B2R2: 이진 분석을 위한 효율적인 프런트 엔드 구축

Minkyu Jung , 김수민 * , 한형석 * , 최재승* , 차상길 *

KAIST

{hestati, soomink, 형석.한, jschoi17, sangkilc}@kaist.ac.kr

개요 - 현재 이진 분석 연구는 주로 백엔드에 초점을 맞추지만 프론트엔드에는 초점을 맞추지 않습니다. 그러나 이진 분석의 효율성을 크게 향상시킬 수 있는 몇 가지 주요 설계 포인트가 프런트 엔드에 있습니다. 우리의 아이디어를 입증하기 위해 우리는 바이너리 코드를 들어 올리고 해당 IR을 평가하는 것과 관련하여 빠른 새로운 바이너리 분석 플랫폼인 B2R2를 설계하고 구현합니다. 우리 플랫폼은 외부 종속성 없이 순수하게 함수형 프로그래밍 언어인 F#으로 작성되었습니다. 따라서 자연스럽게 순수 병렬 처리를 지원합니다.

B2R2의 IR은 데이터 흐름 분석 속도를 높이기 위해 해당 언어로 메타데이터를 포함하며 평가에 효율적으로 설계되었습니다.
따라서 모든 이진 분석 기술은 당사의 IR 설계에서 이점을 얻을 수 있습니다. 효율적인 이진 분석 프런트 엔드를 구축하기 위한 설계 결정에 대해 논의하고 배운 내용을 요약합니다. 또한 GitHub에서 소스 코드를 공개합니다.

I. 서론

이진 분석은 소프트웨어 보안에서 매우 중요하며 오늘날 이를 위한 수많은 도구를 사용할 수 있습니다. 예를 들어, GitHub는 현재 수백 개의 공개 리포지토리1를 수용 하며, 그 설명에는 "바이너리 분석"이라는 용어가 포함되어 있습니다. 많은 퍼징 도구는 용능 향상시키기 위해 [17], [34] 핵심에 이진 분석을 사용합니다.

이진 분석 도구는 일반적으로 프런트 엔드와 백 엔드의 두 가지 주요 구성 요소로 구성됩니다. 주로 디스어셈블러와 리프터로 구성된 프론트엔드는 주어진 바이너리를 디스어셈블하여 IR(Intermediate Representation)이라고 하는 것으로 변환합니다. 백엔드는 들어올려진 IR을 입력으로 받아 CFG 복구[31], 구조 분석[42], [51], 유형 추론[32], [41]과 같은 실제 분석을 수행합니다. 오픈 소스 [4], [13], [18], [19], [44] 또는 폐쇄 소스 [7], [28], [50]에 관계없이 기존의 모든 프레임 워크는 자체 IR. 예를 들어, IDA Pro[28]는 Microcode[26]를 사용하고 angr[43]은 VEX IR[37]과 함께 작동합니다.

백엔드는 프론트엔드와 IR에 의존하기 때문에 프론트엔드의 성능은 분명히 바이너리 분석의 효율성에 영향을 미칠 수 있습니다. 정적 이진 분석은 CFG(Control-Flow Graph) 복구 기술[18], [31]에 의존하며, 기본적으로 시작하려면 해제된 IR이 필요합니다. 동적 기호 실행기 [23], [47] 및 동적 오염 추적기 [38]는 프로그램 실행 중에 발생하는 모든 명령에 대해 리프팅을 수행하므로 효율적인 바이너리 리프터를 사용하여 직접적인 이점을 얻을 수 있습니다. Yunet al. [52] 최근 리프팅의 성능이 상징적 실행자의 병목 현상이 될 수 있음을 확인했습니다.

그러나 바이너리 리프팅은 간단한 번역 프로세스로 간주될 수 있기 때문에 연구자들은 프론트 엔드 디자인에 약간의 주의를 기울였습니다. 간결하고 사용하기 쉬운 IR을 설계하기 위한 여러 노력이 있었지만[10,13], 빠르게 평가할 수 있는 IR을 설계하는 것과 관련이 없습니다.

많은 연구자들은 효과적인 백엔드 알고리즘[18, [25], [31], [32], [41], [42]의 고안에 초점을 맞추었습니다.

이 논문에서 우리는 프론트엔드와 그에 상응하는 IR의 디자인에 효율성 면에서 프론트엔드 자체와 백엔드 모두에 큰 이점을 줄 수 있는 수많은 포인트가 있음을 보여줍니다.

몇 가지 예를 들자면, 바이너리를 잘 최적화된 IR로 변환하는 것은 어렵고 간단한 로컬 옵티마이저를 사용하면 IR의 복잡성을 크게 줄일 수 있다는 것을 발견했습니다. 그러나 이러한 최적화 단계는 해당 AST(추상 구문 트리)를 반복적으로 통과해야 하므로 전체 리프팅 프로세스의 속도를 늦출 수 있습니다. 예를 들어, 우리 시스템의 최적화 단계는 프런트엔드의 총 실행 시간의 1/3 이상을 차지합니다. 다중 코어 병렬 처리를 이용하여 이 문제를 완화할 수 있지만 명령의 크기가 opcode에 따라 다르기 때문에 병렬로 명령어를 분해하는 것은 간단하지 않습니다. 따라서 우리는 이진 명령어를 병렬로 들어 올리는 새로운 기술을 제안합니다.

또한 IR의 구조와 구현이 IR 평가 및 이진 분석의 성능에 큰 영향을 미칠 수 있음을 발견했습니다. 예를 들어, 기계 명령어는 x86의 XMM 및 YMM 레지스터와 같이 64비트보다 큰 숫자를 보유하는 레지스터 간의 산술 연산을 종종 포함하기 때문에 기존 IR은 임의의 정밀도 정수로 숫자를 나타냅니다. 그러나 임의 정밀도 계산은 IR 평가 프로세스를 상당히 느리게 할 수 있습니다. 추가적으로, 각각의 표현식이 표현식에서 사용된 변수에 대한 유용한 메타데이터를 통합하는 IR을 설계할 수 있습니다. 우리는 IR 설계에서 이와 같이 단순한 변형이 효율적인 데이터 흐름 분석을 가능하게 한다는 것을 보여줍니다.

1https://github.com/search?q=%22binary+analysis%22&type=리포지토리.

이진 분석 연구(BAR) 2019 워크숍 2019년 2월 24일, 캘리포니아
주 샌디에이고 ISBN 1-891562-58-4 https://dx.doi.org/
10.14722/bar.2019.23051 www.ndss-symposium.org

우리가 아는 한, 이 백서는 효율적인 이진 분석 프론트 엔드 및 IR 설계에 대한 최초의 엄격한 연구를 제시합니다. 특히 우리는 B2R2라고 하는 새로운 바이너리 분석 프레임워크를 설계하고 구현하며,

구축하기 위해 내린 중요한 설계 결정 요약
효율적인 이진 분석 프런트 엔드.

이 작업의 주요 기여는 다음과 같습니다.

1) 우리는 다음을 활용하는 새로운 IR 리프팅 기술을 제안합니다.
멀티 코어 병렬 처리
2) 최신 바이너리 분석을 연구합니다.
프론트 엔드, 디자인 결정에 대해 논의합니다.
3) 속도를 높일 수 있는 IR의 참신한 디자인을 제시합니다.
IR 평가 프로세스를 향상시킵니다.
4) 효율적인 바이너리인 B2R2를 설계하고 구현합니다.
당사의 기술을 구현하는 분석 프레임워크
및 디자인 선택.
5) GitHub에서 도구를 공개합니다.

