BinSim: Trace-based Semantic Binary Diffing via System Call Sliced Segment Equivalnce Checking

USENIX Security Symposium 2017
Jiang Ming, Dongpeng Xu, Yufei Jiang, Dinghao Wu

김영철

2019. 3. 22.

Scope & Contributions

- is designed for *fine-grained* individual binary diffing analaysis

- present System Call Sliced Segment Equivalence Checking
  - relies on system or API calls
- can detect the similarities or differences across multiple basic blocks
  - overcomes the “block-centric” limitation
- can remove the redundant instructions
  - can produce more precise result

Motivation

- 난독화 된 프로그램 분석은 행동 정보를 사용하는 것이 일반적
  - 동적 분석만 수행하면 의미 분석이 정확하게 되지 않음.

(a)
```
int x, y; // x is an input
HANDLE out = CreateFile("a.txt", …);
y = x + x;
WriteFile(out, &y, sizeof y, …);
CloseHandle(out);
```

(b)
```
int x, y; // x is an input
HANDLE out = CreateFile ( "a.txt", … );
y = x << 1;
WriteFile ( out, &y, sizeof y, … );
CloseHandle ( out );
```

(c)
```
int x, y, z; // x is an input
HANDLE out = CreateFile("a.txt", …);
z = (x >> 31);
z = (x ^ z) - z; // z is the absolute value of x
y = 2 * z;
WriteFile(out, &y, sizeof y, …);
CloseHandle(out);
```

if x>=0, (a), (b), (c)는 모두 같지만
if x<0, (b)는 달라짐
➔ 동적 랜덤 테스팅을 하면 50%확률로 다름.
➔ 이러한 문제를 이용한 “query-then-infect” pattern attack [77]

Motivation

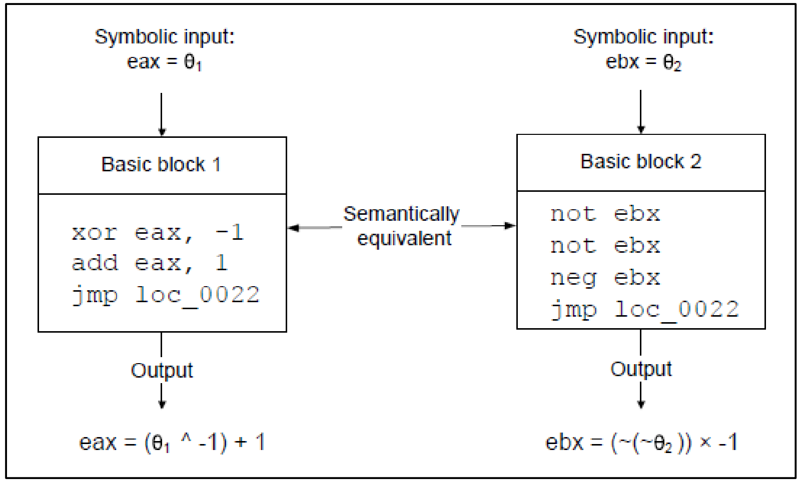
- symbolic execution (SE)을 이용한 의미 분석 방법
  - the core is matching semantically equivalent basic blocks
  - 단순한 구조에는 좋지만 스케일이 커지면 두 가지 문제 발생

    1) 바이너리 코드에서 function boundary 를 식별해야하는 문제 [5]

    2) 적당한 바이너리 크기만 되어도 성능이 매우 떨어짐 [46]

Motivation

- symbolic execution (SE)을 이용한 의미 분석 방법




```
void BitCount1(unsigned int n)
{
  unsigned int count = 0;
  for (count = 0; n; n >>= 1)
    count += n & 1 ;
  printf ("%d", count);
}
```

```
void BitCount2(unsigned int n)
{
  unsigned int count = 0;
  while (n != 0 ) {
      n = n & (n-1);
      count++;
    }
  printf ("%d", count);
}
```

```
void BitCount3(unsigned int n)
{
  n = (n & (0x55555555)) +
       ((n >> 1) & (0x55555555));
  n = (n & (0x33333333)) +
       ((n >> 2) & (0x33333333));
  n = (n & (0x0f0f0f0f)) +
       ((n >> 4) & (0x0f0f0f0f));
  n = (n & (0x00ff00ff)) +
       ((n >> 8) & (0x00ff00ff));
  n = (n & (0x0000ffff)) +
       ((n >> 16) & (0x0000ffff));
  printf ("%d", n);
}
```

