Journal of The Korea Institute of Information Security & Cryptology ISSN 1598-3986(Print)
VOL.30, NO.5, Oct. 2020 ISSN 2288-2715(Online)
https://doi.org/10.13089/JKIISC.2020.30.5.847

복합 에뮬레이션을 이용한 효율적인 커버리지 가이드 IoT 펌웨어 퍼징 기법*

김 현 욱,[1†] 김 주 환,[1] 윤 주 범[2‡]
[1,2]세종대학교(대학원생, 교수)

Efficient Coverage Guided IoT Firmware Fuzzing Technique Using Combined Emulation*

Hyun-Wook Kim,[1†] Ju-Hwan Kim,[1] Joobeom Yun[2‡]
[1,2]Sejong University(Graduate student, Professor)

요 약

IoT 장비가 상용화되면서 IP카메라, 도어락, 자동차, TV 등 일반 생활기기에 블루투스나 유무선의 네트워크가 내재되어 출시되고 있다. IoT 장비는 네트워크를 통해 많은 정보들을 공유하며 개인적인 정보들을 수집하여 시스템을 가동하기 때문에 IoT 장비에 대한 보안은 더욱 중요해지고 있다. 또한, 현재 사이버 위협 중 웹 기반 공격과 애플리케이션 공격이 상당히 많은 비중을 차지하고 있고, 이를 보안하기 위해 보안 전문가들이 수동 분석을 통해 사이버 공격의 취약점들을 분석하고 있다. 그러나 수동 분석으로만 취약점을 분석하기에는 사실상 불가능하기 때문에 현재 시스템 보안을 연구하는 연구원들은 자동화된 취약점 탐지 시스템을 연구하고 있고, 최근 USENIX에서 발표된 Firm-AFL은 커버리지 기반의 퍼저를 사용하여 퍼징의 처리속도와 효율성에 대해 연구를 진행하여 시스템을 제안했다. 하지만, 기존 도구는 펌웨어의 퍼징 처리속도에 초점을 두고 연구를 진행하다 보니 다양한 경로에서 취약점을 발견하지 못했다. 본 논문에서는 기존 도구에서 찾지 못한 다양한 경로에서 취약점을 발견하고자 변이과정을 강화시켜 기존 도구가 찾은 경로보다 더 많은 경로를 찾고, 제약조건을 해결하며 더 많은 크래시를 발견하는 IoTFirmFuzz를 제안한다.

ABSTRACT

As IoT equipment is commercialized, Bluetooth or wireless networks will be built into general living devices such as IP cameras, door locks, cars and TVs. Security for IoT equipment is becoming more important because IoT equipment shares a lot of information through the network and collects personal information and operates the system. In addition, web-based attacks and application attacks currently account for a significant portion of cyber threats, and security experts are analyzing the vulnerabilities of cyber attacks through manual analysis to secure them. However, since it is virtually impossible to analyze vulnerabilities with only manual analysis, researchers studying system security are currently working on automated vulnerability detection systems, and Firm-AFL, published recently in USENIX, proposed a system by conducting a study on fuzzing processing speed and efficiency using a coverage-based fuzzer. However, the existing tools were focused on the fuzzing processing speed of the firmware, and as a result, they did not find any vulnerability in various paths. In this paper, we propose IoTFirmFuzz, which finds more paths, resolves constraints, and discovers more crashes by strengthening the mutation process to find vulnerabilities in various paths not found in existing tools.

Keywords: Fuzzing, Firmware Emulator, Dynamic Analysis, Embedded Device, IoT(Internet of Things)

Received(07. 14. 2020), Modified(09. 02. 2020), Accepted(09. 03. 2020)

* 본 연구는 과학기술정보통신부 및 정보통신기획평가원의 대학ICT연구센터지원사업의 연구결과로 수행되었음 (IITP-2020-2018-0-01423)

† 주저자, hyunwook711@naver.com
‡ 교신저자, jbyun@sejong.ac.kr (Corresponding author)

I. 서 론

오늘날 IoT(Internet of Things)의 상용화가 이루어지면서 IP카메라, 공유기, 자동차, TV, 도어락 등 많은 IoT 장비들이 생산되어지고 있으며, 각자 다르게 동작하는 IoT 장비들의 하드웨어를 제어하는 펌웨어 또한 많이 생산되어지고 있다. 펌웨어는 장비를 제어하는 소프트웨어이기 때문에 소프트웨어가 갖는 동일한 취약점들을 갖게 된다. 우리에게 밀접하게 다가온 IoT 장비들에 대해서 공격을 당하게 될 경우, 개인적인 피해가 막대하기 때문에 IoT 장비들의 위협[1]에 대응하기 위해 펌웨어 취약점을 빠르게 찾고, 보완해야 한다. 펌웨어들에 대해 취약점을 찾는 연구가 이루어지고 있지만, 펌웨어의 종류가 너무 많아 현재 나온 기술들이 완벽하고 빠르게 펌웨어 취약점을 찾지 못하고 있다.

가장 빠르고 효율적이게 소프트웨어의 알려지지 않은 취약점을 찾기 위한 방법으로 퍼징(Fuzzing)[2] 기법이 있으며, 펌웨어의 취약점을 찾는 기법으로도 퍼징은 많이 사용된다. 하지만, 실제 IoT 장비에 퍼징을 하게 될 경우 처리속도와 실제 장비의 고장 등과 같은 많은 문제점들이 있고, 이러한 문제점들을 극복하기 위해 펌웨어를 실제 장비와 똑같이 동작할 수 있도록 에뮬레이팅하여 퍼징을 시도하는 방법으로 연구가 진행되고 있다.