나머지 논문은 다음과 같이 구성된다. 우리는 먼저
IR 설계에 관한 몇 가지 관련 작업을 요약합니다. 우리는 그때
기존 바이너리의 구조에 대한 주요 관찰 사항 제시
분석 프론트엔드. 다음으로, 우리는 디자인 선택 사항을 제시합니다.
B2R2, 우리의 새로운 바이너리 분석 플랫폼. 마지막으로 결론을 내립니다.
우리의 실험 결과를 설명함으로써 논문.

II. 관련 작업

형식 의미를 제공하는 몇 가지 IR이 있습니다.
예를 들어, GAL [9] 및 DBA [10]는
비트 조작 연산자와 같은 유용한 연산자를 제공하여 이진 분석. BAP[13]는 간결한 IR
을 제시합니다.
모든 어셈블리 부작용을 명시적으로 나타낼 수 있습니다. 하지만,
그들 중 누구도 IR 평가의 효율성에 중점을 두지 않습니다.

Kim et al. [30] 기존의 체계화된 첫 번째 연구를 보여줍니다.
리프터. 그들은 다음을 기반으로 각 리프터와 IR을 특성화합니다.
그들의 표현력. 특히, 그들은 공식적으로 두 가지를 정의합니다.
속성: 명시성과 자체 포함. IR은 명시적입니다.
모든 IR 문이 한 번에 하나의 변수만 업데이트하는 경우
IR은 배타적으로 설명할 수 있는 경우 자체 포함됩니다.
기계 명령어의 의미. 그러나 그들의 초점은
IR의 정확성을 테스트하지만 효율적인 설계는 아닙니다.
리프터나 평가자.

Godefroid와 Taly[24]는 템플릿 기반 프로그램 합성을 사용하여 x86 명령어를
IR로 자동 변환합니다. 그들
CPU를 사용하여 각 명령에 대한 입출력 샘플을 수집하고 프로그램 합성 기술을 사용
하여
해당 IR. 합성 기반 접근 방식은 더 나아가
저자가 기존 IR을 활용한 [49]에서 조사
합성을 위한 광산 템플릿으로의 번역. 이 작품에서 그들은
프로그램 합성 기술을 사용하여 리프터를 지원하도록 확장했습니다.
이전에 지원되지 않는 아키텍처. 하사브니스와 세카르 [27]
리프팅을 자동화하기 위해 학습 기반 접근 방식을 채택하십시오. 그들의
접근 방식은 컴파일러를 활용하여 IR에서 어셈블리로의 변환을 위한 데이
터 세트를 생성하고 데이터 세트를 사용하여 매핑을 학습합니다.
어셈블리에서 IR까지. 이러한 작업은 감소를 목표로 하지만
리프터를 구현하기 위한 인간의 노력, 우리는 오히려
고도로 최적화된 리프터를 설계합니다.

III. 예비 연구

우리의 연구는 기존 바이너리를 조사하는 데 영감을 받았습니다.
분석 도구. 특히 10개의 오픈 소스 도구를 선택했습니다.
COTS 소프트웨어에 의존하지 않고 프런트 엔드를 분석했습니다. 표 I은 우리가 분석
한 오픈 소스 도구를 보여줍니다.

표 I: 우리가 연구한 오픈 소스 바이너리 분석 도구.

도구	버전	최신 릴리스	리포지토리 URL W
BAP	1.5.0	2018/10/11	https://github.com/angr/angr
BinNavi	6.1.0	2018/09/27	2008/12/161 https://github.com/BinaryAnalysisPlatform/bap
			https://github.com/binsec/binsec
			https://github.com/google/binnavi
비트 블레이즈	1.0		http://bitblaze.cs.berkeley.edu
인사이트	0.4	잭스탭 2014/06/11	https://github.com/hotelzulima/insight
	0.8.4	2017/03/31	https://github.com/jkinder/jakstab
미아즘	0.1.0	레이더2 2018/11/12	https://github.com/cea-sec/miasm
	3.1.3	2018/12/5	https://github.com/radare/radare2
rev.ng	0.1	2019/02/20	https://github.com/revng/revng
B2R2★	0.1.1	2019/03/22	https://github.com/B2R2-org/B2R2

[1] BitBlaze의 발행일을 사용했습니다.
★ 이것은 우리의 일입니다.

표 II는 그들의 리프터의 특성과 그들의
IR 의존적이기 때문에 여기에서 McSema [48]를 생략합니다.
IDA Pro [28]에서.

가. 관찰

도구를 검토하여 몇 가지 중요한 점을 발견했습니다.
이진 분석 프런트 엔드의 디자인. 우리는 그들을 강조
여기에서 표 II에 나타난 열의 순서대로. 이 모든
관찰 결과는 B2R2의 설계로 요약됩니다.
§ IV에 설명되어 있습니다.

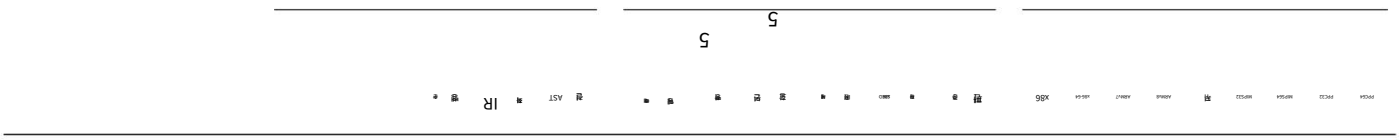
O1 (평행도). 많은 이진 분석 도구가 지원하지 않습니다.
언어 선택으로 인한 순수한 병렬 처리. 절반
도구는 Python(CPython) 또는 OCaml을 사용합니다.
다중 코어를 활용합니다[11]. 의 영향이 있지만
프론트 엔드의 병렬화 가능성이 무시되었습니다.
순수한 병렬 처리는 성능을 크게 향상시킬 수 있습니다.
그것: 우리의 병렬 리프팅 기술은 전체 리프팅을 증가시킵니다.
최신 데스크탑 컴퓨터에서 최대 2배의 속도를 제공합니다(§ VB 참조).

O2 (IR 최적화). BAP와 같은 몇 가지 도구 [13]
및 PyVEX[3]는 바이너리를 들어올리는 동안 IR 최적화를 수행합니다.
IR 최적화는 분석 비용 절감에 도움이 되지 않습니다.
해제된 IR을 평가하여 더 큰 이득이 없다면
진술. 그러나 우리는 중요한 성과를 달성할 수 있음을 보여줍니다.
활용하여 IR 최적화를 위한 성능 향상
다중 코어 병렬 처리: 리프팅 처리량이 다음보다 더 좋을 수 있습니다.
IR 최적화가 없는 것(§ VB).

O3 (AST 건설). 대부분의 도구는 추상 구문을 사용합니다.
IR을 나타내고 백엔드를 수행하기 위한 트리(AST)
복수. 그러나 한 가지 주목할만한 예외가 있습니다: Radae2 [4]
리프팅 프로세스 동안 AST를 구축하지 않습니다. 대신, 그것은
단순히 IR에 대한 문자열 표현을 내보냅니다. 그런 디자인
선택은 효율적인 리프팅을 가능하게 하지만 다른 한편으로는
정적 분석기를 작성하는 것은 어렵습니다.

O4 (AST의 메타데이터). 대부분의 기존 IR은 아무 것도 저장하지 않습니다.
AST의 추가 정보. 특히 PyVEX[3]는
분기 문의 유형 정보(PyVEX는 이를
점프 종류)의 AST. BAP의 AST에는 속성,
전달할 임의의 정보를 지정하는 데 사용할 수 있습니다.
서로 다른 분석 사이에 있습니다. 그러나 우리는 알지 못한다
AST에 특정 메타데이터를 보관하는 기존 도구
바이너리 수준의 데이터 흐름 분석을 향상시키기 위한 것입니다. 이 논문에서 우리는
가능하게 하는 새로운 AST 구축 방법론을 제시합니다.
IR에 메타데이터를 임베딩하여 효율적인 데이터 흐름 분석
표현(§ IV-D 참조).