(좌) single block 내의 syntax 는 다르지만 의미가 같은 경우를 탐지하기 좋음
(우) BitCount 의 3가지 implementation, 세 번째에는 loop 가 존재하지 않기 때문에 매칭되는 블록을 찾을 수 없음

Motivation

- symbolic execution (SE)을 이용한 의미 분석 방법
  - summarizing possible challenges that can defeat the block-centric binary diffing methods
    - (1) The lack of context information
    - (2) Compiler optimizations such as loop unrolling and function inline
    - (3) The chain of ROP gadgets
    - (4) Covert computation
    - (5) The same algorithm but with different implementations
    - (6) Control flow obfuscation schemes
      - ➔ can break up one basic block into multiple ones
    - (7) Virtualization obfuscation decode-dispatch loop

Motivation

- BinSim’s solution is hybrid
  - (1)~(5)를 자연스럽게 해결함.
  - (6), (7)을 해결하기 위해서는 추가적인 작업이 필요함.

Methodology

- BinSim's core method

  step1) 같은 환경에서 두 프로그램 실행

  ➔ 시스템 콜 시퀀스와 함께 트레이스 로깅

  step2) 매칭된 시스템 콜의 인자를 시작 지점으로 backward slicing, 각 slice를 따라 WP 계산

  ➔ WP는 인자 계산에 영향을 주는 Data/Control flow 정보를 가짐

  step3) 슬라이싱 된 세그먼트에서 암호화 기능을 식별

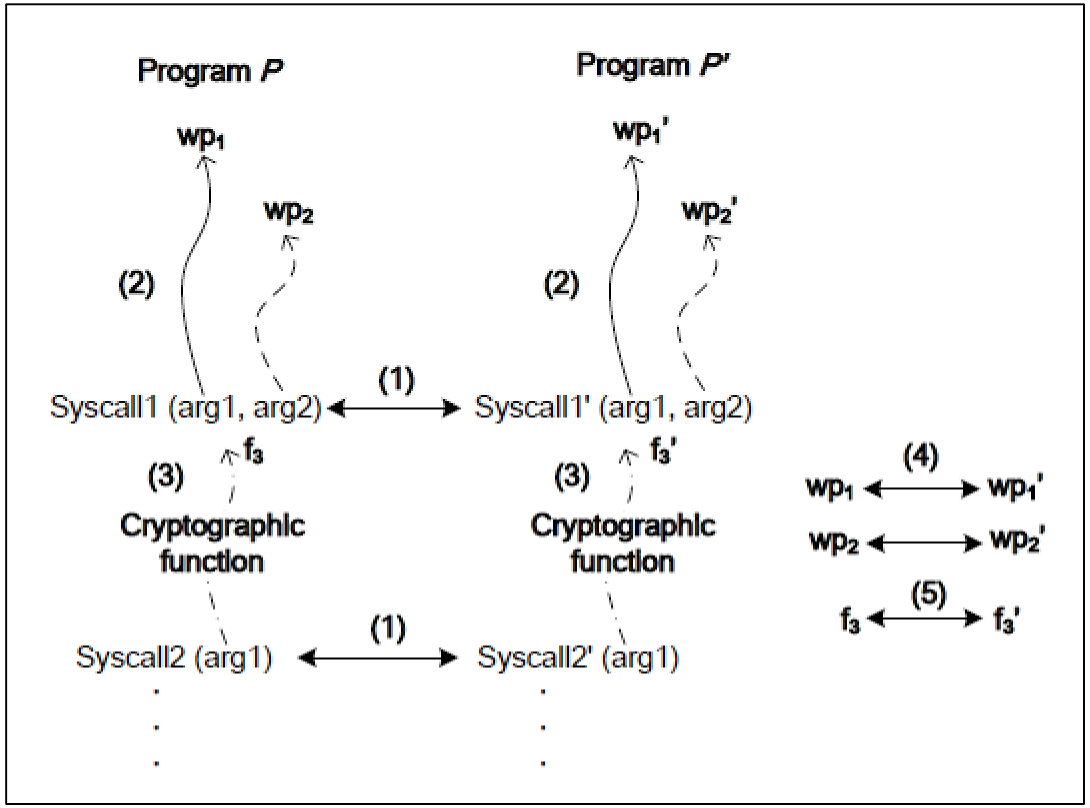
  ➔ 암호화 기능은 복잡한 symbolic form 을 가지기 때문

  step4) Solver 를 통해 두 WP가 같은지 확인

  ➔ 같은 방식으로 시스템 콜 쌍 비교

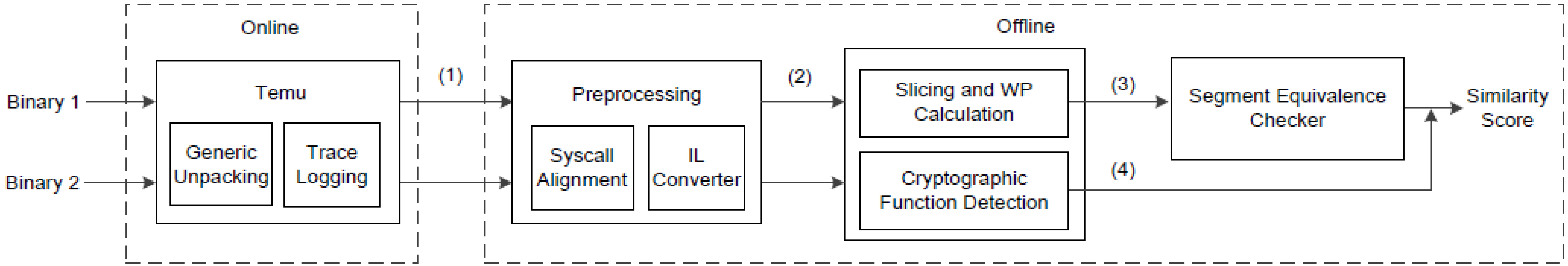
  ➔ 여기서 conditionally equivalent 하다는 것을 아는 듯..

  step5) Similarity score 계산



Architecture

- BinSim's architecture : two stages
  - Online stage : Temu (generic unpacking + trace logging)
  - Offline stage : Preprocessing + Slicing & WP calculation + Segment equivalence checker



On-demand Trace Logging

- The logged trace data
  (1) Instruction log (opcode, values of operands)
  (2) Memory log (memory access address to use binary slicing)
  (3) System calls invoked and their data flow dependencies

- Generic unpacking plug-in
  - Many malware samples exhibit the malicious behavior only after the real payload is unpacked
  - supports recording the execution trace that coms from real payload

On-demand Trace Logging

- Removing irrelevant system calls
