Journal of The Korea Institute of Information Security & Cryptology ISSN 1598-3986(Print)
VOL.31, NO.4, Aug. 2021 ISSN 2288-2715(Online)
https://doi.org/10.13089/JKIISC.2021.31.4.687

바이너리 정적 분석 기반 Out-of-Bounds Read 취약점 유형 탐지 연구*

유 동 민,[1†] 김 문 회,[2] 오 희 국[3‡]
[1]한양대학교 컴퓨터공학과 바이오인공지능융합전공 (대학원생),
[2,3]한양대학교 컴퓨터공학과 (대학원생, 교수)

A Out-of-Bounds Read Vulnerability Detection Method Based on Binary Static Analysis*

Dong-Min Yoo,[1†] Wen-Hui Jin,[2] Heekuck Oh[3‡]
[1]Major in Bio Artificial Intelligence, Department of Computer Science and Engineering, Hanyang University (Graduate student),
[2,3]Department of Computer Science and Engineering, Hanyang University (Graduate student, Professor)

요 약

프로그램에서 취약점이 발생하면 그에 대한 정보가 문서화되어 공개된다. 그러나 일부 취약점의 경우 발생한 원인과 그 소스코드를 공개하지 않는다. 이러한 정보가 없는 상황에서 취약점을 찾기 위해서는 바이너리 수준에서 코드를 분석해야 한다. 본 논문에서는 Out-of-bounds Read 취약점 유형을 바이너리 수준에서 찾는 것을 목표로 한다. 바이너리에서 취약점을 탐지하는 기존의 연구는 주로 동적 분석을 이용한 도구로 발표되었다. 동적 분석의 경우 프로그램 실행 정보를 바탕으로 취약점을 정확하게 탐지할 수 있지만, 모든 실행 경로를 탐지하지 못할 가능성이 있다. 모든 프로그램 경로를 분석하기 위해서는 정적 분석을 사용해야 한다. 기존의 정적 도구의 경우 소스코드 기반의 도구들이며, 바이너리에 수준의 정적 도구는 찾기 어렵다. 본 논문에서는 바이너리 정적 분석을 통해 취약점을 탐지하며, 메모리 구조를 모델링하는 방법으로 Heap, Stack, Global 영역의 취약점을 탐지한다. 실험 결과 기존의 탐지도구인 BAP_toolkit과 비교하였을 때 탐지 정확도 및 분석 시간에서 의미 있는 결과를 얻었다.

ABSTRACT

When a vulnerability occurs in a program, it is documented and published through CVE. However, some vulnerabilities do not disclose the details of the vulnerability and in many cases the source code is not published. In the absence of such information, in order to find a vulnerability, you must find the vulnerability at the binary level. This paper aims to find out-of-bounds read vulnerability that occur very frequently among vulnerability. In this paper, we design a memory area using memory access information appearing in binary code. Out-of-bounds Read vulnerability is detected through the designed memory structure. The proposed tool showed better in code coverage and detection efficiency than the existing tools.

Keywords: Binary Analysis, Vulnerability Detection, Static Analysis, Out-of-bounds

Received(05. 12. 2021), Modified(07. 19. 2021), Accepted(07. 19. 2021)

* 이 논문은 2021년도 정부(과학기술정보통신부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임(NRF-2019R1A2C2003045).

* 이 논문은 ETRI부설연구소의 위탁연구과제[2020-028]로 수행한 연구결과입니다.

† 주저자, ehdals6@hanyang.ac.kr

‡ 교신저자, hkoh@hanyang.ac.kr(Corresponding author)

I. 서 론

보안 취약점은 모든 소프트웨어 플랫폼에서 발생할 수 있다. 따라서 프로그램에 존재하는 취약점을 찾고 발견된 취약점의 원인을 알아내는 것은 어플리케이션의 연구 개발에 매우 중요한 문제이다. 그러나 이미 공개된 취약점이 아닌 이상 다양한 프로그램에서 취약점의 발생과 그 원인을 직접 밝혀내는 것은 매우 어려운 일이다.

일반적으로 소프트웨어 내에서 취약점이 발견되면 CVE(Common Vulnerabilities and Exposures)가 부여되고 이에 대한 내용이 문서화된다. 하지만 일부 취약점의 경우 소스코드가 공개되어 있지 않으며 취약점에 대한 자세한 내용을 설명하지 않는 경우도 많이 존재한다. 예를 들어 애플사의 Mac OS의 XNU 커널의 경우 발견된 취약점의 개수와 그 유형은 공개하지만 자세한 취약점의 발생 원인을 외부에 공개하지 않는다. 또한 애플의 오픈소스 공개 정책으로 업데이트한 소스코드를 매우 늦게 공개하여 곧바로 발견된 취약점들에 대한 보안 검증을 할 수 없다. 현재 이처럼 소스코드가 없는 상태에서 취약점을 찾아내기 위해서는 바이너리 수준의 분석이 필요하다.

프로그램에서 취약점을 검출하기 위해 기준이 될 수 있는 것 중 하나는 취약점 유형이라고 불리는 CWE를 확인하는 것이다. CWE(Common Weakness Enumeration)이란, 소프트웨어 또는 하드웨어에서 발생할 수 있는 취약점을 분류하여 열거해놓은 목록이다.

다양한 취약점 유형 중에서 본 논문이 탐지하고자 하는 Out-of-Bounds 취약점은 정해진 버퍼를 벗어난 주소에서 값을 읽거나 쓰는 경우에 발생하는 매우 위험한 취약점 중에 하나이다. MITRE에서 2020년 가장 위험한 CWE 순위를 발표하였다[1], Table 1.와 같이 2위는 CWE-787: Out-of-bounds Write, 4위는 CWE-125: Out-of-bounds Read, 5위는 CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer로 발표되었다. CWE-119는 CWE-787, 125의 개념을 모두 포함하고 있는 상위개념의 취약점으로 가장 빈번하게 발생하는 상위 5개의 취약점 유형 중에서 3개가 Out-of-bounds 취약점과 관련된 것으로 나타났다.

Out-of-bounds(OOB) 취약점은 일반적으로 매우 많이 발생하는 취약점 유형 중 하나이다. 이를 수동으로 탐지하기에는 시간적 비용이 많이 발생하며, 자동화하여 탐지할 필요성이 있다.

Table 1. The CWE Top 5

Rank	ID	Name
[1]	CWE-79	Cross-site Scripting
[2]	CWE-787	Out-of-bounds Write
[3]	CWE-20	Improper Input Validation
[4]	CWE-125	Out-of-bounds Read
[5]	CWE-119	Improper Restriction of Operation within the Bounds of a Memory Buffer

소스코드가 없는 상황에서 취약점을 자동으로 탐지하기 위한 방법으로는 코드를 기반으로 탐지하는 정적 분석과 프로그램 실행 정보를 기반으로 탐지하는 동적 분석 방법이 존재한다.

