공학석사 학위논문

자연어처리 기반 바이너리 유사성을 활용한 취약점 탐지

2023년 2월

서울대학교 대학원

전기·정보공학부

안 성 관

자연어처리 기반 바이너리 유사성을 활용한 취약점 탐지

지도 교수 백 윤 흥

이 논문을 공학석사 학위논문으로 제출함
2023년 2월

서울대학교 대학원
전기·정보공학부
안 성 관

안성관의 공학석사 학위논문을 인준함
2023년 2월

위 원 장 이병영 (인)

부위원장 백윤흥 (인)

위 원 문태섭 (인)

초 록

최근 off-the-shelf application, system software들은 소스 코드를 재사용하여 실행가능한 바이너리의 형태로 배포되고 있다. 기존의 코드를 재사용하는 것은 생산성을 높였으나 이로 인하여 취약한 코드의 재생산이라는 예상치 못한 위험이 도래했다. 이로 인해 바이너리에서 취약점을 탐지하는 것이 어느 때보다 중요해졌다. 본 연구에서는 바이너리의 취약점 탐지를 위해 Binary Code Similarity Detection(BCSD)을 활용한다. BCSD란 소스 코드에 접근 불가능할 때 어셈블리 함수와 같은 바이너리 코드의 snippet들이 유사한지 판단하는 것이다. 이는 코드 복제 탐지, 취약점 탐지와 같은 다양한 사례에 적용 가능하다. 특히 취약점이 있는 바이너리와 비교를 통해 해당 바이너리에 취약점이 있는지 분석할 수 있다.

하지만 바이너리는 소스 코드에 비해 의미 분석이 어렵다. 이 점으로 인해 최근 BCSD 연구들은 바이너리 유사성 연구에 바이너리의 의미 분석 및 일반화를 위해 AI를 접목하려 한다. 이에 본 연구에서는 BERT 기반 바이너리 코드의 유사성을 학습하는 구조를 가진 모델인 BinShot을 제안한다. BinShot 모델은 샴 네트워크에 weighted distance 벡터를 사용했으며 이전 모델과 달리 binary cross entropy를 손실 함수로 채택하였다. 본 논문에서는 효과성, 전이 가능성, 실용성의 측면에서 기존 state-of-the-art 연구와의 비교를 통해 BinShot의 우수성을 입증한다.

주요어 : 바이너리 분석, 취약점 탐지, 자연어 처리, Few-shot 학습
학 번 : 2021-20916

목 차

표 목차

그림 목차

제 1 장 서 론

제 1 절 연구의 배경

오늘날 프로그램의 기능이 다양화되고 복잡도가 증가함에 따라 프로그램을 제작할 때 소스 코드를 처음부터 직접 작성하기보다 잘 설계된 라이브러리, 모듈, 프레임워크를 활용한다. 이를 통해 off-the-shelf application과 같은 프로그램들은 기존 소스 코드를 재사용하여 실행 가능한 바이너리의 형태로 우리에게 배포되고 있다. 프로그램의 크기와 기능이 증가하는 흐름에 맞춰 기존의 코드를 재사용하는 것은 생산성을 높였으며 코드 유지보수에 편리함을 가져왔다. 하지만 이로 인해 변형된 멀웨어의 확산과 취약한 코드의 재생산이라는 예상치 못한 위험이 도래했다. 이로 인해 바이너리에서 취약점을 탐지하는 것이 어느 때보다 중요한 과제가 되었다.

Binary Code Similarity Detection(BCSD)는 소스 코드에 접근할 수 없을 때 함수와 같은 바이너리 코드의 snippet을 비교하여 두 코드가 유사한지 결정하는 작업이다. BCSD는 실제 취약점이 있는 함수와의 비교를 통해 조사 대상인 바이너리 코드에서 취약한 함수가 있는지 알아볼 수 있다. 이뿐만 아니라 바이너리 코드 간의 비교를 통해 코드 복제 탐지, 멀웨어 탐지에도 응용될 수 있다.

하지만 소스 코드에서 바이너리로 컴파일하는 과정에서 변수의 이름과 타입, 함수 이름 등 코드의 의미적인 정보가 사라지거나 변형될 수 있으며 컴파일러 종류, 최적화 레벨, 컴퓨터 구조에 따라 바이너리의 형태가 달라질 수 있다는 점으로 인해 BCSD는 쉬운 작업이 아니다. 초기 BCSD 연구는 바이너리의 의미를 얻기 위해 graph isomorphism detection, data flow analysis의 정적인 분석과 바이너리의 실행 경로를 분석하는 동적인 분석을 취해 바이너리 코드의 유사성을 탐지해왔다. 하지만 이러한 방법들은 확장성이 떨어지며 코드 탐지 범위가 좁다는 측면에서 한계가 있었다.

최근 AI의 기술이 발전하고 이를 다양한 분야에 적용하여 문제를 해결하려는 흐름에 따라 BCSD에 AI를 접목하여 위의 한계를 극복하려는 시도가 활발해졌다. 예를 들어, [1-6]은 머신러닝을 BCSD에 적용한 사례이다. 이 중 DeepBinDiff[1]는 바이너리 코드의 의미 정보와 CFG(Control Flow Graph)를 학습해 basic block 단위의

임베딩 값을 생성하였다. 그리고 InnerEye[2]는 어셈블리 코드의 instruction, basic block, 함수를 각각 단어, 문장, 문단으로 처리했다. 이에 더 나아가, 최근 NLP에서 사전학습 언어 모델의 등장으로 DeepSemantic[3]에서는 바이너리 코드를 사전학습 언어 모델인 BERT로 학습하여 바이너리 코드의 심층적인 의미를 파악하고자 했다. 하지만 해당 연구는 유사성 분석이라는 작업과 달리 미세 조정 시 코드의 similarity를 학습하지 않지 않아 학습되지 않은 바이너리를 추론할 때 그 정확도가 많이 떨어진다.

이에 본 논문에서는 BERT 기반 similarity learning 구조를 제안하고자 한다. 본 논문에서 제안하는 BinShot은 사전 학습된 BERT 모델에 Siamese network와 weighted distance vector를 사용하였으며 손실 함수로 binary cross entropy를 채택하여 기존의 BCSD 연구보다 월등한 성능을 내는 모델이다. state-of-the-art 연구들과 비교 실험을 통해 해당 모델의 우수성을 effectiveness, transferability, practicality(난독화, 실제 시나리오)의 측면에서 증명하였고, 각각에서 모두 기존 방법보다 높은 성능을 보였다.

본 논문은 ACSAC 2022에 발표된 논문 ‘Practical Binary Code Similarity Detection with BERT-based Transferable Similarity Learning’의 추가 연구이다.

제 2 장 배경지식

본 장에서는 본 논문의 주 작업인 Binary Code Similarity Detection과 이를 위해 활용된 BERT 모델, Few-shot learning에 대해 설명한다.

제 1 절 Binary Code Similarity Detection

Binary Code Similarity Detection(BCSD)는 두 개의 바이너리 코드가 주어지면, 이들이 의미적으로 유사한지 식별하는 연구이다. 바이너리는 instruction(명령어), instruction의 집합인 basic block, basic block의 집합인 function, 마지막으로 함수의 집합인 entire program으로 구성되어 있어 이 중 어떤 요소를 유사성 비교 단위로 설정하는지에 따라 연구 내용이 달라질 수 있다. 본 논문에서는 선행 연구들의 작업에 맞춰 바이너리 코드의 비교 단위를 function으로 한다. 바이너리 코드의 유사성에 대한 라벨을 정의하는 기준은 두 함수가 동일한 바이너리의 동일한 함수여야 하는 것이다. 이에 프로그램을 컴파일하는 과정에서 적용된 컴파일러, 아키텍처 또는 최적화 레벨이 달라도 같은 바이너리, 함수 이름을 가진 함수 쌍을 유사하다고 간주하며, 그렇지 않은 경우 유사하지 않다고 간주한다. 즉, BCSD는 gcc, clang 등 두 가지 컴파일러와 O0부터 O3까지 네 가지 최적화 레벨로 컴파일된 머신 코드에서 비교하는 두 함수가 동일한지 비교하는 것이다.

