Journal of The Korea Institute of Information Security & Cryptology ISSN 1598-3986(Print)
VOL.33, NO.2, Apr. 2023 ISSN 2288-2715(Online)
https://doi.org/10.13089/JKIISC.2023.33.2.305

FunRank: 함수 호출 관계 및 데이터 흐름 분석을 통한 공개된 취약점 식별*

이 재 휴,[1†] 백 지 훈,[2] 문 현 곤[3‡]
[1,2,3]울산과학기술원 (대학원생, 학생, 교수)

FunRank: Finding 1-Day Vulnerability with Call-Site and Data-Flow Analysis*

Jaehyu Lee,[1†] Jihun Baek,[2] Hyungon Moon[3‡]
[1,2,3]UNIST (Graduate student, Undergraduate student, Professor)

요 약

최근 소프트웨어 제품의 복잡성 증가로 오픈소스 소프트웨어를 적극 활용하는 경우가 많아지고 있다. 이는 개발 기간 단축에 도움을 주지만, 동시에 사용된 오픈소스 소프트웨어간의 서로 다른 개발 생명 주기(SDLC)가 전체 제품의 버전 최신화를 어렵게 하기도 한다. 이로 인해 사용된 오픈소스 소프트웨어의 알려진 취약점에 대한 패치가 공개되었음에도 불구하고, 패치를 신속히 적용하지 못해 공개 취약점의 위협에 노출되는 경우가 많다. 특정 장치가 이런 위협에 노출되어있는지를 신속히 판별하기 위한 공개 취약점 식별 기법에 관한 여러 연구 들이 수행되어 왔는데, 기존 기법들은 취약점이 발생하는 함수의 크기가 작거나 인라인되는 경우 취약점 발견에 어려움을 겪는 경우가 많다. 본 연구는 이런 문제를 해결하기 위해 함수 호출 관계 및 데이터 흐름 분석을 통한 바이너리 코드 유사성 비교 도구인 FunRank를 개발하였다. 개발된 도구는 기존 연구들과 달리, 컴파일러에 의해 인라인 될 수 있는 크기가 작은 함수의 코드를 식별해야만 발견할 수 있는 공개취약점 또한 찾아낼 수 있도록 설계되어 있다. 본 연구에서 인위적으로 만들어진 벤치마크 및 실제 펌웨어로부터 추출된 바이너리를 이용해 실험한 결과, FunRank가 바이너리 코드 내에서 인라인 된 함수를 잘 찾아내고, 이를 통해 공개된 취약점의 존재성을 빠르게 확인하는 데에 도움을 줌을 보일 수 있었다.

ABSTRACT

The complexity of software products led many manufacturers to stitch open-source software for composing a product. Using open-source help reduce the development cost, but the difference in the different development life cycles makes it difficult to keep the product up-to-date. For this reason, even the patches for known vulnerabilities are not adopted quickly enough, leaving the entire product under threat. Existing studies propose to use binary differentiation techniques to determine if a product is left vulnerable against a particular vulnerability. Despite their effectiveness in finding real-world vulnerabilities, they often fail to locate the evidence of a vulnerability if it is a small function that usually is inlined at compile time. This work presents our tool FunRank which is designed to identify the short functions. Our experiments using synthesized and real-world software products show that FunRank can identify the short, inlined functions that suggest that the program is left vulnerable to a particular vulnerability.

Keywords: Binary Similarity Comparison, Vulnerability Identification, Inline Function

Received(01. 25. 2023), Modified(02. 15. 2023), Accepted(02. 20. 2023)

* 본 논문은 주 저자의 학위 논문의 내용을 포함하고 있으며, 일부 내용이 국내 학술대회에 발표된 바 있습니다.

* 본 논문은 ETRI부설연구소의 위탁연구과제(2022-041)로 수행한 연구결과입니다.

† 주저자, splendormic@unist.ac.kr
‡ 교신저자, hyungon@unist.ac.kr(Corresponding author)

I. 서 론

1-day 취약점은 소프트웨어 개발 당사자가 취약점을 인지하고 해당 패치가 공개된 취약점을 말한다[1]. 최근 많은 개발사에서 개발에 필요한 시간과 노력을 줄일 수 있으며, 다수의 개발자들에 의해 검증되어있다는 점에서 오픈소스 소프트웨어를 사용하고 있는데, 이는 1-day 취약성 측면에서의 문제를 수반하고 있다. 이는 다수의 오픈소스 소프트웨어들이 사용될 경우, 이를 사용하여 개발한 소프트웨어는 복잡성이 증가하고 파편화되어 관리하는 것이 어려지기 때문으로, 개별 취약 소프트웨어의 패치가 공개되더라도 이를 포함하는 제품 전체를 업데이트하는 과정의 어려움으로 제품은 더욱 오랜 기간 동안 1-day 취약점에 노출될 수 있는 것이다. 실제로 JPCERT의 조사에 의하면 Log4j 취약점으로 알려진 CVE-2021-44228 [2]은 최초 패치가 공개된 이후 10일 동안 공격 시도가 있었는데 [3], 이 기간 내에 패치가 적용되지 않은 시스템의 경우 해당 공격들에 의해 피해를 입었을 가능성이 있음을 시사하고 있으며, 이는 패치가 공개된 1-day 취약점이 여전히 유효한 위협임을 시사하고 있다.

소프트웨어 제품의 잠재적 1-day 취약점 노출 정도를 확인하기 위한 바이너리 코드분석 기법에 대해 많은 연구들이 진행되어 왔다. 그 중 정적 분석 기반 바이너리 코드 비교 기법은 서로 다른 바이너리 코드를 비교함으로써 여러 바이너리 코드의 연관성을 파악하는 방법으로 자동화가 용이하여 공개된 바이너리에 대하여 바이너리 분석이나 악성 코드나 취약점 탐지와 같이 널리 사용되고 있다 [4]. 최근 연구들은 바이너리 내 존재하는 대부분의 함수들에 대해서 추가 실행 없이 높은 정확도를 보장하고 있지만, 인라인 되는 경우가 많은 짧은 함수는 비교적 잘 식별하지 못하고 있다. 본 연구는 바이너리에서 인라인 된 1-day 취약점 관련 함수 (취약함수)를 식별하기위해 개발된 분석기인 FunRank를 소개한다. FunRank는 취약함수를 입력으로 하여 바이너리 분석을 통해 두 가지 특징(feature)을 생성하며, 이를 바탕으로 바이너리가 인라인 되어있을 수 있는 취약함수를 식별한다. 첫 번째 특징은 하나는 바이너리의 함수 호출 그래프를 분석하여 각 함수들을 특징화한 call-site feature로, 각 함수가 최종적으로 어떤 외부 함수 (시스템 콜, 공유라이브러리 등)을 호출하는지를 반영하며, 다른 하나는 함수의 바이너리 코드의 데이터 흐름 그래프 (data-flow graph)을 분석하여 생성한 data-flow feature로, 이는 함수의 의미 (semantic)을 반영한다. FunRank는 call-site feature를 통해 취약함수일 가능성이 극히 낮은 함수들을 비교 대상에서 배제한 뒤, 나머지 함수들의 data-flow feature를 별도로 확보한 취약함수의 것과 비교해 각 함수의 취약함수와의 유사도를 계산한다. 그 결과는 취약함수와의 유사도를 기준으로 한 순위의 형태로 사용자에게 제공되며, 사용자는 분석 바이너리에서 높은 순위의 함수들을 확인함으로써 취약함수 존재여부를 판별하는데에 걸리는 시간을 단축시킬 수 있다.