동적 분석 방법은 프로그램 실행 정보를 바탕으로 탐지하는 방법이다. 실제 프로그램의 실행정보를 바탕으로 취약점을 정확하게 탐지할 수 있지만, 실행 파일의 크기가 커질수록 모든 코드 경로를 탐색하지 못할 가능성이 존재한다. 그와 반대로 정적 분석 방법의 경우 코드를 기반으로 탐지하기 때문에 모든 코드를 확인하여 분석할 수 있지만, 동적인 실행 정보가 없기 때문에 오탐의 가능성이 높다. 따라서 동적, 정적 분석 도구 모두 각 장단점이 있기 때문에 두 분석 방법이 적절하게 활용되어야 한다. 하지만 바이너리에서 OOB를 탐지하기 위한 정적 분석 도구는 찾기 어렵다.

OOB를 탐지하기 위한 기존의 정적 분석 도구는 대부분 소스코드 기반의 방법이다[2-3]. 바이너리에서는 컴파일 과정에서 소스코드의 일부 정보를 누락하기 때문에 소스코드 기반의 탐지 방법을 적용해서는 바이너리에서 취약점을 탐지하기 어렵다. 2019년 공개된 cwe_checker[6]는 바이너리에서 시그니처를 분석하여 취약점을 자동으로 탐지하는 최초의 도구로 소개되었지만, 해당 도구에서는 OOB를 탐지하는 모듈은 포함되지 않았으며, 이외에 바이너리에서 정적기반의 자동탐지 도구는 찾기 어렵다.

바이너리에서 OOB를 탐지하기 위한 기존의 도구는 바이너리 계측, 기호실행, 퍼징과 같은 동적 분석 방법이 주로 발표되었다[7-12]. 컴파일 과정에서 많은 정보가 누락될 수 있는 바이너리 코드의 특성상 이와 같은 동적 분석 방법은 실행 정보를 바탕으로

취약점을 정확하게 탐지할 수 있다. 그러나 동적 바이너리 계측을 사용하는 도구인 Valgrind[4], 기호실행 방법을 사용하는 BAP_toolkit의 경우 Heap 영역만을 고려한 탐지 방식이다[12]. Heap영역에서는 매우 뛰어난 탐지 성능을 보이지만, Stack영역은 탐지할 수 없다. 또한 퍼징과 같은 동적 분석은 무작위 입력값을 기반으로 취약점을 탐지하는 방식이기 때문에 시멘틱한 취약점의 경우 찾을 수 없는 케이스가 존재한다.

기존의 OOB를 탐지하는 바이너리 정적 도구는 찾기 어려우며, 동적 도구의 경우에도 Stack영역과 모든 코드 커버리지를 만족하지 않는 경우가 존재한다. 본 논문에서는 기존에 연구되지 않았던 바이너리 정적 분석을 이용하여 OOB 취약점을 찾는 자동화 도구를 개발하였다.

기존에 동적 도구가 탐지하지 않았던 Stack영역을 탐지하기 위해서는 Stack 구조를 복구할 필요가 있다. 바이너리에서는 컴파일 과정에서 버퍼의 경곗값 정보가 누락되기 때문이다. 해당 문제를 해결하기 위해서 본 논문은 바이너리에서 메모리 접근 시에 발생하는 패턴을 식별한 뒤 코드의 주변 정보를 이용하여 변수의 Base 주소와 크기 값을 가지고 있는 Stack 구조를 복구한다. 이를 통해서 OOB 취약점이 발생했을 때 주소와 크기를 비교하는 것으로 취약점을 탐지한다. 위와 같은 방법을 통해 Heap영역뿐만 아니라 Stack, Global영역을 탐지할 수 있다.

또한 바이너리는 x86, ARM 등 다양한 아키텍처가 존재하며, 이에 따라 각 명령어도 서로 상이하다. 현재 본 논문은 x86을 기반으로 연구를 진행하였다. 그러나 서로 다른 아키텍처에서도 적용할 수 있도록 중간언어 기반의 분석 기술을 사용하는 도구를 활용하여 연구를 진행하였다. 카네기멜론 대학에서 개발한 바이너리 분석 플랫폼인 Binary Analysis Platform(BAP) 도구를 사용하여 분석하였다[13-14].

II. 관련 연구

본 장에서는 Out-of-Bounds 취약점 유형을 탐지하기 위한 기존연구에 대한 설명을 하고 이에 대한 연구와 한계점을 도출한다.

Out-of-bounds를 탐지하기 위한 기존의 연구는 코드를 분석하여 취약점을 탐지하는 정적 탐지와 프로그램 실행정보를 이용하는 동적 탐지로 구분할 수 있다. 정적 분석 방법의 경우 기존의 다양한 탐지 방법이 존재하지만 대부분 소스코드 수준에서 취약점을 탐지한다. 그러나 소스 코드를 확인하지 않고 바이너리 코드만을 이용하여 취약점을 찾는 것은 더욱 어려운 문제이다. 바이너리 수준에서 취약점을 탐지하는 방법은 주로 동적 분석을 활용한 주제로 발표되었다.

2.1 Out-of-bounds 탐지 문제점

Out-of-bounds 취약점 탐지의 문제점은 바이너리에서 변수의 경계 영역이 확인되지 않는 것에 있다. 소스 코드에서 바이너리로 컴파일될 때 프로그램 내부 정보가 누락되어 메모리의 경계를 확인할 수 없다. 예로 Fig. 1.에서 배열 a와 b에 각각 값을 할당할 경우 바이너리 코드에서는 Stack에서의 주소인 〔RBP-0x60〕과 〔RBP-0x1C〕 주소 정보와 할당 값에 대한 정보만 있을 뿐 배열 a와 b의 어떠한 경계 정보도 나오지 않는다. 또한 각 배열의 크기도 확인할 수 없다.

Out-of-bounds 탐지하기 위해서는 메모리 접근 시에 해당 메모리 영역이 다른 메모리 영역을 침범한 것인지 확인할 수 있어야 한다. 그러나 바이너리에서는 메모리 경계를 확인할 수 없으며, 이 때문에 바이너리 코드에서는 Stack 영역의 취약점을 찾는 것이 매우 어려워진다. 이러한 문제를 해결하기 위해서는 바이너리 코드에서 메모리 경계를 추측할 수 있어야 한다.

```
Source Code:
int a[10]; int b[10];
a[0] = 4;
b[5] = 5;

Intermediate Represent in BAP:
mem:= mem with [RBP-0x60, el]:u32<-0Px4
mem:= mem with [RBP-0x1C, el]:u32<-0x5c
```

Fig. 1. Missing buffer boundary information

2.2 기존 연구

바이너리에서 Out-of-bounds를 탐지하는 도구는 모두 동적 분석을 활용한 탐지 도구이다. 동적 분석 방식으로는 Dynamic Binary Instrumentation, Symbolic Execution, Fuzzing을 이용한 방법이 존재한다.