표 II: 바이너리 분석을 위한 기존 오픈 소스 도구1 .



1

2

3

4

O5 (IR 표현력). 하나는 표현력을 나타낼 수 있습니다
자기 억제 및 명시적 개념을 가진 IR의 [30]. 대부분의 IR은 명시적이고 독립적이지만

그들 중 일부는 그렇지 않습니다. 특히 VEX IR은 외부 함수를 사용하여 명령어 의미를
표현하기 때문에
IR로 정적 분석기를 작성합니다. 의 표현력
IR은 바이너리 작성의 용이성에 상당한 영향을 미칠 수 있습니다.
분석기. 예를 들어 기호 실행 엔진은
내부적으로 VEX IR을 활용하는 gr에는
1.7 KLOC의 외부 기능.

O6 (해시 계산). 우리는 기존
바이너리 분석 도구는 기본적으로 해시 개념을 지원합니다.
IR. Hash-consing[21]은 다음에서 결정적인 것으로 널리 알려져 있습니다.
대규모 기호 실행 시스템 구축 [6], [12]
구조적으로 동일한 AST를 동일한 파일에 저장할 수 있습니다.
물리적 메모리를 사용하여 메모리를 크게 줄입니다.
사용. 그러나 기존 도구 중 어느 것도 해시 개념을 제공하지 않습니다.
IR을 위한 기능입니다.

O7 (임의 정밀도 정수). 대부분의 기존 도구는 큰 정수, 즉 임의 정밀도 정수를 사용합니다.
IR 구현. 임의 정밀도 정수는 다음을 나타내는 데 도움이 됩니다.
다음과 같은 벡터 레지스터를 사용하는 SIMD 명령어
Intel의 XMM 및 YMM 레지스터, 다음보다 큰 값 보유
64비트. 우리는 대부분의 바이너리 분석 도구가
SIMD 명령어, 그러나 그 중 어느 것도 큰 정수를 다음으로 분할하지 않습니다.
임의 정밀도 정수의 사용을 피하기 위해 더 작은 청크.
그러나 임의 정밀도 산술은 상당히 느립니다.
직접 실행할 수 있는 고정 정밀도 산술보다
ALU. 우리의 연구는
고정 정밀도 정수가 있는 기계어 코드는 실제로 간단하며 IR 평가(§ VC) 속도를 향상
시킬 수 있습니다.

O8 (아키텍처 지원). 많은 도구는 작은 세트에 중점을 둡니다.
작성해야 하는 ISA(명령어 집합 아키텍처)
각각의 다른 ISA에 대한 리프터는 독립적으로 필요합니다.
상당한 엔지니어링 노력. 이 관찰은
바이너리로 프론트 엔드의 모듈식 개발(§ IV-A)
분석 프레임워크는 전면에 다른 리프터를 쉽게 채택할 수 있습니다.

끝. 표 II의 마지막 열은 각 도구가 어떻게
여러 주요 아키텍처를 지원합니다.

B. 실험 설정

바이너리의 특성을 더 이해하려면
분석 도구와 해당 프론트엔드에서 각 도구를 실행하여
Ubuntu 16.04에서 얻은 실제 바이너리 세트를 들어 올립니다.
여기서 우리의 주요 목표는 리프팅 처리량을 비교하는 것입니다.
표 II의 도구. 의 성능만을 측정하기 위해
리프팅, 우리는 추가 IR 최적화를 수행하지 않고 선형 스윕 방식으로 각 도구로 테스트
바이너리를 실행했습니다.
기능 경계 식별 또는
CFG 재구성. 선형 스위프 방식으로만 리프터를 실행했지만 측정된 리프팅 처리량은

재귀 하강 리프터의 성능에 영향을 줍니다. 우리는 선택했다
우리의 실험에서 재귀 하강 리프팅을 사용하지 않기 위해
모든 도구가 지원합니다.

바이너리를 수집하기 위해 먼저 x86과 x86-64를 모두 설정했습니다.
두 개의 새로운 VM에서 Ubuntu. 그런 다음 바이너리를 추출했습니다.
/bin 및 각 VM의 /usr/bin에 있습니다. 결과적으로,
우리는 총 1,200 및 1,204 ELF 바이너리를 얻었습니다.
x86 및 x86-64 VM.

PyVEX와 같은 일부 바이너리 리프터는
원시 바이너리를 입력으로 사용하는 반면 다음과 같은 다른 도구는
BINSEC는 ELF와 같이 잘 구성된 바이너리만 사용할 수 있습니다.
바이너리, 입력으로. ELF 구문 분석 도구에 지원을 추가한 이후
또는 PE 바이너리는 시간이 많이 걸리고 오류가 발생하기 쉽습니다.
실험에 원시 바이너리만 사용합니다. 구체적으로, 우리는
획득한 각 바이너리의 .text 섹션을 추출했습니다.
objcopy로 그것들을 모두 하나의 raw로 결합했습니다.
각각 32비트 및 64비트 아키텍처용 바이너리 파일입니다. 로
결과적으로 32비트용 78MB blob과 79MB blob을 얻었습니다.
각각 64비트 아키텍처. 우리도 하나 더 얻었다
GNU의 .text 섹션을 추출하여 추가 원시 바이너리
x86 VM의 C 라이브러리(LIBC) 파일. 총 세 가지가 있습니다.
표 III에 표시된 원시 바이너리 명령어의 얼룩.

표 III: 리프팅 시간 비교.

도구	w/ 예쁜 인쇄(들) LIBC	x86 블롭	x86-64 블롭	예쁜 프린팅 없음 LIBC	x86 블롭	x86-64 블롭
분노	39.6	2576.6	2360.2	27.2	512.4	1619.2	1312.2
부드러운 롤빵	13.3	643.8	4.8	N/A1		261.0	221.6
빈섹	25.5	2078.7			12.4	660.8	N / A1
미아즘 레	171.5	9581.6	8686.5	133.3	94.3	7526.3	7448.5
이더2	1.8	104.1	1.7	455.9	2.3	92.3	84.5
B2R2 (0.0.1)2	11.4	568.8	180.7	1.6		86.3	71.8
B2R2 (0.1.1)	3.4	207.7				53.0	42.0

[1] BINSEC는 x86 아키텍처만 지원합니다.
2 대대적인 최적화 이전의 첫 번째 프로토타입.

원래 공부한 모든 도구를 실행하려고 했지만,
우리는 그 중 5개만 실행할 수 있었습니다.
다른 도구를 실행하기 위해 상당한 양의 코드 수정
추가 분석을 수행하지 않고. 표 III에는 5가지 도구가 나열되어 있습니다.
달릴 수 있었다는 것. 테이블의 세 번째 열
각 도구 버전의 릴리스 날짜를 나타냅니다.
우리는 사용했었다. 우리는 여전히 일부 리프터를 수동으로 수정해야 했습니다.
표에 나와 있는 모든 수정 사항을 요약합니다.
다음과 같이 만들었습니다.

파이벡스. 3줄을 수정하여 IR 최적화를 비활성화했습니다.
파이벡스. 우리는 또한 23줄의 Python 스크립트를 작성했습니다.
이진 명령어 시퀀스를 다음 시퀀스로 들어 올립니다.
VEX IR.
부드러운 롤빵. 우리는 Lifter Benchmark2라는 BAP 플러그인을 사용했습니다 .
들어 올리는 동안 다른 분석을 실행하지 않도록 하십시오.
빈섹. BINSEC는 원시 바이너리 파일을 입력으로 사용하지 않습니다.
따라서 리프팅을 지원하기 위해 BINSEC의 38개 라인을 수정했습니다.
원시 바이너리 파일.
독기. 우리는 23줄의 Python 스크립트를 작성했습니다.
원시 바이너리를 입력으로 사용하고 이를 Miasm 시퀀스로 들어 올립니다.
IR 진술.
레이더2. 우리는 독립 실행형 명령줄 도구인 rasm2를 사용했습니다.
원시 바이너리 파일에서 ESIL 문자열을 가져옵니다. 우리는 패치
처리되지 않은 명령을 무시하기 위해 rasm2의 한 줄. 우리
억제하기 위해 한 줄(printf 문)을 제거했습니다.
표 III에 대한 IR 인쇄.

