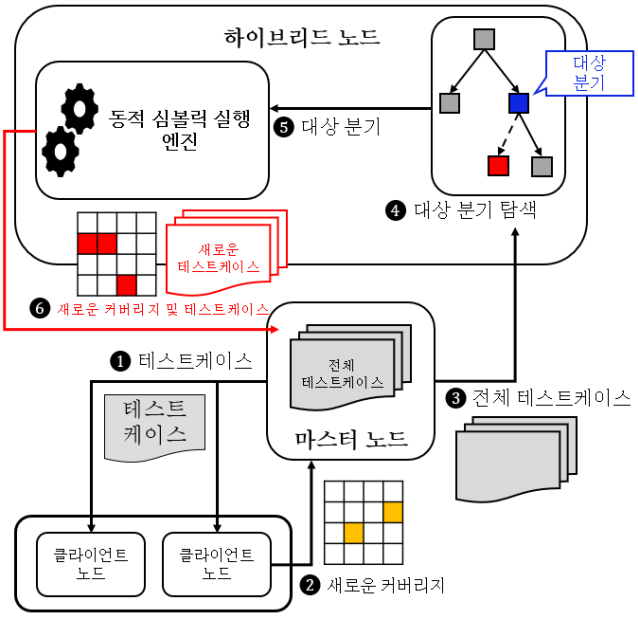
2.4 하이브리드 퍼징

하이브리드 퍼징[3, 4, 5]은 커버리지 기반 퍼징과 동적 심볼릭 테스팅을 함께 사용하여, 두 기술의 장점을 동시에 얻으려는 기술이다. 커버리지 기반 퍼징으로 빠르게 다양한 입력 값들을 찾아나가며, 복잡한 조건의 분기를 만나면 동적 심볼릭 테스팅을 통해 새로운 입력 값을 만들 수 있다.

하이브리드 퍼징은 동적 심볼릭 테스팅에 기반하고 있기에, 큰 프로그램에 적용하였을 때는 확장성 부족 문제가 발생하게 된다. 이를 해결하고자 QSYM[5]에서는 Intel Pin[6]을 기반으로 x86-64 아키텍처에 대해 매우 효율적인 동적 심볼릭 실행 엔진을 구현하였고, 실제로 다양한 유틸리티 프로그램에서 버그를 발견하여 뛰어난 성능을 입증하였다.

3. 본론

본 논문에서는 크고 복잡한 프로그램에 대해서도 실용적으로 적용할 수 있는 스냅샷 기반의 하이브리드 퍼저인 *H-Fuzz* 를 제안한다. QSYM[5]은 x86-64 아키텍처에 특화된 효율적인 동적 심볼릭 실행 엔진을 새로 구현하는 방식으로 확장성 부족 문제를 어느정도 극복하였다. 하지만, 브라우저, 커널 드라이버 등 더욱 크고 복잡한 프로그램에 대해서는 여전히 확장성 부족 문제가 남아있다.



(그림 1) H-Fuzz 구조

H-Fuzz 에서는 동적 심볼릭 테스팅의 확정성 부족 문제를 개선하기 위해, 스냅샷 기반 퍼징을 활용한다. 프로그램의 전체 실행에 대해 동적 심볼릭 테스팅을 적용하면 극심한 성능 저하 및 메모리 사용량 증가 등으로 인해 확장성이 떨어지는 특징이 두드러진다. 우리는 입력 값을 처리하는 특정 함수에만 동적 심볼릭 테스팅을 적용하여도 충분히 효과적이라는 것을 관찰하였다. 이로부터, 프로그램의 실행 중 분석가가 지정한 특정 함수에 도달하였을 때의 전체 메모리를 스냅샷으로 저장한 뒤, 이 스냅샷으로부터 해당 함수에 대해서만 동적 심볼릭 테스팅을 적용하였다. 이러한 방식으로 유저 프로그램, 커널 모듈 등, 맥락에 무관하게 테스팅을 수행하려는 함수 단위로 하이브리드 퍼징을 수행해볼 수 있다.

그림 1 은 H-Fuzz 의 전반적인 구조를 나타내고 있다. H-Fuzz 는 분산화된 퍼징이 가능하며 마스터 노드, 클라이언트 노드, 그리고 핵심적인 하이브리드 퍼징을 수행하는 하이브리드 노드로 구성되어 있다. 마스터 노드는 퍼징 전체 프로세스 내에서 구해지는 모든 테스트케이스들을 유지하며, 이를 변형(Mutation)하여 클라이언트 노드들에게 전달한다(❶). 클라이언트는 받은 테스트케이스를 실행해본 뒤, 새로운 커버리지가 발견된 경우 마스터 노드에게 보고한다(❷). 하이브리드 노드는 마스터 노드로부터 전체 테스트케이스를 공유받고(❸), 모든 테스트케이스를 실행해보며 오직 한 방향으로만 실행된 분기들을 모두 찾는다(❹). 이렇게 찾아진 대상 분기들을 활용하여(❺), 동적 심볼릭 실행 엔진에서는 각 대상 분기에서 반대 방향으로 실행될 수 있는 입력 값을 SMT Solver[7]로 생성해낸다. 하이브리드 노드는 생성된 입력 값으로 프로그램을 스냅샷 지점부터 실행해보며, 이 때 새로운 커버리지가 발견된 경우 마스터 노드에게 해당 테스트케이스 및 증가한 커버리지를 알린다(❻). H-Fuzz 는 이러한 과정을 반복하며 기존 커버리지 기반 퍼징이 도달하지 못한 새로운 커버리지를 발견하는 입력 값들을 생성해낸다.