2.2.1 Dynamic Binary Instrumentation (DBI)

DBI는 프로그램을 실행하는 도중에 취약점을 탐지할 수 있는 코드를 삽입하여 취약점을 탐지하는 방법이다. 예로 현재 상용화되어 있는 관련 도구를 살펴보면 Table 2, Valfrind Memchecks[4]의 경우 malloc과 같은 함수에 사용되는 정보를 이용하여 Heap 영역 메모리 관련한 취약점을 탐지한다. malloc과 같은 메모리 할당 함수는 Base 포인터와 그 크기를 확인할 수 있기 때문에 메모리의 경계를 확인할 수 있다. 이를 활용하면 Heap 메모리 접근 시에 검증 코드를 삽입하는 것으로 취약점을 탐지할 수 있다. 그러나 이와 같은 방법은 Stack 메모리에는 적용하기 힘들다.

Address Sanitizer[2,15]의 경우 Heap, Stack영역의 Out-of-bounds(OOB)를 탐지하지만 소스코드 기반의 compile-time instrument 기법이다.

Table 2. Comparison of memory tools

	Technology	Heap OOB	Stack OOB
Address Sanitizer	CTI	yes	yes
Mudflap	CTI	yes	some
Valgrind	DBI	yes	no
Dr.Memory	DBI	yes	no

2.2.2 Symbolic Execution(기호실행)

기호실행은 바이너리 코드를 기반으로 실행될 수 있는 모든 실행 경로와 프로그램 상태를 실제 실행과 근사하게 에뮬레이션하여 프로그램을 분석하는 방법이다[16-24]. 이는 정적 분석만으로 확인하기 어려웠던 프로그램 실행과정에 연산되는 메모리 값을 예측할 수 있으며, 이를 이용하여 메모리 접근 시 발생하는 취약점을 탐지할 수 있다.

기호실행을 사용하는 대표적인 도구로는 BAP_toolkit이 있다.[12] BAP_toolkit은 Fig. 2.와 같이 기호실행 모듈을 이용하여 malloc과 같은 Heap영역의 메모리 주소와 크기를 에뮬레이션하고 이후 나오는 memcpy와 같은 취약한 함수가 나왔을 경우 해당 주소와 크기를 확인하는 것으로 취약점을 탐지하고 있다.

```
(defmethod call-return (name len ptr)
  (when (and len ptr (= name 'malloc))
    (memcheck-acquire 'malloc ptr len)))

(defmethod call (name dst src len)
  (when (is-in name 'memmove 'memcpy
'memcmp)
    (check/both dst src len)))
```

Fig. 2. Detection code provided by BAP

그러나 BAP_toolkit 도구 또한 기존의 바이너리 분석 도구와 마찬가지로 Heap영역만을 고려한 탐지 방식이며, Stack영역은 고려하지 않는다.

2.2.3 Fuzzing

퍼징은 실행 프로그램에 무작위 데이터를 입력하는 것으로 프로그램 충돌이나 잠재적인 취약점을 찾는 방법이다. 이러한 퍼징 시스템은 무작위로 입력된 입력 값으로 인한 잘못된 메모리 접근을 찾아낼 수 있다. 하지만 Out-of-bounds 취약점을 찾기 위해서는 메모리 버퍼의 정확한 경곗값을 확인할 수 있어야 취약점 탐지가 가능한데, 임의적인 입력값을 사용해서 메모리의 경곗값을 찾는 것은 매우 어려운 일이다.

2019년 H.Wang은 정적, 동적 정보를 모두 활용하는 하이브리드 방식을 발표하였다[25]. 해당 논문에서는 정적 분석을 통해 메모리의 경곗값을 복구한 다음 퍼징을 이용한 동적 실행 정보를 이용하여 Out-of-bounds 취약점을 탐지한다. 퍼징 시스템에서 무작위 입력값을 설정하는 것은 매우 어렵지만, 정적 분석 방식으로 메모리 영역의 경곗값을 복구하는 것으로 취약점 탐지의 효율을 높였다. 하지만 퍼징은 입력값을 기반으로 취약점을 탐지하는 방식이다. 프로그램의 입력 값에 영향을 받지 않는 시멘틱한 취약점의 경우 찾기 어려우며, 동적 분석의 특성상 프로그램의 크기가 커질수록 코드의 모든 영역을 분석하기에 어려움이 있다.

2.3 기존 연구의 문제점

앞선 퍼징을 이용한 논문의 경우 정적 분석을 통해 메모리 경곗값을 복구하여 효율적인 퍼징을 하기 위한 방법을 제시하였다. 그러나 퍼징의 경우 무작위 입력값에 따라서 취약점을 탐지하기 때문에 모든 코드 경로를 탐지 못 할 수 있으며, 입력값이 들어가지

않는 시멘틱한 취약점의 경우 탐지할 수 없다.

이러한 퍼징의 단점은 코드 전체를 분석하는 정적 분석을 이용하거나, BAP_toolkit에서 제공하는 기호실행 도구를 이용하면 해결할 수 있다. 기호실행은 프로그램의 실행 가능한 모든 경로를 에뮬레이션하기 때문에 코드의 모든 경로를 탐색할 수 있으며, 분석 과정에서 프로그램의 메모리 상태를 계산하기 때문에 시멘틱한 취약점의 경우도 정확하게 찾을 수 있다.

BAP_toolkit와 같은 기호실행 도구는 매우 정확한 탐지를 보장하지만, 모든 경로를 탐색하기 위해서 적절한 입력값이 선택되어야 하고, 프로그램 실행과정을 모두 연산하기 때문에 많은 분석 시간이 소요된다. 또한 현재 Heap 영역만을 탐지하고 있으며, Stack영역은 고려하지 않는다.

현재 Out-of-bounds를 탐지하는 기존의 도구는 정적 분석의 경우 소스코드 기반의 도구가 존재하지만, 바이너리 기반의 도구들은 찾기 어렵다. 동적 분석의 경우 많은 도구들이 존재하지만 Heap메모리 영역만을 탐지하는 도구들이며, 실행 정보 바탕으로 탐지하기 때문에 모든 코드를 커버하지 못하는 문제가 있다.

본 논문에서는 바이너리 정적 분석을 이용하여 Heap영역뿐만 아니라 Stack 영역에서 Out-of-bounds 취약점을 찾는 것을 목표로 한다.