C. 리프터의 성능 비교

단일 Intel 2.10GHz Xeon E5에서 5가지 도구를 실행했습니다.
우리가 얻은 세 가지 원시 바이너리 blob이 있는 코어
§ III-B. 표 III는
도구들. 표의 2-4열은 우리가 얼마나 많은 시간을
각 도구에서 IR 설명을 들어 올리고 예쁘게 인쇄하는 데 소비했습니다.
열 5-7은 우리가 리프팅에만 소비한 시간을 나타냅니다.
리프팅 처리량 때문에 결과는 다소 의외였습니다.
최악의 경우 두 배 정도 차이가 납니다. 가장 빠름
Radae2(B2R2 제외)이고 100배 이상이었습니다.
바이너리를 들어 올리는 데 Miasm보다 빠릅니다. 사실 그에도 불구하고
Radae2는 C로 작성되었지만 여전히 훨씬 빠릅니다.
다른 도구. 물론 빠른 리프터가 된다는 것은
Radae2는 최고의 바이너리 리프터입니다: AST를 빌드하지 않습니다.
O3에서 논의한 바와 같이 비교적 쓰기 어렵다.
분석기.

2https://github.com/BinaryAnalysisPlatform/bap-plugins/tree/master/
리프팅 벤치마크

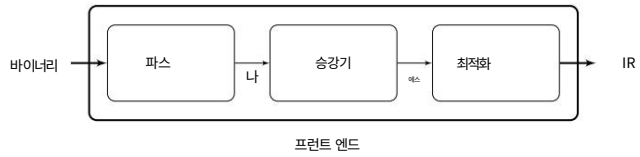


그림 1: B2R2 프런트 엔드의 아키텍처.

우리의 예비 연구는 대부분의 기존 도구가
분석의 용이성에 초점을 맞추되 효율성에는 초점을 두지 않음
프론트엔드 엔진. 사실, 이것은 주요 동기 중 하나입니다.
그것은 우리의 연구에 영감을 주었습니다. 이진 분석을 설계할 수 있습니까?
분석기를 쉽게 작성할 수 있는 프론트엔드와
여전히 효율적입니까? 프로그램을 작성하는 것으로 널리 알려져 있습니다.
대수학을 지원하는 기능적 언어가 있는 분석기
데이터 유형 및 패턴 일치 [46]가 훨씬 더 쉽습니다.
다른 언어보다. 따라서 여기에서 우리의 목표는
다음을 따르는 동안 효율적인 이진 분석 프론트 엔드를 작성하십시오.
기능적 패러다임.

O9 (리프팅 퍼포먼스). 들어 올리는 속도와 관련하여 개선의 여지가 크며 다음과 같이 할 수 있습니다.
적절한 엔지니어링으로 속도를 크게 향상
노력. 표 III에서 보듯이 우리는 B2R2를 만들 수 있었습니다.
덕분에 데이터 세트의 첫 번째 프로토타입보다 1.4배 더 빠릅니다.
시스템에 대한 대대적인 최적화(§ IV-H). 우리의
리프팅 처리량(예쁜 인쇄 없이)이 2배 더 빨랐습니다.
레이더2보다. 의 예쁜 인쇄 오버헤드에 유의하십시오.
Radae2는 AST를 빌드하지 않기 때문에 최소화됩니다.

이 백서에서 우리는 B2R2라고 하는 새로운 바이너리 분석 프레임워크를 설계하고
구현했습니다. 비교하기 위해서
B2R2의 효율성을 기존 도구와 비교하여 실행했습니다.
동일한 데이터 세트에서 최적화된 B2R2 버전 없이
병렬 리프팅 기술을 가능하게 합니다. 우리의 실험 결과
B2R2가 1.7배, 4.9배, 12.5배, 30.6배 빠름을 보여줍니다.
Radae2, BAP, BINSEC 및 angr보다 각각.

IV. B2R2 디자인

이 섹션에서는 B2R2 및 IR의 전반적인 설계 및 구현을 제시합니다. 특히, 우리는
키를 열거합니다
효율적인 이진 분석 프론트 엔드 및 IR 구축을 위한 설계 선택.

A. 모듈식 프런트 엔드 디자인

B2R2는 프런트 엔드를 세 가지 주요 모듈로 나눕니다.
구문 분석, 리프트 및 최적화. 그림 1은 전반적인
바이너리를 다음과 같이 취하는 B2R2 프론트 엔드의 아키텍처
입력하고 들어올려진 IR을 출력으로 반환합니다. PARSE 첫 번째 구문 분석이 제공됨
시퀀스를 나타내는 데이터 구조 I를 얻기 위한 이진 코드
아키텍처 중립적인 방식으로 지침을 제공합니다. 리프트 _
주어진 지침을 자체 IR 문 S로 들어 올립니다. 즉,
LowUIR, 그리고 마지막으로 OPTIMIZE 는 IR 문을 최적화합니다.
최종 IR 진술을 생성합니다.

우리는 다른 바이너리에서 유사한 디자인 패턴을 관찰합니다
BAP[13]와 같은 분석 도구. 그러나 일부 도구는
모듈식 설계 접근 방식을 따르십시오.
각 단계를 독립적으로 실행합니다. 예를 들어, PyVEX [3]
리프팅 기능에 IR 최적화 프로세스를 포함합니다. 우리의

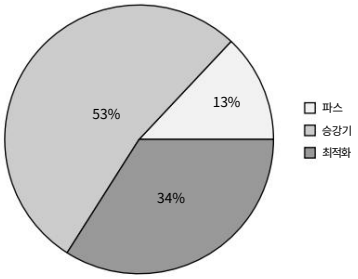


그림 2: B2R2 프런트 엔드의 성능 분석.

모듈식 설계 접근 방식은 병렬 리프팅을 가능하게 하는 핵심 요소 중 하나입니다 (§ IV-B).

B. 병렬 리프팅 및 최적화

병렬 리프팅은 Intel 아키텍처(예: x86 및 x86-64)에서 특히 어려운데, 가변 길이의 바이트 시퀀스로 명령어를 인코딩하기 때문입니다. 즉, 현재 명령을 완전히 구문 분석하고 크기를 파악하지 않으면 다음 명령을 예측할 수 없습니다. 따라서 구문 분석 단계 (PARSE) 는 병렬로 실행할 수 없습니다.

그러나 구문 분석된 명령어 목록이 주어지면 이들 사이에 종속성이 없기 때문에 병렬로 쉽게 들어올릴 수 있습니다.
B2R2는 먼저 PARSE 에서 구문 분석된 명령어를 누적하고 병렬로 리프팅 (LIFT) 및 최적화 (OPTIMIZE) 를 수행합니다. 반면 기존 리프터는 PARSE 와 LIFT 를 명령어마다 반복적으로 실행하기 때문에 병렬 연산이 원천적으로 불가능하다.

또한 예비 실험(§ III-C)에서 PARSE 모듈이 프런트 엔드 전체 실행 동안 CPU 주기의 13%만 차지한다는 것을 발견했습니다. 그림 2는 B2R2 프론트 엔드의 각 모듈에 대한 성능 분석을 설명합니다. 이것은 프런트 엔드 비용의 87%가 우리의 접근 방식과 병렬화될 수 있음을 의미합니다.