  - Temu's multi-tag taint tracking can track data flow dependencies between system calls
    - three possible sources of a system call argument

      (1) the output of a previous system call

      (2) the initialized data section (e.g., .bss segment)

      ~~(3) the immediate argument of an instruction~~

On-demand Trace Logging

- Consider the parameter semantics
  - the fake dependency will be removed

- Another challenge is malware could invoke a different set of system calls to achieve the same effect
  - “replacement attacks” show above threat is feasible

Offline Analysis

- Preprocessing

  (1) First, BinSim lifts x86 instructions to Vine IL

  - SSA style of Vine IL is useful to track the use-def chain when performing backward slicing
  - is also side effect free
  - explicitly represents the setting of the eflags register bits

  (2) Aligns the two collected system call sequences to locate the matched system call pairs

  - [34]'s alignment algorithm is more precise than LCS algorithm

Offline Analysis

- Dynamic Slicing Binary Code
  - The purpose is to examine the aligned system calls to determine whether they are truly equivalent
  - Commencing at a tainted system call's argument
    - track a chain of instructions with data/control dependencies
  - Terminate when the source of slice criterion is one of the following conditions
    - the output the previous system call, a constant value, or the value read from the initialized data section

Offline Analysis

- Dynamic Slicing Binary Code
  - split data/control dependencies tracking into three steps

    1) index and value based slicing that only consider data flow

    2) tracking control dependencies

    3) remove the fake control dependencies caused by virtualization obfuscation code dispatcher

Offline Analysis

- Dynamic Slicing Binary Code
  - split data/control dependencies tracking into three steps

    1) index and value based slicing

       - use-def chain 을 따라 data dependencies 추적

       ex. mov edx [4*eax+4]