특히 최근 연구[3][4]는 IoT 장비에 대한 문제점을 극복하기 위해 에뮬레이션 기법을 사용했고, 기존 펌웨어 퍼징의 문제였던, 퍼징 처리속도와 효율성에 대해서는 많은 연구와 실험 끝에 문제점들을 많이 극복했다. 하지만, Firm-AFL[5]은 펌웨어의 특정 바이너리를 퍼징하게 될 경우, 바이너리에 대한 코드 커버리지를 넓히지 못 하고, 특정 블록만을 퍼징한다. 그래서 본 논문에서는 펌웨어를 퍼징하는 기존 연구에서 코드 커버리지의 효율성을 증가시키기 위해 입력값이 변이되어질 때, 퍼저가 변이를 진행할 때 에너지 할당하는 방법과 변이 연산자의 최적의 효율을 찾아 효율이 좋은 변이 연산자가 선택되어지게 하는 IoTFirmFuzz를 제안한다.

본 논문의 구성은 2장에서 논문을 이해하는데 필요한 배경지식, 3장은 IoTFirmFuzz에서 사용한 기법들을 설명하고, 4장에 제안한 기법들이 얼마나 효과적으로 바뀌었는지 평가하고, 끝으로 본 논문에 대한 결론으로 마무리한다.

II. 관련 연구

IoT 펌웨어를 퍼징하는 방법에는 2가지 방법이 있다. 첫 번째는 실제 IoT 장비에 직접 네트워크 포트나, 직렬 포트를 통해 입력값을 보내면서 장비의 결함을 확인하는 방법이 있다. 두 번째는 펌웨어를 에뮬레이션하여 실제 IoT 장비를 흉내낸 에뮬레이터에 퍼징하는 것이다. 하지만, 첫 번째 방법은 실제 IoT 장비 프로세서가 성능이 좋지 않아 처리속도가 느리고, 실제 IoT 장비가 고장날 경우 장비가 어떤 입력값으로 고장난 것인지 퍼징 정보들에 대한 모니터링 같은 문제로 인해 두 번째 방법인 IoT 펌웨어를 에뮬레이팅하는 방법으로 IoT 장비의 취약점을 찾는 연구가 적합하다.

본 장에서는 펌웨어와 IoT 장비를 퍼징하기 위해 필요한 복합 에뮬레이터와 커버리지 기반 그레이박스 퍼저를 살펴본다.

2.1 펌웨어(Firmware)

펌웨어[6][7]는 IoT 장비의 하드웨어들을 특정 목적을 가지고 장비를 작동하게 제어해주는 소프트웨어이다. 펌웨어 이미지에 기본적으로 압축되어있는 요소들은 장비의 다양한 장치와 아키텍처, 명령어 세트, 운영체제, 사용자 지정 구성 요소(펌웨어 버전, 장치 목적 및 제품 코드)들이 대표적이다. 그래서 펌웨어 이미지는 IoT 장비의 공급 업체에 따라 메타데이터가 달라 실질적으로 여러 펌웨어를 일반화하기는 어렵다. 또한, 특정 하드웨어의 아키텍처들은 종종 알려지지 않은 종류가 있어 펌웨어를 분석하고 에뮬레이팅하는 것은 쉽지가 않고, 실제 IoT 장비의 플래시 메모리에서 펌웨어를 획득해 분석하는 것은 오랜 시간이 필요하다.

논문에서는 펌웨어의 호환성과 처리속도를 해결하기 위해 2.2절에서 복합 에뮬레이션으로 두 가지의 한계점을 해결했고, Firm-AFL[5]의 코드 커버리지 측정을 한층 높인 IoTFirmFuzz를 제안한다.

2.2 복합 에뮬레이션

일반적으로 펌웨어의 에뮬레이팅은 QEMU(quick emulator)[8]를 사용하여 에뮬레이션을 진행하는데, QEMU에는 전체 시스템을 에뮬레이팅하는 전체 시스템 모드(system mode) 에뮬레이션이 있

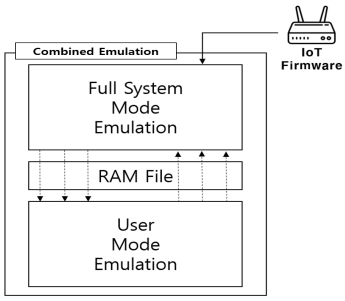


Fig. 1. Combined Emulation

고, 펌웨어의 특정 프로그램이나, 특정 부분을 기본적인 코드변환을 하여 에뮬레이팅할 수 있는 사용자 모드 에뮬레이션(user mode)이 존재한다. 펌웨어를 퍼징하는 이전 연구[5]는 기존의 문제였던, 호환성을 중점으로 연구를 진행하면 퍼징 처리속도에서 발생하고 퍼징 처리속도를 중점으로 연구를 진행하면 호환성에서 발생하는 문제를 QEMU의 두 가지 모드를 함께 적용하여 위와 같은 문제를 동시에 해결하는 연구를 진행했다.

전체 시스템 모드 에뮬레이션은 타켓 IoT 장비를 흉내 내어 IoT 장비와 같이 동작하게끔 한다. 사양이 좋은 PC로 에뮬레이팅을 하여 동작하기 때문에 일반 IoT 장비보다는 실행 부분에서 빠르지만, 전체 시스템 모드에서 퍼징을 실행하면 펌웨어 전체에서 처리하는 부분이 많아 오버헤드가 많이 발생하고 퍼징시에 각종 호출로 인해 퍼징의 처리속도가 느려지는 단점이 있다. 이러한 단점을 보완하고자 복합 에뮬레이션은 전체 시스템 모드에서 펌웨어 전체를 에뮬레이팅을 하고, 사용자 모드 에뮬레이션에서 에뮬레이팅한 일부분의 프로그램을 실행한 메모리 부분을 공유받아 에뮬레이팅하여 실질적으로 적은 프로세스를 처리하여 오버헤드가 적은 사용자 모드에서 퍼징을 실행시키는 방식이다.

