효율적 바이너리 코드 재조립 시스템의 설계*

최지원, 차상길

한국과학기술원

On Designing an Efficient Binary Reassembler

Jiwon Choi, Sang Kil Cha

KAIST

요 약

바이너리 계측(instrumentation)은 악성코드 분석이나 취약점 탐지 등에 있어 핵심적인 기술이다. 최근에는 정적으로 역어셈블(disassemble)된 바이너리에 계측 코드를 삽입한 후 재조립(reassemble)하는 기술이 대두되고 있으나, 대부분 계측의 정확도 또는 효율이 현저히 낮아지는 문제를 가지고 있다. 본 논문에서는 최신 바이너리 재조립 기술들이 갖는 문제점에 대해 고찰하고, 정확도와 효율성 모두를 높일 수 있는 새로운 바이너리 재조립 기술을 제안한다.

I. 서론

소프트웨어의 동작을 관찰하고 측정하기 위한 기술인 바이너리 계측(instrumentation)기술은 악성코드 분석이나 메모리 오류 탐지, 바이너리 취약점 패치 등 다양한 소프트웨어 보안에 필요한 핵심적인 기술이다[1].

바이너리 계측 방식은 크게 동적 계측과 정적 계측 방식으로 나뉜다. 동적으로 코드 캐시를 변형하는 동적 계측 방식은 정적 계측에 비해 속도가 느리다는 단점이 있지만 동적으로 생성되는 코드를 계측할 수 있다. 한편, 정적 계측 방식은 속도가 빠르긴 하지만 동적 코드를 다룰 수 없을 뿐 아니라 역어셈블의 부정확성에 의해 계측의 정확도가 낮다는 단점을 갖는다. 최근에는 이러한 정적 계측의 문제를 개선한 Superset Disassembly 기술[3]이 소개되었는데, 모든 메모리 접근을 동적으로 계측함으로서 정확도가 획기적으로 개선된 반면, 속도 및 공간 효율이 크게 떨어지는 문제를 갖는다.

본 논문에서는 이러한 정적 바이너리 계측 기술이 갖는 문제점들에 대해 진단하고, 이를 경감하기 위한 기술들을 제안한다. 좀 더 구체적으로는 정적 값 분석(value analysis)과 동적 메모리 계산 방식을 결합한 하이브리드 형태의 심볼화(symbolization)를 개발하여, 이를 통해 실제 바이너리 코드를 안정적으로 재조립하는 기술을 제안한다.

우리는 제안하는 방법의 효율성을 검증하기 위해 GNU coreutils의 108개의 스트립(strip)된 바이너리들에 대해 재조립 실험을 진행하였다. 그 결과 우리는 정적 계측이 갖는 속도상의 우월성을 유지한 채 공간 오버헤드 또한 평균 300배 가까이 줄일 수 있었다.

II. 관련 연구

정적 바이너리 계측기술은 최근 들어 바이너리 재작성 기술을 통해 그 실효성이 높아지는 추세이다. 바이너리 재작성의 가장 큰 어려움은

*이 논문은 2018년도 정부(과학기술정보통신부)의 재원으로 정보통신기술진흥센터의 지원을 받아 수행된 연구임 (No.B0717-16-0109, 바이너리 코드 분석을 통한 자동화된 역공학 및 취약점 탐지 기반기술 개발).

역어셈블된 코드의 재조립시 코드와 데이터들이 재배치된다는 점이다. 즉, 원본 바이너리에서 코드나 데이터가 특정 위치에 고정 할당된 경우 이를 재조립한 바이너리에서는 실행 시 해당 코드나 데이터를 참조할 수 없는 문제가 발생한다. 과거 연구에서는 이러한 문제를 효과적으로 핸들링하기 위해 크게 두 가지 방법론이 제시되어 왔는데, 메모리 심볼화[2][4]와 전역 매핑 테이블의 운용[3]이 이에 해당한다.

메모리 심볼화는 바이너리 내부에 명시된 값이 메모리 참조 값인지 여부를 다양한 휴리스틱을 사용해 판단한다. 그리고 메모리 참조 연산들에 대해서는 메모리 주소 대신 심볼(라벨)을 활용하여 해당 영역을 참조할 수 있도록 변경한다. 만약 심볼화 작업이 완벽할 수 있다면, 이후 코드가 추가되거나 삭제되어 재배치가 일어나더라도, 정상적인 동작이 가능하다.