병렬 리프팅 구현에서는 PARSE 에서 N개의 구문 분석된 기본 블록을 누적하고 여러 스레드에서 누적된 명령어 블록을 비동기식으로 리프트 및 최적화합니다. 누적 기본 블록 수 N은 사용자가 설정할 수 있는 파라미터로 기계에 따라 최적값이 다를 수 있습니다. 더 많은 메모리를 사용하고 가비지 수집기에 더 많은 압력을 가하므로 N 값을 단순히 늘릴 수는 없습니다. 최적의 매개변수 값 N을 자동으로 구성하는 것은 이 문서의 범위를 벗어납니다.

우리는 병렬 리프팅이나 최적화를 수행하는 기존 바이너리 분석 도구를 알지 못합니다. 많은 기존 도구가 순수 병렬 처리(O1)를 지원하지 않는 언어로 작성되었음을 상기하십시오. 따라서 단일 프로세스 내에서 여러 코어를 활용하기가 어렵습니다. 그러나 B2R2는 불변성을 지원하는 함수형 언어로 작성되었기 때문에 병렬성을 쉽게 달성할 수 있습니다. 실험에서 병렬 리프팅은 프런트 엔드의 전체 파이프라인 성능을 두 배 향상시킬 수 있습니다(§ VB 참조).

```
방법 μ                          ::= 특급정보                    *단점정보
                                  | 특급정보
엔디안                           ::= 벤디안 | 렌디안
UNOP                  ♦u        ::= 네그 | 아니다
비놉                  ♦b        ::= 추가 | 서브 | MUL | DIV | SDIV | 모드 |
                                    스모드 | SHL | SHR | SAR | 그리고 | 또는 |
                                    XOR | 연결
릴롭                  ♦r        ::= EQ | 네크 | GT | GE | SGT | SGE |
                                    LT | 르 | SLT | SLE
캐스탑                ♦c        ::= ZeroExt | 서명 확장
EXPRESSION exp ::= 숫자 값 크기 |
                                    변수 이름 크기 |
                                    PCVar 이름 크기
                                  | TempVar 이름 크기 |
                                    이름 이름
                                  | UnOp ♦ u exp μ
                                  | BinOp ♦b 경험치 μ
                                  | RelOp ♦r exp exp μ |
                                    하중 크기 exp μ |
                                    ITE 경험치 경험치 μ |
                                    캐스트 ♦c 크기 exp μ |
                                    exp pos 크기 추출
                                  | 정의되지 않은 크기
STATEMENT stmt ::= ISMark addr len
                                    IE마크 주소
                                    엘마크 이름
                                    특급을 넣어 |
                                    상점 경험치
                                  | Jmp 특급
                                  | CJmp 특급 특급 |
                                    InterJmp 특급 |
                                    InterCJmp 경험치 경험치
                                  | 부작용 부작용
```

그림 3: LowUIR의 구문.

C. LowUIR

O5에서 논의된 바와 같이 IR은 백엔드 분석을 쉽게 하기 위해 명시적이고 독립적이어야 합니다. LowUIR이라고 하는 B2R2의 IR도 대부분의 다른 IR과 동일한 설계 선택을 따릅니다. 그림 3은 LowUIR의 형식적인 설명을 보여줍니다.

단일 기계 명령어는 일반적으로 여러 IR 문에 해당하므로 LowUIR은 각각 ISMark 및 IEMark 표현식으로 기계 명령어의 시작과 끝을 명시적으로 표시합니다. 유사한 방식으로 LowUIR는 점프 대상이 동일한 명령어 내에 있는지 여부에 따라 분기 명령어를 구별합니다. 예를 들어, x86의 BSF 명령어는 루프 내 레지스터의 각 비트를 스캔합니다. Jmp 또는 CJmp로 표시하는 명령어 내 점프를 사용하여 의미 체계를 간결하게 작성할 수 있습니다. x86의 jne 명령어와 같이 실제 기계 명령어 사이의 정규 분기 명령어의 경우 InterJmp 및 InterCJmp 문을 사용합니다. 이러한 IR 문은 제어 전송이 명령어 내에 있는지 여부를 명시적으로 식별하는 데 도움이 됩니다.

D. IR 메타데이터 임베딩

특히, LowUIR의 각 표현식에는 해당 표현식에 대한 메타데이터(ExprInfo)가 포함되어 있습니다. mally의 경우 IR 메타데이터는 IR의 작동 의미에 영향을 주지 않는 AST에 저장된 추가 정보입니다. LowUIR에서 각 ExprInfo는 (1) 표현식에 사용된 변수 세트와 (2) 표현식이 메모리에 액세스하는지 여부를 나타내는 부울 값을 저장합니다. 이러한 메타 정보는 use-def 체인을 구성할 때 각 AST의 모든 노드를 순회할 필요가 없기 때문에 데이터 흐름 분석기를 작성하는 데 유용합니다[5].

즉, use-def 체인을 구성하는 데는 일정한 시간이 걸립니다.

우리의 메타데이터. 예를 들어, 메타데이터를 관찰하여 taint 소스(용도)를 파악할 수 있고, taint sink(def)를 확인하여 빠르게 taint sink(def)를 인식할 수 있기 때문에 전체 AST를 순회하지 않고 간단한 overtaining 정책으로 taint 분석 도구를 작성할 수 있습니다. 우리의 IR은 명시적이기 때문에 AST의 루트 노드입니다. 즉, 각 명령문은 단일 변수만 업데이트할 수 있습니다.

불행히도, 그러한 메타데이터를 구성하는 것은 리프팅에 대한 순수한 오버헤드가 될 수 있습니다. 우리의 실험에서 LowUIR에 메타데이터를 추가함으로써 약 3%의 성능 저하를 관찰했습니다.
반면에 메타데이터를 활용하여 로컬 옵티마이저를 향상시킵니다(§ IV-E 참조).

E. 블록 레벨 로컬 최적화

기계 명령에 대한 의미 체계를 작성하는 것은 근본적으로 어렵고 상당한 엔지니어링 노력이 필요합니다. 더군다나 기계 지침을 최적화된 IR 문으로 번역하는 것은 훨씬 더 어렵습니다. 우리는 종종 우리의 IR 문이 반드시 최적은 아니라는 것을 관찰합니다. 그것들에는 중복 변수 할당이나 심지어 데드 코드가 포함되어 있습니다. O2에서 회상하면 PyVEX 및 BAP와 같은 소수의 리프터만이 해제하는 각 기본 블록에 대해 최적화를 수행합니다. 여기에는 상수 폴딩, 데드 코드 제거 및 다양한 구멍 최적화와 같은 전통적인 로컬 최적화 기술이 포함됩니다[5].

B2R2는 (1) 단순 대수 단순화를 통한 상수 접기, (2) 상수 전파, (3) 데드 코드 제거의 세 가지 블록 수준 로컬 최적화를 구현합니다. 최적화 방법을 효율적으로 수행하기 위해 IR 표현식(§ IV-D)의 메타데이터를 활용할 수 있습니다. 블록 수준 최적화는 데이터 세트에서 IR 문의 수를 17% 줄입니다(§ VA 참조).

F. 해시로 압축된 AST

O6에서 회상하면 기존 리프터 중 어느 것도 기본적으로 해시 개념 AST를 지원하지 않지만 해시 개념은 분석기의 성능을 크게 향상시킬 수 있습니다[6]. 기본적으로 지원하지 않는 IR에서 해시 개념을 사용하려면 각 AST를 새로운 유형으로 래핑해야 합니다. 이는 잠재적으로 성능을 저하시키고 상당한 엔지니어링 노력을 필요로 합니다.

B2R2는 명령어 수준 메타 데이터 ExprInfo와 함께 각 표현식별로 태그가 지정된 ConsInfo의 각 해시 개념 AST에 대한 고유 식별자와 해시 키를 저장하여 LowUIR 표현식 내에서 해시 개념을 구현합니다. 병렬 처리를 지원하는 해시 기반 AST를 유지하기 위해 스레드로부터 안전한 약한 해시 테이블[21]을 구현했습니다.