Fig. 1은 복합 에뮬레이터[5]의 간단한 구조를 나타낸다. 이는 Decaf-QEMU[9]를 기반으로 firm adyne[10]을 사용해 펌웨어를 에뮬레이팅하며, 서로의 메모리 파일을 공유하며 사용자 모드에서 처리할 수 없는 시스템 호출이 발생할 경우, 메모리를 전체 시스템 모드와 공유하여 시스템 호출을 처리하고, 처리한 시스템 호출의 메모리를 다시 저장해 사용자 모드와 공유하여 퍼징을 계속해서 진행한다. 이전 연구[5]에서 발표한 연구에 따르면 복합 에뮬레이터는 전체 시스템 모드에서 보다 더 높은 처리량을

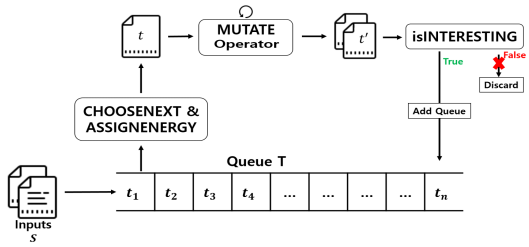


Fig. 2. AFL Input Structure Architecture

보여줬고, 사용자 모드에서 보다 더 많은 IoT 장비에 대한 호환성을 나타냈다. 본 논문에서는 복합 에뮬레이터를 사용하여 AFL의 변이 방식을 개선하였으며, 변경된 변이 방식이 효율성을 4장 평가 부분에서 자세히 설명한다.

2.3 커버리지 기반 그레이박스 퍼저

커버리지 기반 그레이박스 퍼저[5][11][12][13]는 커버리지 정보를 얻기 위해 경량 계측기(lightweight instrument)를 사용한다. 예를 들어 AFL의 계측기는 도달하는 분기별 적중 횟수와 함께 기본 블록을 캡처하며 커버리지를 확인하며 퍼징을 진행한다.

Fig. 2는 커버리지 기반 그레이박스 AFL 퍼저의 입력값 구조 아키텍처를 보여준다. 먼저 입력값 S가 들어오면 큐 T에 저장되고 CHOOSENEXT와 ASSIGNENERGY의 함수에서 입력값을 선택하고 변이가 얼마나 이루어 질지 선택한다. 이후 선택된 입력값에 대해 MUTATE_INPUT 함수에서 변이 연산자를 선택하여 변이가 시작되며 퍼징을 실행한다. 변이된 입력값에 대해 isINTERESTING 함수에서 변이된 입력값에 횟수에 따라 true로 반환되어 T에 저장되고, true가 반환되지 않으면 변이된 입력값은 삭제된다.

III. IoTFirmFuzz 설계

본 논문에서 제안한 IoTFirmFuzz는 IoT 펌웨어 퍼징에 코드 커버리지를 넓히기 위해 입력값 큐 T에서 변이될 입력값 t를 선택하고 몇 번의 변이를 할 것인지에 변이 횟수를 설정하는 기법과 3.1절에서 설명할 AFL의 변이과정 중에서 2단계인 무질서 단계를 거칠 때, t에 대해 변이 연산자가 선택되는 방법에 대한 기능을 강화하여 기존의 Firm-AFL보

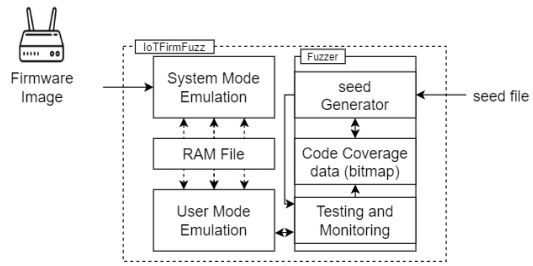


Fig. 3. IoTFirmFuzz Structure

다 경로를 넓히는 효율적인 입력값을 생성하여 많은 크래시를 발견하려는 목적으로 구현하였다.

Fig. 3은 IoTFirmFuzz의 전체적인 구조도를 나타낸다. 가장 처음으로 펌웨어 이미지를 얻어 시스템 모드 에뮬레이션을 통해 에뮬레이팅을 시도하고, 테스트를 진행할 네트워크 프로그램을 실행시켜 RAM file을 생성한다. 그 후, 에뮬레이팅 된 시스템 안에서 사용자가 주는 시드를 넣고 퍼저를 실행한다. 퍼저가 실행될 때, 시스템 모드 에뮬레이션에서 저장된 RAM file을 사용자 모드 에뮬레이션에서 불러드려 프로세스를 에뮬레이팅하고, 사용자 모드 에뮬레이션에서 프로세스를 퍼징한다. 퍼징 중에 사용자 모드에서 처리할 수 없는 syscall이 발생할 경우, RAM file을 갱신하여 시스템 모드로 보내 syscall을 처리하고 사용자 모드로 전환하여 퍼징을 재개한다. 본 논문에서는 Fig. 3과 같이 시드 생성기를 최적화하여 코드 커버리지를 넓혀 더 많은 경로와 크래시를 발견했다.

3.1 변이 연산자 스케줄러

AFL은 Table 1과 같이 11가지 변이 연산자를 미리 정의하는 휴리스틱 기반이고, 그 중에서 하나를 선택하여 실행시간에 입력값을 변화시키며 퍼징을 시도한다. 퍼징을 실행할 때, AFL은 미리 정의된 변이 연산자를 3단계 스케줄링에 따라 변이 연산을 선택한다. 첫 번째, 결정론적 단계 스케줄러(deterministic stage scheduler)이다. 이 단계에서는 AFL에서 선택된 6가지의 변이 연산자를 순서대로 사용하여 입력값을 변이시킨다. 두 번째, 무질서 단계 스케줄러(havoc stage scheduler)는 첫 번째 단계를 통해 변이되었던 입력값이 크래시나 경로 정보를 발견하지 못할 때, 더 많은 변이 연산자를 선택하기 위해 무질서 단계를 거친다. 이 무질서