FunRank의 식별 성능 평가를 위해 우리는 두 가지 벤치마크를 이용해 실험을 진행하였다. 먼저 이전 연구에서 수집 및 정리한 바이너리 코드 비교 기법 성능 평가용 벤치마크인 BinKit[5]을 이용해 FunRank가 인라인된 함수 코드를 탐지할 수 있음을 확인하였다. 또한 공개 펌웨어 이미지에서 수집한 바이너리를 이용한 실험에서는 실제 1-day 취약성을 확인할 수 있음도 확인하였다.

II. 배경지식

2.1 함수 인라인 (function inlining)

Function inlining은 함수 호출을 함수의 정의로 교체하는 최적화 기법이다 [8]. 인라인 결정 기법은 NP-complete 문제만큼 어려우며 [9], 이에 많은 휴리스틱 기반의 인라인 최적화 기법이 제안되어 있다. 제안된 기법들은 주로 함수에 존재하는 명령어의 수나 함수 호출횟수와 같은 프로그램의 특징을 고려하여 인라인 시킬 함수를 결정하는데, GCC의 경우 함수가 인라인 되는 경우는 크게 다음과 같았다. 첫 번째로 inline 키워드를 사용하는 경우로, C에서 inline 키워드는 컴파일러에게 해당 함수가 인라인 될 것을 지시하는 키워드이다 [10]. 이 키워드는 함수의 코드 라인의 개수와 상관없이 지시자가 존재하는 함수에 인라인 최적화를 허용하게 된다. 두 번째는 static 키워드를 사용하고 적은 수의 코드라인으로 구성되어 있을 때이다. 이 때 코드 라인 수는 컴파일러의 정책마다 다르게 설정될 수 있으며, 따라서 이 경우 컴파일러나 빌드 옵션에 의해서 인라인 적용 결과가 다르게 된다. 이렇게 인라인된 함수는 다른 함수의 일부가 되기 때문에 바이너리 코드 상에

서 식별이 용이한 특징을 갖고 있지 않으며, 이는 본 연구의 주요 동기 중 하나가 된다.

2.2 바이너리 유사도 기반의 1-day 취약성 식별

바이너리 유사도 비교 기법을 통해 1-day 취약점을 식별하는 다양한 연구들이 제안되었다. BinGo [11], Firmup [7]은 바이너리 코드로부터 프로그램 의미론을 표현한 특징을 추출하고, 서로 다른 바이너리에서 추출한 특징 사이의 비교를 통해 바이너리 유사도를 계산한다. 이를 위해 바이너리에 존재하는 모든 함수에 대하여 함수별로 분석을 실시하는데, BinGo는 바이너리에서 사용하는 공유 라이브러리 함수들을 추상화하고, 함수에서 호출할 수 있는 공유 라이브러리 함수들을 수집하여 함수를 추상화하여 표현한다. Firmup은 바이너리의 함수들의 제어 흐름 그래프에 존재하는 기본 블록 (basic block)을 대상으로 데이터 흐름의 역추적 분석(backward analysis)를 수행하고, 기본블록의 분석 결과를 모두 수집하여 함수를 추상화하여 비교한다.

BinXray [1], FIBER [12]는 바이너리에서 취약점의 패치를 식별하여 1-day 취약점의 존재성을 결정한다. BinXray는 패치가 적용된 소스 코드를 빌드한 바이너리에서 패치 코드를 식별하기 위하여 함수의 제어 흐름 그래프를 추출하여 그래프의 구조적 정보를 바탕으로 함수의 패치 코드를 식별하고, 이를 추상화하여 패치 특성(patch signature)을 생성한다. 그리고 분석하고자 하는 바이너리에서 식별된 모든 함수들에 대해 패치 특성의 존재성을 검증하여, 패치가 존재하지 않으면 1-day 취약점이 존재한다고 판단한다. FIBER는 Firmup과 유사하게 제어 흐름 그래프를 구성하는 기본 블록들에 대한 데이터 흐름 분석(data flow analysis)을 진행하여 각 기본 블록에 대한 추상화를 수행한다. 이를 통해 함수의 바이너리 특성(binary signature)을 생성하고 이용하여 패치 존재성을 확인, 분석대상 바이너리의 취약성을 검증한다.

최근에는 바이너리 유사도 계산을 위하여 기계 학습을 도입하여 유사도를 계산하는 연구도 제안되었다. Gemini[13], PatchHecko[6]와 같은 연구들은 바이너리 분석을 통해 함수를 구성하는 명령어나 기본 블록의 개수와 같은 여러 특징들을 수집, 조합하여 특징을 표현하고 이를 인공 신경망 기반의 모델을 통해 두 바이너리 코드의 유사도를 계산한다.

앞서 설명한 연구들은 대부분의 함수들에 대해서 잘 동작하지만, 함수의 크기가 작을 경우 탐지 성능이 부족한 편이며, 컴파일러에 의해 인라인 된 함수 코드를 식별하는 것은 고려되지 않는다.

2.3 짧고 인라인 되는 취약함수 예시

CVE-2017-16544는 busybox의 1.27.2에서 발견된 취약점으로 add_match 함수에서 입력 받은 문자열에 대한 길이 검사가 수행 되지 않아 잠재적으로 임의 코드 실행이 가능할 수 있는 취약점이다 [14]. Fig. 1은 busybox 1.25.1 버전의 소스코드에서 확인할 수 있는 취약한 add_match 함수의 정의이다. 패치 이전의 add_match 함수는 3줄로 구현된 크기가 작은 함수이며, 소스코드 분석을 통해 2개의 함수에 의해서만 호출되는 함수임을 확인할 수 있었다. Fig. 2는 1.25.1 버전의 소스코드를 gcc 7.5.0의 기본 최적화 옵션(-O2)와 디버깅 옵션(-g)을 통해 빌드하여 생성한 바이너리에서 add match를 호출하는 complete_cmd_dir_file을 IDA pro [15]를 통해 디컴파일한 결과이다. 컴파일 된 바이너리에서, add_match는 호출자인 complete_cmd_dir_file에 인라인 되어 존재함을 확인할 수 있다.