       (1) index based slicing

       eax 에 영향을 주는 instruction 을 추적

       ➔ indirect memory access is a valid index into a jump table / 이런 경우에만 사용

       (2) value based slicing

       [4*eax+4]에 영향을 주는 instruction 을 추적

       대부분의 경우 value based slicing 을 사용하는 것이 적합함

Offline Analysis

- Dynamic Slicing Binary Code
  - split data/control dependencies tracking into three steps

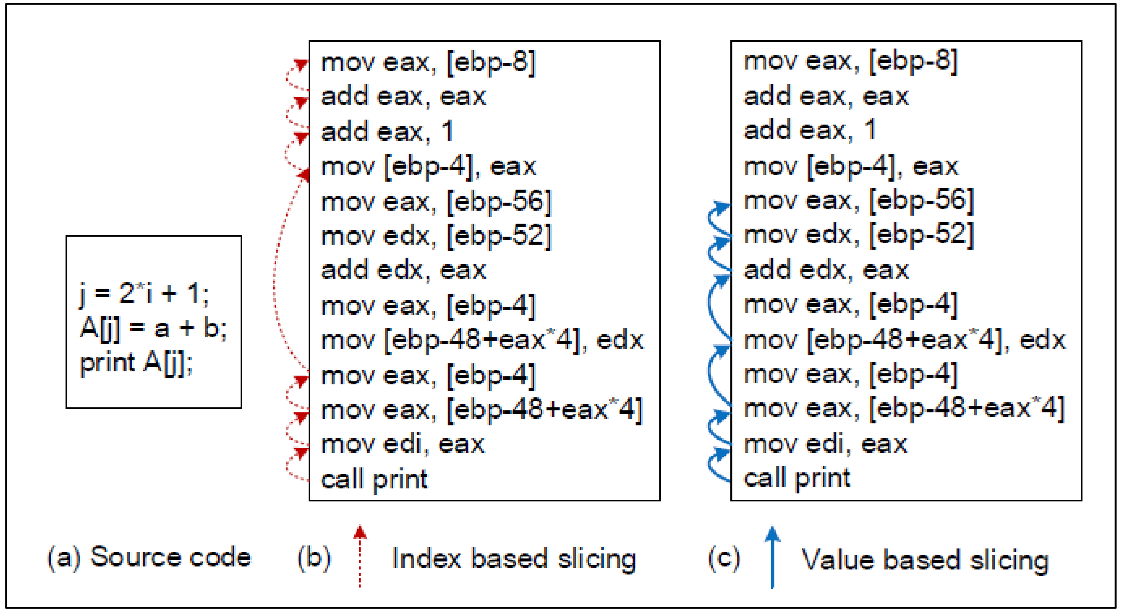
    1) index and value based slicing

index based slicing

➔ j = 2*i + 1 과 관련된 명령어 추적

value based slicing

➔ A[j] = a + b 와 관련된 명령어 추적



Offline Analysis

- Dynamic Slicing Binary Code
  - split data/control dependencies tracking into three steps

    2) tracking control dependency

    - conditional control transfer 는 eflag register (zf, cf, …) 에 영향을 받음.
    - add instructions related to the data flow of eflags bit value into the slice
    - also consider that the conditional logic is implemented without eflags

      ex. jecxz : jumps if register ecx is zero.

    - BinSim supports above all cases

Instructions	Meaning
CMOVcc	Conditional move
SETcc	Set operand to 1 on condition, or 0 otherwise
CMPXCHG	Compare and then swap
REP-prefixed	Repeated operations, the upper limit is stored in ecx
JECXZ	Jump if ecx register is 0
LOOP	Performs a loop operation using ecx as a counter

Offline Analysis

- Dynamic Slicing Binary Code
  - split data/control dependencies tracking into three steps

    3) dispatcher identification

      - virtualization obfuscation 에서 decode-dispatch loop iteration 의 특징 셋

        (1) it is a sequence of memory operation, ending with an indirect jump

        (2) it has an input register as virtual program counter (VPC) to fetch the next code

        ex. VMProtect takes esi as VPC, Code Virtualizer takes al register

        (3) it ends with an indirect jump which dispatches to a bytecode handler table

Offline Analysis

- Dynamic Slicing Binary Code
  - split data/control dependencies tracking into three steps

