Journal of The Korea Institute of Information Security & Cryptology ISSN 1598-3986(Print)
VOL.29, NO.5, Oct. 2019 ISSN 2288-2715(Online)
https://doi.org/10.13089/JKIISC.2019.29.5.1039

프로그램 분석을 위한 정적분석 기반 역추적 제어흐름그래프 생성 방안 모델*

박 성 현,† 김 연 수, 노 봉 남‡

전남대학교 정보보안협동과정

Static Analysis Based on Backward Control Flow Graph Generation Method Model for Program Analysis*

Sunghyun Park,† Yeonsu Kim, Bongnam Noh‡
Interdisciplinary Program of Information Security, Chonnam National University

요 약

프로그램 자동 분석 방법 중 하나인 기호 실행은 지난 몇 해 동안 기술적으로 크게 향상 되었다. 그러나 여전히 기호실행 그 자체만을 이용하여 프로그램을 분석하는 것은 실용적이지 않다. 가장 큰 이유로는 프로그램 분석 중에 발생하는 경로 폭발 문제로 인한 메모리 부족으로, 기호 실행을 이용해 프로그램의 모든 경로의 해를 구할 수 없다. 따라서 분석가는 모든 경로의 해를 구하는 것이 아닌 취약성을 갖는 지점으로 기호 실행 탐색 경로를 구성하는 것이 실용적이다. 본 논문에서는 기호 실행 과정에서 사용될 수 있는 정적분석 기반 바이너리 역방향 제어 흐름 그래프 생성 방법 기술을 제안한다. 역방향 제어 흐름 그래프 생성을 통해 분석가는 바이너리 내의 잠재적인 취약지점을 선정할 수 있고, 해당 지점으로부터 생성된 역추적 경로는 향후 기호 실행을 위해 효율적으로 사용될 수 있다. 우리는 리눅스 바이너리(x86)를 대상으로 실험을 진행하였고, 실제로 잠재적인 취약점 선정 및 역추적 경로 생성이 바이너리의 다양한 상황에서 가능함을 보였다.

ABSTRACT

Symbolic execution, an automatic search method for vulnerability verification, has been technically improved over the last few years. However, it is still not practical to analyze the program using only the symbolic execution itself. One of the biggest reasons is that because of the path explosion problem that occurs during program analysis, there is not enough memory, and you can not find the solution of all paths in the program using symbolic execution. Thus, it is practical for the analyst to construct a path for symbolic execution to a target with vulnerability rather than solving all paths. In this paper, we propose a static analysis - based backward CFG(Control Flow Graph) generation technique that can be used in symbolic execution for program analysis. With the creation of a backward CFG, an analyst can select potential vulnerable points, and the backward path generated from that point can be used for future symbolic execution. We conducted experiments with Linux binaries(x86), and indeed showed that potential vulnerability selection and backward CFG path generation were possible in a variety of binary situations.

Keywords: Static Analysis, Symbolic Execution, Binary Vulnerability, Control Flow Graph

Received(07. 16. 2019), Modified(08. 31. 2019), Accepted(09. 01. 2019)

* 이 논문은 ETRI부설연구소의 2019년도 위탁연구과제로 수행한 결과입니다.

† 주저자, everpall@gmail.com
‡ 교신저자, bbong@jnu.ac.kr(Corresponding author)

I. 서 론

최근 소프트웨어 분석의 자동화 기술 연구는 단일 경로의 테스트 케이스 생성뿐만 아니라, 다양한 테스트 케이스 중에서 취약점으로 도달 할 수 있는 최적화된 경로를 파악하는 것에 중점을 두고 있다[1, 2,3,4]. 이러한 자동화 기술 중 기호 실행(Symbolic Execution)[5] 기술이 각광 받고 있다. 기호 실행은 말 그대로 추적하고 싶은 변수를 기호적인 표현으로 두고 해당 기호 변수로 인해 발생하는 방정식의 해를 풀어내는 기술이다. 즉, 기호 실행은 특정 경로에 도달하기 위핸 하를 풀어내는 방식으로 소프트웨어의 특징이나 행위를 분석하는 연구가 가능하다. 논리적으로는 프로그램의 모든 해가 존재할 경우 모든 영역의 해를 구함으로써 바이너리 코드의 모든 경로를 탐색할 수 있는 장점을 갖고 있다.

이처럼 기호 실행은 소프트웨어 취약점을 찾는데 유용하게 사용될 수 있고 KLEE[6], CUTE[7], DART[8], CREST[9], S2E[10]와 같은 자동화 기호 실행 도구들이 존재한다. 하지만 기호 실행은 프로그램의 규모와 복잡도에 큰 영향을 받는다. 경로가 증가될 때 마다 각 경로를 기술하는 문맥이 생성되어야 하는데, 급격하게 경로가 증가될 경우 이를 관리하는데 어려움이 발생한다. 이러한 문제로 기호 실행은 여러 최적화 기법을 적용하는 연구가 진행되어 왔다. 또한 여러 가지 경로 삭제 알고리즘이 제안되어 관심 없는 프로그램 경로는 기호 실행에서 제외될 수 있도록 하였다[11].