해당 CVE에 대하여 분석자가 released 제품이나 소프트웨어 패키지에 포함되어있는 busybox의 CVE-2017-16544가 위험한지 검증하는 것을 가정하면, 바이너리 비교 기법을 활용한 첫번째 시나리오는 add match의 호출자를 식별하는 것이다. 하지만 취약함수의 호출자를 식별하였다고 하여 취약한 add_match가 반드시 존재한다는 것을 보장할 수 없으므로, 분석자는 add_match의 존재성을 위해 바이너리 코드 비교를 통해 식별한 호출자를 추가적으로 분석해야 한다. 왜냐하면 실제 시나리오에서 분석하고자 하는 바이너리가 사용한 컴파일러, 컴파일 옵션 등을 알 수 없기 때문이다. 또 다른 시나리오는

```
static void add_match(char *matched)
{
   matches = xrealloc_vector(matches, 4, num_matches);
   matches[num_matches] = matched;
   num_matches++;
}
```

Fig. 1. The definition of vulnerable add_match in busybox 1.25.1

```
unsigned int __fastcall complete_cmd_dir_file(...)
{
  // ...
  if ( (st.st_mod & 0xF000) == 0x4000)
  {
    *((_BYTE *)v13 + v15) = 47;
    *((_BYTE *)v13 + v15 + 1) = 0;
LABEL_17:
    v16 = lineedit_ptr_to_statics;
    v17 = (unsigned __int8 **)xrealloc_vector_helper(
        lineedit_ptr_to_statics->matches,
        0x404u,
       lineedit_ptr_to_statics->num_matches);
    v18 = v16->num_mathces;
    v16->num_matches = v17;
    v17[18] = (unsigned __int8 *)
    v16->num_mathces = v18 + 1;
    goto LABEL_5;
  }
  // ...
}
```

Fig. 2. The result of decompiling complete_cmd_dir_files in busybox 1.25.1 binary

취약한 add_match가 바이너리에 존재하도록 컴파일하고, add_match의 바이너리 코드를 비교하는 것이다. gcc는 -fno-inline 옵션을 통해 인라인 최적화를 적용하지 않을 수 있으며, 이를 통해 add_match가 바이너리에 함수로써 존재할 수 있게 컴파일 할 수 있다. 하지만 동일한 컴파일러에서 인라인 최적화가 적용된 busybox와 적용되지 않은 busybox를 BinDiff[22]를 통해 비교를 해보면, BinDiff에서, 취약한 add_match가 인라인 최적화가 적용된 busybox에서 식별되지 않는다.

최근 컴파일러와 최적화 기법의 발전으로 인해 소스 코드 내 많은 함수들이 인라인 될 수 있다. 한 연구[16]에서 51개의 오픈 소스 프로젝트를 대상으로 인라인 최적화 대상의 비율에 대하여 분석하였을 때, O3 최적화에서 소스코드에 표현되어있는 함수 중 평균 30~40%의 함수들이 인라인 최적화가 적용되며, LTO(Linker Time Optimization)가 추가로 적용될 경우, 인라인 최적화가 적용되는 함수가 최대 70%까지 증가할 수 있음을 보여준다. 이는 CVE-2017-16544와 같은 특징을 가진 취약함수가 더 많이 존재할 수 있으며, 따라서 바이너리 유사도 비교 기법에서도 인라인 최적화를 고려해야 함을 시사한다.

2.4 기존 연구의 한계

대부분의 바이너리 비교 분석 기법은 바이너리 함수의 1:1 관계를 통해 두 함수의 유사도를 계산한다. 이는 두 바이너리가 같은 소스코드에서 컴파일된 바이너리들이라면, 두 바이너리에 존재하는 함수들이 1:1로 대응될 것임을 가정한다. 이러한 가정은 소스코드와 컴파일러 및 컴파일 옵션과 관계없이 바이너리에 존재하는 대부분의 함수에 대해서는 효과적이지만, 인라인되는 함수에 대해서는 그렇지 않다. 인라인 되는 함수는 바이너리 코드에서 호출 함수의 일부로 존재하며, 하나의 함수다 다수의 함수를 호출한다는 특성으로 인해 인라인 하는 함수와 인라인되는 함수는 1:1 관계가 아닌 1:N 관계(N은 인라인 함수를 포함하고 있는 호출 함수의 개수)를 갖게 된다. 이러한 특성은 대부분의 바이너리 유사도 비교 연구 결과물들이 인라인 함수를 식별하지 못하거나 식별하더라도 높은 오탐율을 보이는 원인인 것으로 추정된다[16].

이러한 문제를 해결하기 위해, O2NMatcher[17]는 소스 코드와 바이너리 코드 사이의 1:N 관계를 고려해 유사도를 계산함으로써 바이너리 코드에서 인라인된 함수를 더 잘 식별할수 있음을 보여주었다. 하지만 이를 위해 O2NMatcher는 바이너리에 대응되는 소스코드를 활용하기 때문에, 정확히 대응되는 소스코드를 획득하기 어려울 수 있는 바이너리간 유사도 분석에는 활용되기 어렵다. 따라서 바이너리 코드 사이의 유사도 분석 기반의 인라인 함수 탐지 기법은 아직 잘 연구되어있지 않으며, 추가연구를 필요로 하는 기술이라고 말할 수 있다.

III. FunRank 설계

소스코드로부터 생성된 바이너리는 소스코드에 표현된 정보를 모두 포함하는 것이 아니며, strip과 같은 도구를 사용하면 바이너리에 존재하는 함수 이름이나 디버깅 정보들을 제거할 수 있다. 따라서 제한된 정보를 가진 바이너리에서 소스코드에 표현된 함수를 식별하기 위해서는 바이너리에 존재하는 함수를 추상화할 수 있는 기법이 필요하다. 이 때, 인라인 최적화를 고려하기 위하여, 바이너리 코드에서 함수가 인라인 되더라도 보존되는 특징을 추상화해야

하는데, 본 논문은 바이너리 분석을 통해 함수를 2가지 형태의 추상화를 제안한다.

3.1 FunRank 개요

Fig. 3는 FunRank의 전체 구조와 동작 흐름을 보여주며, Table 1은 FunRank에서 사용되는 용어에 대한 설명이다. FunRank는 함수 호출 및 인라인 관계에서의 1:N 특성을 고려한 바이너리 유사도 분석 도구로, 사용자로부터 분석 대상 바이너리(target binary)와 참조 바이너리(reference binary), 분석함수(target function)를 입력으로 받는다. 분석의 첫 단계는 call-site feature 기반의 필터링이다. 먼저 각 바이너리에서 함수 호출 그래프를 생성하여 식별된 모든 함수의 call-site feature를 계산한 뒤, 분석 대상 바이너리에서 생성된 함수들의 call-site feature와 분석함수의 call-site feature를 비교하여, 연관성이 존재하는 함수들만으로 구성된 후보 집합(candidate set)을 구성한다.

분석의 두 번째 단계는 data-flow feature를 이용한 비교이다. 먼저 FunRank는 분석 후보 집합에 존재하는 함수들과 참조 바이너리의 분석함수의 data-flow feature를 각각의 data-flow graph으로부터 생성한다. 다음으로 생성된 분석함수의 data-flow feature와 분석 후보 집합에 존재하는 함수들의 data-flow feature를 비교함으로써 분석 후보 집합에 있는 모든 함수들에 대하여 분석함수와의 유사도를 계산한다. FunRank는 이렇게 계산된 분석 후보 집합 내 함수들의 유사도 순위를 사용자에게 제공하고, 사용자는 이를 바탕으로 유사도 순위가 높은 순서대로 함수들을 검증하여 분석함수 존재 여부를 빠르게 식별할 수 있다.