Table 1. AFL mutation operator types

No	mutation operator	Meaning
1	bitfilp	To flip a single bit or multiple consecutive bits.
2	byteflip	To flip a single byte or multiple consecutive bytes.
3	arithmetic inc/dec	To add or subtract one or several bytes.
4	interesting values	To change the byte of a test case to a byte that has already been defined.
5	user extras	To insert or change a test case byte to a user-supplied value.
6	auto extras	To overwrite bytes in the test case with tokens recognized by the AFL during a bit flip 1/1.
7	random bytes	Randomly select one byte of the test case and set the byte to a random value.
8	delete bytes	Randomly select several consecutive bytes and delete them.
9	insert bytes	Randomly copy some bytes from a test case and insert them to another location in this test case.
10	overwrite bytes	Randomly overwrite several consecutive bytes in a test case.
11	cross over	Splice two parts from two different test cases to form a new test case.

단계에서는 첫 번째 변이 연산자들 가운데 효율이 좋은 연산자가 중복으로 존재하고, 다른 연산자들이 추가된 형태로 8가지 변이 연산자가 존재한다. 세 번째, 스플라이싱 단계 스케줄러(splicing stage scheduler)는 위의 두 가지 단계를 거쳐 크래시나 경로를 발견하지 못할 때 실행되는 단계이고, 스플라이싱 단계에서는 cross over 연산자만 사용하여 변이한다.

앞서 언급한 결정론적 단계는 가장 처음으로 입력값이 거치는 단계이기 때문에 새로 생성된 입력값들은 무조건 거치게 되고, 결정론적 단계에서 크래시나 새로운 경로를 발견하여 입력값이 새로 생성될 경우,

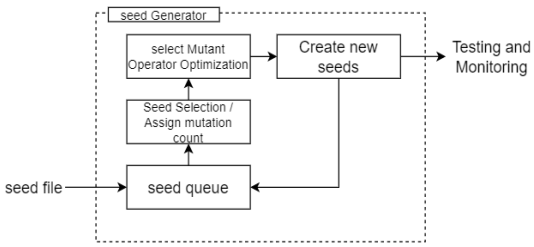


Fig. 4. seed Generator Structure

변이 횟수를 전부 소모해도 다음 무질서 단계로 넘어가지 않는다. 그래서 AFL은 결정론적 단계에서 대부분 시간을 소비하고 마지막 스플라이싱 단계는 거의 사용 되지 않는다. 하지만, MOptAFL[13][14]에 기재된 연구에 따르면 결정론적 단계보다 무질서 단계에서 흥미로운 입력값을 찾는 것이 더 효율적이라는 것을 보여주고, 가장 흥미로운 입력값을 찾아내는 bitflip 1/1, bitflip 2/1은 너무 적은 횟수로 선택되어 비효율적인 모습을 보여준다. 그래서 본 논문은 더 흥미로운 입력값을 생성하는 변이 연산자에 더 많은 시간을 실행하기 위해 PSO 최적화 알고리즘을 기반으로 변이 연산자의 최적 분포를 찾아 퍼저의 효율을 개선한다.

Fig. 4는 시드 생성기의 구조도를 나타낸다. 윗 단락에서 설명한 것처럼 본 논문에서는 시드를 선택하여 변이 횟수를 할당하는 메커니즘과 변이 연산자를 선택하는 메커니즘을 최적화시켜 효율적인 시드를 생성하게 한다. 3.1.1절과 3.2절에서 두 가지 메커니즘에 대해 더 자세히 설명한다.

3.1.1 3PSO 알고리즘 기반 최적의 변이 연산자

PSO(Particle Swarm Optimization) 알고리즘[13][15]은 James Kennedy 등이 제안했으며, 문제에 대한 최적의 해결책을 찾는 것을 목표로 한다. 또한, 구현하는데 있어 적은 비용으로 구현이 가능하여 변이 스케줄링 최적화에 적합하다. 하지만 본 논문에서 AFL의 변이 연산자 스케줄러(mutation operator scheduler)에 PSO 알고리즘을 구현하는 것은 일반적인 PSO 알고리즘을 적용한 것과는 조금 다르다.

입력값마다 변이가 진행되는 변이 연산자들을 모아 하나의 집합을 형성하고 집합 안에서 변이 시간, 실행시간, 새로 찾은 블록이나 크래시를 가지고 집합 가운데서 최적의 변이 연산자의 위치를 탐색하려고 시도한다. 변이 연산자들은 최적의 분포 위치를 분별하기 위해 Local best Position(L_{best}), Global Best Position(G_{bast}), X_{now}로 각 위치를 구별하는데, L_{best}는 하나의 집합에서 변이 연산자 중에 효율이 가장 좋은 것을 가리킨다. 퍼저에 대입해서 말하자면, 현재 퍼징되고 있는 변이 연산자가 아닌, 이전에 퍼징을 시도한 변이 연산자 중 가장 효율이 좋은 변이 연산자를 선택하는 것을 말한다. X_{now}는 다음 퍼징을 시도할 때, 효율이 가장 뛰어나서 선택될 확률이 가장 높은 변이 연산자 가리킨다. G_{bast}는 변이 연산자들의 집합이 여러 가지 집합들 가운데 어느 집합이 최적의 분포를 갖고 있는지 하나의 집합에 대한 위치를 말한다. 이렇게 변이 연산자와 각 집합을 구별하여 변이 연산자의 집합들을 평가하고 전체 집합의 각 변이 연산자의 효율성을 평가하여 최적의 변이 연산자를 찾는다.

