26TH USENIX SECURITY SYMPOSIUM

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

Jiang Ming, Dongpeng Xu, Yufei Jiang, Dinghao Wu



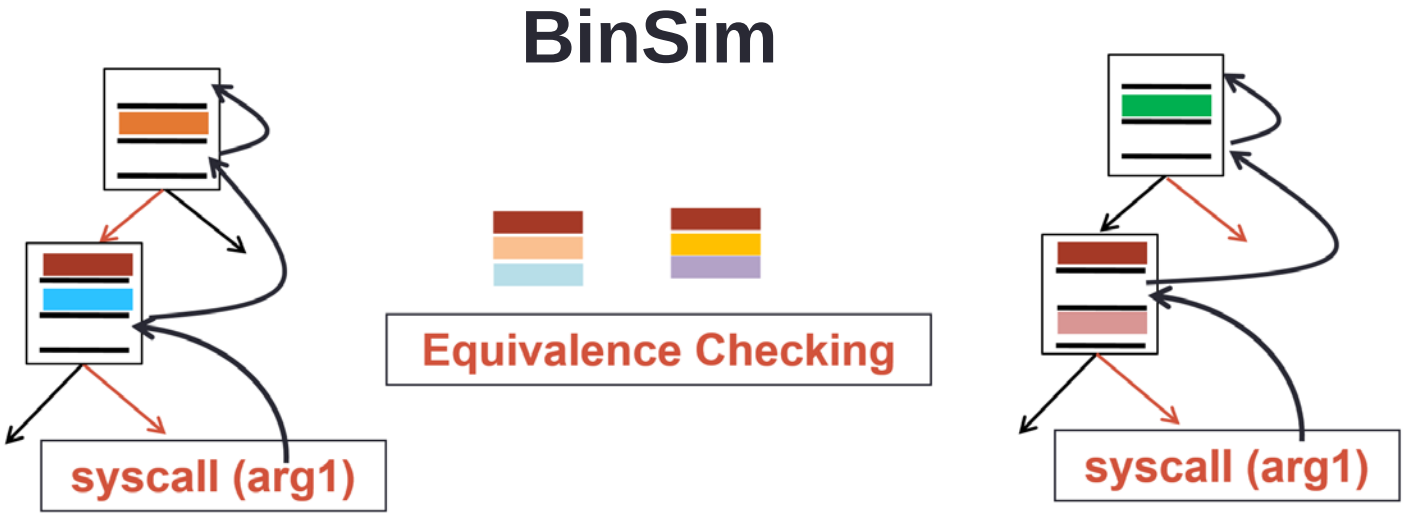
UNIVERSITY OF TEXAS ARLINGTON



PennState

This Talk is About

- A hybrid method to identify fine-grained relations between obfuscated binary code.
- Address "**block-centric**" limitation
- Detect "**conditionally equivalent**" behaviors