본 논문에서는 효율적인 기호 실행을 위한 역방향 CFG(Control Flow Graph, 이하 CFG) 생성 방법에 대해 소개한다. 역방향 CFG는 프로그램을 분석하는 과정에서 잠재적인 취약점을 선정할 수 있고, 기호 실행을 위한 사전 탐색 경로를 생성하기 위한 방법이 될 수 있다. 분석가는 초기 목표 지점을 선정 및 생성된 CFG 경로를 통해 분석 대상에 대한 탐색 영역을 최소화 할 수 있다.

우리는 정적 모델링 방법을 제안하고, 먼저 잠재적인 취약 지점 후보군을 선정한다. 이는 메모리 복사 함수를 호출하는 인스트럭션을 대상으로 하였다. 이후 역추적 분석을 통해 후보군 중에서 사용자 입력에 영향을 미치는 인스트럭션 및 역추적 경로를 파악하는 경로를 생성하는 것을 목표로 한다. 이는 프로그램 분석에 있어서 흥미로운 영역만을 탐색하도록 유도함으로써 향후 기호 실행에서 발생할 수 있는 경로 폭발 문제를 간접적으로 완화시키는 것 또한 가능하다.

본 논문의 구성은 2장에서 현재 사용되고 있는 데이터 흐름 분석 기술에 대해 설명하고, 3장에서는 역방향 CFG 생성을 위한 정적 모델링 방법을 기술한다. 4장에서는 역방향 CFG 생성에 관한 실험 및 평가를 진행하고, 마지막으로 5장에서는 결론에 대한 내용을 기술한다.

II. 관련 연구

2.1 데이터 흐름 분석 기법 연구

2.1.1 정적 오염분석을 통한 실행 경로 탐색 연구

VERIMAG팀은 프로그램 입력으로부터 위험한 함수까지 도달할 수 있는 경로를 밝히는 연구를 진행하였다[12]. 위험한 함수란, strcpy()와 같은 안전하지 않은 함수나 Credential checkings 처리와 같은 Code의 Critical Part를 지칭한다. 결국 이 알고리즘은 임의 프로그램 입력 집합(IS)과 위험한 함수 집합(VF)을 입력하여 IS에서 VF까지 도달 가능한 모든 경로 집합을 돌려준다.

$$x = IS() \cdots \rightarrow \cdots \; y := x \cdots \rightarrow \cdots \; VF(y)$$

Fig. 1. Insecure route acquisition issues

이 문제를 해결하기 위하여 정적 오염분석 알고리즘이 활용되었다. 정적 오염분석 알고리즘은 변수의 데이터 의존관계를 식별하는데 적합하다. 즉 Fig. 1에서 변수 x는 프로그램 입력을 받는다. 그리고 변수 y는 x와 데이터 의존관계가 있다. 마지막 VF함수가 y의 값을 이용하는데 이 y는 프로그램 입력 변수인 x와 의존관계가 있으므로 해당 경로는 안전하지 않은 실행경로이다. 오염분석 알고리즘은 데이터 의존관계를 밝히는 것으로 데이터 흐름분석 프레임웍을 이용 하여 구현할 수 있다.

2.1.2 동적 오염분석을 통한 실행경로 탐색 연구

Rohit Mothe은 프로그램이 크래시가 발생하였을 때, 크래시가 발생한 인스트럭션으로부터 입력지점까지 역추적을 진행하였다[13]. 입력 파일 생성 문제를 해결하기 위해 해당 연구원들은 역방향 및 정

방향 오염분석방법을 제안하였다.

이러한 두 가지 접근 방식을 모두 활용하고 Crash 가 발생한 순간에 Crash 상황에 영향을 주는 입력 영역 매핑, 잠재적 기능 분석을 제공하는 단일 프레임 워크에 코드 실행 달성을 위한 접근 방식을 통합하였다. 마지막으로, 저자가 발견, 분석 / 악용 및 보고 한 몇 가지 사항을 포함하여 공개 된 취약점이 있는 통합 도구의 사례를 보여주었다.

해당 연구는 동적 분석을 기반으로 프로그램이 시작하는 과정에서부터 Crash가 발생할 때까지의 모든 인스트럭션을 기록하였다. 그리고 최종적으로 Crash가 발생하였다면, 해당 지점에서부터 수집된 인스트럭션을 기준으로 역방향 오염분석을 진행한다. 결국 크래시가 발생하였을 때, 분석가는 입력의 오프셋 및 해당 테스트 케이스 값을 추출할 수 있다.

2.1.3 중간 표현식(Intermediate Representation)에 의한 데이터 흐름 분석

Dowser는 오염 추적, 프로그램 분석 및 기호 실행을 결합하여 프로그램의 논리에 깊숙이 묻혀있는 버퍼 오버플로우 및 취약한 버그를 찾는다[14]. Dowser의 핵심 아이디어는 프로그램 분석을 통해 소스코드에서 예비 타겟 지점을 추출하고, 그에 대한 입력 값을 생성하는 테스트 케이스를 추출할 수 있음을 보여준다. 그 중에서 취약점에 사용될 수 있는 복잡한 포인터 연산을 타겟으로 분석을 진행한다.