요약하면, 최적의 변이 연산자 선택은 입력값들에 대한 변이 연산자의 집합들을 각 집합에 대해 PSO 알고리즘을 적용시켜 각 집합에 변이 연산자에 대해 최상의 위치를 찾아서 L_{best}로 표시한 후 집합들 가운데 최적의 효율을 갖는 집합(G_{bast})을 찾고, 최적의 효율을 갖는 집합 안에서 가장 효율이 좋은 변이 연산자(L_{best})를 다음 변이과정을 시도할 때 선택될 변이 연산자(X_{now})로 선정하여 퍼징을 계속 시도한다. 이 절에서 설명한 알고리즘은 결정론적 단계에서 이루어지지 않는다. 변이 연산자를 차례대로 선택하는 결정론적 단계를 거치고 무질서 단계로 들어온 입력값이 최적의 변이 연산자를 찾기 위해 본 알고리즘이 적용된다.

3.2 변이 횟수를 지정하는 파워 스케줄러

AFL[16]은 위 3.1절에서 말했던 것과 같이 결정론적 단계에서 많은 시간을 소비하고 있다. 이러한 이유로 인해 효율이 좋은 변이 연산자로 오랫동안 퍼징을 진행하지 못하고, AFL의 결정론적 단계로 진입하여 오랜 시간 결정론적 단계에서 정해져 있는 변이 연산자만을 선택하여 퍼징을 수행한다. 만약 타겟 프로그램에 대해서 결정론적 단계의 변이 연산자만 사용하여도 최고 효율의 입력값을 생성한다면 좋겠지만, 타겟 프로그램과 입력값에 따라 효율이 좋은 변이 연산자는 다르기 때문에 본 장에서는 파워 스케줄러(power scheduler)[12]를 사용하여 입력값 큐

(T)에서 다른 입력값(t_n)을 선택할 때, t_n의 변이 횟수를 줄여 결정론적 단계에 적은 시간 머물 수 있도록 적은 에너지를 할당하고, 효율이 좋은 변이 연산자에 더 오래 머물 수 있도록 하는 스케줄러를 제안했다.

파워 스케줄러는 T 안의 t_n의 변이 횟수와 우선순위 $p(i)$를 조정한다. 처음 발표된 그레이박스 퍼저의 퍼징 횟수는 $p(i)=\alpha(i)$였다. $\alpha(i)$는 퍼저가 퍼징을 시도할 때 선택된 입력값에 대한 실행시간, 블록 전환 범위, 또 다른 입력값의 생성시간에 따라 에너지가 할당되었다. 하지만 연구를 계속 진행하면서 한 입력값에 대해 더 많은 퍼징을 시도하기 위해 Cut-off Exponential(COE)[17] 스케줄러를 사용하면서 한 입력값에 대한 퍼징 시간을 기하급수적으로 증가시켰다.

$$p(i)=\begin{cases}0 & \text{if } f(i)>\mu \\ \min\left(\frac{\alpha(i)}{\beta}\cdot 2^{s(i)}, M\right) & otherwise.\end{cases} \quad (1)$$

COE의 계산식의 β는 가장 최근 선택된 t_n에 의해 발견된 경로가 본래에 입력값에서 몇 번을 변이했는지 측정한 수이고, $s(i)$는 퍼징을 하기 위해 입력값 큐(T)에서 입력값이 선택된 횟수, M은 퍼저가 반복하면서 생성되는 입력의 수에 대해 상한을 제한한 것이다. 에너지 할당을 늘리면서 입력값의 변이 시간을 늘려 퍼징을 진행하다 보면, 결정론적 단계에서 너무 많은 시간을 보내게 되기 때문에, 특정 변이 연산자만을 사용하는 문제가 발생한다. 이 문제를 개선하기 위해 COE의 확장을 제안했다.

$$p(i)=\begin{cases}0 \\ \min\left(\frac{\alpha(i)}{\beta}\cdot\frac{2^{s(i)}}{f(i)}, M\right)\end{cases} \quad (2)$$

$f(i)$는 새로 생성된 입력값의 횟수이고, μ는 이전의 입력값들이 탐색한 경로를 변이한 횟수의 평균값이다. $f(i)$는 μ값보다 커야 하고, 분모로 $f(i)$를 적용할 시, 반비례하여 t_n을 선택할 때, 에너지 할당을 적게 주고 퍼징을 시도한다. 이렇게 선택된 t_n은 결정론적 단계에 더 적은 시간을 보낼 수 있고, 결정론적 단계에서 새로운 경로나 크래시를 발견하지 못하면 더 많은 변이 연산자가 있는 다음 단계로 넘어가서 더 효율적인 변이 연산자를 선택할 수 있다.

IV. IoTFirmFuzz 실험 평가

본 4장에서는 Table 2에 나와 있는 퍼저들을 비교한다. 3장에서 제시한 2가지 기법을 적용한 IoTF

Table 2. Functions of the target Fuzzer

	Firm-AFL	AFLfast	AFLgo	IoTFirm Fuzz
mutation-based Fuzzer	O	O	O	O
Power Scheduler	X	O	X	O
Select Mutant Operator Optimization	X	X	X	O
Specific Path Concentration Fuzzing	X	X	O	X

Table 3. Experiment firmware information

Exploit ID	Model	Firmware Version	Device	Program	Firm-AFL			IoTFirmFuzz		
					crashes	Total Paths	new edges on(%)	crashes	Total Paths	new edges on(%)
CVE-2016-1558	DAP-2695	1.11.RC044	router	httpd	36	509	15.5	81	1257	96.83
EDB-ID-38720	DIR-817LW	1.00B05	router	hnap	53	201	13.76	118	547	95.95
CVE-2017-3193	DIR-850L	1.03	router	hnap	41	247	16.2	60	856	94.78
CVE-2018-19240	TV-IP110WN	V1.2.2	camera	network.cgi	185	561	20.95	233	2303	96.38

irmFuzz, 파워 스케줄러만 적용된 AFLfast[20], 특정 경로만을 퍼징하는 AFLgo[18][19], 기본 AFL[16] 4가지의 퍼저들 중에 어느 퍼저가 더 효율적인 입력값을 생성하여 얼마나 많은 경로를 찾고, 제약조건을 해결하며, 크래시를 발견할 수 있는지에 대해 성능을 비교한다.

