바이너리 코드 분석을 통한 함수 유사도 분석 도구

Function similarity analysis tools through a binary code analysis

저자
(Authors)

양종환, 이석수, 정우식, 조은선
Jonghwan Yang, Seoksu Lee, Woosik Jung, Eun-sun cho

출처
(Source)

한국정보과학회 학술발표논문집 , 2016.6, 1860-1862 (3 pages)

발행처
(Publisher)

한국정보과학회
KOREA INFORMATION SCIENCE SOCIETY

URL

http://www.dbpia.co.kr/Article/NODE07018007

APA Style

양종환, 이석수, 정우식, 조은선 (2016). 바이너리 코드 분석을 통한 함수 유사도 분석 도구. 한국정보과학회 학술발표논문집, 1860-1862.

바이너리 코드 분석을 통한 함수 유사도 분석 도구

양종환 이석수 정우식O 조은선
충남대학교 컴퓨터 공학과
jhy7185@naver.com sogsu@naver.com jws940814@naver.com eschough@cnu.ac.kr

Function similarity analysis tools through a binary code analysis

Jonghwan Yang, Seoksu Lee, Woosik JungO Eun-sun cho
Dept. of Computer Science & Engineering. Chungnam National University

요 약

우리가 사용하는 프로그램에는 많은 취약점들이 존재한다. 이 취약점들은 해커의 공격에 이용될 수 있기 때문에, 취약한 함수를 보완하는 것이 중요하다. 본 문서는 오픈소스 기반 프로그램의 함수 유사도를 분석하기 위한 다양한 접근 방법을 제시하고, 그것을 토대로 연구한 내용을 소개한다.

1. 서 론

바이너리 코드 분석은 개발에 사용된 소스코드를 필요로 하지 않고 프로그램을 분석을 가능하게 함으로써 코드 도용이나 취약한 코드 검출 등 여러 목적으로 수행 되고 있다.

바이너리 코드 분석을 통해 함수의 유사도를 측정하여 라이선스를 위반하는 코드 도용을 검출하거나 취약한 코드 검출에 도움을 주기도 한다. 함수의 유사도란 기준이 되는 함수와 비교 대상이 되는 함수의 비슷한 정도를 말한다.

유사도를 측정하기 위해 바이너리 코드를 함수별로 나누고 함수로부터 추출된 특징 정보를 기반으로 유사도를 측정하거나[1], 동적으로 스택 사용 패턴을 분석[2]한 결과를 기반으로 유사도를 분석하는 연구가 진행되고 있다.

여러 가지 유사도 측정 방법이 존재하고 있지만 결과에 대한 신뢰도가 검증되지 않아 실제 상황에서 사용하기는 쉽지 않다.

본 논문에서는 컴파일러 버전과 최적화 옵션별로 생성되는 프로그램을 분석하여 추출할 수 있는 공통점을 기반으로 함수의 유사도를 측정하는 기법을 제안한다.

2. 바이너리 코드 비교 방법 및 대상

이 장에서는 바이너리 코드 차이점 분석을 위해 우리가 제안한 방법과 대상을 소개한다. 크게 2가지 방법으로 바이너리 코드를 비교하였는데, 첫 번째 방법은 PIN Tool[3]을 이용한 오염도 분석을 통한 방법이고 두 번째 방법은 OpenREIL[4]을 이용한 REIL[5]이라는 중간언어의 단순 비교이다. 오염도 분석은 프로그램이 입력 받는 외부 데이터가 어느 정도까지 영향을 주는 지를 분석하는 방법이다. 일반적으로 컴파일 하는 과정에서 바이너리 코드의 차이가 발생하기 때문에 컴파일러의 버전별, 최적화 옵션별로 연구를 진행했다.