BCSD 연구는 머신 러닝의 등장 이후 이를 접목한 연구가 활발해져 ML 이전, 이후로 나누어 볼 수 있다. ML 이전의 BCSD 연구는 정적, 동적 방법으로 분류할 수 있다. 정적, 동적 방법의 구분 기준은 유사성 분석 과정에서 직접 바이너리를 실행하며 바이너리의 의미를 분석하는지 여부이다. 정적 방법으로는 [33-35]와 같은 연구들이 있다. 대표적으로 [33, 34]는 바이너리의 call graph나 control flow graph(CFG)에 대한 graph isomorphism detection을 사용해 유사성을 분석했다. 동적 방법의 연구로는 [36, 37]와 같은 방법들이 있다. 동적 접근법은 직접 바이너리를 실행하여 의미를 분석하기에 I/O 쌍을 비교하거나[36] dynamic symbolic execution을 활용하는 방법[37] 등이 있다. 하지만 동적 연구는 바이너리를 실행하는 것으로 바이너리의 전체 의미를 파악하기에 시간이 많이 소요되며, 이로 인한 확장성 문제가 존재한다.

ML의 등장 이후로는 인공 신경망을 통해 바이너리 코드의 의미를

이해하고자 하는 연구가 이뤄졌다. Gemini[4]는 각 바이너리 함수의 CFG에 대한 임베딩 값을 계산하고, 두 함수의 임베딩 벡터에 대한 거리를 계산해 유사성 탐지를 효율적으로 진행한다. Asm2Vec[6]은 PV-DM 모델을 이용하였다. PV-DM 모델은 NLP에서 주변 단어를 참고하여 현재 단어의 임베딩 값을 구하기 위한 방법인 word2vec[7]의 추가 연구이다. Asm2vec[6]은 학습할 명령어의 전, 후 명령어를 참고하여 이에 대한 임베딩을 학습하도록 하였다.

ML을 활용하는 BCSD 연구는 NLP 분야에서 사전학습 모델에 대한 연구가 활발해짐에 따라 사전학습 모델을 바이너리 코드에 적용하려는 시도를 했고, 이로 인해 바이너리 코드의 의미를 더욱 심층적으로 이해할 수 있게 되었다. PalmTree[5]는 어셈블리 코드에 대한 언어 모델을 구축하여 명령어의 임베딩을 생성했고, DeepSemantic[3] 역시 BERT 모델을 이용하여 바이너리 코드를 학습했다. 다만 DeepSemantic[3]은 단순히 바이너리 코드를 BERT 모델에 집어넣은 것이 아니라, 어셈블리 코드의 명령어를 체계적으로 분석하여 normalize하였다. 이에 본 논문은 DeepSemantic[3]에서 사용한 명령어의 normalize 방법을 사용하기로 하였다.

제 2 절 BERT

BERT(Bidirectional Encoder Representations from Transformers)[8]는 NLP 분야에서 개발된 사전학습 언어 모델로, Transformers[attention] 모델의 encoder, decoder layer에서 encoder layer만 취한 뒤 토큰을 masking하는 전략을 취해 언어 이해 작업에 최적화하도록 학습한다.

Transformers[9]는 기존 Recurrent Neural Network (RNN) 계열 모델[10-12]에서 생기는 long term dependency 문제를 해결하여 텍스트 관련 작업에서 높은 성능을 기록한 sequence-to-sequence 학습 모델이다. Transformer의 encoder layer는 self-attention layer와 feed forward neural network으로 구성되어 있다. Self-attention mechanism은 입력 문장에 대한 query 벡터와 key 벡터의 유사도를 구한 뒤 해당 값을 차원의 제곱근으로 나눈다. 그 후 softmax를 취하고 value 벡터와의 곱을 통해 context score를 구하는 방식이다. 이러한 self-attention을 여러 개의 sub-layer로 나누어

계산하는데 이를 multi-head attention이라 한다.

BERT 모델은 transformers 모델의 encoder layer만을 이용해 언어 이해 작업에 특화하도록 학습했다. 초기 BERT 모델은 Masked Language Model(MLM)과 Next Sentence Prediction(NSP)의 두 가지 작업을 학습하였다. 우선 MLM은 입력 토큰 중 15%를 랜덤으로 선택하여 masking한다. 이 때 15%의 토큰 중 80%는 masking하고, 10%는 토큰을 무작위로 바꾼다. 이 과정을 거쳐 만들어진 입력을 transformers 구조에 넣어 주변 단어의 맥락으로 masking된 위치에 어느 단어가 와야 하는지 예측하며 학습한다. 이 학습 방식은 정답인 label을 입력으로 주지 않아도 문장 내에서 label을 만들어 학습하기에 self-supervised learning의 일종이다. NSP는 두 문장의 관계를 이해하기 위한 작업으로, 두 문장이 주어졌을 때 첫번째 문장의 다음에 두번째 문장이 올 것인지 예측하는 작업이다. 입력 문장에서 두 문장의 구분은 [SEP] 토큰이 기준이 된다. 하지만 BERT를 더욱 최적화한 모델인 RoBERTa[13]에 따르면, NSP 작업의 성능을 비교해본 결과 NSP를 하지 않는 것이 하는 것보다 성능이 비슷하거나 더 좋다는 것이 실험을 통해 밝혀졌다. 그래서 RoBERTa[13]는 self-attention의 복잡도로 인해 입력 문장의 길이가 한정된 경우 NSP로 두 문장을 학습하기 보다 한 문장을 더 길게 학습하는 것이 성능에 더 좋은 영향을 미친다고 주장했다. 이에 본 논문에서도 BERT를 학습할 때 NSP 작업은 사용하지 않고, MLM 작업만 수행했다.

BERT의 MLM 학습은 아래의 과정을 통해 파라미터 θ_m 을 최적화하며 수행된다.

$$\theta_m = \ argmin \sum_{t \in T} p(t|y) \log p(t|\hat{y})$$

$$\hat{y} = \ Softmax\big(G_m(X)\big)$$

$t, T, y, \hat{y}$ 은 각각 토큰, 토큰의 집합, 마스킹 이전 원래 토큰, 마스킹된 위치에서 예측된 토큰을 가리킨다. $G_m(X)$ 는 함수 X를 입력으로 받은 MLM 분류기의 fully connected 계층의 출력 벡터를 나타낸다. 이 과정을 거치며 MLM은 마스킹된 토큰의 위치에 올 가장 적절한 토큰이 무엇인지 학습하게 되고, 그 결과 문장의 문맥을 이해하게 된다.

BERT 모델은 언어 이해에 대한 성능이 증명되어 자연어 외에 프로그래밍 언어, 단백질 서열 등 다른 도메인에도 적용되었다.

CodeBERT[14]는 Python, Java, Javascript, Ruby, Go, PHP의 6가지 프로그래밍 언어와 자연어를 동시에 사전학습한 모델로 코드 검색과 코드 문서 생성에 좋은 성능을 보였다. ProteinBERT[15]는 긴 문장에 대해 모델을 매우 효율적이고 유연하게 만드는 새로운 아키텍처 요소를 소개하여 레이블이 지정된 데이터가 제한된 경우에도 단백질 예측 변수를 빠르게 훈련할 수 있도록 했다. 이를 통해 다양한 벤치마크에서 SOTA와 비교해 유사하거나 더 좋은 성능을 보였다. 본 논문에서는 바이너리 코드의 instruction을 체계적으로 normalize하여 함수 단위로 학습한 DeepSemantic[3]의 BERT 모델을 이용하여 실험을 진행했다.

제 3 절 Few-shot Learning

Few-shot Learning(FSL)[16]은 머신 러닝 problems의 한 종류로, 학습할 데이터가 제한된 개수로 주어진 상황에서 이를 효율적으로 학습하여 테스트 셋에서 유의미한 결과를 얻기 위한 방법이다. 이는 데이터 수집의 어려움과 라벨링에 드는 비용 문제로 인해 활발히 연구되고 있다.