포인터 p가 루프 내의 "흥미로운" 배열 액세스 명령 acc_p에 관련된다고 가정 해 보면, acc_pp와 연관된 분석 그룹 인 $AG(Acc_p)$는 루프 실행 중 역 참조된 포인터의 값에 영향을 주는 모든 명령어를 수집한다. $AG(Acc_p)$를 결정하기 위해, Dowser는 acc에서 역 참조 된 p의 값을 계산하는 루프 내의 연산을 나타내는 프로시저 내 데이터 흐름 그래프를 계산한다.

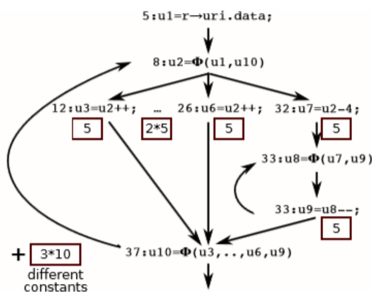


Fig. 2. Static data flow graph with SSA Form

해당 연구는 LLVM 컴파일러의 중간 표현식을 통해 구현되었다. LLVM에서 제공되는 정적 단일 할당(SSA Form : Static Single Assignment Form)[16]형식은 데이터 흐름 그래프로 직접 변환된다. 따라서 프로시저 내의 하나의 변수는 독립적인 version을 갖는 변수로 취급될 수 있다. Fig. 2는 Dowser에서 분석 그룹 데이터 흐름 예제를 보여준다. 그림에서 각각은 소스 라인과 그에 해당하는 소스코드를 가리킨다. 그리고 SSA Form을 통해 포인터 u(version)에 대한 프로시저 내 데이터 흐름을 분석한다. 포인터의 모든 역 참조는 데이터 흐름 그래프를 공유하므로 단일 분석 그룹을 형성한다는 것을 알 수 있다.

2.2 바이너리 기반 기호 실행 방법 연구

Dowser가 오픈소스를 대상으로 테스트 가능 한 반면, BORG는 바이너리에서 작동하며 소스 코드가 필요하지 않은 상태에서 대부분의 Dowser 기능을 흡수하였다[15]. BORG는 실제 프로그램에서 버퍼 오버로드 버그를 감지하기 위해 정적 및 동적 프로그램 분석, 오염 전파 및 기호 실행을 사용한다. BORG는 먼저 오버플로로 이어질 수 있는 버퍼 액세스를 선택하고 실제로 오버로드로 이어질 수 있는 프로그램 경로상의 액세스를 향해 기호 실행을 유도한다.

BORG는 가능한 한 많은 코드를 다루는 방법 대신 테스트중인 프로그램의 "흥미로운" 타겟 부분을 향해 실행을 안내한다. 해당 버그를 위반하는 동안 이러한 흥미로운 부분을 향해 실행을 안내하는 경로 선택 전략을 사용한다. 이러한 안내 된 기호 실행(Guided Symbolic Execution)을 사용하여 오버로드를 허용 할 수 있는 코드로 프로그램 탐색을 안내한다. BORG는 프로그램 흐름 탐색을 위해 타겟 인스트럭션으로부터 가장 최단거리에 있는 블럭을 우선적으로 선택하는 알고리즘을 채택하였다. 하지만 초기 타겟 선정 문제 및 바이너리 정적 분석과정에서 발생하는 간접점프 문제 등에 언급하였고, 이를 한계점으로 삼았다.

III. 역방향 CFG 생성을 위한 정적 모델링 방법

바이너리 기반 정적 역추적 데이터 흐름 분석을 통한 최종 목적은 후보 타겟 지점에서 최종적으로 입력 함수까지의 데이터 흐름을 추적이 가능한가이다. Fig. 3의 최초 후보 지점은 다음과 같이 sub3에 있고 이는 sub2로 부터 호출된 함수이다. 또한 sub2는 sub1로부터 호출된 함수이다. 따라서 sub3에서 호출되는 취약 함수 후보군인 memcpy의 size 인자를 추적하여 sub1의 입력 함수로부터 전달되었는지 경로를 파악하는 것이 본 정적 모델링의 목표이다.

이 논문에서는 이러한 과정은 기호 실행을 위한 사전 단계에서 실행한다고 하여, 예비 분석(preliminary analysis)이라 지칭한다. 이 단계는 보통 정적 분석을 기반으로 이루어지며, 잠재적 취약점 발생 지점과 해당 지점에 영향을 주는 입출력 함수 호출 지점의 경로를 추출하는 데 목적을 둔다. 이를 위해 역방향 정적 프로그램 분석은 세 가지 정적 모델링 적용을 통해 의존 경로를 파악할 수 있다.

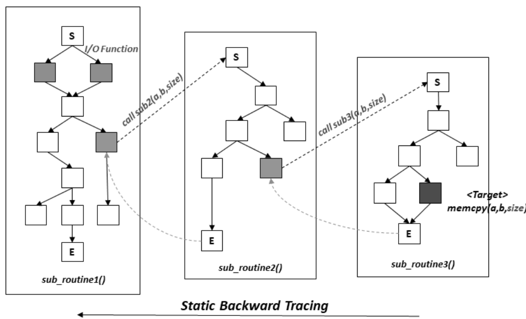


Fig. 3. Static Backward Tracing Overview

3.1 M1: 초기 후보 지점 선정 및 결정적인 값 제거

역추적 CFG 생성을 위한 가장 초기 단계는 후보 지점을 선정하는 것이다. 입출력 함수(I/O function)와 데이터 전달 함수(data transfer functions)가 호출되는 인스트럭션 지점을 모두 수집한다. Fig. 4는 POSIX에서 제공하는 메모리 복사 함수 및 입력 입출력 함수를 보여준다. 각 함수의 심볼(symbol)을 이용하여 해당하는 모든 코드 참조를 식별하고 검토한다. 여기서 심볼은 디버깅 과정에서 발견할 수 있는 함수 명을 가리킨다.

