*Journal of The Korea Institute of Information Security & Cryptology*
VOL.31, NO.4, Aug. 2021

ISSN 1598-3986(Print)
ISSN 2288-2715(Online)
https://doi.org/10.13089/JKIISC.2021.31.4.701

# 바이너리 분석을 통한 UNIX 커널 기반 File System의 TOCTOU Race Condition 탐지*

이 석 원,[1†] 김 문 희,[1] 오 희 국[2‡]
[1,2]한양대학교 컴퓨터공학과 바이오인공지능융합전공 (대학원생, 교수)

# Detecting TOCTOU Race Condition on UNIX Kernel Based File System through Binary Analysis*

SeokWon Lee,[1†] Wen-Hui Jin,[1] Heekuck Oh[2‡]
[1,2]Major in Bio Artificial Intelligence, Department of Computer Science and Engineering, Hanyang University (Graduate Student, Professor)

## 요 약

Race Condition은 둘 이상의 프로세스가 하나의 공통 자원에 대해 입력이나 조작이 동시에 일어나 의도치 않은 결과를 가져오는 취약점이다. 해당 취약점은 서비스 거부 또는 권한 상승과 같은 문제를 초래할 수 있다. 소프트웨어에서 취약점이 발생하면 관련된 정보를 문서화하지만 종종 취약점의 발생 원인을 밝히지 않거나 소스코드를 공개하지 않는 경우가 있다. 이런 경우, 취약점을 탐지하기 위해서는 바이너리 레벨에서의 분석이 필요하다. 본 논문은 UNIX 커널 기반 File System의 Time-Of-Check Time-Of-Use (TOCTOU) Race Condition 취약점을 바이너리 레벨에서 탐지하는 것을 목표로 한다. 지금까지 해당 취약점에 대해 정적/동적 분석 기법의 다양한 탐지 기법이 연구되었다. 기존의 정적 분석을 이용한 취약점 탐지 도구는 소스코드의 분석을 통해 탐지하며, 바이너리 레벨에서 수행한 연구는 현재 거의 전무하다. 본 논문은 바이너리 정적 분석 도구인 Binary Analysis Platform (BAP)를 통해 Control Flow Graph, Call Graph 기반의 File System의 TOCTOU Race Condition 탐지 방법을 제안한다.

## ABSTRACT

Race Condition is a vulnerability in which two or more processes input or manipulate a common resource at the same time, resulting in unintended results. This vulnerability can lead to problems such as denial of service, elevation of privilege. When a vulnerability occurs in software, the relevant information is documented, but often the cause of the vulnerability or the source code is not disclosed. In this case, analysis at the binary level is necessary to detect the vulnerability. This paper aims to detect the Time-Of-Check Time-Of-Use (TOCTOU) Race Condition vulnerability of UNIX kernel-based File System at the binary level. So far, various detection techniques of static/dynamic analysis techniques have been studied for the vulnerability. Existing vulnerability detection tools using static analysis detect through source code analysis, and there are currently few studies conducted at the binary level. In this paper, we propose a method for detecting TOCTOU Race Condition in File System based on Control Flow Graph and Call Graph through Binary Analysis Platform (BAP), a binary static analysis tool.

**Keywords:** TOCTOU Race Condition, Binary Analysis, Vulnerability Detection

Received(05. 20. 2021), Modified(06. 29. 2021), Accepted(06. 29. 2021)

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

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

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

# I. 서 론

현대 사회에서 소프트웨어에 대한 의존도는 점점 증가하고 있다. 일반적인 사무 업무에서부터 정밀 기계 공정이나 의료 기기, 금융 서비스와 같은 핵심적인 기반 서비스까지 크고 복잡한 소프트웨어 시스템들은 널리 사용되고 있다. 이에 따라 다양한 소프트웨어들이 생산되고 있으며, 2019년 기준 세계 소프트웨어 시장 규모는 14,780억 달러에 달한다[1]. 하지만, 소프트웨어의 품질은 이와 같은 규모에 미치지 못하는 경우가 종종 있다.

소프트웨어의 품질에 영향을 미치는 것은 크게 두 가지로 결함(fault)과 취약점(vulnerability)이다. 결함은 소프트웨어로 인해 사고로 연결될 수 있는 큰 오류이며, 취약점은 개발자들의 실수로 인해 소프트웨어에 발생할 수 있는 버그로 공격자에 의해 악의적으로 이용될 수 있다. 방사선 치료기기 Therac-25의 의료 사고는 소프트웨어 결함에 의한 사례로 유명하며, 버퍼 오버플로우, Race Condition 등은 취약점의 사례로 들 수 있다. 결함이나 취약점은 소프트웨어의 품질을 저해시키고 사용자로 하여금 소프트웨어에 대한 안전성과 신뢰도가 떨어지게 되며, 개발 측에서는 유지 관리 비용을 증가시키는 요인이 된다.

효율적인 소프트웨어 품질 관리를 위해 취약점이 발견되면 CVE(Common Vulnerabilities and Exposures)를 부여하여 이에 대한 내용을 문서화하고 기록한다[2]. 하지만 일부 소프트웨어 공급 업체에서는 취약점이 발생한 원인과 같은 상세한 내용에 대해서 설명하지 않는 경우가 있다. 또한, 소스코드를 공개하지 않거나 오랜 시간이 지나서 공개하기도 한다. 그 예로 소프트웨어 공급 업체인 Apple사의 Mac OS의 XNU 커널은 발견한 취약점의 개수와 유형에 대해서는 공개하고 있으나 취약점에 발생한 원인과 같은 자세한 내용은 외부에 공개하지 않는다. 그리고 Apple사의 오픈소스 방침으로 인해 취약점 패치를 수행한 최신 소스코드를 늦게 공개하여 발견된 취약점에 대해 신속한 보안 검증을 수행하기가 어렵다. 이처럼 취약점에 대한 자세한 정보나 소스코드가 없는 상태에서 신속한 보안 검증을 수행하기 위해서는 바이너리 수준의 분석이 필요하다.

또한, 소프트웨어 내에서 취약점을 탐지하기 위해 기준 가운데 취약점 유형을 확인해야 한다. 이는 CWE(Common Weakness Enumeration)이라고 하며, 소프트웨어 또는 하드웨어에서 발생 가능한 취약점을 분류하여 열거한 목록을 말한다.