전역 매핑 테이블 운용 방식은 Superset Disassembly[3]에서 처음으로 제안되었으며, 모든 메모리 참조 명령어가 실행되기 이전에 전역 매핑 테이블을 통해 원본 코드의 위치를 참조하도록 하는 방식이다. 이 방법은 근원적으로 메모리 재배치 문제를 해소하지만, 모든 메모리 참조를 계측함에 따라 높은 런타임 오버헤드를 갖는다. 또한 추가적인 테이블과 원본 역어셈블 이미지에 대한 메모리를 요구하므로, 최소 2배에서 많게는 200배에 이르는 공간 오버헤드 증가를 갖는다.

III. 기존 시스템의 한계점

우리 시스템을 소개하기에 앞서 기존 시스템의 한계점을 크게 두 가지로 정리하였다.

첫째, 심볼화 기반 바이너리 재조립 기술은 기본적으로 PIE 바이너리(Position Independent Executable)를 다루지 못한다. 왜냐하면 바이너리 실행 이전에는 계산된 코드 포인터의 주소를 정적으로 결정할 수 없기 때문이다. 따라서 과거의 UROBOROS[2]나 Ramblr[4]의 경우 상용 바이너리에 적용되기 어려운 한계점을 가지고 있었다. 실제 Ubuntu 18.04 기준 /usr/bin의 98%가 PIE 바이너리이다.

둘째, 기존의 기술에서는 코드 재조립 이후에 발생할 수 있는 코드 및 데이터의 정렬(alignment) 문제를 고려하지 않았다. 코드 실행에 있어서 캐시라인에 정렬되도록 코드를 배치하는 편이 성능 상 유리하며, 특히 SIMD 명령어의 경우에는 정렬이 고려되지 않는다면 명령어의 실행 자체가 불가능한 경우가 발생한다. 따라서 원본 바이너리의 정확도와 효율을 유지하기 위해서는 재조립된 바이너리에서도 정렬을 보존해야 한다.

IV. 시스템 설계 및 구현

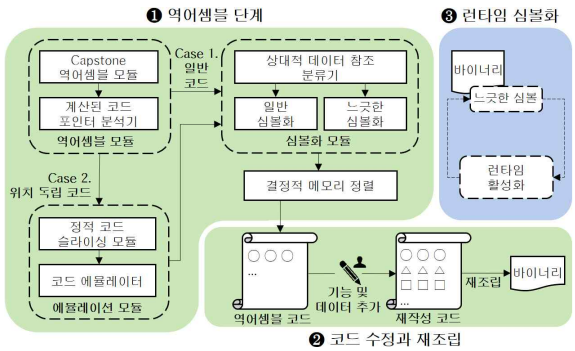

Fig. 1 System overview

우리의 시스템은 바이너리 파일을 입력받아 역어셈블과 재조립 과정을 거친 새로운 바이너리 파일을 생성한다. Fig. 1은 시스템의 전체적인 구조를 나타내는데, 앞 절에서 말한 두 가지 문제를 각각 해소할 수 있는 기술들이 각 단계별로 적용된다. 먼저 PIE 바이너리 문제의 경우 정적 코드 슬라이싱을 통해 코드 포인터의 주소를 계산하게 되며 (섹션 4.1), 미처 정적으로 다 계산되지 못한 심볼들은 느긋한 심볼화 과정을 통해 동적으로 메모리 주소를 해소하는 과정을 거치게 된다 (섹션 4.2). 또한 코드 정렬 문제의 경우에는 원본 바이너리의 정렬을 최대한 보존하는 결정적 메모리 정렬 기술을 사용하여 해소한다 (섹션 4.3).

4.1 정적 코드 슬라이싱

정적 코드 슬라이싱 모듈은 전위 슬라이스(forward slice)방식을 이용해 결정 가능한(deterministic) 레지스터 값을 분석한다. 슬라이

싱은 PIE 바이너리에서 실행 주소를 리턴하는 get_pc_thunk를 기점으로 시작되며 선택적으로 추출된 코드 슬라이스에서 레지스터 실제 값을 evaluation한다. 이때 결정 가능한 값들만을 계산하며, 계산된 값이 메모리 참조 값일 경우에 한해 이를 심볼화한다. 물론 정적으로 결정되지 못한 심볼들이 존재할 수 있는데, 이는 다음 절에서 설명하는 느긋한 심볼화 과정을 통해 동적으로 메모리 주소를 해소한다.