2.1 컴파일러 버전별

컴파일러 버전별로 바이너리 코드를 비교하기 위해서 OpenREIL[4]을 이용해서 바이너리 코드를 REIL[5]이라는 중간언어와 어셈블리어로 출력하여 비교, 바이너리 코드의 차이를 알아보았다. 컴파일러는(GCC)[6]를 이용하였고 해당 컴파일러의 여러 버전 중에 4.4.7버전과 4.6.4버전 그리고 4.8.4버전을 이용하여 바이너리 코드를 비교했다.

2.2 컴파일러 최적화 옵션별

컴파일러 옵션별로 바이너리 코드를 비교한다. 바이너리 코드의 컴파일은 GCC[6] 4.8.4 버전을 사용했고, 컴파일 할 때 최적화 옵션에 차이를 두어 컴파일을 진행했다. 비교하는 과정에서 추가적으로 PIN Tool[3]을 이용한 오염도 분석 방법을 적용하여서 메모리 단에서도 비교를 진행했다.

3. 구 현

앞서 제안한 내용의 타당성을 입증하기 위해 간단한 코드를 구현하여 실험했다.

3.1 컴파일러 버전별

컴파일러 버전별로 바이너리 코드를 비교하기 위해 두 수를 입력하여 계산하고 그 값을 출력하는 [그림1]에서 보이는 간단한 코드를 만들고, 이것을 GCC[6] 컴파일러를 이용해 컴파일 했다.

```
1  #include <stdio.h>
2
3  int cal(int a, int b)
4  {
5      return (a + b) * b;
6  }
7
8  int main(void)
9  {
10     int a = 3;
11     int b = 4;
12     int c = 0;
13
14     c = cal(a,b);
15
16     printf("result = %d\n", c);
17     return 0;
18 }
```

그림 1 테스트 C 코드

컴파일 할 때 버전은 4.4.7, 4.6.4, 4.8.4 버전을 사용하였다. 컴파일한 바이너리 코드를 각각 OpenREIL[4]을 이용해 가상 메모리 주소를 가지고 가상화하여 바이너리 코드와 REIL[5]코드로 출력하는 [그림2]와 같은 파이선 코드를 이용하여 바이너리 코드와 REIL[5]코드를 출력했다.

```
openreil 오픈소스 import

VA_address = 0x0000000000400000
#가상화 머신에 사용될 주소 설정

CodeStorageTranslator(Reader('file_name'))
#실제 바이너리 코드 리더를 통한 코드 변환 부분

DFGraphBuiler().Traverse(VA_address)
#데이터 플로우 그래프 생성(설정한 주소 지점을 기반)

print get_func(VA_address)
#결과 출력
```

그림 2 REIL[5] 변환 파이썬 수도 코드

3.2 컴파일러 옵션별

바이너리 코드를 비교하기 위해 두 수를 입력하면 큰 값을 출력해주는 [그림3]과 같은 간단한 코드를 만들고, 이것을 gcc[6] 컴파일러를 이용해 컴파일 했다. 컴파일하는 과정에서 최적화 옵션을 다르게 해서 3개의 테스트 파일을 만들었다. 최적화 옵션은 기본과 -O2, -O3 옵션을 사용했다. 생성된 바이너리 코드는 GDB[7]를 이용해 역어셈블한 결과로 비교했다. 또 PIN Tool[3]을 이용해서 오염도 분석을 하였다. [그림4]에서 보이는 오염도분석을 하는 코드를 작성 이용하였다. GDB[7]를 이용해 역어셈블 했을 때에는 -O2, -O3 옵션의 결과가 같았고, 옵션을 주지 않았을 때와 결과가 달랐다. 대신에 오염도 분석을 했을 때에는 세 경우 모두 결과가 같았다.

```
#include <stdio.h>

#define max(x,y) ((x)>(y)?(x):(y))

int Max(int a, int b){
    return max(a,b);
}

int main(){
    int a,b;
    scanf("%d %d",&a,&b);
    printf("MAX : %d",Max(a,b));
    return 0;
}
```