여러 취약점 유형 가운데 본 논문에서 대상으로 하고 있는 TOCTOU Race Condition 취약점은 서비스 거부(DoS), 의도하지 않은 권한 상승을 유발하는 위험한 취약점 중에 하나이다. MITRE는 소프트웨어 개발자들이 흔히 유발하는 7개 유형의 취약점인 Seven Pernicious Kingdoms(7PK) 중 프로세스나 스레드와 관련된 Time and State로 분류[3]하였으며 이는 Table 1.에 정리하였다. 또한, 최근 Mac 전용 가상머신 소프트웨어인 VMWare Fusion에서 TOCTOU 문제가 발생하여 권한 상승을 유발하는 사례가 존재한다[4].

일반적으로 소프트웨어에서 발생하는 결함이나 취약점을 찾아내고 해결하기 위해서 사용되는 방법으로는 동적 분석 기법인 검사(testing)와 정적 분석 기법인 감사(auditing)가 있다. 검사 방법은 개발 중 또는 개발된 소프트웨어의 품질을 확인하는 것으로, 의도적으로 계산된 입력 값들을 연속적으로 프로그램에 대입하여 프로그램이 개발자가 의도한대로 동작하는지 확인하는 기법이다. 감사 방법은 프로그램에 존재하는 잠재적인 문제를 찾기 위해 여러 개발자들이 직접 코드를 검토하는 방법이다. 이 두 기법은 실제로도 유용한 방법이며, 지금까지도 사용되고 있다. 하지만 검사 방법은 프로그램이 실행되는 모든 경우의 수에 대해 수행되지 않으며, 테스트 케이스에 한해서 정상적으로 동작한다는 것만 보증되는 한계가 있다. 또한 감사 기법은 애매한 경우나 아직 잘 알려지지 않은 취약점을 찾는 것이 어렵다는 단점이 있다. 특히 소프트웨어의 코드 크기가 클 경우에는 감사 기법을 적용하기 어렵다.

Table 1. 7PK - Time and State

| ID | Name |
|---|---|
| CWE-364 | Signal Handler Race Condition |
| CWE-367 | Time-of-check Time-of-use (TOCTOU) Race Condition |
| CWE-377 | Insecure Temporary File |
| CWE-382 | J2EE Bad Practices: Use of System.exit() |
| CWE-383 | J2EE Bad Practices: Direct Use of Threads |
| CWE-384 | Session Fixation |
| CWE-412 | Unrestricted Externally Accessible Lock |

File System에서 발생하는 TOCTOU Race Condition 취약점을 탐지하기 위해 개발된 기존의 정적 분석 도구는 소스코드를 기반으로 한 탐지 방법을 사용하였다. 바이너리는 컴파일 되는 과정에서 소스코드 정보가 누락이 되는 경우가 있으므로 소스코드 기반의 탐지 방법을 그대로 적용하는 것은 어렵다. 2019년에 발표된 CWE-Checker[5]는 바이너리 분석을 통해 시그니처를 찾아내어 취약점을 탐지하는 자동화 분석 도구를 제안하였으며, TOCTOU Race Condition을 탐지하는 모듈을 포함하였다. 하지만, 단일 함수 내부에서 발생하는 경우만 탐지 가능하며 참조하는 자원이 서로 다르거나 서로 다른 함수 간에서 발생하는 경우(Inter-procedure)는 탐지하지 못하는 한계가 있다. CWE-Checker를 제외하고 TOCTOU Race Condition을 탐지하는 바이너리 정적 분석 도구는 거의 전무하다.

또한 기존의 연구에서는 소스코드를 기반으로 탐지를 수행했기 때문에 Linux 커널의 ELF 바이너리와 XNU 커널의 Mach-O 바이너리가 서로 다른 점을 고려하지 않았으며, CWE-Checker 또한 실제 Mach-O 바이너리까지 커버할 수 있는지 검증할 필요가 있었다.

본 논문에서는 정적 분석 도구를 활용한 바이너리 분석을 통해 UNIX 커널 기반 File System의 TOCTOU Race Condition 탐지 방법을 제안한다. 실제로 취약점을 포함한 소스코드의 바이너리 분석을 수행하였으며, 결과를 토대로 취약점을 유발하는 시그니처를 제안하였다. 또한, 해당 시그니처를 통해 취약점을 유발하는 함수 심볼들의 조합을 정의하였다. 실제 UNIX 커널 기반의 운영체제인 Linux와 XNU 커널의 바이너리인 ELF, Mach-O 바이너리를 대상으로 실험한 결과, TOCTOU Race Condition 취약점을 유발하는 224가지의 함수 조합에 대응 가능하며, 기존의 탐지 방법에서 발생하는 오탐을 개선하였다. 특히 대부분 소스코드 레벨에서 Linux의 ELF 바이너리를 대상으로 실험이 이뤄진 바는 있으나, 바이너리 레벨에서 XNU 커널의 Mach-O 바이너리를 대상으로 한 연구는 없었으므로 의미가 있다고 볼 수 있다.

본 논문은 다음과 같이 구성된다. 2장에서는 TOCTOU Race Condition과 운영체제별 바이너리의 특징과 Mach-O 바이너리 분석에 사용된 분석 도구의 배경지식을 소개한다. 3장에서는 TOCTOU Race Condition 취약점을 탐지하기 위해 연구된 방법에 대해 소개한다. 이후 4장에서는 실제 Mach-O 바이너리 분석을 통해 도출한 시그니처와 이를 탐지하는 방법에 대해 소개하며, 5장에서는 실험 결과를 보이고 6장에서는 결론과 향후 연구에 대해 논하고자 한다.

## II. 배경 지식

### 2.1 TOCTOU (Time-Of-Check Time-Of-Use) Race Condition

TOCTOU Race Condition은 소프트웨어에서 파일이나 디렉토리, 소켓이나 데이터베이스의 트랜잭션 등 특정 자원을 사용하기 전에 자원의 상태를 확인(Check)하는 시점과 사용(Use)하는 시점 사이에 공유 자원을 변경할 수 있는 시간이 존재하며, 공격자에 의해 자원의 상태 확인 결과를 변경하는 방식으로 우회하여 파일이나 메모리 등의 공유 자원에 접근할 수 있는 취약점이다. Fig. 1.은 TOCTOU Race Condition 취약점을 통해 공격자가 자원에 접근하는 과정을 도식화한 것이다.