H-Fuzz 는 오픈소스 분산, 스냅샷, 커버리지 기반 퍼저인 wtf[8]와 동적 심볼릭 실행 엔진인 Triton[9]을 기반으로 구현하였다. wtf 기존 코드에 C++코드 약 2000 줄을 추가하여 Triton API 를 통해 하이브리드 퍼징을 수행하는 하이브리드 노드를 구현하였다.

```
// 1. Challenge for fuzzing
if (x == 0xdead0000) {
    // 2. Challenge for symbolic/concolic execution
    int count = 0;
    for (int i = 0; i < 32; i++) {
        if (buf[i] >='a')
            count++;
    }

    if (count >= 8) {
        // 3. Challenge for fuzzing, again
        if ((x ^ y) == 0x0000beef) {
            ((void(*)())0)(); // 4. bug
        }
    }
}
```

(그림 2) 특정 조건을 만족해야 버그가 발생하는 예제 프로그램

H-Fuzz 의 하이브리드 퍼징의 성능을 검증하기 위해, 간단한 예제 프로그램(그림 2)을 대상으로 실험을 진행하였다. 해당 프로그램은 4 바이트 입력 값으로 *x*, *y* 를 받아서 총 3 가지 조건을 통과해야 널 역참조 버그에 도달할 수 있다. 첫 번째와 세 번째 조건은 기존 커버리지 기반 퍼징으로 통과하기 어렵지만, 심볼릭 실행이나 동적 심볼릭 실행으로 쉽게 통과할 수 있다. 두 번째 조건은 경로 폭발 문제로 심볼릭 실행으로도 통과하기 어렵고, 동적 심볼릭 실행에서도 *count* 자체가 기호화되지는 않아서 조건을 통과하는 입력을 찾아내기 어렵다. 반면 커버리지 기반 퍼징을 여러 번의 재실행을 통해 금방 조건을 만족하는 다양항 입력 값들을 찾아낼 수 있다.

해당 예제는 확장성이 문제가 될 정도로 복잡하지는 않다. 하지만, 이를 큰 프로그램의 중간에 사용되는 코드라고 생각해보면, 전체 프로그램 실행 과정에 하이브리드 퍼징을 적용해야 하므로 확장성 부족 문제가 발생하며 예제 코드에 대해서 원활한 퍼징을 수행하기 어렵게 된다.

H-Fuzz 의 기반 퍼저인 wtf 가 Windows 환경에서만 동작하므로 Windows 의 대표적인 커버리지 기반 퍼저인 WinAFL[10]과 비교해보았다. WinAFL 은 예제 프로그램에서 12 시간 이내에 버그를 찾는데 실패하였는데, 첫번째 조건문을 만족하는 입력 값도 찾아내지 못하였다. 반면, H-Fuzz 에서는 2 분 이내에 버그에 도달하는 입력 값을 찾아내며 하이브리드 퍼징의 성능이 실제로 버그를 찾는데 효과적임을 확인할 수 있었다. 예제 프로그램을 리눅스 환경에서 빌드하여 실험해본 결과, QSYM 도 2 분 이내에 버그를 발견하였다. 예제 프로그램이 기존 하이브리드 퍼징의 확장성 부족 문제를 겪을 정도로 복잡하지 않기 때문에 QSYM 역시 버그를 찾는데 뛰어난 성능을 보였다. 하지만 대규모 프로그램이나 커널 드라이버 등에 하이브리드 퍼징을 적용하는 경우에는 QSYM 에 비해 H-Fuzz 의 스냅샷 기반 퍼징이 큰 이점으로 작용할 것이다.

4. 결론

H-Fuzz 는 스냅샷 기반의 실용적인 하이브리드 퍼저이다. H-Fuzz 는 하이브리드 퍼징에 여전히 남아있는 확장성 부족 문제를 해결하기 위해 프로그램 상의 특정 지점에서 남긴 스냅샷으로부터 동적 심볼릭 실행을 수행한다. 이로 인해 H-Fuzz 는 기존 커버리지 기반 퍼징에 비해서 프로그램의 버그를 찾는데 효과적인 동시에, 리얼월드 프로그램에도 실용적으로 적용할 수 있다.

Acknowledgement

이 논문은 2024 년도 정부(과학기술정보통신부)의 재원으로 정보통신기획평가원의 지원을 받아 수행된 연구임 (No.2020-0-01840,스마트폰의 내부데이터 접근 및 보호 기술 분석)

참고문헌

[1] M. Zalewski, “american fuzzy lop,” http://lcamtuf.coredump.cx/afl/, 2015.

[2] P. Godefroid, M. Y. Levin, and D. A. Molnar, “Automated whitebox fuzz testing,” in Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA, Feb. 2008.

[3] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, “Driller: Augmenting fuzzing through selective symbolic execution,” in Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA, Feb. 2016.

[4] R. Majumdar and K. Sen, “Hybrid Concolic Testing,” in Proceedings of the 29th International Conference on Software Engineering (ICSE), Minneapolis, MN, May 2007.

[5] Yun, Insu, et al. "{QSYM}: A practical concolic execution engine tailored for hybrid fuzzing." 27th USENIX Security Symposium (USENIX Security 18). 2018.

[6] 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,” in Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Chicago, IL, Jun. 2005.

[7] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS’08/ETAPS’08). 337–340.

[8] A. Souchet, “what the fuzz”, https://github.com/0vercl0k/wtf, 2021.

[9] F. Saudel and J. Salwan, “Triton: A dynamic symbolic execution framework,” in Symposium sur la securit ´ e des technologies de l’information ´et des communications, SSTIC, France, Rennes, June 3-5 2015.

[10] Google Project Zero, “WinAFL,” https://github.com/googleprojectzero/winafl, 2016