본 논문에서는 기본적으로 Out-of-bounds 취약

```
<I/O function symbols>
read, open, fread,
fscanf, scanf, sscanf,
fwscanf, swscanf, wscanf, poll ..

<Data transfer function symbols>
memcpy, memset, strcpy, strcpyA, strcpyW, tcscpy,
mbscpy, StrCpy,StrCpyW,
lstrcpy,lstrcpyA,tccpy,mbccpy,ftcscpy,wcsncpy,
tcsncpy, mbsncpy,StrCpyN, StrCpyNA,
StrCpyNW,strcpynA, StrNCpyA, StrNCpyW,
lstrcpynA, lstrcpynW ..
```

Fig. 4. I / O and data transfer function symbols for code reference identification

점을 유발하는 함수를 대상으로 하였다. 일반적으로 모든 메모리 복사 함수를 호출하는 인스트럭션을 취약 대상 후보군으로 선정하는 것은 옳지 않다. 따라서 수집된 메모리 복사 함수의 호출 지점 중 실제로 취약하다고 판단되는 지점만을 식별해야 한다. 먼저, 데이터 전달 함수에서 사용되는 크기 파라미터가 사용자 입력 데이터에 영향을 받는지 확인한다. 이는 가장 먼저 함수 내 탐색(Infra-procedure) 과정에서 레지스터 변수의 상태를 확인할 수 있다. 즉, 크기 파라미터를 가리지는 레지스터 변수는 분명하지 않은 값(undetermined value)이라는 조건을 만족시켜야 하며, 이는 1차적으로 잠재적 취약점 후보군으로 속할 수 있게 된다. 이와 반대로 Determind Value(결정적인 값)은 함수 내 탐색 과정에서 값이 결정되기 때문에 입력에 의존하지 않고 제외된다.

3.2 M2: 역방향 CFG 생성 방법 연구

3.2.1 정적 역추적 프로그램 분석

크기 파라미터를 시작점으로 입출력 함수의 파라미터에 도달할 때까지 코드를 역추적 한다. 이를 위해 크기 파라미터가 의존하는 모든 변수의 추적이 필요하다. 변수는 종종 이전 할당 값에 종속적인 재 할당 값이다. 예를 들어 다음과 같이 명령어를 반복하는 것을 발견할 수 있다.

```
mov eax, ebx
lea eax, [ecx+eax*4]
```

이와 같은 명령어를 만났을 때, 다음과 같이 모델링을 한다면 변수 추적이 어려워질 수 있다. 왜냐하면 eax 레지스터는 매번 새롭게 값이 바뀔 수 있기 때문이다. 프로그램이 복잡해질수록 여러 값에 의존하는 레지스터에 대한 값의 추적이 직관적으로 판단하기 어렵다. 이러한 문제는 변수의 임시적 요소(temporal element)를 전혀 고려하지 않았기 때문에 발생한다. 임시적 요소를 극복하고, eax와 같은 변수를 정확하게 추적하기 위해 이 논문에서는 BNIL (Binary Ninja Intermediate Language)[16]을 활용한다. BNIL 형식의 SSA Form[17]은 Binary Ninja Platform에서 제공하는 중간언어(IL) 형식으로 주로 함수 호출 및 메모리, 스택 관리 등을 좀 더 직관적으로 관찰할 수 있다. 또 프로그램 수명 기간(lifetime) 동안 변수의 정의(Def)와 사용(Use)을 알 수 있고 관리될 수 있기 때문에 후보 타겟 변수를 명시적으로 추적하는데 용이하다. 이를 이용하여 레지스터, 스택 변수 및 메모리를 모델링하는 것이 가능하다.

후보군 중 하나인 데이터 전달함수의 크기 파라미터를 시작점으로 크기 파라미터가 의존하는 모든 변수를 추적하기 위해 역방향 정적 프로그램 분석(backward static program analysis) 기법을 사용한다.

```
var_def =
self.function.get_ssa_var_definition(self.var.src)

self.to_visit.append(var_def)

while self.to_visit:
    idx = self.to_visit.pop()
    if idx is not None:
        self.visit(self.function[idx])
...
def visit(self, expression):
    method_name = 'visit_{}'.format
(expression.operation.name)
    if hasattr(self, method_name):
        value = getattr(self, method_name)
(expression)
    else:
        value = None
    return value
```