G. Bigint 분할

최신 CPU는 벡터 레지스터를 사용하여 명령 수준 병렬 처리를 지원합니다. 예를 들어 Intel의 SSE(Streaming SIMD Extensions)는 128비트 값을 보유할 수 있는 XMM0과 같은 XMM 레지스터와 함께 SIMD 명령어 세트를 추가합니다. 벡터 레지스터는 64비트보다 큰 크기의 값을 저장할 수 있지만 x86-64 프로세서의 ALU는 여전히 기본 워드 크기, 즉 64비트의 데이터로 작동합니다. 따라서 대부분의 SIMD 명령어는 벡터 레지스터의 값을 나누고 64비트보다 작은 크기의 값과 병렬로 여러 산술 연산을 수행합니다. 예를 들어 "VADDPS x86의 XMM0, XMM1" 명령어. 이 명령어는 XMM0 및 XMM1에 저장된 각 값을 4개의 32비트 값으로 분할하고 각 쌍을 병렬로 추가합니다.

O7에서 관찰했듯이 기존의 모든 이진 분석 도구는 벡터 레지스터를 나타내기 위해 임의 정밀도 정수에 의존합니다. 임의의 정밀도 정수로 SIMD 명령어의 의미를 구현하는 것이 간단하기 때문에 놀라운 일이 아닙니다. 그러나 임의 정밀도 정수 산술은 ALU에서 직접 실행할 수 없으므로 상당한 런타임 오버헤드가 발생합니다. BAP 및 BINSEC와 같은 일부 도구는 큰 숫자가 레지스터(64비트 또는 32비트 값)에 들어갈 수 있을 때마다 기본 정수 연산을 사용하는 Zarith[36]와 같은 임의 정밀도 산술을 위한 효율적인 라이브러리를 활용하지만 우리의 관찰은 우리가 그러한 라이브러리에 의존할 필요조차 없다는 것을 암시합니다.

LowUIR은 고정 정밀도 정수만으로 기계 명령어의 의미를 구현합니다. 벡터 레지스터를 처리하기 위해 64비트 의사 레지스터로 분할합니다. 예를 들어 x86의 XMM0 레지스터를 XMM0A와 XMM0B라는 두 개의 의사 레지스터로 나눕니다. 이진 의미 체계를 구현하는 이 방법을 bigint 분할이라고 합니다. 우리의 접근 방식은 대부분의 SIMD 명령어가 어쨌든 작은 데이터 값으로 작동한다는 사실에서 영감을 받았습니다. 여기서 단점은 bigint 분할이 bigint 분할이 없는 것보다 더 많은 수의 IR 문을 생성할 수 있다는 것입니다. 그러나 우리의 경험적 결과는 bigint 분할이 IR 평가 속도(§ VC)를 개선하는 데 도움이 된다는 것을 보여줍니다.

또한 bigint 분할은 벡터 레지스터에서 하위 값을 명시적으로 추출할 필요가 없으므로 SIMD 명령어의 전체 의미를 단순화할 수 있습니다.

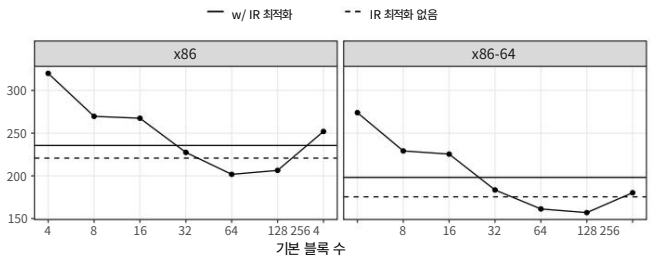
H. 구현

B2R2는 45 KLOC의 F# 코드로 구성됩니다. 다른 외부 종속성에 의존하지 않으므로 .NET CLR(공용 언어 런타임)에서만 실행됩니다[35]. 병렬 리프팅 및 IR 최적화를 구현하기 위해 F#의 비동기 워크플로[39]를 사용합니다(§ IV-B 참조).

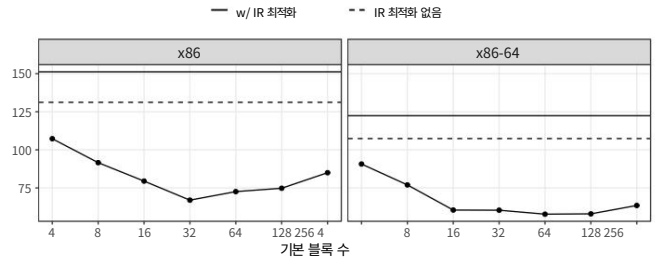
프론트 엔드를 효율적으로 만들기 위해 여러 반복에 대해 B2R2를 리팩토링하고 크게 최적화했습니다. 특히 불필요한 클로저와 연속을 제거하고 '구조체 튜플'[2]과 같은 로컬 데이터 구조를 사용하여 가비지 수집기에 대한 압력을 낮추어 힙 할당을 최대한 줄였습니다. 또한 컬렉션에 대한 임의 액세스가 필요할 때마다 목록을 배열로 교체하고 성능에 중요한 기능을 인라인했습니다. 마지막으로, 우리는 구별된 공용체에서 열거형으로 전환하고 중요한 실행 경로에서 예외를 제거했습니다. 결과적으로, 다중 코어 병렬 리프팅 방법을 사용하지 않고 단일 코어에서 B2R2의 리프팅 속도는 엄격한 최적화 후 2.8배 빨라졌습니다. 여기서 엄청난 성능 향상은 엔지니어링 노력의 중요성을 의미합니다.

V. 평가

이 섹션에서는 다음 연구 질문과 관련하여 B2R2를 평가합니다. (1) 최적화가 IR 진술을 단순화하는 데 도움이 됩니까? 그렇다면 IR 최적화를 적용하면 얼마나 느려지는가? (2) 다중 코어 병렬 처리를 활용하여 리프팅 프로세스의 속도를 높일 수 있습니까? (3) bigint 분할은 IR 문을 평가할 때 얼마나 많은 속도 향상을 제공합니까?



(a) 당사 서버 시스템(Xeon E5-2620 v4).



(b) 데스크톱 컴퓨터(i7-8700)에서.

그림 4: 누적할 기본 블록의 수에 따른 병렬 리프팅 성능 비교.

표 IV: 전후의 IR 문 수
IR 최적화를 적용합니다.

	선택 전	선택 후	감소율
LIBC 얼룩 x86	2,297,506	1,906,024	17%
얼룩 x86-64	138,882,883	114,718,934	17%
얼룩	103,848,026	87,456,016	16%

A. IR 최적화

IR 최적화는 이진 분석에 어떤 영향을 줍니까? 에게
이 질문에 답하기 위해 먼저 IR의 수를 비교했습니다.
IR 최적화를 적용하기 전과 후의 진술.
특히 사용된 동일한 대상 바이너리 blob을 들어 올렸습니다.
§ III에서, 해제된 IR에 IR 최적화를 적용했습니다.
진술. 그 결과, 우리는 수를 줄일 수 있었습니다.
x86 및 x86-64에서 각각 16% 및 17% 증가했습니다.
표 IV는 결과를 요약합니다. IR 수 줄이기
진술은 IR 평가뿐만 아니라
일반적으로 이진 분석에 사용됩니다. 예를 들어 다음과 같이 예상할 수 있습니다.
IR 최적화는 결과 경로의 크기를 줄이는 데 도움이 됩니다.
방식.