    3) dispatcher identification
      - identify possible decode-dispatch loop iteration in the backward slice
      - mark the input registers and output registers for each instruction sequence ending with an indirect jump
        - check there is an output register meets the two heuristics

          (1) b_i is tainted by the data located in ptr[a_j] (b : output, a : input)

          (2) the sequence ends with jmp ptr [b_i*(table stride) + table base]
      - remove the fake control dependencies caused by virtualization code dispatcher

Offline Analysis

- Handling Cryptographic Functions
  - cryptographic functions make SMT-based security analysis hard.

    because of the complicated input-output dependencies
  - BinSim's backward slicing step will be long, and equivalence checking will become hard to solve as well.

  - cryptographic function execution has almost no interaction with system calls except the ones are used for input/output

    ex. crypto ransomware take the original user's file as input, and then overwrite it.

Offline Analysis

- Handling Cryptographic Functions
  - BinSim does a “stitched symbolic execution” [11]

    1) First, make a forward pass over the sliced segments

    2) apply the advanced detection heuristics proposed by [27]

       e.g., too many bitwise operations, instruction chains, mnemonic const values

Offline Analysis

- Weakest Precondition Calculation
  - the result of slicing $S = [i_1, i_2, \ldots, i_n]$
  - $WP(S, P) = \{ wp(i_n, P) = P_{n-1}, wp(i_{n-1}, P_{n-1}) = P_{n-2}, \ldots,$
    $wp(i_1, P_1) = P_0 \} = P_0$

  ➔ WP(S, P) is the condition satisfying to reach the given point P

  ➔ $WP = F_1 \wedge F_2 \wedge \ldots \wedge F_n$ (이전 지점들의 조건을 모두 만족하는 것)

- Opaque predicates [17] can lead to a complicated WP
  - [45] can detect opaque predicate
  - remove the identified that

Opaque predicate

From Wikipedia, the free encyclopedia

In computer programming, an **opaque predicate** is a predicate—an expression that evaluates to either "true" or "false"—for which the outcome is known by the programmer *a priori*, but which, for a variety of reasons, still needs to be evaluated at run time. Opaque predicates have been used as watermarks, as it will be identifiable in a program's executable. They can also be used to prevent an overzealous optimizer from optimizing away a portion of a program. Another use is in obfuscating the control or dataflow of a program to make reverse engineering harder.

Offline Analysis

- Segment Equivalence Checking
  - identify whether two API calls are semantically equivalent

    ex. $wp_1 \equiv wp_2 \wedge arg_1 = arg_2$???

    check the equivalence of their arguments' wp

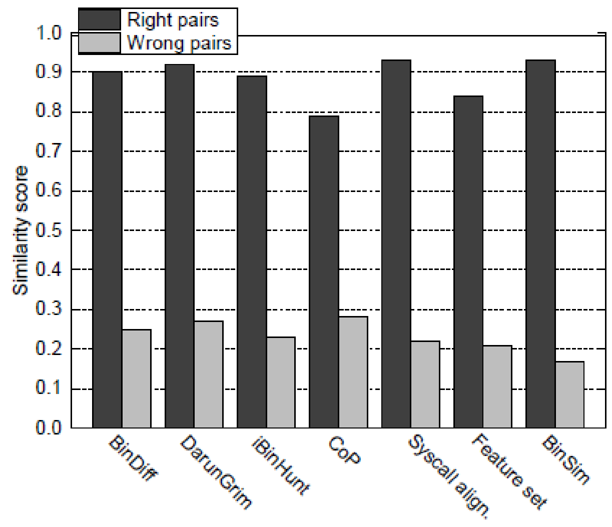
    validity checking for the above formula

$$\mathrm{Sim}(a,b) = \frac{\sum_{i=1}^{n} \textit{Similarity Score}}{\mathrm{Avg}\{|T_a|,|T_b|\}}$$

a, b : program

T_a, T_b : system call sequences collected a, b

n : the number of aligned system call



Discussion

- BinSim's limitations

  1) incomplete path coverage

  2) environment-sensitive malware which can detect sandbox environment

  3) generic unpacking can be defeated by more advanced packing methods

  4) slice size explosion

  5) customize an unknown cryptographic algorithm

E N D