Fig. 5. Backward Static Program Code-snippet for Dependent Variable Tracking

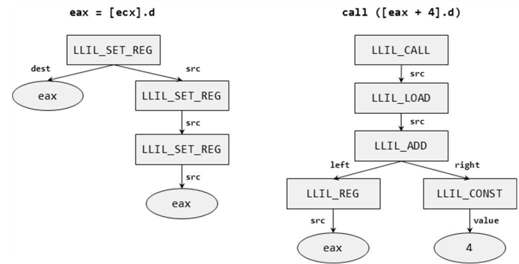


Fig. 6. BNIL(Binary Ninja IL) Tree

Fig. 5의 초기 추적 변수(self.var.src)는 해당 정의(definition) 지점을 추적할 수 있다. 프로그램의 변수는 항상 “use-def” 형식으로 정의될 수 있다. def는 변수의 정의 및 재 할당 되는 dest 이고, use는 해당 변수가 사용되는 source라고 볼 수 있다. visit 메소드는 MediumLevelILInstruction object[17]를 사용하고, 명령어(expression)의 오퍼랜드에 따라 다른 메소드를 디스패치 할 수 있다. Fig. 6은 하나의 인스트럭션이 각각의 명령어 트리로 구성될 수 있음을 보여준다. BNIL은 트리 기반 언어(tree-based language)이기 때문에 트리의 종료 노드에 도달할 때까지 각각의 명령어에 대한 오퍼랜드 호출을 재귀적으로 수행한다. 최종적으로 역추적 CFG 생성을 통해 특정 파라미터가 의존하는 모든 변수의 추적을 함으로써 해당 파라미터가 사용자 입력에 영향을 받는지 직접적으로 확인할 수 있다.

단일 함수 내부에서의 역추적은 어렵지 않지만 만약 추적중인 변수가 함수 호출의 변수로 사용 중인 경우에는 좀 더 복잡할 수 있다. 함수가 직접 호출(direct calls)상황은 기존의 정적 분석 기반으로도 추적이 가능하다. 반면 추적중인 변수가 간접 호출(indirect calls)인 경우에는 추적이 어렵다. 정적 분석 기반의 역추적이 어려운 문제를 다음과 같이 정리할 수 있다.

① 가상 함수 사용으로 인한 간접 호출(indirect calls)
② 함수 테이블 사용으로 인한 간접 호출(indirect calls) / 콜백(callback) 함수
③ 함수 포인터 사용으로 인한 간접 호출(indirect calls)
④ 재귀 함수

재귀 함수는 추가적인 추적이 불필요하기 때문에 단순히 함수 호출 체인에서 삭제하는 것만으로 해결할 수 있다. 그 외 간접 호출의 경우, 데이터 흐름

그래프(data flow graph) 복구 이후 3.2.1 과정을 진행해야 한다.

3.2.2 vtable 간접호출 복구

클래스 생성자(constructor)는 vtable에 대한 포인터를 메모리의 객체 구조체에 저장한다. vtable은 C++ 객체 생성과정에서 부모 함수와 자식 함수간 버전 별 함수 호출방식을 용이하기 위해 관리되는 함수 테이블이다. vtable의 경우 함수 심볼이 따로 존재하지 않으며 포인터 연산으로 접근하기 때문에 추적이 어려워질 수 있다. 일반적으로 vtable 포인터를 저장하는 두 가지 방법은 다음과 같다.

① vtable 포인터의 하드 코딩된 값을 직접 참조
② vtable 포인터를 레지스터에 저장한 다음 해당 레지스터의 값을 메모리에 복사

따라서 클래스 생성자 내부를 탐색하여 오프셋이 없는 레지스터에서 메모리 주소로의 쓰기를 찾으면 vtable이라고 예측할 수 있다. 이는 Fig. 7의 text:400BDA가 이에 해당한다.

vtable의 위치를 알았다면, 어떤 오프셋이 호출되는지 알아야한다. 이 값을 얻기 위해서는 호출 명령어(CALL instruction)에서 vtable 포인터가 메모리에서 검색될 때까지 프로그램 state를 추적하고, 가상 테이블의 오프셋을 계산하여 호출되는 함수를 찾아야 한다. 이를 위해 호출 명령어가 포함된 베이직 블록(basic block)을 사전 처리하는 데이터 흐름 분석이 필요하다.

vtable 포인터 역참조를 모델링하고, 가상 함수 오프셋을 계산하는데 필요한 데이터를 수집할 수 있다. 해당 내용은 Fig. 7에 대한 내용을 바탕으로 다음과 같이 요약될 수 있다.

① 메모리에 있는 객체의 구조체에서 vtable에 대한 포인터를 읽는다.
② 디스패치 할 함수가 첫 번째 함수가 아닌 경우 포인터 값에 오프셋을 더한다.
③ 계산된 오프셋에서 함수 포인터를 읽는다.
④ 함수를 호출한다.

3.2.3 함수 포인터 간접 호출 복구

함수 포인터를 사용하는 경우에도 간접 호출이 발생한다. 이 경우는 일정한 패턴을 지닌 가상 함수와는 달리 개발자의 의도에 따라 서로 다른 형태를 보일 수 있다. 함수 포인터 테이블 사용으로 인한 간접 호출을 복구하기 위한 시나리오는 다음과 같다.