4.2 느긋한 심볼화

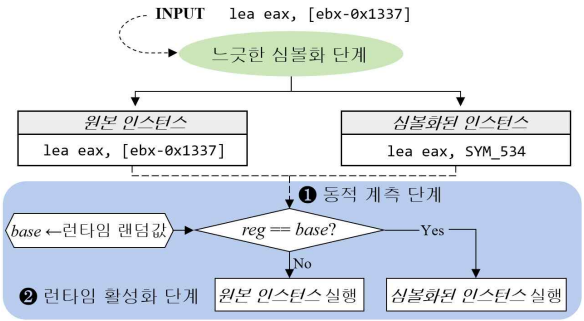

Fig. 2 Lazy symbolization

PIE 바이너리에서는 데이터 참조 시 바이너리의 특정 지점을 기준으로 하는 상대적인 주소를 사용한다. 기준점의 위치는 실행 시 랜덤하게 결정되기 때문에 기존의 기술이나 앞 절에서 논의한 정적 슬라이싱을 활용하더라도 추출이 불가능하다. 우리는 느긋한 심볼화(lazy symbolization) 기술을 통해 이를 해소한다.

먼저, 역어셈블된 코드 중에서 메모리 포인터 관련 명령어(LEA등)를 추출한다. 해당 명령어들은 데이터를 접근하기 위한 주소의 계산일 수도 있고, 단순 메모리 계산 명령일수도 있다. 하지만 후자를 심볼화하는 경우 잘못된 데이터 값을 사용하게 되는 문제가 존재한다. 따라서 우리는 동적 계측을 이용하여 두 경우를 판별한다. GCC 컴파일러의 경우 메모리 연산의 베이스 메모리 주소로 항상 고정된 위치(예를 들면 GOT)를 사용하는 점에 착안하여 동적으로 베이스 레지스터의 값을 확인하고 해당 값이 알려진 베이스 포인터 값이면 메모리 주소 연산으로 판별하도록 하였다.

따라서 Fig. 2에서 보는 바와 같이 메모리 관련 명령어들에 대한 동적 계측 코드를 추가적으로 삽입한다. 동적으로 계산된 베이스 주소가 GOT 등의 고정된 위치 값을 갖는 경우에는 심볼화된 인스턴스를 실행하며, 그 외에는 원본 인스턴스를 실행한다. 심볼화된 인스턴스가 사용할 심볼의 주소를 계산하기 위해 정적으로 베이스 레지스터의 값을 GOT주소라 가정하고 임의의 심볼을 부여한다. 단, 이때 가정이 잘못되었을 경우에도 의미 없는 심볼이 추가될 뿐 코드의 시멘틱을 변화시키지 않기 때문에 재조립된 바이너리는 정상적으로 동작하게 된다.

4.3 결정적 메모리 정렬

코드 재조립 이후에 코드 및 데이터 정렬이 보존되지 않는다면 바이너리의 성능이 떨어질 뿐 아니라 실행 자체가 불가능해질 수 있다. 이를 해소하기 위한 방법으로는 SIMD 명령어에서 참조하는 메모리 주소를 모두 조사하는 것이지만 이는 바이너리의 모든 실행 흐름을 복원할 수 있어야만 가능하다는 문제점이 있다.

결정적 메모리 정렬 기술은 원본 바이너리에 존재하는 SIMD명령어의 연산 대상 주소를 활용하여 해당 데이터를 정렬한다. 이때 정렬의 크기는 가장 큰 32바이트에서부터 2바이트에 이르기까지 내림차순으로 원본 주소의 정렬 크기를 근사하여 결정하는데, 이를 통해 정렬이 요구될 가능성이 있는 모든 데이터들을 정렬할 수 있으며, Superset Disassembly[3] 방식에 비해 적은 공간 오버헤드로도 구현이 가능하다.

4.4 구현

본 논문에서 제안하는 시스템은 Python을 사용하여 약 2,500줄로 구현하였다. 우리의 시스템은 크게 바이너리의 메타데이터 분석을 위한 Pyelftools, 바이너리 역어셈블을 위한 Capstone, 그리고 재조립시 기계어의 유효성 검증을 위한 Keystone 모듈로 구성된다.

V. 평가

제안하는 시스템의 효과를 확인하기 위해 바이너리 데이터 셋에 대한 실험을 수행했다. 실험은 8개의 Intel i7-7700K 4.20GHz 코어 및

4GB RAM을 갖는 Ubuntu 16.04 32bit 가상 시스템에서 진행하였다.

5.1 데이터 셋(Data Set)

제안하는 시스템의 성능 측정을 위해 GNU coreutils 8.30.10 에 포함된 108개의 스트립된 리눅스용 바이너리에 대해 테스트를 진행했다. 상용 바이너리에서의 다양한 컴파일 특성을 나타내기 위해 우리는 O0부터 O3에 이르는 4단계의 최적화 단계별로 생성했으며, 그 결과 총 432개의 바이너리를 얻을 수 있었다.