특히 root 권한으로 실행되는 setuid 프로그램의 경우, 권한이 상승된 상태에서 자원에 접근할 수 있으므로 계정의 비밀번호와 같은 중요한 정보가 유출될 수 있는 큰 문제가 있다. 가장 일반적으로 System Call 함수인 access와 open 사이에서 발생하는 경우를 예로 들 수 있다. Fig. 2.는 일반적인 access/open을 사용하는 예시이다.

공격자는 해당 프로그램을 통해서 권한 상승을 통해 접근 불가능한 파일에 접근하는 것을 목표로 한다. 여기서 access/open 사이에는 TOCTOU Race Condition으로 인해 자원이 변경될 수 있으므로 안전하지 못한 코드이다. 공격자는 해당 취약점을 이용한 공격 코드를 작성할 수 있으며, Fig. 3.의 코드를 통해 access를 수행한 후 open이 수행되기 전에 원래 자원의 링크를 해제하고 시스템의 암

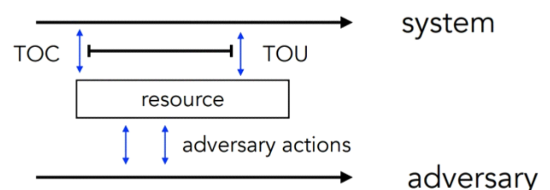


Fig. 1. TOCTOU Race Condition

```
void main(int argc, char **argv)
{
  int fd;
  /* If my invoker cannot access
     argv[1], then exit. */
  if (access(argv[1], R_OK) != 0)
    exit(1);
  fd = open(argv[1], O_RDONLY);
  /* Use fd... */
}
```

Fig. 2. Examples of general access/open usage including TOCTOU Race Condition vulnerability

```
void main(int argc, char **argv)
{
  /* Assume "file" refers to a file
     readable by the attacker. */
  if (fork() == 0) {
    system("victim file");
    exit(0);
  }
  usleep(1);

  unlink("file");
  link("/etc/shadow", "file");
}
```

Fig. 3. Attack code using TOCTOU Race Condition vulnerability

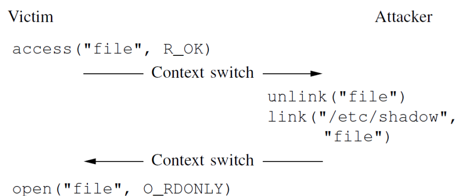


Fig. 4. Process flow when the TOCTOU Race Condition attack is successful

호를 가지고 있는 shadow 파일에 대한 링크를 생성하여 접근할 수 있게 된다.

Fig. 4.는 공격이 성공한 경우의 프로세스의 흐름을 도식화한 것이며, access가 호출된 이후 open이 수행되기 전에 공격자의 코드가 먼저 수행된다.

## 2.2 ELF (Executable and Linkable Format)

ELF(Executable and Linkable Format)는 실행 파일, 오브젝트 파일, 공유 라이브러리 및 코어 덤프를 위한 표준 파일 포맷이다. 1999년 86open 프로젝트에 의해 X86 프로세서 기반의 Unix 및 Unix 계열의 시스템에서 표준 바이너리 파일 포맷으로 채택되었다. ELF 포맷은 다양한 환경에서 오래된 실행 파일 포맷들을 대체했으며, 지금까지 Linux, FreeBSD, Solaris 등 다양한 운영체제에서 바이너리 표준으로 사용되고 있으며, Sony사의 Playstation 등 게임 콘솔에서도 사용하고 있다. ELF는 ELF Header와 파일 데이터로 구성되어 있으며, 32/64bit에 따라 Header의 형태가 약간 다르다[6].

## 2.3 Mach-O (Mach-Object)

Mach-O(Mach-Object)는 실행 파일, Object 코드, 라이브러리 등의 파일을 포함한다. 해당 포맷은 Mach 커널을 기반으로 하는 운영체제에서 사용된다. Mach-O 바이너리 포맷은 Mach-O Header와 SymTab Command, Main Command, Segment Command 등의 여러 Command와 __TEXT, __DATA 등의 이름을 지닌 Section이 존재한다[7].

## 2.4 중간표현 (Intermediate-Representation)

중간표현(Intermediate-Representation, IR)은 소스코드를 표현하기 위해 컴파일러 또는 가상의 시스템에서 내부적으로 사용하는 데이터 구조나 코드를 뜻한다. IR은 최적화나 번역과 같은 추가적인 처리를 용이하게 할 수 있도록 설계되었으며, 정보의 손실 없이 소스코드를 나타내는 것이 가능하다. 또한, 특정 언어에 구애받지 않고 소스코드를 표현하는 것이 가능하다는 점이 특징이다[8].

## 2.5 바이너리 분석 도구

바이너리 분석 도구는 바이너리를 IR로 변환하는 방식을 통해 분석을 지원한다. 그 중 대표적인 도구는 IDA Pro, Angr, BAP, Radare2가 있다.

IDA Pro[9]는 Hex-Rays사에서 개발한 상용 바이너리 분석 유틸리티로 현존하는 바이너리 분석 유틸리티 중 가장 강력한 기능을 지원하고 있다. 바이너리 분석 외에도 C/C++ 컴파일러로 컴파일 된 프로그램의 경우에는 디컴파일러 기능까지 지원하며, 대부분의 바이너리 포맷과 CPU 아키텍처에 대해 IR 변환을 지원하고 있다. 사용하는 IR 언어는 REIL이며, CFG 생성을 통한 분석 또한 지원한다.

Table 2. Performance comparison of binary analysis tools for Mach-O binaries

| | IDA Pro | Angr | BAP | Radare2 |
|---|---|---|---|---|
| IR Language | REIL | VEX | BIL | ESIL |
| File Parsing | O | △ | O | O |
| Function Identification Analysis | O | △ | O | O |
| Disassembly | O | O | O | O |
| IR Lifting | △ | △ | △ | O |
| CFG Construction | O | △ | O | O |
| Built-in Static Analysis | X | O | O | X |

Angr[10][11]는 UC Santa Barbara와 Arizona State University가 합작하여 만든 오픈소스 바이너리 분석 유틸리티로 C와 Python으로 개발되었다. 사용하는 IR은 VEX이며, 기호 실행, CFG 분석, Data-dependency 분석, Value-set 분석과 같은 정적 분석 기법을 지원한다.