기존의 FSL의 문제들은 주로 지도학습 문제이다. 구체적으로, FSL을 통한 분류 문제는 각 클래스에서 라벨이 있는 몇 가지의 데이터만으로 분류기를 학습한다. 일반적으로 N개의 클래스에서 K개의 예시를 학습하는 경우 N-way K-shot 분류라고 한다. 예를 들어 이미지에서 강아지, 고양이, 사자, 양이라는 4개의 클래스를 구분할 때 각 클래스별로 2개의 이미지만으로 학습한다면 4-way 2-shot 분류가 된다. 특별한 경우로, 만약 K가 1인 경우에는 one-shot learning[17]이라 하며 K가 0인 경우, 즉 클래스에 대한 학습 데이터가 하나도 없는 경우 zero-shot learning[18]이라 한다. 이 때 학습 예시와 라벨로 구성된 적은 양의 학습 데이터셋을 support set, 모델의 성능을 평가할 데이터셋을 query set이라 한다.

FSL의 방법으로는 크게 데이터, 모델, 알고리즘의 3가지로 나눌 수 있다. 데이터 측면으로는 데이터를 증강하여 성능을 올릴 수 있다. 학습 데이터의 샘플을 변형하거나 유사한 데이터셋에서 샘플을 추출하여 기존의 학습 데이터 수를 늘릴 수 있다. 모델 측면에서는 사전 지식을 통해 hypothesis space를 제한하는 방법이다. Parameter sharing 또는 tying으로 multitask learning을 하거나 데이터의 임베딩을 학습하는 방식 등이 있다. 본 논문에서는 데이터의 임베딩을 학습하고, 임베딩

벡터 간의 거리를 학습하는 방식으로 FSL을 수행한다. 마지막으로 알고리즘 측면에서는 fine-tuning을 통해 기존 parameter를 조절하거나 optimizer를 학습시키는 등의 방법을 통해 FSL을 수행한다.

제 3 장 제안 방법

제 1 절 디자인 및 설계

본 논문의 전반적인 수행 흐름은 [그림 1]과 같다. 바이너리에서 어셈블리 코드를 만들어내 이를 normalize 함으로써 데이터를 전처리하고, BERT 모델을 통해 어셈블리 코드에 대한 사전학습 모델을 제작한다. 그 후 사전학습 모델을 BCSD 작업에 맞게 미세 조정을 실시하여 본 논문에서 제안한 모델인 BinShot을 생성한다. 마지막으로, BinShot을 이용해 두 함수가 주어지면 그 함수들이 유사한지 여부를 예측하도록 한다.

[그림 1]에서 바이너리 전처리는 크게 두 가지 과정으로 나눌 수 있다. 첫번째는 바이너리 disassemble, 두번째는 normalize이다. 바이너리 disassemble은 state-of-the-art 바이너리 분석 툴인 IDA Pro[31]를 사용해 실행가능한 바이너리에서 디버깅 정보와 함께 어셈블리 코드 코퍼스를 생성한다. 그 후 DeepSemantic[3]에서 제안한 방법인 [표 1]에 따라 명령어를 normalize한다.

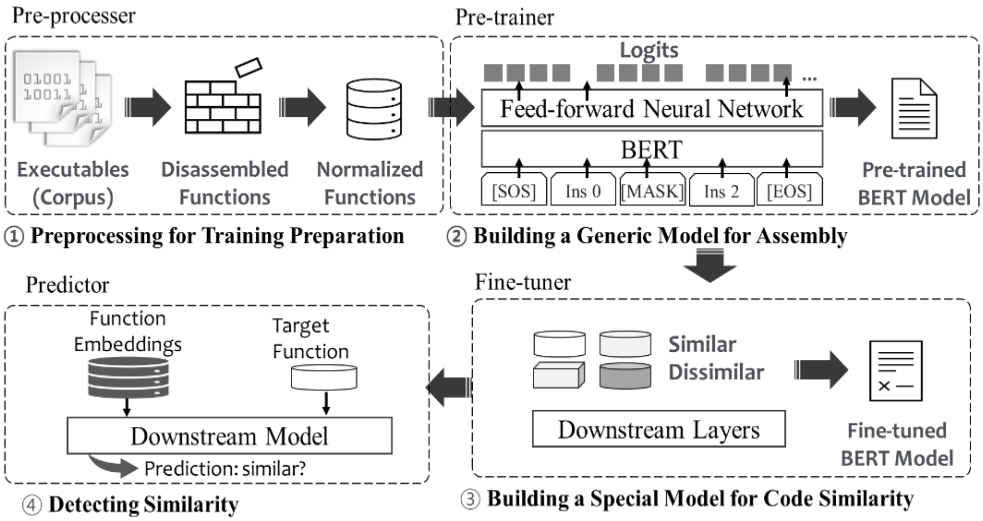

그림 1. 수행 흐름도

Rule	Notation	Example
[I] libc library call	libc[name]	call_libcprintf
[I] recursive call	self	call_self
[I] call within a binary	innerfunc	call_innerfunc
[I] call out of a binary	externfunc	call_externfunc
[I] jump to a destination	jmpdst	jmp_jmpdst
[I] referring a string	dispstr	mov_reg4_dispstr
[I] referring a static variable	dispbss	mov_reg8_dispbss
[I] referring non-string data	dispdata	movabs_reg8_dispdata
[I] other immediates	immval	sar_reg8_immval
[R] register size	reg[1|2|4|8]	rax → reg8, ebx → reg4
[R] stack register	rsp	rsp → sp8
[R] base register	rbp	rbp → bp8
[R] instruction register	rip	rip → ip8
[P] direct pointer	[byte|word |dword|qword]ptr	mov_qwordptr [reg8-8]_reg8
[P] indirect pointer	[base_index*scale +displacement]	or_dwordptr [reg8+disp]+immval

표 1. Instruction normalization 방법

어셈블리 코드 사전학습 모델 제작은 BERT 모델을 사용하여 학습한다. 본 논문에서는 BERT의 두 가지 작업 중 NSP 작업은 하지 않고, MLM만 시행했다. 그 이유로 NLP의 인접 문장과 달리 바이너리 코드에서는 함수 간의 관계가 위치가 아닌 함수 호출에 의해 결정되어 다음 함수 예측(NSP 작업)이 무의미해지기 때문이다. 본 논문에서는 함수 호출을 직접 고려하지는 않는 대신 instruction normalize 과정에서 libc 호출을 별도의 단어로 정의하여 묵시적으로 처리한다. 그렇지 않으면 normalize 과정에서 외부 라이브러리 호출과 내부 함수 호출(실행 파일 내에 정의된 다른 함수)이 동일하게 처리되기 때문이다. 그리고 normalize 시 간접 호출(call reg8)이나 재귀 호출(call self)을 더 나은 문맥 추론을 위해 별개의 instruction으로 본다.

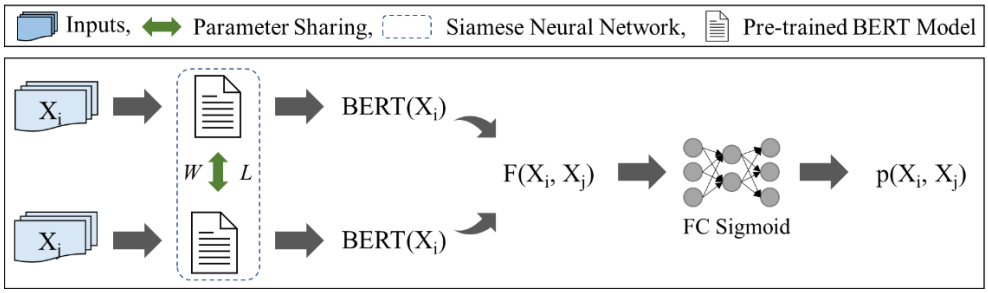

그림 2. BinShot 모델 구조

사전학습 모델 제작 후 BCSD 작업을 위해 파라미터를 미세 조정한다. 본 논문에서 제안하는 BinShot의 모델 구조는 [그림 2]와 같다. [그림 2]에서 X, W, L, F, FC, p 는 각각 input, weights, loss, distance vectors, fully connected 계층, 코드 유사도에 대한 확률이다. 즉, 사전 학습된 BERT모델을 샴 네트워크 구조로 만든 후 라벨링된 데이터셋(정규화된 함수 한 쌍과 유사성에 대한 라벨)을 입력으로 받아 BERT 모델을 통해 각 함수에 대한 임베딩 벡터를 출력하여 그 두 벡터 간의 weighted distance를 학습하는 것이다.