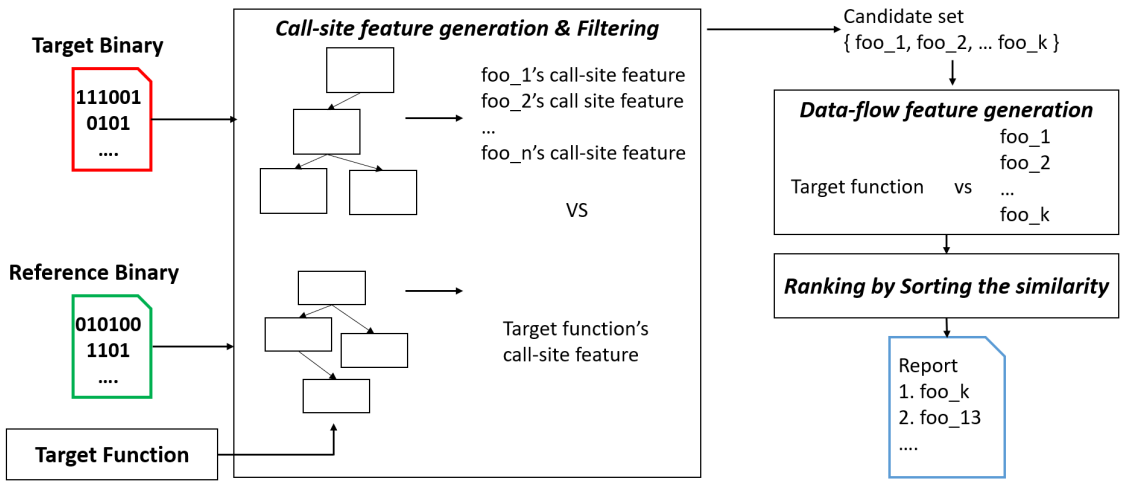


Fig. 3. FunRank overview

Table 1. Term and Description for overview.

Term	Description
Target Binary	The binary to wants to check if the Target Function code is included. Assume that the binary is stripped.
Reference Binary	The binary that is compiled with debugging symbol like function name and not performed inline expansion optimization.
Target Function	A Function to verify that it exists in the Target binary. It must be exist in Reference binary
Candidate set	A set of functions selected by filtering with call-site feature from Target binary

3.2 Call-site Feature 및 함수 필터링

FunRank는 바이너리 분석을 통해 획득한 함수 호출 그래프를 이용해 각 함수의 call-site feature를 계산한다. 함수 호출 그래프는 함수들을 정점(node)들로 하고 호출자(caller)와 피호출자(callee)의 관계를 간선(edge)으로 하는 유방향 그래프를 말한다. FunRank는 대부분의 함수가 최종적으로 glibc의 printf와 같은 외부 공유 라이브러리의 함수들을 호출하게되며, 이러한 외부 공유 라이브러리의 함수는 함수 호출 그래프에서 진출 차수(out degree)가 0인 정점으로 표현된다는 점을 이용한다. 즉, 다음 두 가지 조건을 만족하는 함수의 집합을 함수 F의 call-site feature로 계산하는 것이다: 1) 바이너리의 함수 호출 그래프의 정점 F에서 도달 가능한 진출 차수가 0인 정점이며, 2) 외부 공유 라이브러리에 존재하는 함수. 즉 함수 F의 call-site feature는 F의 호출 스택에 존재하는 외부 공유 라이브러리의 함수들의 집합이다.

Fig. 4는 함수 호출 그래프와 함수 호출 그래프에 존재하는 함수들의 call-site feature의 예시이다. 정의한 call-site feature는 함수 호출 그래프에 존재하는 모든 정점들을 대상으로 그래프 탐색을 통해 계산할 수 있으며, 함수의 호출 관계에서 호출자는 자신이 호출한 함수의 call-site feature를 반드시 포함하는 특성을 가진다. 이러한 특성은 분석함수가 분석 대상 바이너리에서 인라인 되었더라도 분석함수 코드를 포함하는 함수를 식별하는데 효과적인데, 이는 함수가 인라인 되더라도 함수의 호출 관계는 보존되기 때문이다. 예를 들어, Fig. 4에서 small vulnerable 함수는 함수 foo에 의해 호출되며, foo는 피호출자인 small_copy와 small_vulnerable의 call-site feature를 모두 포함하고 있다. 만약 small vulnerable이 인라인 되어 있다면, 분석자는 함수 foo를 식별해야만 하며, 이미 foo는 small vulnerable의 call-site feature를 포함하고 있기 때문이다. 따라서 서로 다른 함수의 연관성을 두 함수의 call-site feature간의 교집합의 존재를 통해 확인할 수 있다.

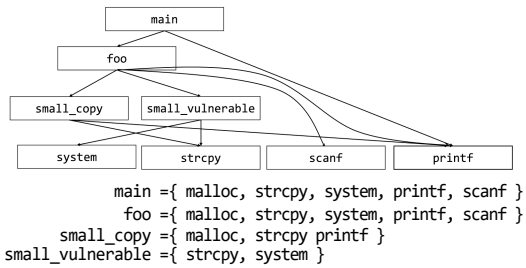


Fig. 4. An example of a call-site feature computed from call graph.

FunRank는 이와 같은 과정을 통해 각 함수의 call-site feature를 계산한 뒤, 이를 참조 바이너리에 있는 분석함수의 call-site feature와 비교하여 call-site feature의 교집합이 지나치게 적은 함수들을 연관성이 적은 함수로 보고 분석대상에서 제외하며, 나머지 함수로 후보집합을 구성한다.

3.3 Data-flow Feature

FunRank는 call-site feature를 통해 선정된 분석 후보 집합 내 함수들에 대해 인라인된 함수 코드를 포함하고 있을 가능성을 점수화하며, 이를 위해 data-flow graph 기반으로 생성되어 각 함수의 의미를 반영하는 data-flow feature를 활용한다.

Data-flow feature는 함수 내의 데이터 흐름을 추상화한 것으로, 각 함수의 핵심 의미에 따라 결정되며 그 핵심의미는 인라인되더라도 유지되어야 한다는 점에서 인라인 되는 함수의 식별에 도움을 줄 수 있는 지표이다. 함수의 데이터 흐름은 함수를 표현하는 주요 특징 중 하나로, 근본적으로 함수의 의미(semantic)에 의해 결정된다. 때문에 인라인 최적화가 적용되더라도 인라인 함수의 데이터 흐름 중 함수의 핵심 의미에 해당하는 데이터 흐름은 보존되는데, 이는 인라인 함수를 포함하게 된 호출자(caller)가 인라인 되기 전의 피호출자(callee)함수와 data-flow feature 측면에서의 유사성을 갖게 한다. 이런 이유로 본 연구에서 정의 및 사용하는 data-flow feature는 인라인되는 함수의 식별에 유용한 것으로 추정되며, 본 장의 나머지 부분에서 보다 구체적인 정의, 계산방법 및 비교방법을 설명한다.