BAP(Binary Analysis Platform)[12][13]은 Carnegie Mellon University에서 개발한 오픈소스 바이너리 분석 유틸리티로, Ocaml로 개발되었다. 사용하는 IR은 BIL이며, Angr와 마찬가지로 바이너리에 대한 정적 분석 기법을 지원한다. 그리고 별도의 수정을 통해 다른 바이너리 포맷들에 대한 분석이 가능하도록 확장이 가능하며, CWE-Checker, BAP-Toolkit과 같은 취약점 분석 모듈을 추가할 수 있다는 특징이 있다.

Radare2[14]는 오픈소스 바이너리 분석 유틸리티로 C언어로 개발되었다. 사용하는 IR은 ESIL이며, 상용 바이너리 분석 유틸리티인 IDA Pro만큼 다양한 바이너리 포맷과 CPU 아키텍처에 대해 지원한다.

## III. 관련 연구

Unix 커널 기반의 File System TOCTOU Race Condition 취약점을 탐지하기 위한 연구는 정적/동적 분석을 통해 다양하게 진행되어 왔다.

Matt Bishop의 연구[15]는 소스 코드 레벨에서의 정적 분석을 통해 취약점을 탐지하는 방법을 제안하였으며, 호출 의존성 그래프(Call Dependency Graph)를 통해 프로그램에서 발견될 수 있는 잠재적 Check/Use 함수 Call의 Interval을 결정하고, 데이터 흐름 분석(Data Flow Analysis)을 통해 System Call에 대한 Argument가 Interval을 생성하는지를 결정하는 방식을 사용하여 탐지하는 방법을 제시하였다.

Hao Chen의 연구[16]에서는 MOPS라는 정적 분석 도구를 제시하였다. 탐지 방식은 흔히 알려진 보안 속성인 Temporal safety Property를 유한 상태 오토마톤(Finite State Automaton)을 사용하여 정의, 소스코드를 Pushdown Automation을 사용하여 모델링하고 사전에 정의한 FSA와 소스코드를 통해 생성된 모델 검사를 통해 Temporal safety property의 위반 여부를 판단하여 취약점을 탐지하는 방법을 제시하였다. Control Flow 분석을 주로 하며 Data Flow는 배제하는 방식을 취하여 확장성과 효율성은 높였지만 정확성이 떨어진다는 점이 있다.

Brian V. Chess의 연구[17]는 Eau Claire라는 소스코드 기반의 정적 분석 도구을 제안하였다. Function Specification이라는 개념을 사용하여 해당 함수가 특정 조건을 벗어나는 경우에는 취약점이 발생할 수 있다고 사전에 정의하는 방식을 사용하였다. 전체적인 탐지 과정은 탐지 대상 소프트웨어의 소스코드와 취약점의 발생 조건을 정의한 Function Specification을 Dijkstra Guarded Command[18]로 변환한다. 그리고 변환한 Dijkstra Guarded Command를 Verification Condition으로 변환, 마지막으로는 Simplify라는 자동 정리 증명기를 통해 검증이 유효하지 않다면 취약점이 존재한다고 판단하는 방법을 제안하였다. 하지만 사전에 충분한 Function Specification이 정의되어 있지 않은 경우에는 오탐(False Negative)가 발생할 수 있으며 확장성이나 효율성이 낮은 단점이 있다.

Jinpeng Wei의 연구[19]에서는 정적 분석 기법이 아닌 동적 분석 기법을 제안하였으며, TOCTOU

Race Condition을 유발할 수 있는 함수의 모든 조합을 Check-Use Call의 Pair를 정의하였으며, 이를 동적으로 모니터링하여 탐지하는 방법을 제시하였다. 하지만 모든 실행 경로를 고려하지 못하며, Check와 Use가 서로 다른 자원을 참조하더라도 동시에 두 System call이 의도치 않게 발생하는 경우 오탐이 발생한다.

그리고 상대적으로 가장 최근인 2019년에 공개된 CWE-Checker는 소스코드 레벨이 아닌 바이너리 레벨에서의 정적 분석을 통해 취약점을 탐지하는 기법을 제안하였다. BAP의 확장 모듈로 본 논문에서 대상으로 하고 있는 CWE-367 외에도 다양한 취약점 탐지가 가능하며, 바이너리의 CFG를 생성하여 Check에 사용되는 함수를 Source, Use에 사용되는 함수를 Sink로 하여 도달 가능한 경우를 판단하여 TOCTOU Race Condition 취약점을 탐지한다. 하지만 이 기법은 단일 함수 내에서만 발생하는 경우만 탐지가 가능하며, 참조하는 공유 자원이 서로 다른 경우와 서로 다른 함수 내에서 발생 가능한 경우는 고려하지 않았기 때문에 오탐이 발생한다. 또한, ELF 바이너리를 주 대상으로 하였기 때문에 XNU 커널의 Mach-O 바이너리에 대한 Coverage가 낮다.

정적 분석 방법을 채택한 기존의 연구에서는 소스 코드 레벨에서만 수행되었기 때문에 소스 코드가 없는 경우에는 적합하지 않다는 단점이 있으며, 사전에 정의된 조합이 충분하지 않을 경우에는 탐지 정확성 또한 저하된다. 또한, 동적 분석 방법은 정확성은 높지만 정적 분석 방법에 비해 모든 실행 경로를 확인하지 못하며, 사전에 정의한 Check-Use Pair가 의도치 않게 발생하는 경우에도 취약점으로 판단하는 오탐이 발생한다. 그리고 이전의 연구가 소스코드를 기반으로 탐지하는 방식을 채택하였기 때문에 Linux 의 ELF 바이너리에서는 정확한 탐지 결과가 나올 수 있지만 XNU 커널에서도 동일한 결과가 나온다고 보장할 수는 없다.

바이너리 레벨에서의 정적 분석을 수행하는 BAP의 확장 모듈인 CWE-Checker 또한 기존에 연구된 기법들과 비교하였을 때는 정확성과 Coverage 면에서는 완전하지 못하다. 본 논문에서는 단일 함수에서만 발생하는 취약점 외에도 서로 다른 자원을 참조하는 경우와 Inter-procedure 간 발생 가능한 취약점을 탐지하는 것을 목표로 한다.