아래 수식은 임베딩 벡터 간의 거리를 어떻게 계산하는지 나타낸다. $BERT(X)$ 는 정규화된 함수 X가 BERT 모델을 거쳐 생성된 임베딩 벡터이고, $F(X_i, X_j)$는 두 임베딩 벡터 간의 distance vector를 나타낸다. D는 거리를 측정하는 함수이다. 본 논문에서는 k=2로, squared error를 사용했다.

$$D(X,Y) = (x_i - y_i)^k, \qquad F(X_i, X_j) = D(BERT(X_i), BERT(X_j))$$

거리를 계산한 후 아래 수식을 통해 이진 분류기가 binary cross entropy 손실 함수를 통해 weighted distance vector를 학습한다.

$$L = Y \log p(X_i, X_j) + (1 - Y) \log(1 - p(X_i, X_j))$$

$$p(X_i, X_j) = \sigma(FC(F(X_i, X_j)))$$

$$\sigma(z) = \frac{1}{1 + e^z}$$

이 때 Y 는 입력 (X_i, X_j) 에 대한 라벨이며 $FC(X) = WX + b$ 는 가중치(W)를 가진 fully connected 계층의 출력 벡터를 의미한다. σ는 시그모이드 함수로, 입력 함수 쌍에 대한 유사도 값을 0~1 범위의 확률로 나타낸다.

본 논문에서 제안하는 방법의 특징은 BCSD에 적합한 손실 함수를

사용했다는 것이다. 이전의 학습 기반 BCSD 모델[4,5]은 종종 contrastive 손실 함수를 사용했다. 아래 수식은 contrastive 손실 함수 L_c를 나타낸다.

$$L_c(W, Y, X_i, X_j) = Y\frac{1}{2}(F_W)^2 + (1-Y)\frac{1}{2}(\max(0, \alpha - F_W))^2$$

Contrastive 손실 함수는 입력 함수의 쌍이 positive인 경우 임베딩 벡터 사이의 거리를 최소화하고, negative인 경우 두 임베딩 벡터의 거리를 마진(α , 본 논문에서는 1로 설정)을 넘지 않는 범위에서 최대화한다.

하지만 본 논문은 이진 분류를 위해 binary cross entropy로 손실 학습을 정의하는 OSL(one-shot learning)[17]에서 제안한 학습 방법을 사용한다. Binary cross entropy 손실 함수를 사용함으로써 다른 손실 함수를 사용하는 state-of-the-art BCSD 모델보다 학습하지 않았던 함수의 추론에 있어 훨씬 더 나은 성능을 발휘함을 확인하였다. 또, 본 논문에서 제안한 방법에 binary cross entropy 손실 함수 대신 contrastive 손실 함수를 사용하여 그 결과를 비교해 보았다.

OSL 방법을 사용하여 바이너리 코드 유사성에 대한 미세 조정이 된 모델로 추론을 한다. 모델은 한 쌍의 함수를 입력으로 받는다. 이 때 하나의 함수는 비교할 타겟 함수이며, 다른 하나는 데이터베이스에 저장된 함수이다. 데이터베이스에 저장된 함수에 대한 임베딩 값을 계산할 때 ① 데이터베이스 함수의 임베딩을 미리 계산해 놓으며 ② 학습하지 않은 함수에 대한 임베딩 벡터도 정의해두면, 입력 함수 쌍에 대한 연산을 더욱 빨리 할 수 있다. 이 과정을 통해 BERT 모델로 각 함수의 임베딩 벡터 값을 구하면 두 벡터 값의 distance vector를 계산하고, 해당 값이 이진 분류기와 시그모이드 함수를 통해 0과 1 사이의 확률로 출력된다. 최종적으로 임계값 p=0.5를 기준으로 0(negative) 또는 1(positive)의 값을 얻어 함수 쌍에 대한 유사도를 예측하게 된다.

제 4 장 실험

제 1 절 실험 설계

Dataset	Binaries	Functions	Basic blocks	Instructions	Tokens
GNU utilities	1,000	439,036	3,965,532	22,137,920	1,244
SPEC2006	176	407,277	4,661,761	28,307,441	9,631
SPEC2017	120	755,297	9,336,587	52,940,593	11,932
Real-world programs	104	169,065	2,422,651	14,269,468	8,892
Total	1,400	1,770,675	20,386,531	117,655,422	18,449

표 2. 데이터셋

본 논문에서 학습 및 모델 평가를 위해 사용된 데이터셋은 [표 2]와 같다. GNU utilities, SPEC2006, SPEC2017, 실생활에 주로 쓰이는 프로그램들로 선정한 real-world programs(RWP)이다. GNU utilities에는 선행 연구[1, 6, 36]에서 벤치마크로 사용되었던 바이너리들인 binutils(v2.26), coreutils (v8.30), diffutils (v2.8), and findutils (v4.7.0)이 있다. SPEC2006과 SPEC2017은 일련의 표준화된 CPU 집약적인 벤치마크이다. 또 RWP에는 Github에서 오픈 소스 프로젝트로 인기 있는 실제 프로그램 11개를 수집했다. BusyBox (v1.34.1) [19], Libgmp (v6.2.1) [20], ImageMagick (v7.0.10) [21], Libcurl (v7.78.0) [22], LibTomCrypt (v1.18.2) [23], OpenSSL (v1.1.1f) [24], SQLite (v3.30.1) [25], zlib (v1.2.8) [26], PuTTYgen (v0.76) [27], Nginx (v1.16.1) [28], and vsftpd (v3.0.3) [29]이 RWP에 해당된다. 각 데이터셋의 소스 코드로부터 2개의 다른 컴파일러(gcc, clang)와 4개의 다른 최적화 레벨(O0-O3)을 사용하여 전체적으로 1,400개의 실행 가능한 바이너리를 생성하였다.

실행 가능한 바이너리에서 학습 및 실험을 위한 함수의 쌍을 만들기 전에, 본 논문에서는 함수의 크기(instruction 개수)를 기준으로 함수를 필터링했다. 구체적으로 instruction의 개수가 m보다 작거나 같은 함수 또는 n보다 크거나 같은 함수를 아웃라이어로 설정했다. 그 이유로는 ① 너무 작은 함수는 그 의미가 유의미하지 않을 수 있고 ② 취약점 여부를 확인하고 싶은 타겟 함수의 instruction 개수가 적을 가능성이 희박하며 ③ BERT가 너무 긴 입력 문장을 처리하는 데에는 한계가 있기 때문이다. 실험을 위해 본 논문에서는 m = 5, n = 250으로 설정했다. 전체 함수 데이터셋에서 각 임계값으로 인해 필터링된 작은 함수와 큰

함수의 비율은 각각 16.47%, 4.66%이다.

이 결과 바이너리 데이터셋의 전체 1,770,675개의 함수에서 117,655,422개의 instruction이 생성된다. 그 후 실험을 위한 코퍼스를 만들기 위해 DeepSemantic[3]의 normalize 방법을 사용하여 전체 instruction에 대한 vocab을 만든다. 이 과정으로 총 18,449개의 vocab이 생성된다. 각 vocab이 토큰으로 처리되기 때문에, 18,449개의 토큰에 더해 BERT에 대한 5개의 특수한 토큰을 추가하여 총 18,454개의 토큰을 얻었다. 특수한 토큰으로는 문장의 시작을 의미하는 [SOS](start of sequence), 문장의 끝을 의미하는 [EOS](end of sequence), 알 수 없는 토큰을 의미하는 [UNK](unknown), 마스킹을 의미하는 [MASK], 패딩을 의미하는 [PAD]토큰이 이에 해당된다. 이렇게 구축된 vocab을 바탕으로 BERT 사전학습 및 미세 조정에 대한 데이터셋을 구성한다.

기존 실험에 더해, 본 논문은 난독화된 바이너리를 대상으로 실험하여 논문의 실용성을 보였다. 이를 위해 BinKit[30]에서 제공하는 난독화된 바이너리 데이터셋을 사용했으며, Obfuscator-LLVM[38]에서 사용 가능한 기법 중 가짜 control flow를 추가한 bogus control flow(BCF), 기존 명령어를 다른 명렁어로 대체하는 instruction substitution(SUB), control flow 형태를 변환한 control flow flattening(FLA), 그리고 3가지 옵션 모두 적용된 (ALL) 기법을 사용했다. 바이너리 데이터셋에서 코퍼스는 BinShot에서 코퍼스를 제작하는 과정과 동일하게 생성되었다.