3.3.1 정의 및 계산방법

FunRank는 함수에 대한 데이터 의존성 그래프(data dependency graph, DDG)를 생성하고, 이를 분석하여 추상화한 data-flow feature를 생성한다. 이 때, 인라인 최적화를 고려하기 위하여, FunRank는 함수의 제어 흐름 그래프를 구성하는 노드인 기본 블록(Basic Block, BB)단위로

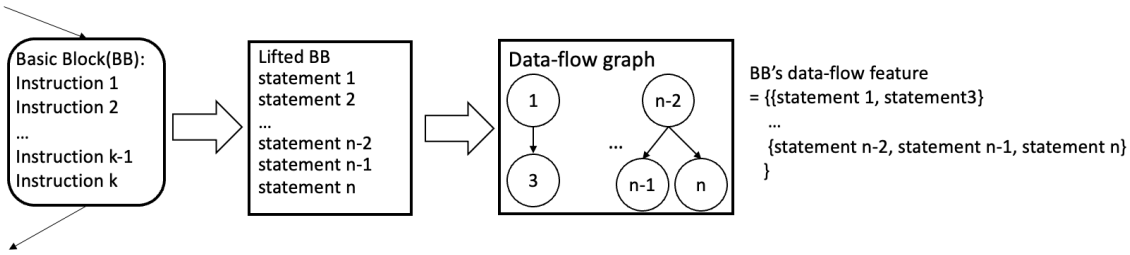


Fig. 5. The process to compute data-flow feature from one basic block

data-flow feature를 계산하는데, 이는 호출 함수에서 인라인 함수의 코드 위치를 정확히 식별할 수 없기 때문이다. FunRank는 Angr.io [18]를 통해 바이너리 코드 역변환을 수행하며, 이를 통해 VEX-IR [19]을 획득하여 데이터 흐름 분석을 수행한다.

Fig. 5는 함수의 CFG를 구성하는 기본 블록의 data-flow feature의 계산 과정이다. 기본 블록의 data-flow feature는 기본 블록의 DDG에서 존재하는 각 연결 성분(connected component)을 구성하는 노드들의 집합으로 정의한다. FunRank가 데이터 흐름 분석을 통해 생성한 DDG는 역변환 된 IR statement를 노드로 하여 use-def 관계를 간선으로 표현한 것으로, Fig. 5와 같이 하나의 기본 블록에는 여러 개의 연결 성분이 존재할 수 있다. 즉 기본 블록의 data-flow feature는 기본 블록의 DDG를 구성하는 정점과 간선의 관계를 통해 데이터 흐름 관계가 비교적 큰 IR statement들의 집합

```
IR statements from BB

t5 = Sub32(t0,0)
PUT(offset=60) = t5
t6 = Sub32(t0, 0)
t7 = GET:I32(offset=64)
STle(t6) = t0
t8 = t5
t9 = GET:I32(offset=24)
STle(t0) = t9
PUT(pc) = 0x492f0
t4 = LDle:I32(0x49308)
```

→

```
Data-flow feature
{
  t4 = LDle:I32(0x49308),
  PUT(offset=8) = t0
},
{
  STle(t0) = t9,
  STle(t6) = t0,
  t6 = Sub32(t0, 0),
  PUT(offset=60) = t5,
  t9 = GET:I32(offset=24),
  t5 = Sub32(t0, 0),
  t8 = t5,
  t7 = GET:I32(offset=64)
}
```

Fig. 6. An example the data-flow feature about one basic block

으로 표현되는 것이다. Fig. 6은 역변환된 하나의 기본 블록에서 생성한 data-flow feature를 보여주는데, FunRank는 생성한 기본 블록의 DDG에서 DFS 기반 알고리즘을 통해 data-flow feature를 계산한다. 함수의 data-flow feature는 함수를 구성하는 기본 블록의 data-flow feature의 합집합으로 정의한다.

3.3.2 IR 정규화

정의한 data-flow feature는 바이너리 코드를 역변환한 결과인 IR을 통해 생성되므로, 바이너리 코드를 생성할 때 컴파일러에 의해 결정되는 값이나 역변환 과정에서 생성되는 정보를 포함하고 있다. 이러한 정보들은 바이너리 코드의 데이터 흐름을 표현하는데 있어 불필요한 정보들이므로, 관찰을 통해 불필요한 정보들을 제거하는 정규화를 진행하였다.

첫 번째 정규화 대상 정보는 임시 변수(temporary variable)이다. VEX-IR에서 IR operation의 결과를 임시 변수에 저장하게 되며, 이는 CPU의 레지스터와 유사하다. 이 때, 중요한 정보는 임시 변수에 특정 연산의 결과를 쓰거나, 임시 변수의 값을 읽었다는 프로그램 의미론이 중요하므로 이러한 임시 변수는 모두 0으로 변경하는 정규화를 진행하였다

두 번째 정규화 대상은 메모리 오프셋(memory offset)으로, 메모리 오프셋은 load, store와 같은 연산에서 특정 메모리에 접근할 때 사용되는 값이다. 메모리 오프셋은 컴파일러에 의해 결정되는 정보이며, 또한 데이터 흐름을 표현할 때 메모리에 접근했다는 의미가 더 중요하므로, 이러한 값들도0x0으로 변경하는 정규화를 진행하였다.

마지막 정규화 대상은 게스트 머신 상태(guest machine state)이다. Angr에서는 특정 아키텍처를 추상화하여 표현하게 되며, IR statment의 부

작용(side effect)를 표현한다. 이러한 정보는 함수의 의미론에서 불필요한 정보이므로, 이를 모두 값은 상태로 통일하는 정규화를 진행하였다.

3.3.3 유사도 및 순위

FunRank는 정규화가 적용된 함수의 data-flow feature 사이의 유사도를 계산하고, 분석 후보 집합에 대하여 유사도 기반의 순위를 부여한다. 함수의 data-flow feature는 IR statement의 집합으로 구성된 data-flow feature 구성요소의 집합으로 표현된다. 따라서 두 함수의 data-flow feature의 유사도는 data-flow feature 구성요소의 교집합의 길이를 이용하여 계산이 가능하며, Fig. 7는 두 함수 f와 g의 data-flow feature를 이용한 유사도 계산 과정을 보여준다. 이 때 f는 유사도 계산의 기준이 되는 함수로 FunRank에서는 취약함수가 된다. FunRank는 먼저 f의 data-flow feature 구성요소 f1을 선택하여 함수 g에 존재하는 모든 data-flow feature 구성요소와의 교집합의 길이를 계산하여 수집한다. 그리고 f2에 대하여, 동일한 과정을 수행하게 되면, f의 data-flow feature 구성요소 별로 수집된 g의 data-flow feature의 구성요소 중 가장 큰 값을 찾을 수 있다. Fig. 7에서 f1은 g1에서 가장 큰 값인 11일 갖게 되고, f2에서는 g3과 가장 큰 값 18을 갖게 된다. FunRank는 기준이 되는 함수의 data-flow feature 구성요소 별로 가장 큰 값을 모두 합산하여 두 함수의 유사도를 측정하게 되며, 예시에서는 29가 되게 된다.

FunRank는 분석 후보 집합에 존재하는 함수들에 대하여 분석함수와의 유사도를 계산한다. 그러면 분석 후보 집합에 존재하는 함수들을 유사도를 기준으로 내림차순을 하여 순위를 부여하게 되며, 순위가 부여된 결과를 사용자에게 제공하게 된다. 이 때, 유사도 순위가 높은 함수는 분석함수이거나 분석함수의 코드를 가질 확률이 높다는 것을 의미한다.

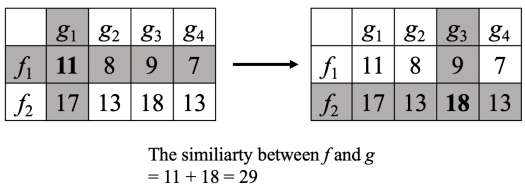


Fig. 7. The process to calculate the similarity between functions.