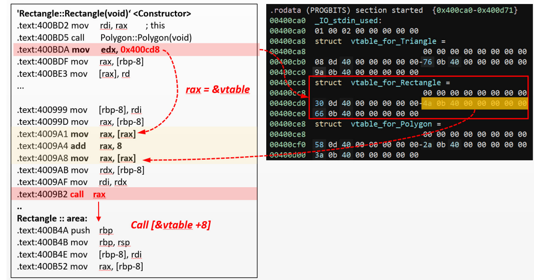


Fig. 7. Function Call Process in vtable

① 전체 심볼 정보를 수집하고, 이 중 함수 포인터 테이블로 예상되는 심볼을 추출한다.
② CALL 명령어에서 사용된 오프셋 및 파라미터 정보를 획득한다.
③ 획득한 정보를 바탕으로 간접 호출로 인해 실행되는 실제 함수를 식별한다.

Fig. 8은 이러한 시나리오를 도식화한 것이다. 함수 포인터 테이블로 예상되는 심볼을 수집하고, caller에서 획득한 정보를 바탕으로 간접 호출 명령어가 가리킬 수 있는 모든 함수를 매칭 한다.

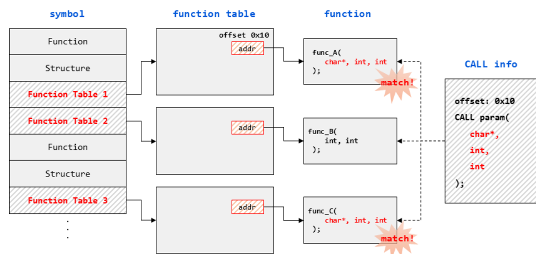


Fig. 8. Indirect call recovery due to the use of function pointer table

3.3 M3: 메모리 포인터 역 추적을 위한 정방향 분석

역추적을 진행하는 과정에서 추적중인 변수가 메모리 포인터로 사용될 경우 추적이 실패할 수 있다. Fig. 9의 sub_function03의 첫 번째 인자를 추적하는 과정에서 더 이상 상위 함수로 추적이 진행되지 않고 중간에 종료된다.

```
sub_function01(arg1, arg2):
var_c#1 = arg1
var_10#1 = eax#1
var_1c @ mem#0 = 0        Tracing End!
eax_1#2 = var_c#1
ecx_1#2 = var_10#1
var_5c#1 = &var_1c
var_54#1 = eax_1#2
var_50#1 = ecx_1#2
eax_2#3 = sub_function02(var_5c#1, 0, var_54#1, var_50#1)
...

var_5c_1#2 = var_1c @ mem4 // aliased value?
..

eax_4#5 = sub_function03(var_5c_1#2, var58_1#1, var_54_1#2)
```

Fig. 9. Backtrace failed due to memory pointers

sub_function03의 첫 번째 인자인 var_5c_1#2의 Definition을 추적한 결과 var_1c에 의존하고, var_1c는 Memory aliased value 로 인해서 definition을 찾지 못하고 끝나게 된다. 하지만 실제 var_1c는 sub_function02의 첫 번째 인자로 들어가게 되고 타입 형태는 포인터 타입으로 전달된다. 따라서 본래 sub_function03의 첫 번째 인자는 추적이 끝나기 이전에 그 값이 sub_function02에 의해서 할당될 수 있음을 암시한다.

보통 우리가 자주 보았던 [OUT] 함수들이 모두 이러한 형태의 바이너리를 띄고 있다. 따라서 var_1c는 sub_function02에 의해서 값이 할당될 수 있고, 또 그 값이 sub_function02의 ARG2, ARG3, ARG4에 의해서 의존될 수 있다면 ARG1은 ARG2, ARG3, ARG4를 정의(Definition)로 볼 수 있다.

따라서 역추적이 끝난 시점에서의 변수가 Memory Aliased Value 라면, 이는 다른 함수에 의해서 값이 할당될 수 있다고 할 수 있다. 또 변수가 어떠한 함수의 인자로 포함된다면 다른 인자와 함께 그 함수를 탐색할 필요가 있다. 함수 탐색 과정은 정방향 분석(Forward Analysis)를 진행하고, 모든 Forward 인스트럭션을 조사한 뒤 메모리 복사 과정이 이루어지는 구간을 찾아야 한다. 만약 ARG1이 다른 인자에 의해서 Definition 되었다면 해당 ARG를 대상으로 역추적을 재개한다.

IV. 실험 및 평가

제안하는 방법의 검증을 위해 정적 분석 기술을 기반으로 초기 취약점이 발생할 수 있는 함수 호출 지점을 수집하고, 제안하는 모델링 과정을 통해 이와 관련된 입출력 함수식별을 수행한다. 해당 실험을 위해 Binary Ninja Version 1.1.1470 Personal (a3b48b43)[17] 의 BNIL을 활용하여 Plugin Code을 작성하였다.

4.1 CVE-2014-0160(Openssl v_1.0.1f)

CVE-2014-0160은 Openssl[18]에서 발생하는 힙 오버플로우 취약점으로 'Heartbleed' 라는 명칭으로 알려져있다. 해당 취약점은 주어진 크기를 검증하지 않고, 공격자의 메시지 필드를 malloc 및 memcpy 호출의 크기 파라미터로 사용했기 때문에 가능했다. Openssl 바이너리의 온전한 역방향 CFG를 생성하기 위해 간접 호출(indirect calls) 지점을 복구한다. 먼저 간접 호출 복구를 위해 호출되는 함수 오프셋과 전달되는 파라미터 정보를 수집한다. 수집한 정보를 기존에 존재하는 Openssl 함수 테이블의 함수 정보 (오프셋, 파라미터)와 대조한다. 이후 동일한 정보를 확인하고, 각각의 간접 호출 지점을 같은 방식으로 매칭 한다. Fig. 10는 Openssl의 간접 호출을 복구한 역추적 CFG의 함수 호출 깊이를 비교한 것이다. 모델을 적용하지 않을 경우 낮은 깊이에서 추적이 중단된 반면 (Indirect Call로 인한 추적 중단)에, 모델을 적용할 경우 입력지점까지의 높은 깊이까지의 추적이 가능함을 보였다. 검증을 위해 실제 소스코드에 존재하는 함수 호출 정보를 확인한 결과 깊이에 따른 올바른 매칭이 이루어졌음을 확인할 수 있었다.