본 논문에서 바이너리의 disassemble과 유의미한 정보 추출을 위해 state-of-the-art 바이너리 리버스 엔지니어링 도구 중 하나인 IDA Pro[31]를 활용한다. 어셈블리 함수 목록, 상호 참조(예: call graph), 섹션 이름, 문자열 참조 및 외부 라이브러리 호출을 추출할 수 있는 내장 IDAPython API로 스크립트를 작성하여 instruction 정규화를 용이하게 했다.

본 논문에서는 ML을 위한 인기 있는 프레임워크 중 하나인 PyTorch로 BinShot을 개발한다. 프로토타입은 BERT와 샴 네트워크를 사용하여 구현했다. 거리 함수의 경우, 실험을 통해 제곱 오차의 전체 성능이 절대 오차의 성능보다 약간 더 우수하다는 것을 확인했기 때문에 제곱 오차(k=2)를 채택하였다.

하이퍼 파라미터는 다양한 옵션 중에서 최고의 성능을 보여주는 것으로 선택했다. 명령어 임베딩을 위한 256개 차원, 128개의 숨겨진

레이어, 8개의 어텐션 레이어 및 헤드(Transformer 구조용), 256개의 최대 토큰 길이, 3개의 토큰 인코더 레이어. 학습률이 0.0005인 Adam 옵티마이저와 사전 훈련 및 미세 조정 모두에 드롭아웃 알고리즘을 사용한다. 드롭아웃 비율은 토큰 인코더 레이어의 경우 0.2이고 다른 레이어의 경우 0.1이다. 이에 더해, 안정적인 학습 결과를 위해 선형 학습 속도 워밍업 전략과 그래디언트 클리핑 전략을 사용한다. 미세 조정은 배치 크기 32로 5 epoch 동안 훈련했다.

본 논문에서 사용된 Metric은 정밀도(P), 재현율(R), 정확도(A), F1 점수(F)이다. 아래 수식에서 TP, FP, TN, FN은 각각 true positive, false positive, true negative, false negative의 수이다.

$$P = \frac{TP}{TP+FP},\ R = \frac{TP}{TP+FN},\ A = \frac{TP+TN}{TP+TN+FN+FP},\ F = \frac{2\times P\times R}{P+R}$$

실험은 2개의 Intel Xeon Gold 6226R CPU, 256GB RAM 및 2개의 NVIDIA RTX A6000 GPU가 장착된 서버에서 진행했다.

제 2 절 실험 결과

ML 기반 접근 방식을 사용한 이전 BCSD 연구는 효과성(정확도, F1 점수) 및 효율성(런타임 성능)에 주로 중점을 두었다. 본 논문에서는 취약한 함수를 데이터베이스에 저장하여 실제 CVE 함수를 탐지할 수 있는 실용성 측면에서 BinShot을 평가한다. 또 일반 바이너리가 아닌 난독화[38]한 바이너리에 대한 유사도 탐지 결과를 통해 BinShot이 난독화된 바이너리에도 강건한지 평가한다. 그리고 시각화 기술(t-SNE[32])을 사용하여 유사한 함수 임베딩 벡터 그룹이 차원 축소된 공간에서 실제로 서로 가깝다는 것을 입증한다.

난독화된 바이너리를 제외한 모든 실험들은 기준 모델로 Gemini[4], Asm2Vec[6], PalmTree[5] 및 DeepSemantic[3]과 같은 최신 BCSD 모델을 선정했다. PalmTree의 경우 논문의 저자들이 직접 공개한 사전 훈련된 모델을 활용하였다. 공정한 비교를 위해 동일한 데이터 셋으로 다운스트림 모델을 구축하였고, 각 모델의 논문에서 제시한 하이퍼 파라미터를 따랐다.

또 손실 함수의 효율성을 보여주기 위해 BinShot의 변형인 BinShot-CTR을 추가로 비교한다. 이 BinShot-CTR은 거리 함수로 L2 distance를 선정했으며 contrastive 손실 함수를 적용했다.

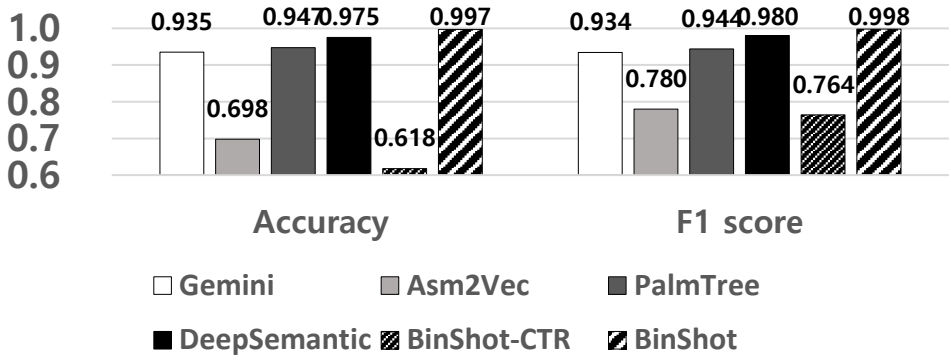

그림 3 (a). 전체 데이터셋 대상 실험 결과

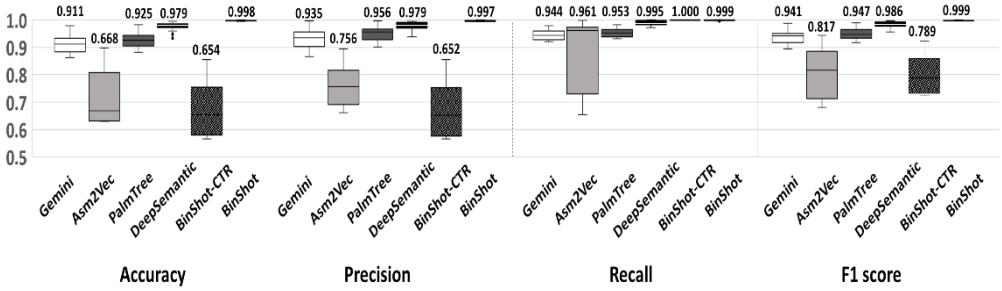

그림 3 (b). 서로 다른 컴파일러 및 최적화 레벨 실험 결과

[그림 3]은 4개의 기준 모델(Gemini[4], Asm2Vec[6], PalmTree[5], DeepSemantic[3]) 및 BinShot 변형 모델(BinShot-CTR)과 비교하여 BinShot의 효과성을 보여준다. [그림 3의] (a)는 전체 데이터셋, (b)는 서로 다른 컴파일러 및 최적화 레벨(예: (gcc-O0, gcc-O0), ..., (clang-O3, clang-O3))에 따른 결과를 나타낸다. (b) 결과는 코드 유사성을 추론하기 위해 다양한 구성(즉, 교차 컴파일러, 교차 최적화 수준)의 영향을 더 잘 나타내도록 설계되었다. [그림 3]의 (a)는 BinShot이 BCSD에 대한 다른 모든 state-of-the-art 모델을 능가한다는 것을 명확하게 보여준다. 마찬가지로, [그림 3]의 (b)는 BinShot의 성능이 컴파일러와 최적화 수준의 조합에 관계없이 낮은 분산으로 안정적으로 유지됨을 증명한다. 이에 본 논문에서는 BinShot이 거리 벡터의 모든 요소 간의 관계를 이해하여 정교한 추론을 할 수 있기 때문에 weighted distance를 학습하는 것이 BinShot이 다른 모델을 능가하는 데 도움이 된다는 가설을 세웠다. 한편, 다른 모델은 스칼라 값(특징 벡터로부터의 거리)을 기반으로 하기에 두 함수 간의 관계를 지나치게 단순화한다.