IV. 실험 결과

4.1 실험 구성

FunRank는 Python3 [20] 기반의 IDA pro[15]와 Angr.io [18]를 이용하여 구현하였다. IDA pro는 자동 분석을 통해 바이너리에서 함수를 식별하고 call-site feature를 계산하기 위한 바이너리의 함수 호출 그래프를 생성한다. Angr.io는 함수의 data-flow feature를 계산하기 위하여, 바이너리 코드를 VEX-IR[19]로 역변환을 하고 데이터 흐름 분석을 통한 데이터 의존성 그래프를 생성하게 된다.

그리고 FunRank의 탐지 성능을 확인하기 위하여 바이너리 코드 비교 기법 벤치마크인 BinKit [5, 27]을 이용하여 벤치마크를 구성하였다. Binkit에는 여러 바이너리 데이터 셋이 존재하는데, 이 중 Noinline-dataset과 Normal-dataset을 활용하여 벤치마크를 구성하였다.

Noinline-dataset은 바이너리를 빌드 할 때, 인라인 최적화를 적용하지 않는 옵션을 주어 컴파일한 바이너리로 구성되어 있고, Normal-dataset은 일반적인 컴파일러 옵션을 적용하여 생성한 바이너리로 구성되어 있다. 여기서 Normal-dataset의 gcc 5.5.0으로 기본 최적화 옵션(-O2)와 ARM으로 컴파일된 바이너리 10개를 선정하여 분석 대상 바이너리 집합을 구성하였다. 그리고 Noinline-dataset에서도 동일한 옵션의 바이너리를 10개를 선정하여 참조 바이너리 집합을 구성하였다. 또한, 참조 바이너리 집합의 바이너리를 모두 분석하여 기본 블록이 5개미만으로 구성되어 있는 함수들을 각 바이너리별로 5개씩 수집하여 총 10개의 바이너리에 대한 50개의 분석함수를 선정하였다. 그리고 분석 대상 바이너리 집합의 바이너리들을 분석하여 분석함수에 대한 정답 함수를 선정했는데, 분석함수가 인라인 되어 있을 경우 분석함수의 호출자가 정답함수가 되고, 분석함수가 인라인 되어있지 않을 때는 분석함수와 동일한 이름의 함수가 정답함수로 지정되었다. 이를 통해 50개의 함수 중 46개의 함수가 분석 대상 바이너리에서 인라인 되어있는 것을 확인하였다.

마지막으로 FunRank의 탐지 성능을 평가하기 위하여, 수동 검증 노력(manual effort)를 정의하

였다. 수동 검증 노력은 FunRank의 결과에서 정답 함수를 찾기 위해 분석한 함수들 중 실제로 검증해봐야 하는 함수의 개수를 비율로 정의한다.

Fig. 8은 FunRank의 실제 분석 결과이다. line 1은 분석함수를 보여주고, line 2에서는 분석 대상 바이너리에서 선정한 분석 후보 대상의 개수를 나타낸다. line 4~5에서는 유사도 기준 순위(ranking)가 1위인 함수가 2개 존재하며, 그 중 하나가 treat_file이고, 유사도가 30임을 나타낸다. Fig. 8에서 정답 함수는 분석 대상 바이너리에서 동일한 이름을 가진 함수이다. line 8에는 정답함수의 유사도와 유사도 기준 순위를 보여주는데, 정답 함수는 분석함수와 data-flow feature 유사도가 29이며, 유사도 기준 순위는 3이고 동일한 유사도를 가진 함수가 8개 존재한다. 이는 유사도 기준 순위 1위부터 10개의 함수를 수동으로 검증하였을 때 정답 함수를 찾을 수 있으며, 따라서 예시에서 볼 수 있는 수동 검증 노력은 10/87 * 100 ≒ 11%로 표현된다. 이를 통해 FunRank의 탐지 성능을 계산할 수 있으며, 수동 탐지가 낮을수록 FunRank의 성능이 우수하다고 말할 수 있다.

```
01: vulnerable function: remove_output_file
02: the number of functions to compare = 87
03: call-site feature of vulnerable function
    = {'unlink', 'sigprocmask', 'close'}
04: compare function in target = treat_file
05: similarity = 30 / ranking = 1(2)
06: ...
07: compared function in target =
remove_output_file
08: similarity = 29 / ranking = 3(8)
```

Fig. 8. The result of FunRank.

4.2 Call-site Feature의 효용성

Fig. 9는 call-site feature를 이용하여 분석 대상 바이너리에서 분석해야할 함수의 개수에 대한 감소율을 나타낸다. 자체 벤치마크의 분석함수 50개에 대하여 call-site feature를 사용할 경우, 바이너리에서 식별된 함수의 개수 중 최소 50% 이상의 감소율을 보여주며, 이 중 35개의 분석함수가 감소율이 80% 이상임을 보여준다. 이를 통해, call-site feature를 사용할 경우 바이너리에서 분석해야할 함수의 개수를 효과적으로 줄일 수 있으며 이는 곧 분석에 필요한 시간과 노력을 효과적으로 줄일 수 있음을 의미한다.

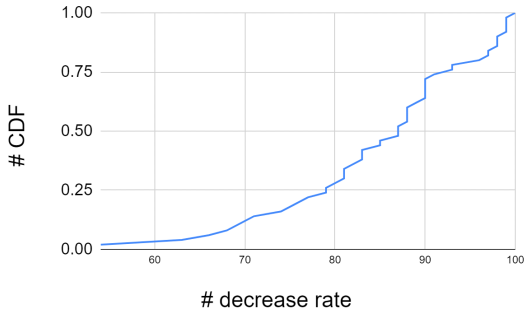


Fig. 9. The effect to filter functions with call-site feature.

4.3 정규화의 영향

Table 2는 논문에서 제안한 각 정규화에 따른 수동 검증 노력의 변화를 보여준다. base는 정규화를 적용하지 않은 data-flow feature를 사용했을 때의 결과를 보여주며, Level-1은 임시 변수, Level-2는 메모리 오프셋, Level-3는 게스트 머신 상태를 정규화하였을 때를 의미한다. 50개의 함수를 대상으로 정규화 미적용 시, 수동 검증 노력이 1% 이하인 함수의 개수가 6개이며, 유사도가 존재하지 않는 함수가 2개 존재한다. 임시 변수 정규화를 적용할 경우, 유사도가 존재하지 않는 함수는 없지만 수동 검증 노력이 1% 이하인 함수의 개수가 정규화를 적용하지 않았을 때보다 감소하였으며, 메모리 오프셋과 게스트 머신 상태를 정규화 할 경우 각각 수동 검증 노력이 1% 이하인 함수의 개수가 증가하였지만, 유사도가 존재하지 않는 함수가 여전히 존재하였다. 이를 통해, FunRank의 탐지 성능을 개선하고자 정규화 대상을 추가하는 방식으로 정규화 전략

Table 2. The effect for each normalization about data-flow feature.

	base	Level-1	Level-2	Level-3
1%↑	6	4	12	14
10%↑	12	19	9	8
30%↑	14	13	11	10
50%↑	10	6	10	8
100%↑	6	8	6	8
100%=	2	0	2	2

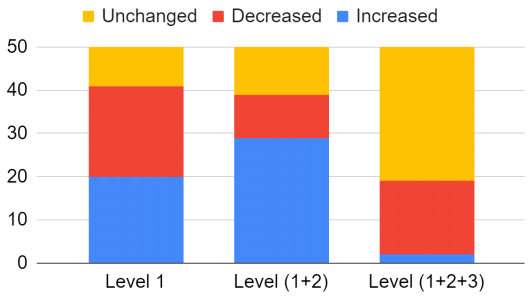


Fig. 10. The result for different normalization strategy about data-flow feature.