간접 호출 복구 수행(3.2)을 바탕으로 제안하는 모델링을 적용한 결과는 Fig. 11과 같다. Openssl은 최초 321개의 메모리 전달 함수 중 모델링을 통해 34개의 취약 후보 지점을 선정할 수 있었으며, 해당 경로는 실제 취약점 3종을 모두 포함하는 것으로 확인됐다.

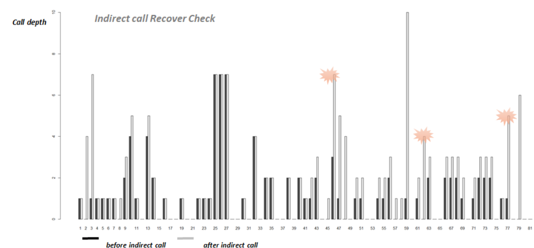


Fig. 10. Comparison of search depth before and after indirect call recovery

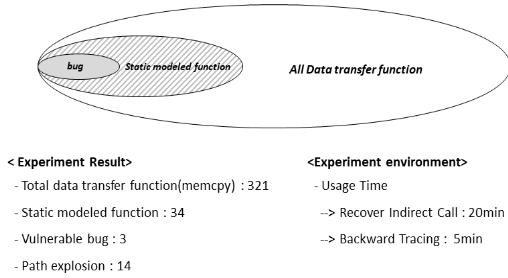


Fig. 11. CVE-2014-0160 Experiment result

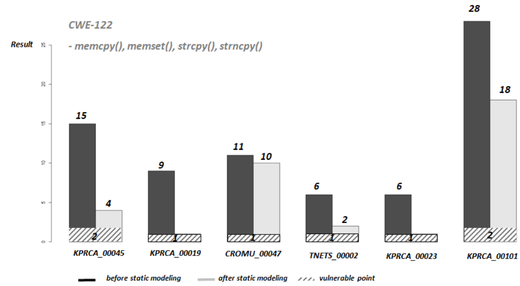


Fig. 12. CGC Binary Experiment result

4.2 CGC Binary Test

CGC 바이너리[19]는 총 6종을 대상으로 실험하였다. 2014년도부터 DARPA는 100개 이상의 취약한 프로그램에 대한 소스코드를 공개했다. 이 프로그램들은 다양한 소프트웨어 결함을 나타내는 취약점으로 특별하게 설계된 것이다. Table .1의 CGC 바이너리 (KPRCA_45, KPRCA_19, KPRCA_23, KPRCA101, CR OMU_47, TNETS_02)의 일부로 사용자 입력으로 인해 발생하는 Heap overflow(CWE-122) 취약점을 갖는다. CGC 바이너리는 실제 대회에서 사용했던 바이너리로 기존 프로그램에서 발생하는 특정 취약점의 형태를 잘 포팅한 대회용 바이너리이다.

Fig . 12은 CGC 바이너리 실험의 전체적인 결과이다. 대부분 초기 후보군 선정 이후 정적 모델링을 통한 역추적 CFG를 생성한 결과가 더 나은 것을 볼 수 있다. CROMU _47의 경우 타겟-입력 지점까지 도달하는 갯수와 모델링된 개수가 별 차이 없었던 이유는 CROMU_47의 경우 타겟 함수가 대부분의 입력 버퍼의 데이터를 파싱하는데 쓰였기 때문이었다. KRRCA_45의 경우 역추적 과정에서 메모리 포인터 문제(3.3)가 발생하였고 이를 해결해야 했다. KRRCA_23의 경우 vtable 간접 호출문제(3.2.2)가 발생했고 이를 해결하였다. 나머지 바이너리의 경우 기존의 정적 역추적 분석을 통해서 역방향 CFG를 생성할 수 있었다(3.2.1).