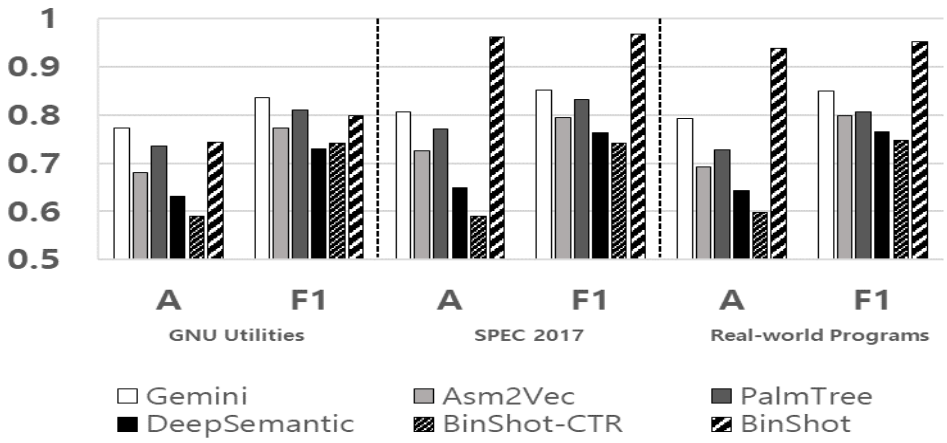

그림 4 (a). SPEC2006을 학습하여 다른 데이터 셋으로 추론한 결과

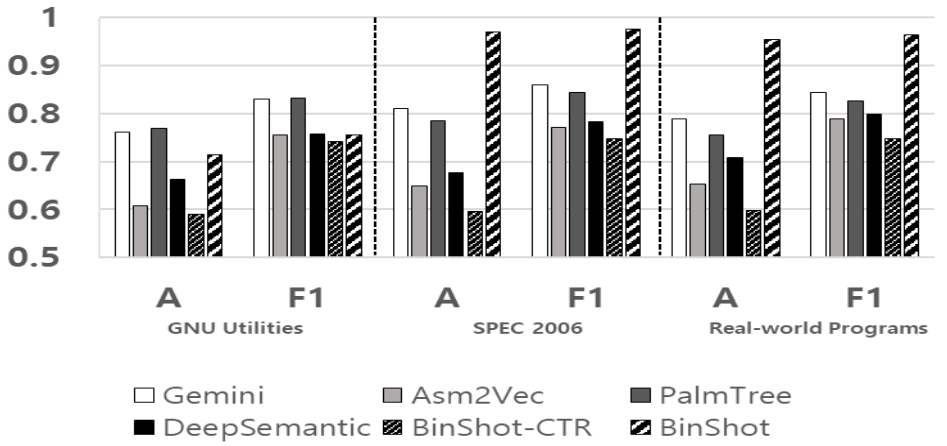

그림 4 (b). SPEC2017을 학습하여 다른 데이터 셋으로 추론한 결과

[그림 4]는 본 논문의 Transferability 결과를 보여준다. 구체적으로 transferability 실험은 다음과 같은 질문을 제기하며 실시되었다. ① 데이터 세트 X에서 학습된 모델이 다른 데이터 세트Y에서 바이너리 유사성을 추론할 수 있는 능력은 얼마나 될까? ② 어떤 최첨단 모델이 가장 좋은 성능을 낼 것인가? Transferability가 높다는 것은 모델이 더 일반화되거나 확장 가능하다는 것을 의미한다. 이에 본 논문의 전이 가능성 실험은 특정 모델을 실제로 완전히 다른 바이너리에 적용할 때 일반화할 수 있는지를 보고자 하였다.

이에 본 논문에서는 SPEC2006 및 SPEC2017의 바이너리

코퍼스만을 가지고 학습을 수행하여 2개의 미세 조정 모델을 제작하였다. 기준 모델인 Gemini[4], Asm2Vec[6], PalmTree[5], DeepSemantic[3] 및 BinShot-CTR에도 동일한 데이터 셋(SPEC2006, SPEC2017)으로 총 10개의 서로 다른 모델을 구축했다. 그런 다음 각 모델이 학습되지 않은 데이터 셋(예: GNU 유틸리티, 실제 프로그램)에서의 유사성을 추론하도록 하였다. [그림 4]는 정확도와 F1 점수를 포함하여 다양한 모델에 대한 추론 결과를 간략하게 보여준다. [그림 4] (a)와 [그림 4] (b) 모두에서 볼 수 있듯 BinShot은 GNU 유틸리티를 제외한 다른 데이터 셋에서 더 나은 성능을 달성했다. 즉, BinShot 모델의 정확도 및 F1 점수는 4개의 기준 모델 및 BinShot-CTR의 정확도 및 F1 점수보다 높다. 예를 들어, SPEC2017의 전이 학습을 적용할 때 real-world 프로그램의 F1 점수([그림 4] (b)의 가장 오른쪽 막대)는 최고의 성능을 나타냈다.

실험 결과를 통해 다음과 같은 통찰을 얻을 수 있었다. 1. BinShot으로 미세 조정된 모델은 일반적으로 다른 (학습된 적 없는) 데이터 셋에 적용할 수 있으며, 이는 샴 신경망을 사용한 weighted distance 학습이 의도한 대로 작동하였다. 2. SPEC2006의 함수는 SPEC2017의 함수와 공통점이 거의 없다. 왜냐하면 SPEC2006과 SPEC2017의 상호간 추론 결과와 real-world 프로그램 추론 결과가 큰 차이가 없기 때문이다. 마지막으로 BinShot은 GNU 유틸리티에서 상대적으로 낮은 성능을 보였다. 조사를 통해 본 논문에서는 GNU 유틸리티가 서로 다른 바이너리에 걸쳐 꽤 많은 함수가 공유됨을 발견했다. 이는 coreutils가 다양한 유틸리티 프로그램을 컴파일할 때 공통으로 정적 라이브러리(libcore.a)를 가지기 때문이다. 이 프로그램은 본질적으로 여러 바이너리에 동일한 함수를 포함한다.

Program	CVE	Vulnerable function	Gemini		Asm2Vec		PalmTree		DeepSemantic		BinShot-CTR		BinShot	
			O0-O3	A/R	O0-O3	A/R	O0-O3	A/R	O0-O3	A/R	O0-O3	A/R	O0-O3	A/R
OpenSSL v1.0.1e*	2014-0160	tls1_process_heartbeat	✓✓✓✓		✓✓✓✓		✓✓✓✓		✓✗✓✗		✓✓✓✓		✓✓✓✓	
		dtls1_process_heartbeat	✓✓✓✓		✓✓✓✓		✓✓✓✓		✓✗✓✗		✓✓✓✓		✓✓✓✓	
	2014-0221	dtls1_get_message_fragment	✓✓✓✓	0.0033/ 1.0000	✓✓✓✓	0.1179/ 1.0000	✓✓✓✓	0.0140/ 1.0000	✓✗✓✗	0.3656/ 0.6000	✓✓✓✓	0.0033/ 1.0000	✓✓✓✓	0.9009/ 1.0000
	2014-3508	OBJ_obj2txt	✓✓✓✓		✓✓✓✓		✓✓✓✓		✓✓✓✗		✓✓✓✓		✓✓✓✓	
	2015-1791	ssl3_get_new_session_ticket	✓✓✓✓		✓✓✓✓		✓✓✓✓		✗✓✓✓		✓✓✓✓		✓✓✓✓	
NTP v4.2.7p10	2014-9295	crypto_recv	- - - -		- - - -		- - - -		- - - -		- - - -		- - - -	
		ctl_putdata	✓✓✓ -	0.0055/ 1.0000	✓✓✓ -	0.1588/ 1.0000	✓✓✓ -	0.0083/ 1.0000	✓✓✓ -	0.4505/ 1.0000	✓✓✓ -	0.0064/ 1.0000	✓✓✓ -	0.7940/ 1.0000
		configure	✓ -✓✓		✓ -✓✓		✓ -✓✓		✓ -✓✓		✓ -✓✓		✓ -✓✓	
libav v0.8.3	2012-2776	decode_cell_data	✓✓✓✓	0.0007/ 1.0000	✓✓✓✓	0.1215/ 1.0000	✓✓✓✓	0.0065/ 1.0000	✗✓✗✓	0.0003/ 0.5000	✓✓✓✓	0.0007/ 1.0000	✓✓✓✓	0.9497/ 1.0000