## IV. 바이너리 분석을 통한 탐지 방법 제안

이전의 연구에서 TOCTOU Race Condition을 탐지하기 위해 사용된 정적 분석 기법들은 Linux 환경을 기반으로 하며, 소스 코드가 있다는 가정 하에 수행되었다. 따라서 동일한 UNIX 커널 기반의 운영체제인 Linux나 XNU라도 서로 다른 결과가 나올 수 있으며, 소스 코드가 없는 경우에는 적용하기 어렵다는  단점이 있다.

본 논문에서 제안하는 기법의 목표는 소스 코드가 없는 상태에서 바이너리 분석을 통해 취약점을 탐지하고 Linux와 Mac OS에 상관 없이 UNIX 커널 기반의 OS에서는 동일한 탐지 결과가 나오는 것이다. 특히 Mach-O 바이너리에서도 정확한 분석을 수행하기 위해 다양한 바이너리 분석 도구의 성능을 비교하였으며, 그 결과는 Table 2.에 정리하였다.

본 논문에서 제안하는 탐지 도구는 바이너리 분석 도구로 BAP(Binary Analysis Platform)을 사용하였다. 이는 서로 다른 기종의 아키텍처에서도 적용 가능한 중간표현 기반의 분석 기술을 사용하기 때문이다. 이를 이용한 분석 도구는 대표적으로 IDA Pro, Angr, BAP, Radare2, Ghidra와 같은 다양한 분석 도구들이 있으며, 각각의 도구마다 장단점이 있다. 그 중에서도 알려진 시그니처 패턴을 이용하여 취약점 탐지에 가장 적합한 것은 BAP이며, 확장 모듈인 CWE-Checker를 제공하므로 이를 기반으로 기존의 정적 분석 탐지 도구를 확장하여 보다 정확한 탐지 도구를 개발하였다.

### 4.1 Check-Use Pair 시그니처

UNIX 커널 기반의 File System을 대상으로 한 TOCTOU Race Condition은 특정한 함수의 조합으로 인해 유발할 수 있음을 Wei의 연구[19]에서 상세하게 정의하였으며, 이를 Check-Use Pair라 한다. 해당 연구 내용에서 정의한 취약점을 유발하는 함수의 Symbol은 크게 Creation, Remove, NormalUse, Check의 네 가지 유형으로 분류되며, 이는 Table 4.에 정리하였다. 또한 정의에 따라 각각의 유형에 속하는 함수의 심볼을 Table 5.에, 취약점을 유발하는 Check-Use Pair 조합을 Table 3.에 정리하였다.

Table 3. Classification of TOCTOU Pairs

| Use | Explicit check | Implicit check |
|---|---|---|
| Create a regular file | CheckSet×FileCreationSet | FileRemoveSet×FileCreationSet |
| Create a directory | CheckSet×DirCreationSet | DirRemoveSet×DirCreationSet |
| Create a link | CheckSet×LinkCreationSet | LinkRemoveSet × LinkCreationSet |
| Read/Write/Execute or Change the attribute of a regular file | CheckSet×FileNormalUseSet | (FileCreationSet×FileNormalUseSet)∪ (LinkCreationSet×FileNormalUseSet)∪ (FileNormalUseSet×FileNormalUseSet) |
| Access or change the attribute of a directory | CheckSet×DirNormalUseSet | (DirCreationSet×DirNormalUseSet)∪ (LinkCreationSet×DirNormalUseSet)∪ (DirNormalUseSet×DirNormalUseSet) |

Table 4. Classification of Check-Use Set

| Set Type | Subtype |
|---|---|
| CreationSet | FileCreationSet |
| | LinkCreationSet |
| | DirCreationSet |
| RemoveSet | FileRemoveSet |
| | LinkRemoveSet |
| | DirRemoveSet |
| NormalUseSet | FileNormalUseSet |
| | DirNormalUseSet |
| CheckSet | none |

Table 5. Classification of Function Symbol

| Set Type | Function Symbol |
|---|---|
| FileCreationSet | {creat, open, mknod, rename} |
| LinkCreationSet | {link, symlink, rename} |
| DirCreationSet | {mkdir, rename} |
| FileRemoveSet | {unlink, rename} |
| LinkRemoveSet | {unlink, rename} |
| DirRemoveSet | {rmdir, rename} |
| FileNormalUseSet | {chmod, chown, truncate, utime, open, execve} |
| DirNormalUseSet | {chmod, chown, utime, mount, chdir, chroot, pivot_root} |
| CheckSet | {stat, access} |

### 4.2 XNU 커널 바이너리 분석

동일한 UNIX 기반의 커널이라 하더라도 Linux의 ELF 바이너리와 Mac OS의 Mach-O 바이너리는 분명한 차이가 존재한다. 대부분의 기존 연구는 ELF 바이너리를 대상으로 수행되었기 때문에 Mach-O 바이너리에 그대로 적용 가능한지에 대해서는 검증이 되어있지 않다.

Mach-O 바이너리 탐지 또한 목표로 하고 있기 때문에 바이너리를 직접 확인할 필요가 있었으며, 바이너리 분석 도구인 BAP을 사용하여 실제 XNU 커널의 IR을 분석을 수행하였고 Linux에서 나타나는 함수 심볼과 동일한 지 확인하였다. 실제로 확인한 결과 Linux에서 나타나는 함수의 심볼과 미세한 차이가 있음을 확인하였으며, 함수 심볼 앞에 '_'가 추가되어 나타나는 것을 확인할 수 있었다. Fig. 5.는 실제 XNU 커널 기반의 Mach-O 바이너리를 분석한 결과를 나타낸 것이다.

```
cwe_367_mac:   file format Mach-O 64-bit x86-64

SYMBOL TABLE:
0000000100000000 g     F __TEXT,__text          __mh_execute_header
0000000100000e50 g     F __TEXT,__text          _main
0000000000000000         *UND*        ___memset_chk
0000000000000000         *UND*        _access
0000000000000000         *UND*        _close
0000000000000000         *UND*        _exit
0000000000000000         *UND*        _free
0000000000000000         *UND*        _malloc
0000000000000000         *UND*        _open
0000000000000000         *UND*        _write
0000000000000000         *UND*        dyld_stub_binder
```

Fig. 5. Function symbol form found in actual Mach-O binary

### 4.3 취약점 탐지 전략

본 논문의 목표는 Linux 커널의 ELF 바이너리만을 대상으로 하는 것이 아닌, XNU 커널의 Mach-O 바이너리를 포함하여 UNIX 커널 기반의 File System을 대상으로 한 TOCTOU Race Condition 취약점을 탐지하는 것을 목표로 하고 있다. 취약점 탐지 전략은 크게 두 가지의 경우로 나눠

서 수행하고자 한다. 첫 번째는 단일 함수 내에서 취약점이 발생하는 경우이며, 다른 하나는 서로 다른 함수 사이에서 취약점이 발생하는 경우이다. 그리고 이 두 가지 경우 모두 공통적으로 참조하고 있는 자원이 동일한 자원인가 아닌가에 대한 검증을 통해 취약점을 탐지한다.