을 변경하여 실험을 진행하였다.

Fig. 10은 정규화 대상 누적 시 수동 검증의 노력의 변화를 보여준다. 임시 변수와 메모리 오프셋 정규화를 적용할 경우, 임시변수 정규화만 적용하는 것보다 수동 검증 노력이 증가하는 것을 볼 수 있다. 그리고 제안한 정규화를 모두 적용 시, 임시 변수와 메모리 오프셋을 정규화 하는 것보다 수동 검증 노력이 증가하는 함수의 갯수가 93% 감소하는 것을 알 수 있다. 이를 통해 제안한 정규화를 모두 적용하면 FunRank의 탐지 성능을 크게 개선됨을 확인할 수 있었으며, 이에 FunRank는 모든 정규화를 활용하고 있다.

4.4 실제 바이너리에서의 1-day 탐지

FunRank의 실용성을 확인하기 위하여 ipTime, TP-Link, OpenWRT 등 실 제품에서 수집한 8,000여개의 펌웨어 내에 포함된 바이너리들을 이용해 실험을 진행하였다. 구체적으로, 수집된 펌웨어 이미지들로부터 binwalk [21]를 이용해 3개 프로그램의 오픈 소스 바이너리(busybox, libcurl, lighttpd)를 추출하였다. 추출한 바이너리들에 대하여 바이너리 내 존재하는 문자열 중 버전과 관련된 문자열을 식별한 뒤 해당 바이너리의 버전에 존재하는 것으로 제보된 CVE를 확인함으로써 공개 취약점 관련 함수의 샘플을 얻었다. 이때, 수집된 바이너리에서 컴파일러 및 최적화 정도에 대한 정보를 식별할 수 없지만 컴파일러 옵션 및 컴파일러 버전의 차이가 있을 가능성이 높다. 그러므로, 참조 바이너리는 식별한 버전의 소스 코드를 gcc 7.5.0 컴파일러와 O2 최적화와 인라인 최적화 미적용을 적용하여 생성하였다.

Table 3은 FunRank를 통해 수집한 실제 바이너리대상 공개 취약점 식별 실험 결과이다. FunRank는 취약 함수의 코드를 식별한 경우 최대 9%의 수동 검증 노력을 필요함을 보여주고 있다. 그리고 FunRank의 탐지 성능을 비교 분석하기 위하여, IDA pro의 바이너리 비교 플러그인인 BinDiff [22]를 동일한 바이너리들에 대하여 진행하였으며, Table 4는 동일한 real-world 바이너리들에 대한 BinDiff의 결과이다. BinDiff의 경우, 취약 함수가 인라인 최적화가 적용되지 않을 경우, 탐지를 할 수 있었지만 신뢰도를 표현하는 confidence와 similarity의 평균값이 1개의 함수를 제외하고는 낮음을 보여준다.

Table 3과 4는 FunRank와 BinDiff의 오탐률 또한 미탐지 (false-negative) 및 오탐지 (false-positive)의 형태로 보여주고 있다. FunRank와 BinDiff의 오탐지와 미탐지는 각 도구의 특성상 서로 다르게 정의되는데, BinDiff의 경우, 1:1 대응 방식의 바이너리 유사도 비교 분석이라는 점에서 서로 다른 함수가 1:1 대응되는 경우를 오탐지로 정의하며, 미탐지는 동일한 함수가 존재함에도 1:1 대응이 존재하지 않는 경우로 정의한다. FunRank는 1:N 매칭 방식의 바이너리 유사도를 계산하여 순위를 부여하므로, 오탐지는 발생하지 않으며, 같은 함수가 위의 매칭된 N에 포함되지 않을 경우를 미탐지로 정의한다. Table. 3에 나타난 것처럼 FunRank의 경우, lighttpd에 대하여 미탐지가 24개의 바이너리에 대해서 존재하였다. 이는 IDA pro의 자동 분석 기능을 통해 함수를 식별하는 과정에서 취약 함수를 포함한 함수가 식별되지 않아 발생한 문제였다. 반면에 Table. 4에서 보듯 BinDiff의 경우, 오탐지는 발생하지 않았으나, 취약 함수가 인라인 되어 있을 경우 탐지하지 못하는 미탐지가 발생하였다. 이를 통해, FunRank는 함수가 인라인 되어 있지 않을 경우, BinDiff 정도의 탐지 성능을 가지며 함수가 인라인될 경우 BinDiff보다 좋은 탐지 성능을 갖고 있음을 알 수 있다.

V. 한 계

FunRank는 정적 바이너리 분석을 통한 바이너리 비교 기법을 통해 인라인 된 함수 코드를 식별하였다. 이를 위해 함수를 추상화하여 서로 다른 특성을 가진 두 개의 feature를 제안하였지만, 실험을 통해 몇 가지 한계가 존재함을 확인하였다.

첫 번째로 call-site feature의 soundness이다. call-site feature는 바이너리의 함수 호출 그래프로부터 생성되며, 이는 함수 호출 그래프가 소스 코드로부터 보존되는 정보라는 것을 가정한다. 하지만, 관찰을 통해 컴파일러의 최적화를 통해 함수 호출 그래프가 변화할 수 있음을 확인하였다. 예를 들어 꼬리 재귀 최적화(tail recursion optimization) [23]의 경우, ARM과 같은 아키텍처에서 꼬리 재귀를 통해 호출된 함수는 함수 호출 명령어가 아닌 점프(jump) 명령어를 통해 호출되는 것을 확인하였다. ARM에서 점프를 통해 함수 호출을 진행할 경우, 해당 함수 호출은 함수 호출 그래프에 반영이 되지 않는다. 또 다른 경우는 컴파일러에 의한 함수 변경으로 그 예는 GCC의 경우 fortification [24]이 있다. GCC는 컴파일 된 바이너리에서 버퍼 오버플로우를 방지하고자 소스코드 분석을 통해 printf나 memcpy와 같이 메모리를 접근하는 함수를 변경하는데, printf의 경우 해당 옵션이 적용되어 있으면 인자가 문자열 상수 일 경우, 소스코드에는 printf을 사용했지만 빌드한 바이너리에서는 puts를 호출하는 형태로 변경한다. 이러한 컴파일러의 최적화로 인해 함수 호출 그래프는 변화할 수 있으며, 따라서 call-site feature는 최적화 정도 에 따라 안전성이 변화할 수 있다.

두 번째 한계는 정확한 함수 매칭의 부재이다. FunRank는 바이너리에 존재하는 함수 중 일부에 대하여 data-flow feature를 활용한 유사도를 계산하고 이를 통해 순위를 부여해 사용자에게 제공한다. 실험 결과에서도 알 수 있듯이 사용자는 높은 순위의 함수부터 순차적으로 검증하면서 공개 취약점을 식별할 수 있지만, 여전히 바이너리 분석을 위한 추가적인 노력이 요구된다. 이를 해결하기 위해서는 분석 대상 바이너리에서 data-flow feature를 통해 분석함수에 정확히 매칭 되는 함수를 제공해야 하지만, FunRank에는 이러한 기능이 부족하다. 이를 해결하기 위해서는 data-flow feature의 설계나 비교 방식에 대한 개선이 필요하다.