본 논문의 실험 환경은 cpu i5-6400 2.70GHz에 메모리 8GB이고 운영체제는 Ubuntu 16.04 버전을 사용했고, 실험에 사용된 펌웨어는 Table 3에서 제시한 펌웨어들이다. Table 3은 펌웨어의 특징과 실험을 진행할 펌웨어 안의 바이너리에 대한 정보를 담고 있고, 본 장에서는 펌웨어를 퍼징하고 측정 그래프들을 통해 논문에서 제시한 기법들의 효과를 살펴본다.

4.1 IoTFirmFuzz의 코드 커버리지 평가

Table 3의 동일한 펌웨어로 펌웨어의 특정 바이너리를 정해 실험 대상 퍼저들의 코드 커버리지를 측정했다. 측정 방법은 각 퍼저에 대해 퍼징이 진행될 때 4시간 단위로 퍼저가 찾은 total paths 데이터를 측정하고 퍼저가 찾은 경로마다 얼마나 많은 제약조건을 해결했는지 4시간 별로 퍼저의 new edges on 데이터를 측정했다. new edges on은 퍼저가 찾은 경로에 대한 제약조건을 해결하여 블록 안으로 접근하는 횟수를 말하고, total paths는 퍼저가 실행되는 동안 찾은 전체 경로의 횟수를 말한다.

Fig. 5는 대상 펌웨어들을 퍼저들로 경로를 측정한 그래프이다. Fig. 5의 그래프를 보면 IoTFirmFuzz 다음으로 경로를 많이 찾은 퍼저는 AFLfast이다. 상위의 2개의 퍼저는 하위의 퍼저들 보다 2~3배 정도의 경로를 더 찾았는데, 그 이유는 변이의 횟수를 정하는 파워 스케줄러가 적용되어 있기 때문인 것으로 판단된다. Table 2에서 우수한 성능을 내는 상위 두 개의 퍼저의 공통점은 파워 스케줄러를 가지고 있다는 점이기 때문이다. 3장 초반에 설명했던 AFL의 변이과정에서 AFL기반의 퍼저들은 3단계의 변이과정을 거친다. 하지만, 기본 AFL은 1단계의 결정론적 단계에서 많은 시간을 보내는데, 파워 스케줄러를 적용한 퍼저는 1단계에서 변이하는 실행 시간을 기본 AFL보다 적게 보내고 2단계 무질서 단계에서 더 효율이 좋은 변이 연산자를 선택해 2단계에서 더 오랜 시간 실행된다는 장점으로 인해 하위의 퍼저보다 많은 경로를 찾을 수 있는 걸로 판단된

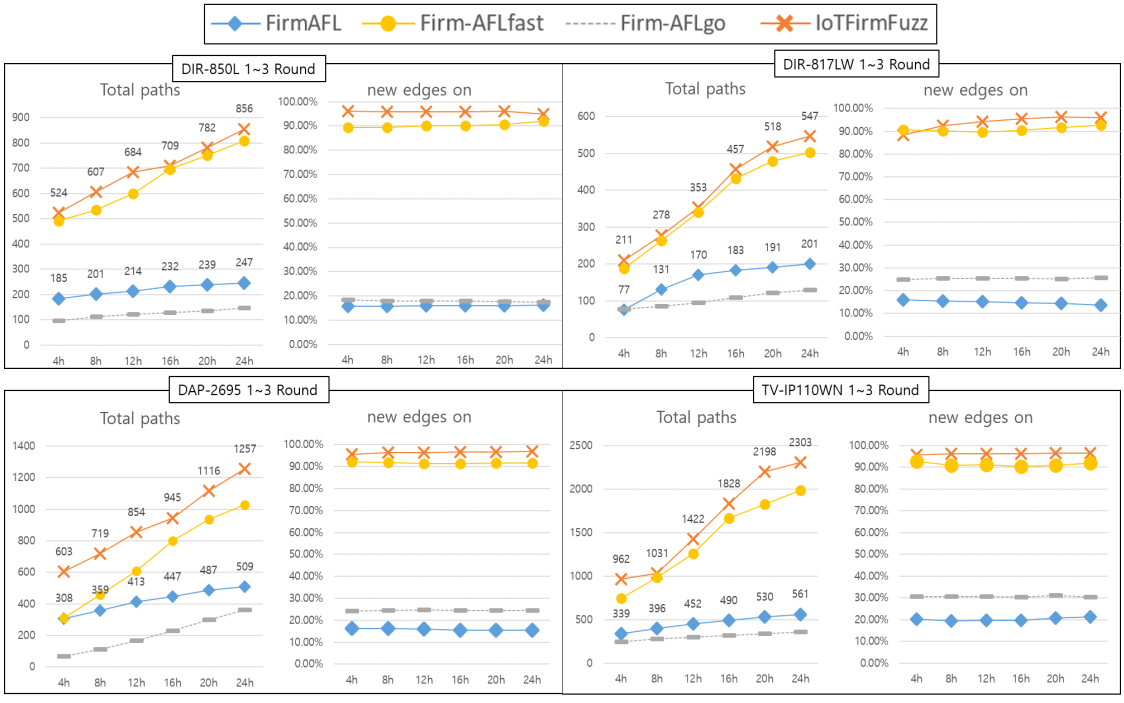


Fig. 5. Code coverage graphs of target firmwares

다. 하지만, 파워 스케줄러가 적용된 상위 2개의 퍼저인 IoTFirmFuzz와 AFLfast는 코드 커버리지 평가 부분에서 차이를 보인다. 그 차이는 2단계의 무질서 단계에 들어와 어떤 방식으로 변이 연산자를 선택하는지 얼마나 효율적인 변이 연산자를 선택하는지에 따라서 다양한 경로를 찾는 것과 제약조건을 해결하는 것이 차이가 나기 시작한다. IoTFirmFuzz는 무질서 단계에 들어와 더 최적화된 변이 연산자를 선택하기 위해 PSO 알고리즘을 사용해 최적의 변이 연산자를 찾아 선택한다. 반면에, AFLfast는 무작위로 2단계 안에서 변이 연산자를 선택하여 사용하기 때문에 AFLfast 보다 IoTFirmFuzz가 더 많은 경로를 찾고 더 많은 제약조건을 해결하는 것으로 판단된다.