그러나 IR 최적화로 인해 추가 오버헤드가 발생할 수 있습니다.
리프팅 프로세스만 고려합니다. 실험에서 IR 최적화는 데이터 세트에서 평균 26.5%의
오버헤드를 발생시켰습니다.
이 결과는 자연스럽게 다음 연구 질문에 동기를 부여합니다.
다중 코어 병렬 처리(§ VB) 활용.

B. 멀티코어 병렬화 활용

§ IV-B에서 우리의 병렬 리프팅 기술이 활용합니다.
프론트 엔드의 성능을 향상시키기 위해 다중 코어.
평행 리프팅의 효과를 측정하기 위해 다음을 비교했습니다.
평행 리프팅이 있거나 없는 총 리프팅 시간
기술. 평행 리프팅으로 달릴 때 측정한
다음을 포함한 B2R2 프런트 엔드의 전체 파이프라인에 대한 시간
구문 분석, 리프팅, 최적화 및 예쁜 인쇄. 우리는 2개를 사용했다
이 실험을 위한 다른 기계: (1) 6년 된 Linux
Intel Xeon E5-2620 v4(32코어)가 있는 서버 시스템; 그리고
(2) Intel i7-8700이 탑재된 최신 Windows 데스크탑 시스템
(12 코어). 영향을 측정하기 위해 이 기계를 선택했습니다.
다른 환경에서 병렬 리프팅.

각 기계에서 기본 수인 N을 변경했습니다.
블록을 4개에서 256개까지 누적하고 전체를 측정했습니다.
리프팅 시간(§ IV-B 참조). 그림 4는 결과를 보여줍니다. 그만큼
점이 있는 선은 평행 리프팅을 위한 리프팅 시간을 나타냅니다.
축적할 기본 블록의 수를 늘리면서,
숫자가 64에 도달할 때까지 더 나은 처리량을 관찰했습니다.
그러나 64개 이상의 블록을 누적하면 시작했습니다.
메모리 압력 때문에 더 나쁜 성능을 관찰하기 위해
가비지 컬렉터는 더 기본적인 것을 축적할수록 증가합니다.
블록 및 해당 AST.

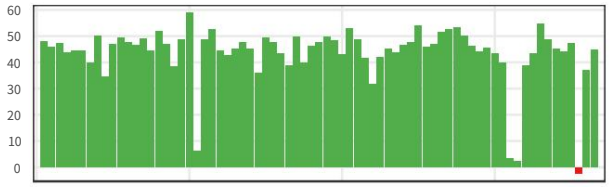
그러나 두 기계 모두에서 총 리프팅 시간은
병렬 리프팅은 IR을 수행하지 않은 것보다 적었습니다.
최적화. 이것은 우리의 평행 리프팅 기술이
프론트엔드 엔진에 상당한 이점을 제공하고
IR 최적화 없이 들어 올리는 속도보다 빠릅니다.
또한, 우리의 평행 리프팅은 우리의 모든 경우에 우세했습니다.
데스크탑 머신. 속도보다 최대 77% 빠릅니다.
IR 최적화 없이 리프팅. 이는 해당
우리가 예비에서 보여준 Radae2의 리프팅 속도
연구(§ III-C). 우리는 우리가 두 기계 사이에 상당한 차이를 관찰하는 이유가 우리 서버
가
기계의 메모리 대역폭이 약 절반입니다.
데스크탑 중 하나.

C. Bigint 분할

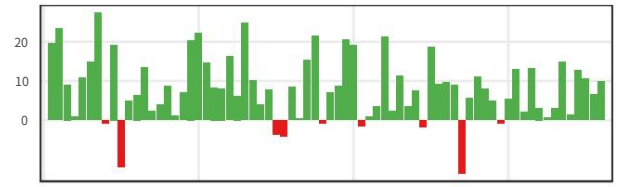
§ IV-G에서 우리의 bigint 분할은 다음을 가능하게 합니다.
고정 정밀도 정수가 있는 이진법의 의미. 우리는 할 수 있습니다
IR 문을 평가하여 bigint 분할의 영향을 관찰합니다. 이러한 영향을 측정하기 위해 실
행 추적을 수행했습니다.
GNU coreutils에서 실행을 구체적으로 에뮬레이트했습니다.
구체적인 실행 컨텍스트를 기반으로 합니다.

특히, 우리는 먼저 73개와 72개의 coreutils 바이너리를 수집했습니다.
x86 및 x86-64에 대해 각각. 그런 다음 무작위로 선택한
각 프로그램에 대한 단위 테스트 중 하나를 수행하고 대상을 실행했습니다.
로 구현된 자체 실행 추적 프로그램이 있는 프로그램
핀 [33]. 추적자는 대상 프로그램을 실행하고 기록은
및 실행 전반에 걸쳐 액세스되는 메모리 값
프로그램. 우리는 어려운 몇 가지 coreutils 바이너리를 생략했습니다.
단위 테스트 스크립트에서 테스트 케이스를 추출합니다. 주어진
실행 추적, 우리의 실행 에뮬레이터는 저장된 명령을 들어 올립니다.
추적에서 LowUIR로 이동하고 IR 문을 다음과 같이 평가합니다.
추적에 저장된 레지스터 및 메모리 값을 사용합니다.

그림 5는 실행 시간 사이의 비율을 보여줍니다.
각 coreutils 바이너리에 대한 bigint 분할 없이. 밖으로
총 145개의 바이너리 중 135개의 바이너리만이 유의미한 결과를 보였습니다.



(a) x86 coreutils에 대한 비교.



(b) x86-64 coreutils에 대한 비교.

그림 5: bigint 분할의 효율성.

성능 향상. 전반적으로 x86 및 x86-64에서 각각 43.5% 및 8.3%의 성능 향상을 관찰했습니다.
우리는 실험에서 x86 coreutils 바이너리가 곱셈 명령어와 같은 복잡한 산술 명령어를 더 많이 가졌기 때문에 x86에서 훨씬 더 많은 성능 향상을 얻었습니다. x86 바이너리에서 3배 더 많은 imul 명령어를 관찰했습니다. 임의 정밀도 정수를 사용한 곱셈 연산은 고정 정밀도 정수를 사용한 곱셈 연산보다 훨씬 느립니다. 반면에 bigint 분할은 처리할 변수의 수가 증가함에 따라 다른 10개 바이너리의 평가 속도를 늦출 수 있으며, 이는 차례로 삽입 및 사전의 찾기 작업을 느리게 하고 변수에서 다음으로 매핑을 저장합니다. 해당 값. 우리의 현재 구현은 내부적으로 이진 트리를 사용하여 사전을 구현하는 기능적 유한 맵을 사용합니다. 그러나 여기서 해시 테이블을 사용하여 성능 병목 현상을 완화할 수 있습니다.

우리는 bigint 분할이 특히 복잡한 산술 연산과 관련된 명령이 있는 경우 IR 평가 속도를 상당히 높일 수 있다고 결론지었습니다.

D. 유효성에 대한 위협

1) 대상 바이너리의 대표성: 대상 바이너리의 명령어 유형이 제한되면 실험 결과가 손상될 수 있습니다. 이 문제를 완화하기 위해 우리는 x86 및 x86-64라는 두 가지 다른 ISA에 대해 실제 OS에서 많은 수의 바이너리 실행 파일을 수집하려고 시도했습니다.

/bin 및 /usr/bin에 다양한 기능을 가진 적절한 양의 바이너리가 포함되어 있다고 가정했습니다.

2) 소스 수정의 정확성: § III-B에서 기억하고 우리는 다른 바이너리 분석 도구의 소스 코드를 수정했습니다. 코드를 진지하게 검토했지만 수정에 예상치 못한 부작용이 있을 수 있으며 이는 리프팅 성능에 영향을 미칠 수 있습니다. 우려를 완화하기 위해 GitHub에서 사용한 diff 파일을 시스템의 소스 코드와 함께 넣습니다.