2.3.1 Stack 메모리 복구

본 연구에서는 바이너리 코드 기반의 정적 분석 방법을 이용하여 Stack의 구조를 복구한다. Stack에서 메모리의 경계를 복구하기 위한 연구가 존재한다[26-30]. 해당 연구에서는 바이너리 코드에서 메모리 접근 시에 발생하는 명령어를 확인하여 해당 주소에 할당되는 값을 확인하여 해당 변수 크기를 예측한다. 해당 방법을 통해 Stack의 메모리 영역을 복구하고 Heap영역의 경우 Memory Allocation함수를 식별하여 Heap의 메모리 영역을 복구하여 취약점을 탐지한다.

III. 제안 방법

3.1 Binary Analysis Platform (BAP)

본 논문에서는 이기종의 아키텍처에도 적용할 수 있는 중간언어 기반의 분석기술을 고려한다. 중간언어 기반의 분석기술을 사용하는 대표적인 도구는 IDA, Angr, Radare2, Ghidra, BAP와 같은 도구들이 존재하며, 이와 같은 바이너리 분석 도구는 각자의 중간언어표현인 REIL(IDA), VEX(Angr), ESLR(Radare2), PCode(Ghidra) BIL(BAP)으로 변환한 후 분석을 하는 단계를 거친다.

위 도구들 중에서 IDA, Ghidra, Radare2의 경우는 취약점 탐지 모듈을 제공하지 않으며, BAP와 Angr는 취약점을 탐지하기 위한 내부 모듈을 오픈소스로 제공하고 있다. 본 논문에서는 바이너리 패턴을 분석하기 용이한 BAP를 기반으로 정적 탐지 도구를 개발하였다.

3.2 취약점 탐지 전략

본 논문의 목적은 Heap영역뿐만 아니라 Stack 영역에서 Out-of-bounds Read 취약점을 탐지하는 것을 목표로 한다. 취약점 탐지의 순서는 다음과 같다. (1). 우선 바이너리 코드에서 메모리 주소 접근 시 나타나는 변수 주소를 식별한 뒤 수집한 주소를 바탕으로 Stack 구조를 복구한다. (2). 복구된 Stack 구조를 바탕으로 코드의 주변 정보를 활용하여 해당 변수가 어느 정도의 버퍼 크기를 가질 수 있는지 정보를 수집한다. (3). 코드에서 취약한 함수가 식별되면, 수집한 변수 주소와 버퍼 크기 정보를 바탕으로 취약점을 탐지한다.

1. Stack 구조 복구
2. 버퍼의 크기 정보 수집
3. 취약점 탐지

3.3 Stack 구조 복구

일반적으로 코드에서 선언되는 변수들은 코드에서 사용되어야 한다. 이는 선언된 변수 대부분 Stack 내에서 주소를 식별할 수 있다는 뜻이다. Stack의 구조를 복구하기 위해서는 메모리 접근 시에 발생하는 모든 변수 주소를 식별해야 한다. 코드에서 선언되는 변수들은 Stack 내에서 [RBP - offset]:size형태로 나타난다. 이러한 변수들은 값이 Load되거나 Store될 때 식별할 수 있으며, offset 정보를 통해 대략적인 위치와 size를 통해 데이터의 타입을 어느 정도 예측할 수 있다. 메모리 주소를 수집하기 위한 패턴은 Fig. 3.과 같다.

Memory Store:
mem := mem[RBP + offset]:size <- [value]

Memory Read:
Reg := RBP + offset

Memory Read (Dereference):
Reg := mem[RBP + offset]:size

Fig. 3. Variable address collection pattern

앞서 수집된 Base 주소를 기반으로 주소의 크기 정보를 예측하여 Stack의 구조를 모델링할 수 있다. 우선 수집된 변수 주소를 정렬한 후 각 주소 간 간격을 계산하여 대략적인 Stack을 복구한다. Fig. 4.는 Stack 구조를 복구하기 위한 과정을 보여준다.

1) BAP에서 제공하고 있는 중간언어표현을 기반으로 코드상에 나타나는 메모리 접근 패턴을 탐지한다. 2) 이를 파싱하여 변수 주소를 수집한다. 3) 수집된 정보를 바탕으로 Stack구조를 모델링한다.

변수를 통해 Stack 구조를 복구하였지만, 해당 정보만 가지고는 버퍼의 크기를 정확하게 탐지할 수 없다. 변수에서 버퍼의 크기를 가질 수 있는 경우는 일반적인 Stack의 배열일 수 있으며, Malloc을 통해 Heap공간이 할당된 주소 값이 들어올 수도 있으며, 전역 변수가 들어올 수 도 있다.

1) IR in BAP
mem := mem with [RBP - 0xD] <- 0
RCX := mem[RBP - 0xE8, el]:u64
RAX := low:64[RBP - 0x70]

2) Collected variable address
RBP-0x8:u64, RBP-0xE0, RBP-0x70
RBP-0x7D, RBP-0xE8:u64, RBP-0xD
RBP-0x78:u64

3) Recovered Stack structure

RBP - 0xE8: 8(Byte)
RBP - 0xE0: 99(Byte)
RBP - 0x7D: 1(Byte)
RBP - 0x78: 8(Byte)
RBP - 0x70: 99(Byte)
RBP - 0x0D: 1(Byte)
RBP - 0x08: 8(Byte)

Fig. 4. Stack Memory Layout Recovering

3.4 버퍼의 크기 정보 수집

본 논문에서는 memcpy, stncpy와 같은 일부 취약한 함수의 정보를 이용하여 Out-of-bounds Read를 탐지한다. 이를 위해서는 Stack, Heap 메모리에서 버퍼 영역을 가리키고 있는 주소의 경곗값 즉 크기를 식별해야한다. 본 논문에서 탐지하고자 하는 취약 함수의 경우 버퍼 정보를 가져올 때 Fig. 5.와 같이 [Stack, Heap, Global, Dereference] 4가지 형태로 나타난다.

Stack 영역의 변수일 경우 [RBP - offset] 정보를 이용하여 해당 버퍼의 주소를 식별할 수 있으며, 앞서 복구한 Stack 구조에서 간격을 확인하여 크기를 식별할 수 있다.

Heap의 경우 memory allocation함수인 malloc과 같은 함수의 호출 규약을 확인하여 주소와 크기를 식별할 수 있다. malloc함수 호출 전 RDI 레지스터를 추적하여 크기를 식별할 수 있으며, malloc 함수 호출 후 RAX 레지스터를 추적하여 Base포인터를 식별할 수 있다.