VI. 결 론

본 논문에서는 바이너리에서 인라인 된 공개 취약점을 식별하기 위해 개발된 정적 바이너리 비교 기반의 분석 도구인 FunRank를 소개하였다. FunRank는 이를 위해 공개 취약점 식별을 위해 call-site feature와 data-flow feature계산 및 활용하는데, 실험을 통해 call-site feature는 통해 바이너리에 존재하는 함수 중 분석해야 하는 함수의 개수를 효과적으로 줄여주며 data-flow feature는 인라인된 취약함수 식별에 유효한 도움을 준다는 점을 확인할 수 있었다. 또한, 공개된 펌웨어 이미지들

Table 3. The result to identify 1-day vulnerability about binary from firmware with *FunRank*.

Binary	Version	CVE	Vulnerable function	Confirms	False-negative	Avg. manual efforts
lighttpd	1.4.26	CVE-2012-5533	http_request_split_value	104	24	1%
libcurl	7.29.0	CVE-2013-1944	curl_maprintf	3	0	9%
libcurl	7.40.0	CVE-2016-8618	tailmatch	3	0	1%
busybox	1.25.1	CVE-2017-16544	add_match	245	0	3%
libcurl	7.71.1	CVE-2022-35252	Curl_cookie_add	242	0	8%

Table 4. The result about real-world binary with BinDiff.

Binary	Function	Inlined	Confirms	Avg. confidence	Avg. similarity	False-negative
libcurl	tailmatch	X	3	0.98	0.97	0
libcurl	curl_maprintf	X	3	0.13	0.08	0
libcurl	Curl_cookie_add	X	245	0.23	0.46	0
busybox	add_match	O	0	0	0	245
lighttpd	http_request_split_value	O	0	0	0	128

로부터 수집된 바이너리에 대한 취약성 분석에도 도움을 줄 수 있음도 확인되었다.

References

[1] Y. Xu, Z. Xu, B. Chen, F. Song, Y. Liu, and T. Liu, "Patch based vulnerability matching for binary programs," in Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 376 - 387, Jul, 2020.

[2] NVD, "CVE-2021-44228." https://nvd.nist.gov/vuln/detail/CVE-2021-44228, Jan. 2023.

[3] JPCERT/CC Eyes, "Observation of Attacks Targeting Apache Log4j2 RCE Vulnerability (CVE-2021-44228)." https://blogs.jpcert.or.jp/en/2021/12/log4j-cve-2021-44228.html#1, Feb. 2023.

[4] I. U. Haq and J. Caballero, "A survey of binary code similarity," ACM Computing Surveys (CSUR), vol. 54, no. 3, pp. 1 - 38, Apr. 2021.

[5] D. Kim, E. Kim, S. K. Cha, S. Son, and Y. Kim, "Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned," IEEE Transactions on Software Engineering, pp. 1-23, Jul. 2022.

[6] P. Sun, L. Garcia, G. Salles-Loustau, and S. Zonouz, "Hybrid firmware analysis for known mobile and iot security vulnerabilities," in 2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 373 - 384, IEEE, Jun. 2020.

[7] Y. David, N. Partush, and E. Yahav, "Firmup: Precise static detection of common vulnerabilities in firmware," ACM SIGPLAN Notices, vol. 53, no. 2, pp. 392 - 404, Feb. 2018.

[8] P. P. Chang and W.-W. Hwu, "Inline function expansion for compiling c programs," in Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation, pp. 246 - 257, Jun. 1989.

[9] R. W. Scheifler, "An analysis of inline substitution for a structured programming language," Communications of the ACM, vol. 20, no. 9, pp. 647 - 654, Sep. 1977.

[10] GCC, An Inline Function is As Fast As a Macro.", https://gcc.gnu.org/onlinedocs/gcc/Inline.html, Jan. 2023.

[11] M. Chandramohan, Y. Xue, Z. Xu, Y. Liu, C. Y. Cho, and H. B. K. Tan, "Bingo: Cross-architecture cross-os binary search," in Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 678 - 689, Nov. 2016.

[12] H. Zhang and Z. Qian, "Precise and accurate patch presence test for binaries," in 27th USENIX Security Symposium (USENIX Security 18), pp. 887 - 902, Aug. 2018.

[13] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pp. 363 - 376, Oct. 2017.

[14] NVD, "CVE-2017-16544." https://nvd.nist.gov/vuln/detail/CVE-2017-16544, Feb. 2022.

[15] hex-rays, "IDA pro." https://hex-rays.com/ida-pro/, Jan. 2023.

[16] A. Jia, M. Fan, W. Jin, X. Xu, Z. Zhou, Q. Tang, S. Nie, S. Wu, and T. Liu, "1-to-1 or 1-to-n? investigating

the effect of function inlining on binary similarity analysis," ACM Transactions on Software Engineering and Methodology, Just accepted, https://doi.org/10.1145/3561385

[17] A. Jia, M. Fan, X. Xu, W. Jin, H. Wang, Q. Tang, S. Nie, S. Wu, and T. Liu, "Comparing one with many - solving binary2source function matching under function inlining," arXiv preprint arXiv:2210.15159

[18] angr, "A powerful and user-friendly binary analysis platform!." https://github.com/angr/angr/, Jan. 2023.

[19] N. Nethercote and J. Seward, "Valgrind: a framework for heavy-weight dynamic binary instrumentation," ACM Sigplan notices, vol. 42, no. 6, pp. 89 - 100, Jun. 2007.

[20] python, "python3." https://www.python.org, Jan. 2023.

[21] binwalk, "Firmware Analysis Tool", https://github.com/ReFirmLabs/binwalk, Jan. 2023.

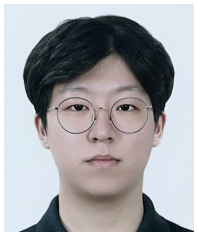
[22] zynamics, "BinDiff." https://www.zynamics.com/bindiff.html/, Jan. 2023

[23] W. D. Clinger, "Proper tail recursion and space efficiency," in Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation, pp. 174 - 185, May. 1998.

[24] GCC, "[PATCH] Object size checking to prevent (some) buffer overflows." https://gcc.gnu.org/legacy-ml/gcc-patches/2004-09/msg02055.html, Jan. 2023.

〈저자소개〉

이 재 휴 (Jaehyu Lee) 학생회원
2021년 2월: 충북대학교 컴퓨터공학과 졸업
2023년 2월: 울산과학기술원 컴퓨터공학과 석사
〈관심분야〉 시스템 보안, 프로그램 분석, 취약점 식별



백 지 훈 (Jihun Baek) 학생회원
2020년 3월~현재: 울산과학기술원 컴퓨터공학과 학사과정
〈관심분야〉 시스템 보안, Neural Network

문 현 곤 (Hyungon Moon) 종신회원
2017년 2월: 서울대학교 전기컴퓨터공학부 박사
2017년 5월~2018년 8월: 미국 조지아 공과대학교 박사후 연구원
2018년 8월~현재: 울산과학기술원 컴퓨터공학과 교수
〈관심분야〉 시스템 보안, 운영체제, 컴퓨터 구조, 프로그램 분석