4.2 IoTFirmFuzz의 크래시 평가

Fig. 6는 대상 펌웨어들을 퍼저들로 퍼징한 크래시를 측정한 그래프이다. 크래시도 경로와 마찬가지로 본 논문에서 제시한 2가지 기법을 적용시킨 IoTFirmFuzz가 더 많은 크래시를 발견했고, 나머지 퍼저들은 비슷한 크래시를 발견했다.

IoTFirmFuzz가 AFLfast와 비슷하게 경로를 찾았지만, 더 많은 크래시를 발견하였다. 이는 경로 평가에서 IoTFirmFuzz가 PSO 알고리즘을 사용하여 AFLfast보다 더 효율성 있는 변이 연산자를 선택하기 때문에 더 많은 경로를 찾고, 제약조건을 해결하는 것을 Fig. 5의 그래프를 보며 확인했다. 크래시 정보 또한, 코드 커버리지 평가와 마찬가지로 더 많은 경로를 찾아 제약조건을 해결하기 때문에 더 많은 경로 접근하여 퍼징하게 되고, 최적의 변이 연산자를 선택해 나아가면서 변이과정을 진행한다. 이때 퍼징을 실행하는데 있어 효율이 좋은 입력값들을 생성해가며 퍼징을 실행하기 때문에 더 많은 크래시를 발견할 수 있다.

Table 3을 살펴보면 본 연구에서 제안한 IoTFirmFuzz가 기존 연구 Firm-AFL[5]에 비해 크래시와 경로 정보를 더 많이 찾은 것을 확인할 수 있다. 이는 IoTFirmFuzz가 기존 연구에서 전체 시스템 모드 에뮬레이터와 사용자 모드 에뮬레이터를 사용하

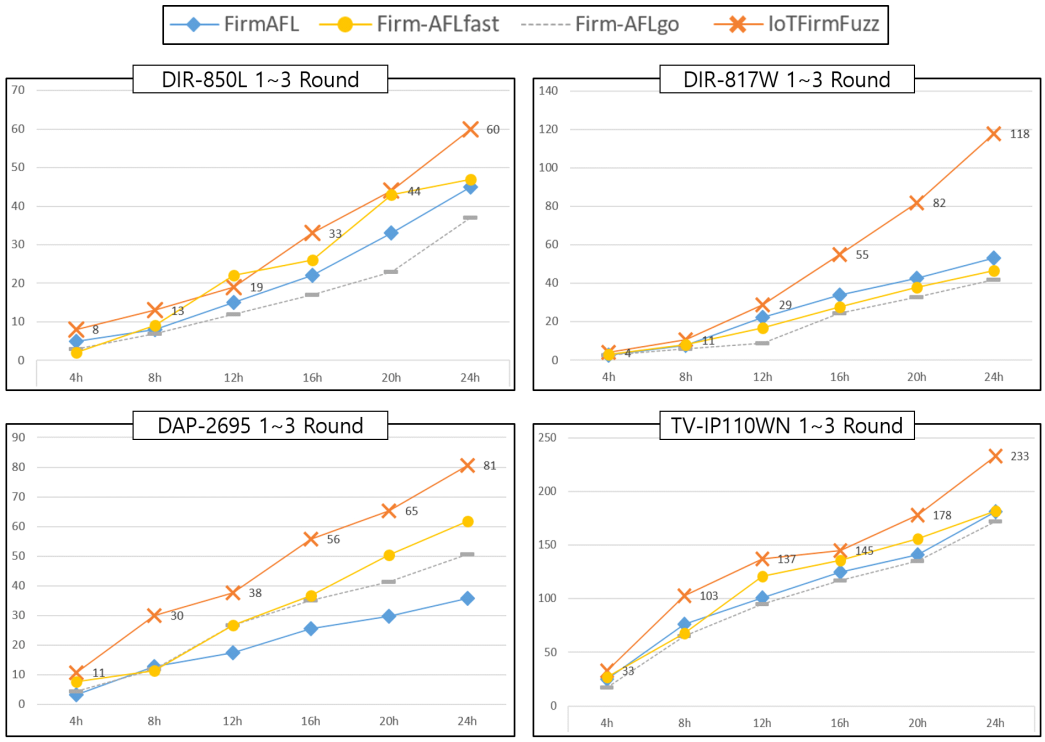


Fig. 6. Crash graphs of target firmwares

는 복합 에뮬레이터를 사용하여 처리속도와 호환성을 높여 사용하는 방식이기 때문에 이전 연구와 처리속도는 같지만, 파워 스케줄러와 PSO 알고리즘을 사용하여 입력값에 대한 효율성을 높여 퍼징을 실행하기 때문에 기존 연구보다 많은 양의 코드 커버리지와 크래시를 발견할 수 있었다. 평가를 수행한 펌웨어 중에서 크래시 부분은 DAP-2695 펌웨어에서 약 2.2배 더 많은 크래시를 발견했고 코드 커버리지 부분은 TV-IP110WN 펌웨어에서 약 4.3배 더 많은 실행 경로를 찾아냈다.