표 3. 취약한 함수 탐지 결과

다음으로는 본 논문에서 제안하는 방법의 실용성을 보이기 위해 바이너리에서 실제 취약한 함수를 탐지하는 실험을 수행했다. [표 3]은 6개의 CVE, 9개의 취약한 함수가 포함된 3개의 프로그램을 나타낸다. 본 논문에서는 실제 취약점 탐지 시나리오에 가깝도록 다음의 세 가지를 가정했다. ① 관심 있는 취약한 함수가 gcc로 컴파일 되었으며, 해당 함수의 임베딩 벡터가 데이터베이스에 존재한다. ② 쿼리 바이너리는 strip되고 clang으로 컴파일 되었다. ③ 쿼리 바이너리에서 데이터베이스에 저장된 함수를 기준으로 취약한 함수를 탐지한다. 이전의 접근법[4, 6, 36]은 함수에 취약점이 포함되어 있는지 확인하기 위해 탐색하고 싶은 함수만 쿼리한다. 그러나 취약한 함수만 쿼리하여 모델을 평가하는 것은 바이너리의 어느 함수가 취약한지 알 수 없을 때 다수의 false positive(예: 모든 함수가 취약하다고 말할 때)가 있을 수 있으며, 이는 정확도 측면에서 문제가 있다. 이에 본 실험에서는 정확한 결과를 위해 함수가 아닌 전체 바이너리 자체를 쿼리하여 모든 취약한 함수를 찾을 수 있도록 한다.

즉, Clang O0-O3로 컴파일된 3개 프로그램, 총 12개 쿼리 바이너리로 BinShot 및 BinShot-CTR, 그리고 4개 기준 모델을 평가했다. [표 3]은 취약한 함수를 탐지한 결과를 요약한 것이며 ✓와 ✗는 각각 탐지 성공/실패를 의미한다. 리버싱 툴은 종종 strip된 바이너리에서 함수 경계를 인식하지 못하기 때문에 몇 가지 함수(crypto_recv, ctl_putdata-O3, configure-O1)는 배제했다. 실험 결과, DeepSemantic[3]을 제외한 모든 모델이 쿼리 바이너리에서 취약한 함수를 발견했다. 그러나 기준 모델 모두 매우 낮은 정확도(즉, 50% 미만)를 보인 반면, BinShot만이 높은 정확도(평균 88.2%)를 달성하여 실제 시나리오에 적용할 때 효율성을 입증했다. 이는 대개 데이터베이스의 모든 비교 중 benign 함수를 오탐하여 취약한 함수로 반환하여 전체 정확도가 떨어지기 때문이다.

Model Type	Option	Acc	Prec	Recall	F1
BinShot	ALL	0.759	0.881	0.600	0.714
	BCF	0.820	0.948	0.679	0.791
	FLA	0.916	0.977	0.851	0.910
	SUB	0.960	0.992	0.928	0.959
BinShot-Obfuscation	ALL	0.972	0.960	0.986	0.973
	BCF	0.953	0.961	0.946	0.953
	FLA	0.973	0.956	0.991	0.973
	SUB	0.979	0.968	0.991	0.979

표 4. 난독화된 바이너리 실험 결과

[표 4]는 난독화된 바이너리 대상 실험 결과이다. 지금까지 데이터셋은 난독화가 적용되지 않은 바이너리였다. 하지만 실제 상황에서 바이너리를 추론할 때는 난독화된 바이너리가 존재한다. 이에 모델의 실용성과 강건함을 입증하기 위해 본 논문에서는 4-1절에서 언급한대로 ALL, BCF, FLA, SUB 총 4가지의 옵션으로 난독화된 바이너리를 사용하여 성능을 측정했다.

[표 4]의 BinShot은 모델에 난독화된 바이너리를 학습시키지 않았을 때의 추론 결과이다. BCF와 ALL 옵션에서의 F1 점수는 각각 0.714, 0.791로 이전 실험에 비해 좋은 결과를 얻지 못했다. 다만 SUB와 FLA 옵션에서는 F1 점수가 각각 0.910, 0.959로, 학습되지 않은 형태의 바이너리임을 감안하면 좋은 결과를 얻었다. SUB 옵션은 바이너리 연산자를 복잡한 형태의 명령어로 변경하는데 BinShot은 명령어의 형태보다 의미를 이해하는 모델이기에 성능이 유지된 것으로 보인다. FLA 옵션은 기존 control flow를 평평하게 만드는 기법이지만 BinShot에서는 control flow를 고려하지 않으므로 기존 결과와 유사한, 즉 강건한 결과를 보였다. 이에 반해 BCF 옵션은 의미 없는 명령어로 구성된 새로운 basic block을 추가한다. 명령어가 추가된 부분이 노이즈로 작용해 BCF와 ALL 옵션의 결과가 낮게 나온 것으로 보인다.

이에 난독화된 바이너리도 BinShot 학습에 포함한다면 모델의 성능이 유지될 수 있는지 보기 위해 난독화 옵션 전체(ALL)가 적용된 바이너리 코퍼스를 학습 코퍼스에 추가하여 BinShot-Obfuscation을 제작했다. 그 결과, [표 4]의 BinShot-Obfuscation 결과에서 가리키듯 4가지 난독화 옵션 모두에서 성능이 향상됐으며 특히 BCF와 ALL

옵션에서 크게 개선되었다. 이는 BinShot 모델이 난독화된 바이너리도 학습한다면 일반화가 가능함을 의미한다. 특히 모든 난독화 옵션에서 F1 점수가 0.95 이상인데, 이를 통해 난독화된 바이너리도 학습한다면 BinShot은 충분히 실용적으로 사용될 수 있음을 확인했다.

방대한 양의 데이터에 대한 분류 모델을 이해하는 방법 중 하나는 시각화이다. 이를 통해 유사한 함수 쌍의 임베딩이 가까운 반면 유사하지 않은 함수 쌍의 임베딩은 충분히 떨어져 있는지 확인해야 한다. 그러나 고차원 공간(본 실험에서는 128차원)에서 벡터를 나타내는 것은 어렵다. 이에 본 연구에서는 유사한 데이터가 저차원 공간에서 서로 가깝게 위치하는, 비선형 차원 축소를 수행하는 t-분산 확률적 이웃 임베딩(t-SNE)[32]을 활용하였다. [그림 5]는 BinShot에서는 올바르게 분류(true positive)하였지만 기준 모델은 잘못 분류한 5개의 함수를 선정했다. 각 함수가 2개의 컴파일러와 4개의 최적화 레벨의 조합으로 컴파일 되었기 때문에 함수당 8개의 서로 다른 임베딩을 나타낸다. [그림 5]에서 BinShot의 경우는 유사한 함수들 간에 잘 군집화 되어있다. 그 중 unicodeOpen(빨간색 O 표시) 함수에는 두 개의 서로 다른 그룹이 있다. 하나는 gcc에서, 다른 하나는 clang에서 컴파일된 그룹이다.