Global 변수를 사용할 경우 RBP 레지스터를 이용하여 접근하는 것이 아닌 상수 주소를 이용하여 메모리에 접근한다. 코드에서 declare하여 선언하였을 경우에는 Stack과 동일하게 상수로 되어 있는 주소의 간격 차이를 이용하여 크기를 구하고, malloc을 사용할 경우 Heap영역과 마찬가지로 malloc의 호출 규약을 확인하여 버퍼의 크기를 구한다.

Dereference의 경우 중간 변수를 통해 배열의 주소를 가져오는 경우이다. 일반적인 지역 변수를 통해 버퍼의 주소를 가져올 수 있으나, 함수의 파라미

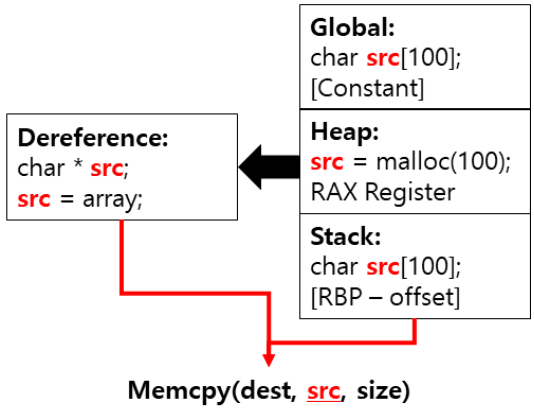

Fig. 5. Patterns of buffers at source address

터 또는 다른 함수의 리턴값으로 들어오는 경우도 존재한다. 이 경우 해당 주소를 역추적하여 변수의 크기를 식별한다.

3.4.1 Stack에서 크기 정보 수집

버퍼가 Stack영역에 있을 경우 복구된 Stack 구조에서 주소 간 간격 차이를 확인하여 크기를 식별할 수 있다. 하지만 3.3절에서 복구한 Stack 구조에서는 버퍼의 정보가 정확하지 않을 수 있다. Stack에서 버퍼에 접근하는 경우 두 가지 경우가 존재하기 때문이다. 하나는 Base 주소와 RAX와 같은 레지스터를 통해 멤버로 접근하는 경우와 또 하나는 곧바로 버퍼의 멤버로 접근하는 경우이다.

Fig. 6.에서 상단의 행은 버퍼에 접근할 경우 중간 변수를 통해 접근한다. 이 경우 RBP 레지스터를 통한 Base 주소에 RAX 값을 연산하여 멤버로 접근한다. 코드에서 볼 수 있듯이, 배열의 Base 주소만 식별되기 때문에 복구한 Stack 구조의 크기 정보는 올바르게 유지된다. 그러나 상수값을 이용하여 곧바로 배열의 멤버로 접근하는 경우 RAX 레지스터 없이 〔RBP - offset〕 형태로 나타난다. 이럴 경우 코드에서는 두 개의 주소가 식별될 수 있는데, 이는 동일한 버퍼에 속해있지만, 단일 변수인지 배열의 값인지 판단하기 어렵다. 제안하는 방법에서는 Stack 구조를 복구할 때 식별된 두 주소를 모두 수집하기 때문에 버퍼의 경곗값이 부정확할 수 있다.

본 논문에서는 Fig. 7.와 같이 특정 패턴을 가지는 경우 휴리스틱 방법으로 Stack의 구조를 변경하였다. 예를 들어 설계한 Stack 구조에서 char형의 버퍼 값으로 추정되는 변수가 식별이 되고 그 다음 식별되는 변수의 값에 0(char)을 할당해주는 패턴이 있다면, 해당 패턴이 char형 문자열의 마지막 항

Int val = 99; Buf[val] = '\0' strlen(Buf);	mem := mem with [RBP + RAX - 0x70] <- 0 RAX := low:64[RBP - 0x70] RDI := RAX RSP := RSP - 8 mem := mem with [RSP, el]:u64 <- 0x6DE call @strlen with return %0000042a
Buf[100-1] = '\0' strlen(Buf);	mem := mem with [RBP - 0xD] <- 0 RAX := low:64[RBP - 0x70] RDI := RAX RSP := RSP - 8 mem := mem with [RSP, el]:u64 <- 0x8B call @strlen with return %00000126

Fig. 6. Two patterns for accessing buffers on the Stack

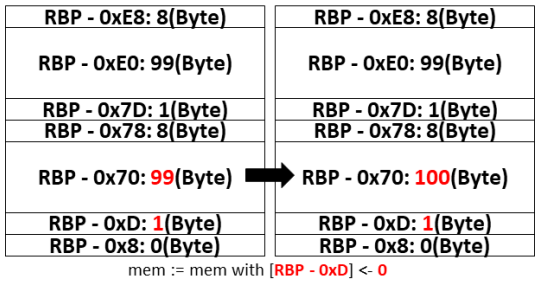

Fig. 7. Adjust the Stack buffer size

목을 0으로 넣는 패턴으로 판단하여 버퍼 크기를 1만큼 증가시켰다.

3.4.2 Heap에서 크기 정보 식별

Heap 영역에서 크기 정보를 식별하기 위해서는 Memory Allocation함수인 malloc과 같은 메모리 할당 함수를 확인하여 찾을 수 있다. Fig. 8.에서 Allocation 함수의 호출 규약을 확인하여 Heap 버퍼를 가리키고 있는 변수 주소와 크기정보를 쉽게 찾을 수 있다. 이를 식별하여 앞서 복구한 Stack에서 변수의 크기 정보를 업데이트한다.

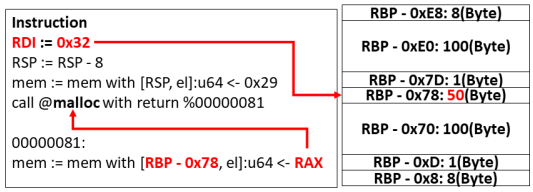

Fig. 8. Find size information in malloc

3.4.3 Global에서 크기 정보 식별

Global 영역의 경우 코드상에서 메모리 공간이 Fig. 9.와 같이 상수값의 형태로 나타난다. Global 변수의 경우에도 일반적으로 변수를 declare하여 버퍼를 생성하는 경우도 있고, 포인터 변수를 사용하여, Heap 주소를 받아오는 경우도 있다. 하지만 〔RBP - offset〕 형태에서 상수로 형태만 변형되었을 뿐 크기의 식별 방법은 동일하다. 그러나 앞선 Stack 구조를 복구할 경우에는 해당 Subroutine (함수)가 끝나면 복구한 Stack 구조를 사용하지 않는 것과는 다르게 Global 영역의 경우 다른 함수에

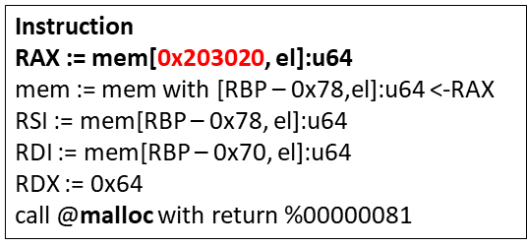

Fig. 9. Global address in binary code

결쳐 사용될 수 있기 때문에 함수가 끝나도 따로 Global 변수의 구조를 유지해야한다.