5.2 바이너리 동작의 정확도

우리는 재조립된 바이너리의 정확도를 검증하기 위해 다양한 입력을 정상적으로 실행할 수 있는지 테스트하였다. 이를 위해 coreutils에 포함된 608개의 유닛테스트 스크립트를 이용하였으며, Table 1의 두 번째 열에서 확인할 수 있듯이 모든 재조립된 바이너리에서 100%의 실행 정확도를 보였다.

최적화 단계	정확도	전체 크기 (원본 바이너리)	전체 크기 (재조립 바이너리)	증가율
-O0	100%	5009592 bytes	6040456 bytes	20.57%
-O1	100%	4751072 bytes	5315480 bytes	11.87%
-O2	100%	4966708 bytes	5550780 bytes	11.75%
-O3	100%	7016680 bytes	7524020 bytes	7.23%

Table 1 Spatial overhead

5.3 재조립된 바이너리의 크기 증가

Table 1의 3-5열에서 볼 수 있듯, 재조립된 바이너리의 크기 증가는 원본 바이너리에 비해 평균 12.85% 수준이었다. 최신 기술인 Superset Disassembly[3]에서 최소 2배의 사이즈 증가가 있었던 것을 감안할 때, 크기증가분이 획기적으로 감소한 것을 확인할 수 있다.

5.4 실행 오버헤드

재조립된 바이너리의 실행 오버헤드를 측정하기 위해 우리는 각 바이너리별로 100회에 걸쳐 coreutils test suite를 실행하였고, 그 결과를 Fig. 3에 정리하였다. 네 가지 최적화 수준에 대하여 평균 0.0788%의 오버헤드를 보였으며, 이는 무시할 수 있는 수준의 오버헤드이다. O1이나 O2 바이너리들에 대해서는 오히려 오버헤드가 감소하는 경향이 나타났는데, 이는 우리의 정적 코드 슬라이싱 기술을 통해 일부 간접 분기문이 직접 분기문으로 변화하면서 생긴 현상으로 수동 검증을 통해 해당 변환이 정상적임을 입증하였다.

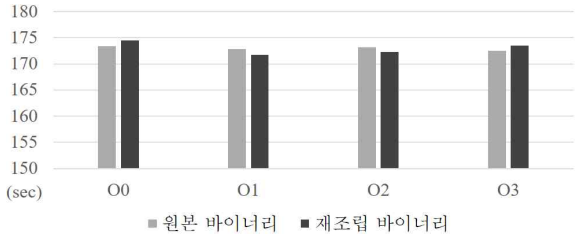

Fig. 3 Performance overhead

		UROBOROS [2]	*Ramblr [4]*	*MULTIVERSE [3]*	*Suggested System*
역어셈블과 재조립 가능	일반 바이너리	O	O	O	O
	PIE 바이너리	✗	✗	O	O
	라이브러리	✗	✗	O	O
기능 수정 데이터 수정 모두 가능	일반 바이너리	*partially*	*partially*	O	O
	PIE 바이너리	✗	✗	O	O
	라이브러리	✗	✗	O	O
w/o 모든 메모리 참조를 계측		O	O	✗	O
w/o 전역 맵핑 테이블 필요		O	O	✗	O
w/o 모든 분기문 재작성 필요		O	O	✗	O
x2 미만의 공간 오버헤드		O	O	✗	O

※ *partially* : SIMD 명령어가 데이터참조를 사용하지 않는 경우 부분적으로 가능

Table 2 Compare with existing system

VI. 결론

본 논문에서는 정확도와 효율성 모두를 높일 수 있는 바이너리 재조립 기술을 제안하였다. 끝으로 우리 시스템의 우수성을 Table 2를 통해 정리하였다.

[참고문헌]

[1] 차상길, “소프트웨어 보안과 바이너리 분석", 정보과학회지 제36권 제3호(통권 제346호), pp. 11-16, 2018.

[2] S. Wang, P. Wang, “Reassembleable Disassembling", in proc. of the USENIX security, 2015.

[3] E. Bauman, Z. Lin, “Superset Disassembly :Statically Rewriting x86 Binaries Without Heuristics", in proc. of the NDSS 2018.

[4] R. Wang, Y. Shoshitaishvili, “Ramblr : Making Reassembly Great Again", in proc. of the NDSS, 2017.