VI. B2R2 기능 및 애플리케이션

이 문서의 주요 초점은 프런트 엔드이지만 B2R2에는 현재 다양한 백 엔드 모듈도 포함되어 있습니다.
이 섹션에서는 B2R2에서 개발한 응용 프로그램과 함께 가지고 있는 두 가지 백엔드 모듈에 대해 설명합니다.

A. 반환 지향 프로그래밍 컴파일

ROP(Return-Oriented Programming) 체인을 구축하는 것은 익스플로잇 개발의 중요한 부분입니다. B2R2는 ROP를 사용합니다.

주어진 바이너리를 분석하고 주로 Q에서 영감을 받은 ROP 페이로드를 반환하는 컴파일 모듈 [40]. 먼저 유용한 ROP 가젯을 검색하고 가제트를 결합하여 임의의 함수 호출, 시스템 호출 호출, 셸 생성 또는 스택 피벗 수행을 위한 ROP 페이로드를 빌드합니다.

B. 그래프 시각화

B2R2는 자동 바이너리 분석을 위한 풍부한 기능을 제공할 뿐만 아니라 분석가가 대상 바이너리를 수동으로 검사하는 데 도움이 됩니다. 특히 B2R2에는 CFG(Control-Flow Graphs)를 그리기 위한 자체 그래프 시각화 엔진이 있습니다. BAP, BINSEC 및 PyVEX와 같은 기존 바이너리 분석 프레임워크는 외부 그래프 레이아웃 도구에 의존하므로 대화형 방식으로 CFG를 분석하기 어렵습니다. 우리의 시각화 알고리즘은 Sugiyama 프레임워크를 기반으로 하는 표준 계층적 그리기 방법을 따릅니다[45].

C. 상징적 실행

바이너리 코드에 대한 심볼릭 실행 [8], [16], [43], [52]은 취약점 탐지, 자동 익스플로잇 생성, 난독화 해제 등 다양한 분야에서 유용하기 때문에 연구 커뮤니티의 주목을 받고 있습니다. B2R2 위에 우리는 B2R2의 유용한 기능을 활용하는 프로토타입 심볼릭 실행기를 구현했습니다. 예를 들어 LowUIR(§ IV-D) 표현식의 메타 정보를 이용하여 기본 블록을 기호적으로 평가할지 여부를 결정할 수 있었습니다. 이 기능은 또한 AST를 걷지 않고 제약 조건 독립성을 구현할 수 있게 합니다. 우리는 또한 기호 엔진의 속도를 높이기 위해 표현식 메모와 함께 해시 압축 AST(§ IV-F)를 사용했습니다.

또한 블록 수준 로컬 최적화(§ IV-E) 및 bigint 분할(§ IV-G)을 활용하여 의미 있는 성능 향상을 관찰했습니다. 심볼릭 실행 엔진의 세부 설계 및 구현에 대해 논의하는 것은 이 백서의 범위를 벗어납니다.

VII. 결론

이 논문에서 우리는 이진 분석의 프런트 엔드의 여러 디자인 측면을 연구했습니다. 우리는 현재 바이너리 분석 프레임워크가 대부분 분석의 용이성에 중점을 두고 있지만 효율성에는 중점을 두지 않는다는 것을 발견했습니다. 또한 병렬 처리 및 고정 정밀도 정수 사용과 같은 프런트 엔드에 대한 몇 가지 중요한 설계 포인트를 무시하므로 효율성에 상당한 영향을 미칠 수 있습니다. 우리의 주장을 입증하기 위해 우리는 기능 우선이고 병렬화 가능하며 리프팅 및

IR 평가. GitHub (https://github.com/B2R2-org)에서 소스 코드를 공개합니다.

우리 도구 개발에 끝없는 지원을 해주신 정승일, 오동엽, KAIST 사이버보안 연구센터(CSRC)에 감사드립니다. 또한 유익한 피드백을 주신 익명의 검토자 Ivan Gotovchits와 David Brumley에게도 감사드립니다.

이 작업은 한국 정부(과기부)의 지원을 받는 정보통신기술진흥원(IITP) 보조금 (No.B0717-16-0109, 이진 코드 분석을 통한 자동 역공학 및 취약점 탐지 플랫폼 구축)의 지원을 받았습니다.

esil.html.

[11] D. Beazley, "파이썬 GIL 내부", http://www.dabeaz.com/python/
길.pdf.

[15] CEA IT Security, "파이썬의 리버스 엔지니어링 프레임워크", https: //github.com/cea-sec/miasm.

[29] INRIA, "멀티코어 OCaml", https://github.com/ocamllabs/ocaml-multicore/ .

[34] VJM Manes, H. Han, C. Han, SK Cha, M. Egele, EJ Schwartz, M. Woo, “Fuzzing: Art, science, and engineering,” CoRR, vol. abs/1812.00140, 2018. [온라인]. 사용 가능: http://arxiv.org/abs/
1812. 00140

[35] Microsoft, "공용 언어 런타임(CLR) 개요." https://docs.
microsoft.com/en-us/dotnet/standard/clr.

[36] A. Mine, X. Leroy, P. Couq 및 C. Troestler, "자리스 라이브러리", https://github.com/ocaml/Zarith.

[41] EJ Schwartz, CF Cohen, M. Duggan, J. Gennari, JS Havrilla 및 C. Hines, "컴파일된 실행 파일에서 C++ 클래스 및 메서드 복구를 위한 논리 프로그래밍 사용", Proceedings of the ACM Conference on Computer and Communications 보안, 2018, pp. 426–441.

[42] EJ Schwartz, J. Lee, M. Woo, D. Brumley, "의미론 보존 구조 분석 및 반복 제어 흐름 구조화를 사용한 네이티브 x86 컴파일", USENIX 보안 심포지엄 회보, 2013, pp 353~368.

[43] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel 및 G. Vigna, “(state of) the art of war: Offensive technologies in binary analysis,” in Proceedings of the IEEE Symposium on Security and Privacy, 2016, pp. 138–157.

[44] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, MG Kang, Z. Liang, J. Newsome, P. Poosankam 및 P. Saxena, “BitBlaze: A new approach to 바이너리 분석을 통한 컴퓨터 보안”, 정보 시스템 보안에 관한 국제 회의 회보, 2008, pp. 1–25.

[45] K. Sugiyama, S. Tagawa 및 M. Toda, "계층적 시스템 구조의 시각적 이해를 위한 방법", IEEE Transactions on Systems, Man, and Cybernetics, vol. 11, 아니. 2, 109–125페이지, 1981.

[46] D. Syme, G. Neverov 및 J. Margetson, "경량 언어 확장을 통한 확장 가능한 패턴 일치", ACM SIGPLAN International Conference on Functional Programming, 2007, pp. 29–40.

[47] N. Tillmann 및 J. De Halleux, ".NET용 Pex-화이트 박스 테스트 생성", 테스트 및 증명에 관한 국제 회의 회보, 2008, pp. 134–153.

[48] 비트의 흔적, "McSema", https://github.com/trailofbits/mcsema.

[49] R. van Tonder 및 C. Le Goues, "Cross-architecture lifter synthesis", Proceedings of the International Conference on Software Engineering and Formal Methods, 2018, pp. 155–170.

[50] 벡터 35, "바이너리 닌자", https://binary.ninja/.

[51] K. Yakdan, S. Eschweiler, E. Gerhards-Padilla, M. Smith, "No more gotos: Decompilation using pattern-independent control-flow structuring and semantics-preserving transformations," in Proceedings of the Network and 분산 시스템 보안 심포지엄, 2015.

[52] I. Yun, S. Lee, M. Xu, Y. Jang, T. Kim, “QSYM: 하이브리드 퍼징에 맞춘 실용적인 concolic 실행 엔진”, USENIX 보안 심포지엄 회보, 2018, pp. 745-762.