3.4.4 Dereference 주소에서 크기 정보 식별

Dereference는 해당 변수가 가리키고 있는 정보가 다른 버퍼의 주소를 가리키고 있는 경우이다. Dereference의 경우 여러 형태가 있을 수 있다. 함수 내에서 Stack이나 Heap 버퍼를 참조하는 경우도 존재하며 해당 함수의 파라미터나 다른 함수의 리턴값을 통해 함수 외부의 버퍼를 참조할 수 있다.

함수 내를 참조하고 있는 버퍼의 경우 Fig. 10.와 같이 해당 주소가 가리키는 레지스터를 역추적하여 추적한 주소의 크기를 식별하여 변수의 크기 정보를 업데이트 한다.

그러나 함수의 파라미터나 다른 함수의 리턴값을 참조하는 경우 버퍼는 외부 함수에서 오는 것이기 때문에 해당 함수만 역추적하면 안 된다.

함수의 파라미터를 참조하는 경우 분석하고 있는 함수가 호출되어 파라미터를 통해 외부 버퍼를 참조하는 경우이다. 함수 파라미터의 경우 RDI, RSI, RDX 레지스터를 이용하여 전달된다.

Fig. 11.와 같이 호출하는 함수 A가 call 하기 전 RDI와 같은 레지스터를 역추적하여 참조하는 버퍼의 크기를 역추적한다. 이 경우 호출하는 함수의 Subroutine Stack 구조를 복구해야 한다. 호출되는 함수 B는 A로부터 전달받은 버퍼의 크기를 이용

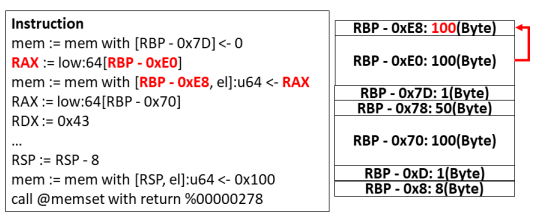

Fig. 10. Find size information in dereference

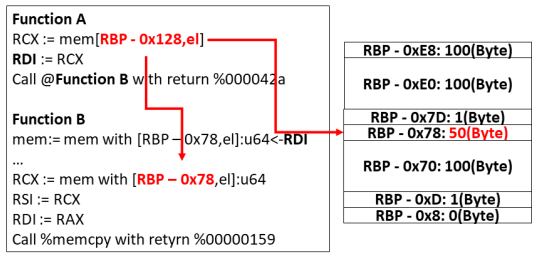

Fig. 11. Function Parameter

하여 Stack의 크기 정보를 업데이트 할 수 있다.

버퍼를 가리키고 있는 주소 변수가 호출되는 함수의 리턴값을 참조하는 경우도 마찬가지로 분석하고 있는 함수만 고려해서는 버퍼의 크기를 알 수 없다. 호출되는 함수의 Subroutine Stack 구조를 복구해야한다.

Fig. 12.와 같이 함수의 리턴값을 전달할 때 RAX 레지스터를 이용하기 때문에 호출되는 함수의 마지막 RAX 레지스터를 역추적하여 호출되는 함수 A의 Stack 구조에서 크기 정보를 식별한 뒤 호출하는 함수 B의 RAX 레지스터를 확인하여 호출하는 함수의 Stack의 크기 정보를 업데이트 한다.

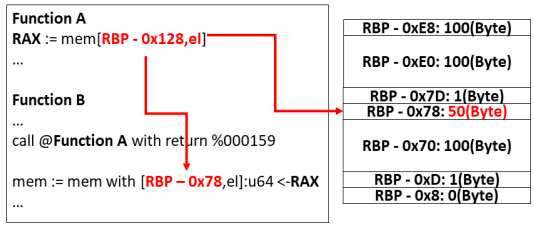

Fig. 12. Function Return

3.5 취약점 탐지

본 논문에서는 memcpy, memmove, strncpy 등 함수 심볼 정보를 이용하여 취약점을 탐지한다. 해당 함수를 호출할 때 발생하는 호출 규약을 확인하여 해당 함수의 source 주소와 size값을 확인한다.

Out-of-bounds Read의 경우 Fig. 13.와 같이 하위 취약점으로 Buffer Under-read와 Buffer Over-read가 존재한다. Buffer Under-read의 경우 source 정보를 확인하여 해당 주소 보다 작은 주소를 읽으면 탐지한다. Buffer Over-read의 경우 source의 주소와 size의 값을 확인하여 해당 주소에서 정해진 버퍼보다 큰 값을 읽으면 탐지한다.

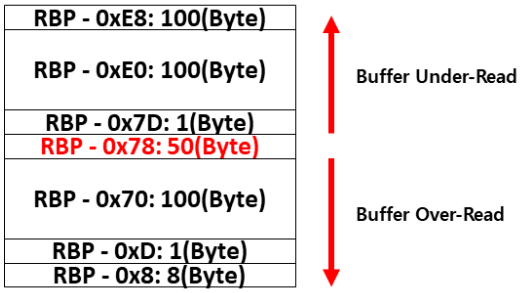

Fig. 13. Out-of-bounds Read

3.5.1 Buffer Under-Read 탐지

취약한 함수에서 Buffer Under-Read를 탐지하기 위해서는 source의 주소만 추적하면 된다. Fig. 14.와 같이 버퍼의 크기에 상관없이 source에서 식별된 변수 주소가 바이너리 연산으로 본래의 주소보다 낮은 주소로 변경이 발생하면 탐지한다.

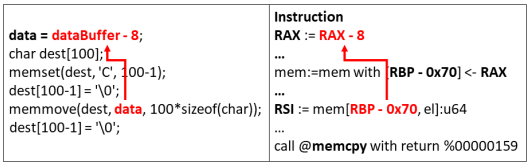

Fig. 14. Buffer Under-Read Detection Method

3.5.2 Buffer Over-Read 탐지

취약한 함수에서 Buffer Over-Read를 탐지하기 위해서는 source의 주소와 size를 확인해야 한다. Over_Read의 경우 해당 버퍼의 크기를 기준으로 크기를 넘으면 Buffer Over-Read, 넘지 않으면 정상적인 접근이기 때문이다. 해당 취약점을 탐지하기 위해서 복구한 Stack구조를 사용한다.