Table 1. Experimental CGC Binary Vulnerability

ID	CWE	Vulnerability
KPRCA_45	122	Privilege
KPRCA_19	122	Privilege
KPRCA_23	122	Privilege
KPRCA101	122	Privilege
CROMU_47	122	Privilege
TNETS_02	122	Privilege

V. 결 론

본 논문에서는 효율적인 프로그램 분석을 위한 역방향 CFG 생성 방법에 대해 소개하였다. 역방향 CFG는 프로그램 분석 방법 중 하나인 기호 실행 과정에서 발생하는 경로 폭발 문제를 간접적으로 완화시킬 수 있다. 생성된 역방향 CFG 생성은 향후 기호 실행의 탐색을 효율적으로 가이드 할 수 있다. 우리는 바이너리를 대상으로 정적 모델링 방법을 제안했다. 모델링 과정에서 발생하는 가상 함수 및 함수 포인터 호출에 따른 문제, 그리고 메모리 포인터 변수에 따른 역추적 문제를 해결하였다. 제안하는 방법을 통해 분석가는 생성된 역방향 CFG는 바이너리 내의 취약점이 발생할 수 있는 타겟 후보군을 추출할 수 있고, 해당 지점으로부터 사용자 입력 지점까지의 탐색 경로 추출을 통해 향후 프로그램 분석을 용이하게 할 수 있다. 실제 취약점을 갖는 Openssl 바이너리와 CGC 바이너리를 대상으로 실험을 진행한 결과 바이너리의 다양한 상황에서 역방향 CFG가 생성 가능함을 보였다.

References

[1] Heelan, Sean. Automatic generation of control flow hijacking exploits for software vulnerabilities. 2009.

[2] Avgerinos, Thanassis, et al. "Automatic exploit generation." Communications of the ACM 57.2 .

2014.

〔3〕 Cha, Sang Kil, et al. ″Unleashing mayhem on binary code.″ Security and Privacy (SP), 2012 IEEE Symposium on. IEEE. 2012.

〔4〕 Huang, Shih-Kun, et al. ″Software crash analysis for automatic exploit generation on binary programs.″ IEEE Transactions on Reliability 63.1 : 270-289. 2014.

〔5〕 Cadar, Cristian, et al. ″Symbolic execution for software testing in practice: preliminary assessment.″ Proceedings of the 33rd International Conference on Software Engineering. ACM. 2011.

〔6〕 Cadar, Cristian, Daniel Dunbar, and Dawson R. Engler. ″KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.″ OSDI. Vol. 8. 2008.

〔7〕 Sen, Koushik, Darko Marinov, and Gul Agha. ″CUTE: a concolic unit testing engine for C.″ ACM SIGSOFT Software Engineering Notes. Vol. 30. No. 5. ACM. 2005.

〔8〕 Godefroid, Patrice, Nils Klarlund, and Koushik Sen. ″DART: directed automated random testing.″ ACM Sigplan Notices. Vol. 40. No. 6. ACM. 2005.

〔9〕 Burnim, Jacob, and Koushik Sen. ″Heuristics for scalable dynamic test generation.″ Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on. IEEE. 2008.

〔10〕 Chipounov, Vitaly, Volodymyr Kuznetsov, and George Candea. ″S2E: a platform for in-vivo multi-path analysis of software systems.″ ACM SIGPLAN Notices 46.3 : 265-278. 2011.

〔11〕 Cadar, Cristian, and Koushik Sen. ″Symbolic execution for software testing: three decades later.″ Commun. ACM 56.2 : 82-90. 2013.

〔12〕 Sanjay Rawat, Laurent Mounier, and Marie-Laure Potet, “Lightweight Static Taint Analysis for Binary Executables Vulnerability Testing”, University of Grenoble, 2009.

〔13〕 Rohit Mothe, “DPTrace: Dual Purpose Trace for Exploitability Analysis of Program”, BlackHat US. 2016.

〔14〕 Haller, Istvan, et al. ″Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations.″ Presented as part of the 22nd USENIX Security Symposium pp. 49-64. 2013.

〔15〕 Neugschwandtner, “The borg: Nanoprobing binaries for buffer overreads”. In Proceedings of the 5th ACM Conference on Data and Application Security and Privacy (pp. 87-97). ACM. 2015

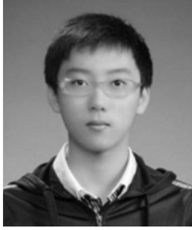
〔16〕 Static simgle assignment form, https://en.wikipedia.org/wiki/Static_single_assignment_form. 2019

〔17〕 Binary Ninja, https://binary.ninja/. 2019

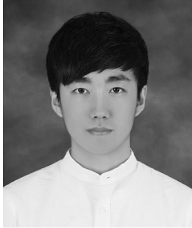
〔18〕 CVE-2016-0160, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-0160

〔19〕 CGC-Binary, https://github.com/ CyberGrandChallenge/

〈저자소개〉



박 성 현 (Sunghyun Park) 학생회원
2016년 8월: 전남대학교 컴퓨터정보통신공학 공학사
2018년 2월: 전남대학교 정보보안협동과정 이학석사
2018년 3월~현재: 전남대학교 정보보안협동과정 박사과정
〈관심분야〉 취약점 분석, 악성코드 탐지, 시스템 보안



김 연 수 (Yeonsu Kim) 학생회원
2017년 8월: 전남대학교 소프트웨어공학 공학사
2017년 9월~현재: 전남대학교 정보보안협동과정 석·박사통합과정
〈관심분야〉 사이버 위협 인텔리전스, 머신러닝, 침입탐지

노 봉 남 (Bongnam Noh) 종신회원
1978년: 전남대학교 수학교육과 학사
1982년: KAIST 대학원 전산학과 석사
1994년: 전북대학교 대학원 전산과 박사
1983년~현재: 전남대학교 전자컴퓨터공학부 교수
2000년~현재: 전남대학교 시스템보안연구센터 소장
〈관심분야〉 정보보안, 시스템 및 네트워크 보안