1.단일 함수 내부에서 발생하는 경우
2.서로 다른 함수 내부에서 발생하는 경우

#### 4.3.1 단일 함수 내에서의 취약점 탐지

CWE-Checker는 제어 흐름 그래프(Control Flow Graph, CFG)를 통해 Check-Use Pair를 토대로 TOCTOU Race Condition을 탐지하는 모듈을 제공하고 있다. 하지만 기존의 방식은 서로 다른 자원에 접근하는 경우 오탐(False Positive)을 유발하였으며, 이는 정확성을 크게 저해하는 요소이다.

따라서 오탐을 줄이고 정확성을 향상시키기 위해 취약점을 유발하는 함수의 원형을 분석할 필요가 있었다. 그 결과, 4.1에서 정의된 Check-Use Pair 함수들은 공통점을 가지고 있었으며, 이는 함수의 첫 번째 인자로 file 또는 directory의 path를 입력받는 것을 확인할 수 있었다. 따라서 Check와 Use 함수가 호출 될 때, 첫 번째 인자 값을 비교하는 로직을 추가하였다. 그리고 4.2에서 확인된 XNU 커널 기반의 바이너리에서 확인된 함수의 심볼과 4.1에서 정의한 Check-Use Pair 시그니처를 추가하여 오탐 발생을 개선하였다.

개선된 CWE-Checker의 모듈은 다음과 같은 순서로 탐지를 수행한다.

1) BAP에서 생성한 IR을 통해 바이너리에서 Check-Use Pair에 상응하는 System call이 존재하는지 확인한다. 만약 존재한다면 Check Call은 CFG의 Source가, Use Call은 Sink가 된다. 이 때, Check와 Use System Call에 사용된 함수의 첫 번째 인자 값을 기록한다.
2) CFG를 통해 Check Call에서 Use Call까지 도달 가능한가에 대한 가능성을 판단한다. 도달할 수 있다고 하면 Interval이 존재하는 것이다.
3) Interval이 존재하는 것을 검증한 후에는 Check/Use Call의 첫 번째 인자 값으로 들어가는 path의 값을 비교한다. 두 Call에서 path의 값이 일치하는 경우, 최종적으로 TOCTOU Race Condition 취약점이 있는 것으로 판단한다.

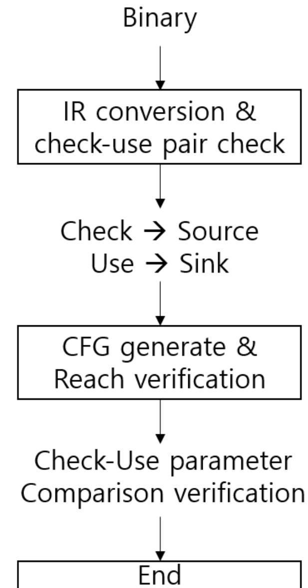


Fig. 6. Vulnerability detection process within a single function

#### 4.3.2 서로 다른 함수 간의 취약점 탐지

기존의 CWE-Checker는 CFG를 통한 취약점 탐지만 수행하기 때문에 서로 다른 함수 간에서 유발 가능한 TOCTOU Race Condition에 대해서는 CFG를 생성하지 못하므로 탐지하지 못하는 (False Negative) 한계가 있다. 특히 기존의 연구에서는 소스코드를 기반으로 한 정적 분석 기법을 통해 서로 다른 함수 간에서 발생 가능한 취약점까지 탐지할 수 있었으며, 이는 바이너리를 분석하였을 때에도 충분히 탐지할 수 있다고 판단하였다. 단, 소스코드가 없이 바이너리만 가지고 분석을 수행하기 때문에 기존의 탐지 방법을 그대로 적용할 수는 없다.

BAP에서는 Call Graph (CG)를 이용한 분석 방법을 제공하고 있다. BAP에서 생성된 IR에는 함수의 Call site의 정보 또한 유지된다. 이를 토대로 Call Graph를 생성할 수 있으며, 함수가 호출되는 순서를 분석할 수 있다.

CFG를 이용한 기존의 탐지 방법은 적용할 수 없

으므로 CWE-Checker의 탐지 모듈에 Call Graph를 이용한 분석 방법을 직접 구현하여 서로 다른 함수 간 발생할 수 있는 취약점 탐지 로직을 추가하였다.

Call Graph를 이용한 탐지 모듈은 다음과 같은 순서로 탐지를 수행한다.

1) BAP에서 생성한 IR을 통해 바이너리에서 Check-Use Pair에 상응하는 System call이 존재하는지 확인한다. 단일 함수 내에서의 취약점 탐지 로직을 통해 탐지할 수 있는지 확인한다. 만약 CFG가 생성되지 않을 경우에는 Call Graph를 생성한다. 여기서도 마찬가지로 Check는 Source로, Use Call은 Sink가 되며, Check와 Use System Call에 사용된 함수의 첫 번째 인자 값을 기록한다.
2) Call Graph를 통해 Check Call에서 Use Call의 선후 관계를 확인한다. Check Call이 먼저 발생하고 그 이후, 다른 함수를 통해서 Use Call이 발생하는 것을 확인하여 Interval 발생 가능성을 유추한다. Check Call이 먼저 발생한 이후 Use Call이 다른 함수 내부에서 호출 되는 경우 Interval이 존재하는 것이다.

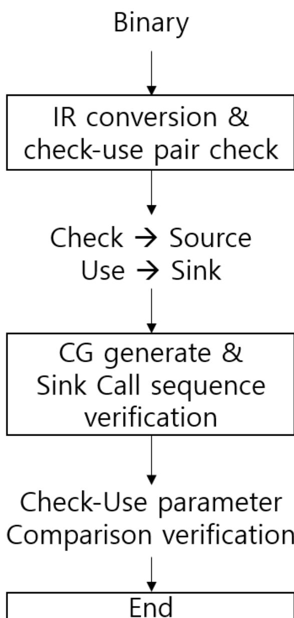


Fig. 7. Vulnerability detection process between different functions

3) Interval이 존재하는 것을 검증한 후에는 Check/Use Call의 첫 번째 인자 값으로 들어가는 path의 값을 비교한다. 두 Call에서 path의 값이 일치하는 경우, 최종적으로 TOCTOU Race Condition 취약점이 있는 것으로 판단한다.