Fig. 15.와 같이 취약한 함수를 식별한 뒤 source에서 식별되는 버퍼의 주소를 추적하여 해당 버퍼의 크기 정보를 가져온다. 해당 크기 정보와 취

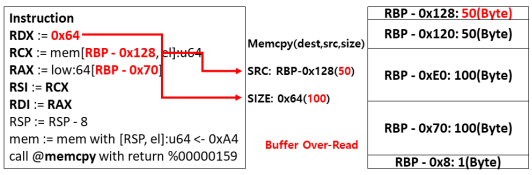

Fig. 15. Buffer Over-Read Detection Method

약한 함수에서 인자로 들어오는 크기 값을 비교하여 취약점을 탐지한다.

IV. 실 험

본 논문이 제안한 방법의 성능을 평가하기 위해 기존에 개발된 취약점 탐지 도구인 BAP_toolkit과 비교 분석하였다. 성능 평가를 위한 테스트셋으로는 NIST에서 제공하고 있는 Juliet Test Suite C/C++를 대상하였다[31]. Juliet Test Suit는 112개의 CWE 샘플을 제공하고 있으며, 각 취약점 샘플들은 코드 내부에 실제 취약점이 존재하는 Bad 함수와 비슷한 패턴을 가지지만 정상적인 함수인 Good함수로 이루어져 있다. 해당 테스트셋에서는 CWE:126 Buffer Overread 870개의 실행파일과 CWE:127 Buffer Underread 1168개의 실행파일이 존재한다.

해당 취약점 샘플들은 Table 3.와같이 다양한 형태로 발생할 수 있는 취약점의 종류로 이루어져 있다. 본 논문이 제안한 도구는 BAP_toolkit과 마찬가지로 취약점을 탐지할 때 memcpy, memmove와 같은 일부 위험한 함수를 식별하여 취약점을 탐지한다. 현재 제안한 도구는 memcpy, memmove의 케이스만 탐지하고 있지만 오탐을 확인하기 위해 취약점 탐지는 위 케이스 모두 섞어 탐지하였다.

Table 3. CWE-125 Vulnerability types

Testcase	CWE126	CWE127
loop	194	176
memcpy	176	176
memmove	176	176
fgets	48	48
fscanf	48	48
rand	48	48
socket	96	96
strcpy	36	352
large	48	0
negative	0	48
Total	**870**	**1168**

4.1 실험 결과

Table 4.은 BAP_toolkit과 본 논문이 제안한 도구의 비교 결과이다. 총 2038개의 실행파일 중에

Table 4. Summary of comparison for the proposed approach vulnerability detection in Juliet test suite.

Test case	CWE	Function	Program	Functions	Known Vulns	(proposed approach/BAP_toolkit)			
						FP	FPR	TP	TPR
memcpy ,memmove	Buffer Over read	malloc	96	322	96	6/0	2%/0%	80/39	83%/41%
		alloca	80	268	80	8/0	3%/0%	8/0	10%/0%
		new_char	96	322	96	10/0	4%/0%	10/0	11%/0%
		decalre	80	268	80	6/0	2%/0%	54/0	68%/0%
	Buffer Under read	malloc	96	322	96	8/0	3%/0%	46/0	48%/0%
		alloca	80	268	80	6/0	3%/0%	46/0	58%/0%
		new_char	96	322	96	8/0	3%/0%	46/0	48%/0%
		decalre	80	268	80	6/0	3%/0%	46/0	58%/0%
Sum		**1184**	**704**	**2360**	**704**	**58/0**	**3%/0%**	**336/39**	**48%/6%**

서 memcpy, memmove 관련 실행파일 704개만 나타낸 표이며, 이외에 경우 모두 오탐이 없었다. BAP_toolkit의 경우 전체 취약점 중에서 malloc을 사용한 39개의 샘플에 대해서만 탐지하였으며, Heap영역을 제외한 Stack영역에서의 취약점은 탐지하지 못하였다. 특히 Buffer Underread의 취약점은 전혀 탐지하지 못하였다. 그 이유는 BAP_toolkit의 경우 malloc의 Base포인터와 그 크기를 구하여 취약점을 탐지하기 때문이다. memcpy, memmove와 같은 취약점의 경우 size의 인자가 unsigned값이기 때문에 Base 포인터가 변경되어야 underread가 발생한다. 하지만 BAP_toolkit에서는 Base 포인터가 변경되는 경우는 고려하지 않았다.

본 논문에서 제안한 도구의 경우 Stack 구조를 복구하는 방법으로 Stack에 있는 버퍼의 크기를 예측할 수 있기 때문에 Stack영역에서 발생하는 취약점을 탐지할 수 있었다. 더욱이 malloc의 True Positive에서 확인할 수 있듯이, Heap영역에서 또한 본 논문이 제안한 도구가 더 높은 탐지율을 보였다. 이는 memcpy와 같은 취약한 함수에서 복사한 크기를 할당받을 때 strlen과 같은 버퍼의 크기로 받아오는 경우가 존재하는데 이 경우 stack의 크기를 모르면 탐지할 수 없기 때문에 제안한 도구가 BAP_toolkit보다 Heap영역에서도 더 높은 탐지를 하였다.

Bap_toolkit에서는 기호실행을 사용한 방법으로 프로그램 실행과정을 에뮬레이션하기 때문에 많은 분석 시간이 소요되었다. 그러나 본 논문의 도구는 정적 분석만을 사용하였으며, Stack구조를 복구하기 위한 일부 연산 작업만 필요하기 때문에 2038개의 샘플을 동일한 환경에서 구동하였을 때 Bap_toolkit의 분석 시간은 10시간 58분이며, 제안된 도구의 분석 시간은 3시간 06분으로 기존의 도구보다 더욱 효율적으로 분석하였다.

V. 결 론

기존의 Out-of-bounds Read를 탐지하기 도구는 동적 분석을 이용한 탐지 방식이 주를 이루었다. 동적 분석의 경우 프로그램 실행 과정에서 모든 코드를 커버하지 못할 수 있으며, 프로그램의 크기가 커질수록 분석 시간과 코드 커버리지에서 효율적이지 않을 수 있다. 또한 기존의 동적 분석 도구의 경우 Heap 영역을 기반으로 한 도구이며 Stack 영역은 탐지하지 않는다. 이를 보완하기 위해 본 논문에서는 바이너리 정적 분석을 이용하여 Heap영역뿐만 아니라 Stack영역의 취약점을 찾는 방법을 제안하였다.

기존의 도구와 제안한 도구를 비교한 결과 제안한 도구가 Heap영역뿐만 아니라 Stack 영역, Global 영역을 고려하여 탐지 커버리지를 높였으며, 기존의 기호 실행 도구보다 탐지 효율에서도 의미 있는 결과를 얻었다. 앞으로 도구를 추가로 개선하였을 때 더욱 긍정적인 결과가 기대된다.