그림 3 두 수를 비교하는 간단한 C 코드

```
VOID Instruction(INS ins, VOID *v)
{
    if (INS_OperandCount(ins) > 1 && INS_MemoryOperandIsRead(ins, 0) && INS_OperandIsReg(ins, 0))
    {
        readMem(UINT64 insAddr, std::string insDis, UINT32 opCount, REG reg_r, UINT64 memOp)
        // 메모리가 읽혀지는 경우
    }
    else if (INS_OperandCount(ins) > 1 && INS_MemoryOperandIsWritten(ins, 0))
    {
        writeMem(UINT64 insAddr, std::string insDis, UINT32 opCount, REG reg_r, UINT64 memOp)
        // 메모리가 쓰여지는 경우
    }
    else if (INS_OperandCount(ins) > 1 && INS_OperandIsReg(ins, 0))
    {
        spreaedRegTaint(UINT64 insAddr, std::string insDis, UINT32 opCount, REG reg_r, UINT64 memOp)
        // 위에 메모리에 읽고쓰는것을 레지스터 까지 확장
    }

    if (INS_OperandCount(ins) > 1 && INS_OperandIsReg(ins, 0))
    {
        followData(UINT64 insAddr, std::string insDis, UINT32 opCount, REG reg_r, UINT64 memOp)
        // 오염된 레지스터의 있는 데이터의 이동을 따라감
    }
}
int main(int argc, char *argv[])
{
    if(PIN_Init(argc, argv))
        return Usage();

    INS_AddInstrumentFunction(Instruction, 0);
    //명령어 사이사이에 Instruction 함수 실행
    PIN_StartProgram();

    return 0;
}
```

그림 4 오염도 분석을 위한 PIN Tool[3]을 활용한 C 코드

4. 결과

위의 3장에 각 항목마다의 구현을 통한 결과는 다음과 같다.

4.1 컴파일러 버전별

컴파일러 버전별로 바이너리 코드를 비교한 결과, 4.4.7버전과 4.6.4버전에서는 주소 값의 차이만 있을 뿐 명령어 단에서의 차이는 발생하지 않았다. [그림5]에서 보이듯이 알 수 있다.

```
; sub_0040052b()
; Stack args size: 0x0
; Code chunks: 0x40052b-0x400577
; ------------------------------
;
; asm: push ebp
; data (1): 55
;
    STR         R_EBP:32,                ,          V_00:32
    STR         R_ESP:32,                ,          V_01:32
    SUB          V_01:32,           4:32,          V_02:32
    STR          V_02:32,                ,        R_ESP:32
    STM          V_00:32,                ,          V_02:32
;
; asm: dec eax
; data (1): 48
;
```

그림 5 4.4.7버전 컴파일 코드 일 부분

```
; sub_0040052a()
; Stack args size: 0x0
; Code chunks: 0x40052a-0x400576
; ------------------------------
;
; asm: push ebp
; data (1): 55
;
    STR         R_EBP:32,                ,          V_00:32
    STR         R_ESP:32,                ,          V_01:32
    SUB          V_01:32,           4:32,          V_02:32
    STR          V_02:32,                ,        R_ESP:32
    STM          V_00:32,                ,          V_02:32
;
; asm: dec eax
; data (1): 48
;
```

그림 6 4.6.4버전 컴파일 결과 일 부분

그러나 4.8.4버전에서는 이전의 버전과의 차이점이 발생하는데 우리가 사용한 코드를 예로 들어서 설명하면 함수를 부르는 부분의 EDI레지스터를 이용하는 부분에서 이전의 버전은 EAX레지스터를 활용하여서 이동하였지만 4.8.4버전에서는 EDI레지스터를 직접 이용하는 것을 [그림6]과 같이 확인 할 수 있었다. 이를 통해서 버전별로 레지스터를 활용하는 부분이 다르다는 것을 알 수 있다.

```
;
; asm: mov esi, edx
; data (2): 89 d6
;
    STR         R_EDX:32,                ,          V_00:32
    STR          V_00:32,                ,        R_ESI:32
;
; asm: dec eax
; data (1): 48
;
    STR         R_EAX:32,                ,          V_00:32
    SUB          V_00:32,           1:32,          V_01:32
     OR           R_CF:1,            0:1,          V_02:32
    SHR          V_02:32,           0:32,          V_03:32
    AND          V_03:32,           1:32,          V_05:32
     OR          V_05:32,           0:32,           V_04:1
```