## V. 실험 및 평가

본 논문에서 제안한 방법의 성능을 평가하기 위해 기존의 개발된 취약점 탐지 도구인 CWE-Checker를 대상으로 비교 분석하였다. 성능 평가를 위한 실험에 사용한 데이터 셋은 NIST에서 제공하는 Juliet Test Suites 1.3[20]을 사용하여 진행하였다. 해당 샘플은 TOCTOU Race Condition을 유발하는 {access, open}의 Pair와 {stat, open}의 Pair로 구성된 각각 18개의 소스 코드이다. 각각의 코드마다 Good 함수와 Bad 함수를 정의하여 취약점이 발생할 수 있는 경우를 분류하였다. 하지만, 해당 샘플은 오탐이 발생하는 경우가 반영되어 있지 않았다. 따라서 보다 정확한 성능 평가를 수행하기 위해 실험 데이터 셋을 일부 변경하여 추가하였다. 크게 다섯 가지의 유형으로 나눴으며, 하나는 정상 동작하는 Good 함수를 그대로 두었고, 두 번째는 취약점이 존재하는 Bad1 함수이다. 세 번째는 False Positive가 발생할 수 있는 서로 다른 자원을 참조하는 FP1 Case를 추가하였다. 네 번째 유형은 서로 다른 함수 간에서 취약점이 발생하는 경우인 Bad2이며, 마지막으로는 서로 다른 함수 간에서 서로 다른 자원을 참조하는 FP2 케이스를 추가하였다. 각 유형별 데이터 셋에 대해서는 Table 6.에 정리하였다.

총 180개의 샘플 데이터는 각각 Linux와 Mac OS에서 컴파일하여 ELF 바이너리와 Mach-O 바이너리로 별도로 생성하여 탐지를 수행하였다.

Table 6. CWE-367 Evaluation Dataset

| Test Case | access, open | stat, open |
|---|---|---|
| Good | 18 | 18 |
| Bad1 | 18 | 18 |
| Bad2 | 18 | 18 |
| FP1 | 18 | 18 |
| FP2 | 18 | 18 |
| **Total** | **90** | **90** |

### 5.1 실험 결과

Table 7.은 CWE-Checker를 통해 확인한 결과이다. Linux의 ELF 바이너리 180개, XNU의 Mach-O 바이너리 180개의 샘플 데이터를 통해 실험을 진행한 결과로는 서로 다른 함수 내에서 발생하는 Bad2 Case와 FP2 Case에 대해서는 완전히 탐지하지 못하는 것을 확인할 수 있었다. 또한 단일 함수 내에서 서로 다른 자원에 접근하는 경우에 False Positive가 발생하는 것이 확인되었다. 이는 Inter-procedure 간 발생할 수 있는 취약점에 대해서 반영하지 않은 것이며, 또한 Data Flow를 고려하지 않았기 때문에 FP1 Case로 인한 오탐이 다량으로 발생하였다. 또한, 초기의 CWE-Checker는 확장이 되어있지 않으므로 Mach-O 바이너리를 아예 인식하지 못하지만, 확장을 한 후 Mach-O 바이너리에서 발견되는 심볼까지 추가로 반영한 결과, Mach-O, ELF 바이너리 모두 탐지되는 것을 확인할 수 있었다.

Table 8.은 본 논문에서 제안한 방법을 통해 확인한 결과이다. 이전과 동일하게 Linux의 ELF 바이너리 180개, XNU의 Mach-O 바이너리 180개의 샘플 데이터를 통해 실험을 진행하였으며, 기존 탐지 도구에서 탐지하지 못한 서로 다른 함수 내에서 발생하는 Bad2 Case와 FP2 Case에 대해서도 대응할 수 있음을 확인하였다. 또한 단일 함수 내에서 서로 다른 자원에 접근하는 경우에 발생하는 FP1 Case에 대해서도 오탐 없이 정확한 탐지를 수행하는 것을 확인하였다.

Table 9.는 본 논문에서 제안한 방법과 CWE-Checker의 결과를 비교하여 요약한 것이다. 본 논문에서 제안한 방법은 오탐을 줄임으로써 정확성을 보다 높아졌음을 확인할 수 있었다.

Table 7. CWE-Checker Performance Valuation

| Test Case | Detection/Set | Rate (%) |
|---|---|---|
| Good | 0/72 | 0 |
| Bad1 | 72/72 | 100 |
| Bad2 | 0/72 | 0 |
| FP1 | 72/72 | 100 |
| FP2 | 0/72 | 0 |

Table 8. Proposed Approach Performance Valuation

| Test Case | Detection/Set | Rate (%) |
|---|---|---|
| Good | 0/72 | 0 |
| Bad1 | 72/72 | 100 |
| Bad2 | 72/72 | 100 |
| FP1 | 0/72 | 0 |
| FP2 | 0/72 | 0 |

Table 9. Summary of comparison for the proposed approach vulnerability detection in Juliet test suite

| False Alarm | Proposed | CWE-Checker |
|---|---|---|
| True Positive | 144 | 72 |
| False Negative | 0 | 72 |
| False Positive | 0 | 72 |

### 5.2 평가 및 논의

실험 결과에서는 본 논문에서 제안한 방법이 기존의 탐지 도구인 CWE-Checker와 비교하여 False Positive, False Negative가 모두 발생하지 않았다. 특히, False Positive 발생이 하나도 나오지 않은 부분에 대해서는 실험 대상으로 하고 있는 데이터 셋이 실제 상용 소프트웨어들과 비교하였을 때, 규모가 크지 않고 복잡도 또한 높지 않기에 본 논문에서 제안한 기법을 적용할 수 있었으며, 이와 같은 결과가 나타난 것으로 보인다.