향후 연구로는 현재 Out-of-bounds Read만을 고려하고 있지만 Write까지 범위를 넓힐 수 있다. 또한 현재 도구는 특정 함수 심볼을 이용한 탐지 방식으로 제한적으로 취약점을 탐지하고 있지만, 일반적인 메모리 접근 시에 발생하는 취약점을 분석하고 탐지하는 것도 향후 과제이다.

References

[1] Common Weakness Enumeration, "CWE Ranking" https://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html,

2021-04-30.
[2] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker." in USENIX Annual Technical Conference, pp.309 - 318, 2012.
[3] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "Softbound: Highly compatible and complete spatial memory safety for c," ACM Sigplan Notices, vol. 44, no. 6, pp. 245 - 258, 2009.
[4] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in ACM Sigplan notices, vol. 42, no. 6. ACM, pp.89 - 100, 2007.
[5] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "Cets: compiler enforced temporal safety for c," in ACM Sigplan Notices, vol. 45, no. 8. ACM, pp.31 - 40, 2010.
[6] cwe_checker github repo, "cwe checke r" https://github.com/ fkie-cad/cwe_ch ecker, 2021-07-19.
[7] American fuzzy lop, "American fuzzy lop" http://lcamtuf.coredump.cx/afl/.", 2021-04-30.
[8] C. Lemieux, R. Padhye, K. Sen, and D. Song, "Perffuzz: automatically generating pathological inputs," in Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis. ACM, pp. 254 - 265, 2018.
[9] C. Lemieux and K. Sen, "Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," in Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. ACM, pp. 475 - 485, 2018.
[10] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. ACM, pp. 2329 - 2344, 2017.
[11] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. ACM, pp. 1032 - 1043, 2016.
[12] bap-toolkit github repo, "BAP Toolkit" https://github.com/BinaryAnalysisPlat form/bap-toolkit-manager, 2021-04-30
[13] BRUMLEY, David, et al, "BAP: A binary analysis platform," International Conference on Computer Aided Verification. Springer, Berlin, Heidelberg, p. 463-469, 2011.
[14] Binary Analysis Platform github repo, "Binary Analysis Platform" https://git hub.com/BinaryAnalysisPlatform/bap, 2021-04-30.
[15] AddressSanitizerComparisonOfMemory Tools, "sanitizers" https://github.com/ google/sanitizers/wiki/AddressSanitize rComparisonOfMemoryTools, 2021-04-30.
[16] A. Filieri, C. S. Păsăreanu, and W. Visser, "Reliability analysis in symbolic pathfinder," in 2013 35th International Conference on Software Engineering. IEEE, pp.622 - 631, 2013.
[17] S. Person, G. Yang, N. Rungta, and S. Khurshid, "Directed incremental symbolic execution," in Acm Sigplan Notices, vol. 46, no. 6. ACM, pp. 504 - 515, 2011.
[18] J. H. Siddiqui and S. Khurshid, "Staged symbolic execution," in Proceedings of the 27th Annual ACM

Symposium on Applied Computing. ACM, pp. 1339 - 1346, 2012.

[19] Y. Li, Z. Su, L. Wang, and X. Li, "Steering symbolic execution to less traveled paths," in ACM SigPlan Notices, vol. 48, no. 10. ACM, pp. 19 - 32, 2013.

[20] T. Xie, N. Tillmann, J. de Halleux, and W. Schulte, "Fitness-guided path exploration in dynamic symbolic execution," in IEEE/IFIP International Conference on Dependable Systems & Networks. Citeseer, pp. 359 - 368, 2009.

[21] [34] B. C. Parrino, J. P. Galeotti, D. Garbervetsky, and M. F. Frias, "Tacoflow: optimizing sat program verification using dataflow analysis," Software & Systems Modeling, vol. 14, no. 1, pp. 45 - 63, 2015.

[22] C. Cadar, D. Dunbar, D. R. Engler et al., "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs." in USENIX Symposium on Operating Systems Design and Implementation, vol. 8, pp. 209 - 224, 2008.

[23] W. Visser, C. S. Pasareanu, and S. Khurshid, "Test input generation with java pathfinder," ACM SIGSOFT Software Engineering Notes, vol. 29, no. 4, pp. 97 - 107, 2004.

[24] S. Anand, C. S. Păsăreanu, and W. Visser, "Jpf - se: A symbolic execution extension to java pathfinder," in International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, pp. 134 - 138, 2007.

[25] WANG, Haijun, et al, "Locating vulnerabilities in binaries via memory layout recovering," Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 718-728, 2019.

[26] G. Balakrishnan and T. Reps, "Analyzing memory accesses in x86 executables," in International conference on compiler construction. Springer, pp. 5 - 23, 2004.

[27] J. Lee, T. Avgerinos, and D. Brumley, "Tie: Principled reverse engineering of types in binary programs," 2011.

[28] Z. Lin, X. Zhang, and D. Xu, "Automatic reverse engineering of data structures from binary execution," in Proceedings of the 11th Annual Information Security Symposium, pp. 1-1, 2010.

[29] Slowinska, Asia, Traian Stancescu, and Herbert Bos, "Howard: A Dynamic Excavator for Reverse Engineering Data Structures," NDSS, 2011.

[30] "Body armor for binaries: Preventing buffer overflows without recompilation." in USENIX Annual Technical Conference, pp. 125 - 137, 2012.

[31] NIST, "Juliet Test Suite" https://samate.nist.gov/SARD/testsuite.php, 2021-04-30

〈저자소개〉

유 동 민 (Dong-Min Yoo) 학생회원
2019년 2월: 한양대학교 ERICA 컴퓨터공학과 학사
2019년 8월~현재: 한양대학교 컴퓨터공학과 석사과정
〈관심분야〉 바이너리 분석, 취약점 분석, 디지털포렌식

김 문 회 (Wen-Hui Jin) 학생회원
2013년 9월: HEILONGJIANG University 소프트웨어공학과 학사
2016년 3월~현재: 한양대학교 일반대학원 컴퓨터공학과 석 박사 통합과정
〈관심분야〉 바이너리 분석, 취약점 분석, 난독화

오 희 국 (Heekuck Oh) 종신회원
1982년: 한양대학교 전자공학과 졸업
1989년 아이오와주립대학 전자계산학과 석사
1992년 아이오와주립대학 전자계산학과 박사
1993년~1994년: 한국전자통신연구원 선임연구원
1995년 3월~현재: 한양대학교 컴퓨터공학과 교수
〈관심분야〉 암호기술응용, 시스템보안