V. 결 론

본 논문에서는 실제 IoT 장비 없이 펌웨어만을 가지고 에뮬레이팅하여 시스템을 구축하고, IoT 장비 펌웨어의 취약점을 빠르고 효율성 있게 찾고자 기존 연구의 퍼징 변이 방식을 개선했다. 또한, 본 논문에서 제안한 IoTFirmFuzz는 기존 연구들에 비해 많은 경로를 찾고 제약조건을 해결하며 퍼징하여 더 많은 크래시를 발견했다. 4장에서 찾은 크래시 정보들은 기존에 알려진 취약점을 유발했으며, 더 다양한 경로에서 취약점을 발견했다. 끝으로 더 다양한 펌웨어를 퍼징할 수 있도록 펌웨어에 대한 에뮬레이션의 호환성 부분을 더 연구하면 다양한 펌웨어의 취약점을 발견할 수 있을 것으로 기대한다.

References

[1] N. Zhang, S. Demetriou, X. Mi, W. Diao, K. Yuan, P. Zong, F. Qian, X. Wang, K. Chen, Y. Tian, C. A. Gunter, K. Zhang, P. Tague, and 0Y. Lin, "Understanding IoT Security Through the Data Crystal Ball: Where We Are Now and Where We Are Going to Be," CoRR, vol. abs/1703.09809, 2017.

[2] Barton P. Miller, Louis Fredriksen and Bryan So, "An empirical study of the reliability of UNIX utilities", Communications of the ACM, Dec. 1990.

[3] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti, "AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares," in Proceedings of the 21st Annual Network and Distributed System Security Symposium ,NDSS ,Feb. 2014.

[4] Jiongyi Chen,Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Keh uan Zhang. "IoTFuzzer: Discovering memory corruptions in iot through app-based fuzzing," In Networked and Distributed System Security Symposium, NDSS, Feb. 2018.

[5] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu and Limin Sun, "Firm-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation," in Proceedings of the USENIX 2019 Annual Technical Conference, Aug, 2019

[6] Andrei Costin, Jonas Zaddach, Aur´elien Francillon and Davide Balzarotti, "A Large-Scale Analysis of the Security of Embedded Firmwares", in Proceedings of the 23rd USENIX Security Symposium, Aug, 2014

[7] Daming D. Chen, Maverick Woo, David Brumley, and Manuel Egele. Towards automated dynamic analysis for Linux-based embedded firmware. In Network and Distributed System Security Symposium, NDSS, February 2016.

[8] F. Bellard, "QEMU, a fast and portable dynamic translator," in Proceedings of the USENIX 2005 Annual Technical Conference, pp. 41-41, Apr. 2005,

[9] Andrew Henderson, Aravind Prakash, Lok Kwong Yan, Xunchao Hu, Xujiewen Wang, Rundong Zhou, and Heng Yin. Make it work, make it right, make it fast: Building a

platform-neutral whole-system dynam ic binary analysis platform. In Interna tional Symposium on Software Testing and Analysis (ISSTA'14), July 2014.

[10] Daming D. Chen, Maverick Woo, David Brumley, and Manuel Egele. "Towards automated dynamic analysis for Linux-based embedded firmware," In Network and Distributed System Security Symposium, NDSS, Feb. 2016.

[11] NCC-Group, FirmAFL, "https://githu b. com/zyw-200/FirmAFL", 2019

[12] M. Bohme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greyb ox fuzzing as markov chain," in CCS, 2016.

[13] C. Lyu, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song and R. Beyah, "MOPT: Optimized Mutation Scheduling for Fuzzers, Technical Report," in Proceedings of the USENIX 2019 Ann ual Technical Conference, Aug, 2019.

[14] C. Lyu, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song and R. Beyah, MOptAFL, "https://github.com/HexH ive/magma/tree/master/fuzzers/moptaf l", 2019.

[15] James Kennedy and Russell Eberhart, "Particle Swarm Optimization", in IEEE International Conference, 1995.

[16] M. Zalewski, American fuzzy lop(AFL), " http://lcamtuf.coredump.cx/afl"

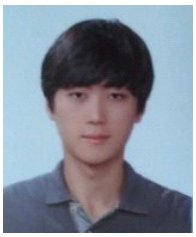
[17] Smolinsky. L, "Discrete Power Law with Exponential Cutoff and Lotka's Law.", J. Assoc. Inf. Sci. Technol, 68, 1792 - 1795, 2017.

[18] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen and Abhik Roychoudhury, "Directed Greybox Fu zzing", in CCS, 2017.

[19] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen and Abhik Roychoudhury, AFLgo, " https://github. com/aflgo/aflgo", 2017.

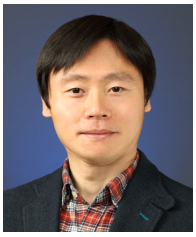
[20] M. Bohme, V.-T. Pham, and A. Roychoudhury, AFLfast https://githu b. com/mboehme/aflfast, 2016.

〈저자소개〉



김 현 욱 (Hyun-Wook Kim) 학생회원
2018년 8월: 숭실대학교 전산원 정보보호학 전공 학사
2018년 9월~현재: 세종대학교 정보보호학과 석사과정
〈관심분야〉 시스템 보안, 임베디드 보안, 네트워크 보안

김 주 환 (Juhwan Kim) 학생회원
2016년 2월: 학점은행제 정보보호학 전공 학사
2019년 8월: 세종대학교 일반대학원 정보보호학과 석사
2019년 9월~현재: 세종대학교 일반대학원 정보보호학과 박사과정
〈관심분야〉 악성코드 분석, 시스템 보안, 소프트웨어 취약점



윤 주 범 (Joobeom Yun) 종신회원
1999년 2월: 고려대학교 컴퓨터학과 학사
2001년 2월: 서울대학교 컴퓨터공학과 석사
2012년 2월: KAIST 전산학과 박사
2001년 3월~2015년 2월: ETRI부설연구소 선임연구원
2015년 3월~현재: 세종대학교 정보보호학과 부교수
〈관심분야〉 네트워크 보안, 시스템 보안, 클라우드 컴퓨팅 보안