또한, 제안하는 방법에서는 동일한 파일 포인터를 사용하거나 파일의 경로가 같은 경우만 대상으로 하였으며 절대/상대 경로를 대상으로 하거나 파일 경로를 입력 받는 경우는 고려하지 않았으므로 보다 복잡한 경우에 대해서는 결과가 다르게 나올 수 있을 것으로 판단하며, Flow 위주의 분석 외에도 Context를 기반으로 한 분석 기법을 적용하여 정확도를 높이는 방법도 고려할 수 있을 것으로 보인다.

향후 연구에서는 실험에서 사용한 데이터 셋만이 아닌 실제 상용 소프트웨어를 대상으로 규모가 크고 복잡한 경우를 다룰 필요가 있으며, Flow 기반의 분석 외에도 Context 기반의 분석을 통해 정확성을 높이고자 한다. 그리고 본 논문에서 제시한 취약점이나 오탐이 발생하는 경우 외에도 보다 다양한 경우에

대응할 수 있도록 제안한 방법을 개선하고자 한다.

## VI. 결 론

기존의 Unix 커널 기반의 File System TOCTOU Race Condition 취약점을 탐지하기 위해서 정적/동적 분석 방법을 막론하고 다양한 탐지 방법들이 제안되어 왔으며, 현재 CWE-Checker와 같이 바이너리 분석을 통해 취약점을 탐지하는 방식까지 제안이 되었다.

과거의 분석 기법은 주로 소스코드가 있다는 상황을 전제하에 둔 소스코드 기반의 취약점 탐지 방식이었으므로, 소스코드가 없는 특수한 상황에서 해당 기법을 적용하는 것을 매우 어려울 것이다. 또한 동적 분석의 경우, 정확성이 높으며 잘 알려지지 않은 경우의 TOCTOU Race Condition을 탐지하는 데에 특화되어 있어 새로운 유형의 취약점을 찾는 데에 보다 유용하다.

본 논문에서는 새로운 유형의 취약점을 찾는데 주를 둔 것이 아닌 기존의 알려진 취약점 유형에 대해서 효율적으로 찾아내기 위해 동적 분석이 아닌 정적 분석 방식을 채택하게 되었으며, 소스코드가 없는 특수한 상황 하에 신속한 보안 검증을 할 수 있는 바이너리 레벨의 분석을 통한 UNIX 커널 기반의 File System TOCTOU Race Condition 탐지 방법을 제안하였다.

기존의 탐지 도구들과 비교하였을 때, 본 논문에서 제안한 탐지 방법을 적용한 도구는 바이너리를 이용한 분석 도구보다 오탐이 적게 나오는 결과를 보였으며, 소스코드가 없이도 취약점을 정확하게 찾아낼 수 있음을 확인하였다.

향후 연구를 통해 취약점 탐지를 위해 만들어진 코드가 아닌 실제 소프트웨어의 바이너리 분석을 통해 도구를 보다 개선하는 것을 목표로 하고 있으며, 실제 어플리케이션 내에서 발생하는 다양한 경우의 취약점 발생 유형에 대해서 분석하고 탐지하는 것을 향후 과제로 한다.

## References

〔1〕 SPRi, “Global Software Market Stat,” https://stat.spri.kr/posts/view/22299?code=stat_sw_market_global, last accessed Feb. 2021.

〔2〕 MITRE CVE, “Common Vulnerability and Exposure”, https://cve.mitre.org/, last accessed Feb. 2021.

〔3〕 MITRE CWE, “7PK - Time and State,” https://cwe.mitre.org/data/definitions/361.html, last accessed Feb. 2021.

〔4〕 NIST National Vulnerability Database, “CVE-2020-3957,” https://nvd.nist.gov/vuln/detail/CVE-2020-3957, last accessed Feb. 2021.

〔5〕 Github, “CWE Checker,” https://github.com/fkie-cad/cwe_checker, last accessed Feb. 2021.

〔6〕 Wikipedia, “Executable and Linkable Format,”https://en.wikipedia.org/wiki/Executable_and_Linkable_Format, last accessed Feb. 2021.

〔7〕 Github, “Mach-O Format,” https://github.com/aidansteele/osx-abi-macho-file-format-reference, last accessed Feb. 2021.

〔8〕 Wikipedia, “Intermediate Representation,” https://en.wikipedia.org/wiki/Intermediate_representation, last accessed Feb. 2021.

〔9〕 Hex-rays, “IDA Pro Decompiler,” https://www.hex-rays.com/products/idahome/, last accessed Feb. 2021.

〔10〕 Wang, Fish, and Yan Shoshitaishvili. “Angr-the next generation of binary analysis,” IEEE Cybersecurity Development (SecDev), pp.8-9, 2017.

〔11〕 Github, “Angr,” https://github.com/angr/angr, last accessed Feb. 2021.

〔12〕 Brumley, David, et al. “BAP: A binary analysis platform,” International Conference on Computer Aided Verification, pp.463-469, 2011.

〔13〕 Github, “Binary Analysis Platform,” https://github.com/BinaryAnalysisPlatform/bap, last accessed Feb. 2021.

〔14〕 Github, “Radare2,” https://github.com/radareorg/radare2, last accessed Feb. 2021.

〔15〕 Bishop, Matt, and Michael Dilger. "Checking for race conditions in file accesses," Computing systems, vol. 9, no. 2, pp.131-152, 1996
〔16〕 Chen, Hao, and David Wagner. "MOPS: an infrastructure for examining security properties of software," Proceedings of the 9th ACM Conference on Computer and Communications Security, pp.235-244, 2002.
〔17〕 Chess, Brian V. "Improving computer security using extended static checking," Proceedings 2002 IEEE Symposium on Security and Privacy, pp.160-173, 2002.
〔18〕 Dijkstra, Edsger W. "Guarded commands, nondeterminacy and formal derivation of programs," Communications of the ACM, vol. 18, no. 8, pp.453-457, Aug. 1975.
〔19〕 Wei, Jinpeng, and Calton Pu. "TOCTTOU Vulnerabilities in UNIX-Style File Systems: An Anatomical Study," Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST), pp.155-167, 2005.
〔20〕 NIST SARD, "Juliet Test Suite," https://samate.nist.gov/SARD/testsuite.php, last accessed Feb. 2021.

## 〈저자소개〉

이 석 원 (SeokWon Lee) 학생회원
2018년 2월: 인하공업전문대학 컴퓨터정보공학과 학사
2019년 8월~현재: 한양대학교 컴퓨터공학과 석사과정
〈관심분야〉 바이너리 분석, 취약점 분석

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

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