그림 7 4.6.4버전 컴파일 버전 일 부분

```
;
; asm: mov esi, eax
; data (2): 89 c6
;
    STR         R_EAX:32,                ,          V_00:32
    STR          V_00:32,                ,        R_ESI:32
;
; asm: mov edi, 0x400614
; data (5): bf 14 06 40 00
;
    STR         400614:32,               ,        R_EDI:32
;
; asm: mov eax, 0
; data (5): b8 00 00 00 00
;
```

그림 8 4.8.4버전 일 부분

4.2 컴파일러 최적화 옵션별

컴파일러 최적화 옵션별에 따른 결과는 아래와 같다. 아래의 그림을 통해 알아보면 최적화 옵션에 따른 바이너리 코드 비교 결과는 -O2옵션과 -O3옵션의 바이너리 코드 결과는 같았지만 아무것도 주지 않은 옵션과는 다르게 나옴을 [그림9]에서 보이다 시피 확인 할 수 있다.

```
<main>:
push   %rbp
mov    %rsp,%rbp
sub    $0x10,%rsp
lea    -0x4(%rbp),%rdx
lea    -0x8(%rbp),%rax
mov    %rax,%rsi
mov    $0x400684,%edi
mov    $0x0,%eax
callq  4004a0 <__isoc99_scanf@plt>
mov    -0x4(%rbp),%edx
mov    -0x8(%rbp),%eax
mov    %edx,%esi
mov    %eax,%edi
callq  40059d <Max>
mov    %eax,%esi
mov    $0x40068a,%edi
mov    $0x0,%eax
callq  400470 <printf@plt>
mov    $0x0,%eax
leaveq
retq
nopl   0x0(%rax)
```

```
<main>:
sub    $0x18,%rsp
mov    $0x400694,%edi
xor    %eax,%eax
lea    0xc(%rsp),%rdx
lea    0x8(%rsp),%rsi
callq  4004c0 <__isoc99_scanf@plt>
mov    0xc(%rsp),%edx
cmp    %edx,0x8(%rsp)
mov    $0x40069a,%esi
cmovge 0x8(%rsp),%edx
mov    $0x1,%edi
xor    %eax,%eax
callq  4004b0 <__printf_chk@plt>
xor    %eax,%eax
add    $0x18,%rsp
retq
```

그림 9 어셈블리어(왼쪽 기본옵션, 오른쪽 -O2옵션)

그리고 PIN Tool[3]을 이용한 오염도 분석방법으로 비교한 결과 중 일부를 [표3],[표4]에서 볼 수 있는데 주소값의 차이만 있을 뿐 옵션에 따라서는 차이가 없음을 확인 할 수 있다.

[TAINT]
bytes tainted from ***0x7f8b6c569000*** to ***0x7f8b6c569400***(via read)
[READ in 7f8b6c569000] 7f8b6c6996c5: movzx eax, byte ptr[rax]
eax is now tainted
[FOLLOW] 7f8b6c69a62e: cmp eax, 0xffffffff
[READ in 7f8b6c569000] 7f8b6c69a63f: movzx eax, byte ptr [rax]
eax is already tainted
[FOLLOW] 7f8b6c677892: cmp eax, 0xffffffff
[SPREAD] 7f8b6c673bea: movsxd rdi, eax
output: rdi | input: eax
rdi is now tainted