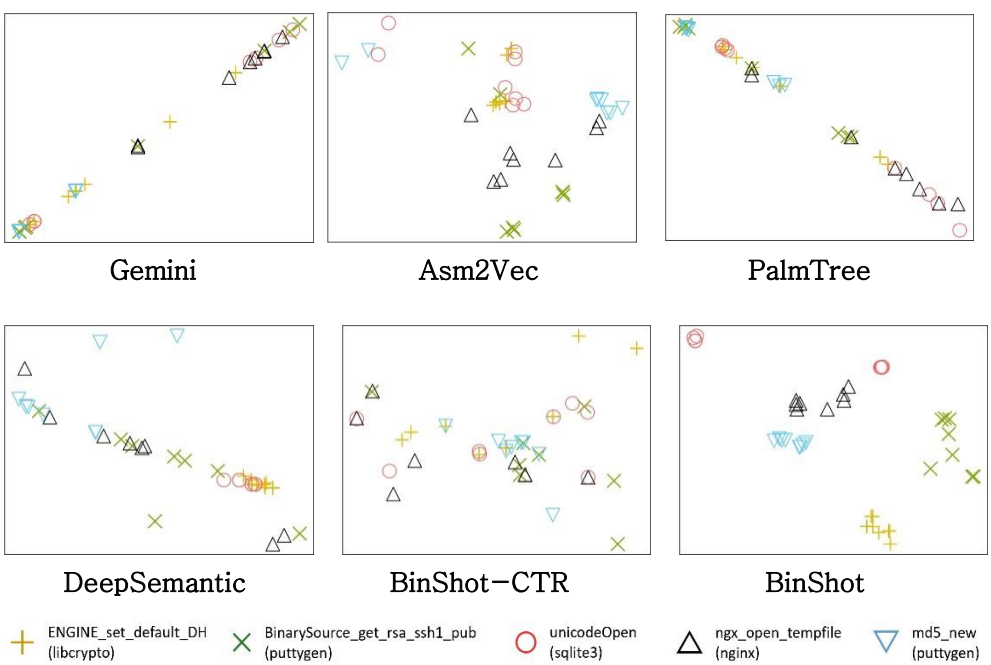

그림 5. 임베딩 벡터의 시각화 결과

제 5 장 결론

본 논문에서는 바이너리 코드 유사도 검출을 위해 BERT 기반의 샴 아키텍처로 weighted distance vector를 학습하는 BinShot을 제안한다. BERT를 채택하여 어셈블리 언어의 의미를 학습하고 이진 교차 엔트로피를 손실 함수로 활용하였다. 기존의 state-of-the-art 모델을 능가하는 성능을 보여주는 본 논문의 실험 결과는 BinShot이 효율성과 실용성을 입증했음을 보여준다.

참고 문헌

[1] Duan, Yue, Li, Xuezixiang, Wang, Jinghan, and Yin, Heng. DeepBinDiff: Learning Program-Wide Code Representations for Binary Diffing. Network and Distributed System Security Symposium, 2020.

[2] Zuo, Fei, et al. "Neural machine translation inspired binary code similarity comparison beyond function pairs." arXiv preprint arXiv:1808.04706 (2018).

[3] Koo, Hyungjoon, et al. "Semantic-aware Binary Code Representation with BERT." arXiv preprint arXiv:2106.05478 (2021).

[4] Xu, Xiaojun, et al. "Neural network-based graph embedding for cross-platform binary code similarity detection." Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. 2017.

[5] Li, Xuezixiang, Yu Qu, and Heng Yin. "Palmtree: learning an assembly language model for instruction embedding." Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security. 2021.

[6] Ding, Steven HH, Benjamin CM Fung, and Philippe Charland. "Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization." 2019 IEEE Symposium on Security and Privacy (SP). IEEE, 2019.

[7] Mikolov, Tomas, et al. "Distributed representations of words and phrases and their compositionality." Advances in neural information processing systems 26 (2013)

[8] Devlin, Jacob, et al. "Bert: Pre-training of deep bidirectional transformers for language understanding." arXiv preprint arXiv:1810.04805 (2018).

[9] Vaswani, Ashish, et al. "Attention is all you need." Advances in neural information processing systems 30 (2017).

[10] Schuster, Mike, and Kuldip K. Paliwal. "Bidirectional recurrent neural networks." IEEE transactions on Signal Processing 45.11 (1997): 2673-2681.

[11] Hochreiter, Sepp, and Jürgen Schmidhuber. "Long short-term

memory." Neural computation 9.8 (1997): 1735-1780.
[12] Chung, Junyoung, et al. "Empirical evaluation of gated recurrent neural networks on sequence modeling." arXiv preprint arXiv:1412.3555 (2014).
[13] Liu, Yinhan, et al. "Roberta: A robustly optimized bert pretraining approach." arXiv preprint arXiv:1907.11692 (2019).
[14] Feng, Zhangyin, et al. "Codebert: A pre-trained model for programming and natural languages." arXiv preprint arXiv:2002.08155 (2020).
[15] Brandes, Nadav, et al. "ProteinBERT: A universal deep-learning model of protein sequence and function." Bioinformatics 38.8 (2022): 2102-2110.
[16] Wang, Yaqing, et al. "Generalizing from a few examples: A survey on few-shot learning." ACM computing surveys (csur) 53.3 (2020): 1-34.
[17] Koch, Gregory, Richard Zemel, and Ruslan Salakhutdinov. "Siamese neural networks for one-shot image recognition." ICML deep learning workshop. Vol. 2. 2015.
[18] Xian, Yongqin, et al. "Zero-shot learning-a comprehensive evaluation of the good, the bad and the ugly." IEEE transactions on pattern analysis and machine intelligence 41.9 (2018): 2251-2265.
[19] BusyBox. 2022. The Swiss Army Knife of Embedded Linux. https://busybox.net.
[20] GNU. 2022. Libbgmp: The GNU Multiple Precision Arithmetic Library. https://gmplib.org.
[21] ImageMagick. 2022. ImageMagick. https://imagemagick.org.
[22] Curl. 2022. libcurl - the multiprotocol file transfer library. https://curl.se/libcurl.
[23] LibTom. 2022. LibTomCrypt. https://www.libtom.net/LibTomCrypt.
[24] OpenSSL. 2022. Cryptography and SSL/TLS Toolkit. https://www.openssl.org.
[25] SQLite. 2022. SQLite. https://www.sqlite.org.
[26] Zlib. 2022. A Massively Spiffy Yet Delicately Unobtrusive Compression Library. https://zlib.net.
[27] PuTTygen. 2022. Download PuTTYgen - Putty key generator.

https://www.puttygen.com.
[28] Nginx. 2020. High Performance Load-balancer and Web Server. https://nginx.com.
[29] vsftpd. 2022. Probably the Most Secure and Fastest FTP server. https://security.appspot.com/vsftpd.html.
[30] Kim, Dongkwan, et al. "Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned." IEEE Transactions on Software Engineering (2022).
[31] Hex-Rays. 2022. IDA Pro Disassembler. https://www.hex-rays.com/products/ida/.
[32] Van der Maaten, Laurens, and Geoffrey Hinton. "Visualizing data using t-SNE." Journal of machine learning research 9.11 (2008).
[33] Dullien, Thomas, and Rolf Rolles. "Graph-based comparison of executable objects (english version)." Sstic 5.1 (2005): 3.
[34] Bourquin, Martial, Andy King, and Edward Robbins. "Binslayer: accurate comparison of binary executables." Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop. 2013.
[35] Eschweiler, Sebastian, Khaled Yakdan, and Elmar Gerhards-Padilla. "discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code." Ndss. Vol. 52. 2016.
[36] Chandramohan, Mahinthan, et al. "Bingo: Cross-architecture cross-OS binary search." Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. 2016.
[37] Ming, Jiang, Meng Pan, and Debin Gao. "iBinHunt: Binary hunting with inter-procedural control flow." International Conference on Information Security and Cryptology. Springer, Berlin, Heidelberg, 2012.
[38] Junod, Pascal, et al. "Obfuscator-LLVM--software protection for the masses." 2015 IEEE/ACM 1st International Workshop on Software Protection. IEEE, 2015.

Abstract

Vulnerability Detection using Binary Similarity based on Natural Language Processing

Seong-gwan Ahn

Department of Electrical and Computer Engineering

The Graduate School

Seoul National University

Recently, off-the-shelf applications and system software are being distributed in the form of executable binaries by reusing source codes. Reusing existing code increases productivity, but it introduces the unexpected risk of reproducing vulnerable code. This makes detecting vulnerabilities in binaries more important than ever. In this study, Binary Code Similarity Detection (BCSD) is used to detect binary vulnerabilities. BCSD is to determine whether snippets of binary code such as assembly functions are similar when the source code is not accessible. This can be applied to various cases such as code duplication detection and vulnerability detection. In particular, it is possible to analyze whether the corresponding binary has a vulnerability by comparing it with a binary having a vulnerability.

However, semantic analysis of binary is more difficult than source code. Due to this point, recent BCSD studies are trying to combine AI for binary semantic analysis and generalization in binary similarity research. In this study, we propose BinShot, a model with a structure that learns the similarity of BERT-based binary codes. The BinShot model uses weighted distance vectors for Siamese networks and, unlike previous models, adopts binary cross entropy as a loss function. In this paper, we demonstrate the superiority of

BinShot through comparison with existing state-of-the-art studies in terms of effectiveness, transferability, and practicality.

Keywords : Binary Analysis, Vulnerability Detection, Natural Language Processing, Few-shot Learning
Student Number : 2021-20916