표 2 기본옵션 컴파일 오염결과

[TAINT]
bytes tainted from ***0x7fcfb6db8000*** to ***0x7fcfb6db8400***(via read)
[READ in 7fcfb6db8000] 7fcfb6ee86c5: movzx eax, byte ptr [rax]
eax is now tainted
[FOLLOW] 7fcfb6ee962e: cmp eax, 0xffffffff
[READ in 7fcfb6db8000] 7fcfb6ee963f: movzx eax, byte ptr [rax]
eax is already tainted
[FOLLOW] 7fcfb6ec6892: cmp eax, 0xffffffff
[SPREAD] 7fcfb6ec2bea: movsxd rdi, eax
output: rdi | input: eax
rdi is now tainted

표 3 -O2옵션 컴파일 오염결과

5. 결론 및 향후 연구 방향

결론적으로 컴파일러 버전별 바이너리 코드 분석에서는 4.8.4 버전부터 변화가 생김을 알 수 있었다. 최적화 방법에 따라서 비교를 할 경우에 최적화 방법에 따라 바이너리 코드가 달라질 수 있지만, 오염도 분석을 했을 때에는 같은 결과를 가진다는 결론을 얻었다. 이 부분을 통해서 함수의 유사도를 파악하는 과정에서 컴파일러의 옵션과 버전에 따른 차이점을 고려하여 함수를 분석하는 과정에서의 접근 방법이 다름을 알 수 있다.

향후에는 오픈소스 버전별로도 차이점을 연구할 계획이다. 현재 오픈소스 별로 함수를 비교하기 위해서 시중에서 사용되고 있는 오픈소스 중 OpenSSL[8]를 중점으로 잡았다. OpenSSL[8] 함수를 몇 개 선정해서 OpenSSL[8] 버전별로 어떤 차이를 갖게 되는지 연구할 계획이다. 우리가 선정한 OpenSSL[8] 함수들은 아래 표와 같다.

OpenSSL 함수	함수 용도
SSL_CTX_use_certificate(SSL_CTX *, X509 *)	서버 인증서 로딩
SSL_CTX_use_PrivateKey(SSL_CTX *ctx, EVP_PKEY *pkey);	개인키 로딩
int password_callback(char *buf, int size, int rwflag, void *userdata);	패스워드를 얻기 위한 callback

표 1 OpenSSL[8] 함수

우리는 몇 가지 OpenSSL[8] 버전을 기준으로 이 함수들이 어떻게 변했는지 리눅스 환경에서 그 차이점을 연구할 예정이다.

또, 다양한 테스트 케이스를 만들어서 연구한 결과에 예외나 오류는 없는지 더 확고하게 할 계획이다. 최종적으로 연구한 결과를 바탕으로 오픈소스를 이용한 프로그램의 함수 유사도 분석 도구를 제작하는데 사용할 것이다.

6. 참고 문헌

[1]박희완, 최석우, 서신애, 한태숙 ,"프로그램의 구조와 상수 값을 이용하는 바이너리 실행 파일의 차이점 분석", 정보과학회논문지: 소프트에어 및 응용 제35권 제7호, 2008.7
[2]박영성, 최용석, 최종무 "동적 스택 사용 패턴을 이용한 바이너리 유사도 분석", 2013 한국정보과학회 제40회 정기총회 및 추계 학술 발표회 , http://www.dbpia.co.kr/Article/NODE02323311,2013.11
[3]]Intel, "Pin tool", https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool
[4]OpenReil, https://github.com/Cr4sh/openreil#_3
[5]Thomas Dullien, Sebastian Porst ,"REIL:A platform-independent intermediate representation of disassembled code for static code analysis", CanSecWest Applied Security Conference, 2009
[6] GNU, "GCC", https://gcc.gnu.org/
[7] GNU, "GDB", https://www.gnu.org/software/gdb/
[8] OpenSSL, "OpenSSL", https://www.openssl.org/