Security-relevant Software **Analysis** for ***Binary Code***



Commercial-Off-The-Shelf



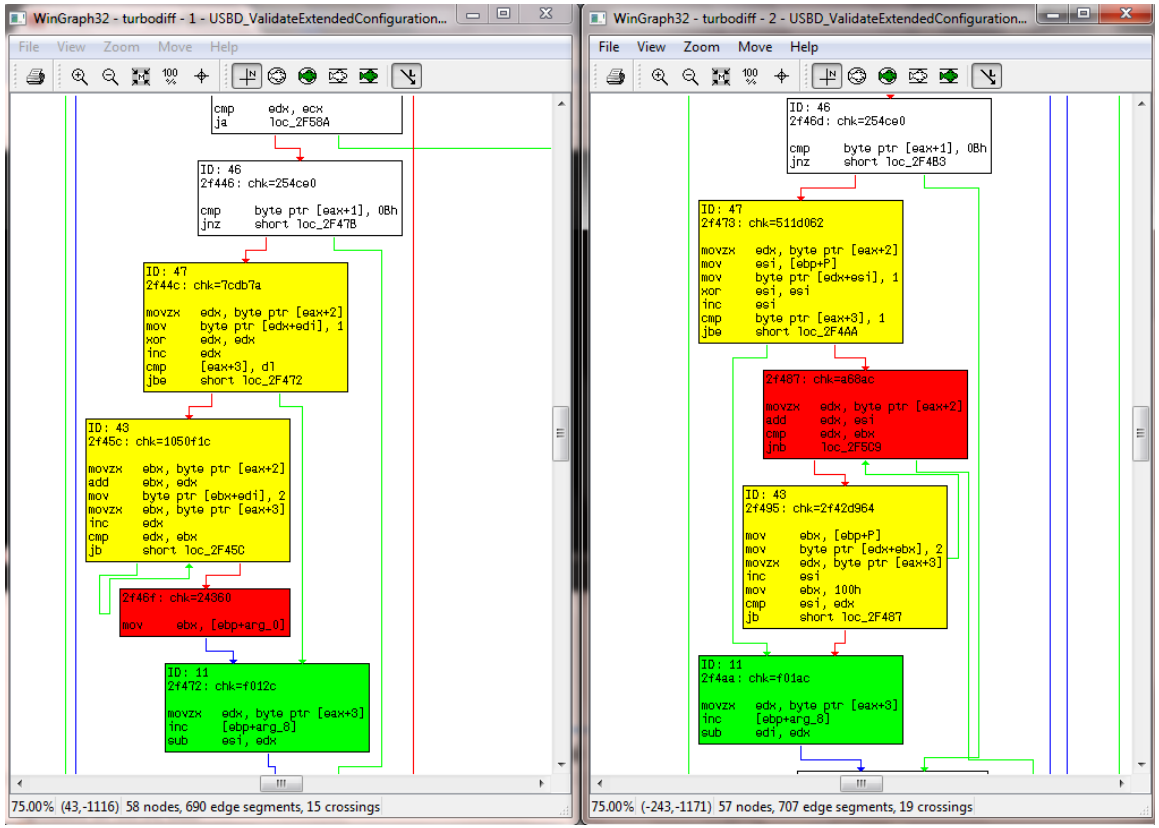


Binary Code



Binary Difference Analysis (Binary Diffing)

- MS12-005: Object Packager Patch Diffing



Security Applications of Binary Diffing

- **“1-day exploit”** generation

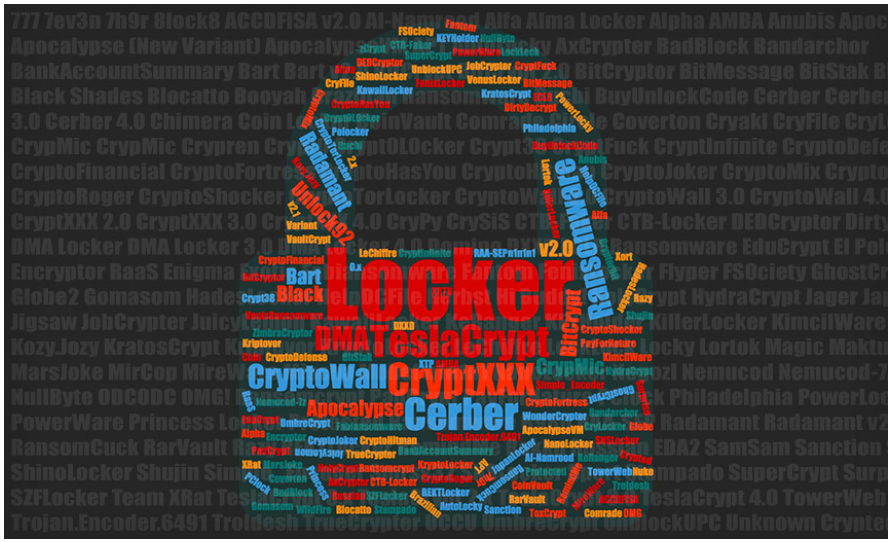


- Software plagiarism detection



- Malware analysis

Ransomware Family Count Surpasses 200



Source: http://www.bankinfosecurity.com/blogs/now-ransomware-goes-to-200-p-2287

Challenge: Code Obfuscation

- Packer, polymorphism, metamorphism, and code virtualization



Original binary



Suspicious binary

Compare Runtime Execution Behavior

- Measure the similarities of program behavior features
  - E.g., system call sequences and system call dependency graphs
  - Limitation: many binary code differences do not reflect on the behavior change; neglect *conditionally equivalent* behaviors

```
…
//  modify registry key
1: RegOpenKeyEx(…);
2: RegSetValueEx(…);
3: RegCloseKey (…);
```

(a) Zbot.a

```
…
//  modify registry key
1: GetLocalTime(&systime);
2:  if ( systime.Day < 20)
3:  {
4:      RegOpenKeyEx(…);
5:      RegSetValueEx(…);
6:      RegCloseKey (…);
7:  }
```

(b) Zbot.b

Conditional equivalent behaviors between Trojan-Spy.Win32.Zbot variants

Basic Block Semantics Modeling

- Represent the input-output relations as symbolic formulas
  - Constraint solver, random sampling, and hashing
- Resilient to moderate obfuscation within a basic block
  - Register swapping, instruction reordering/substitution, and junk code
- **"Block-centric" limitation**

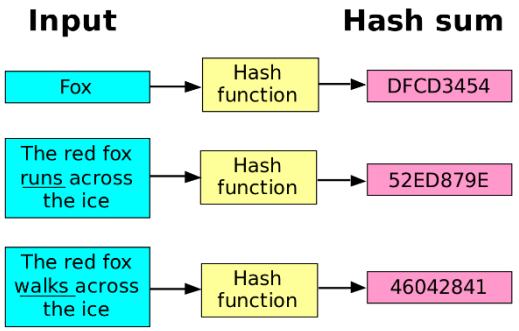
STP constraint solver
Simple Theorem Prover SMT solver

Microsoft Research
z3

Constraint Solver



Random sampling



Hashing

Binary block 1:

```
mov edx, ecx
mov ebx, 0x000A
add edx, ebx
mov ebx, ecx
sub ecx, eax
mov eax, ecx
dec eax
and ecx, 0
jmp 0x401922
```

Binary block 2:

```
lea eax, [ebx]
mov edx, 0x000A
nop
nop
add eax, edx
xchg eax, eax
not ecx
add ecx, ebx
lea edx, [ebx]
xor ebx, ebx
jmp 0x401A22
```



Are they semantically equivalent?

Symbolic inputs to basic block 1:
ecx_0 = symbol1;
eax_0 = symbol2;

```
mov edx, ecx
mov ebx, 0x000A
add edx, ebx
mov ebx, ecx
sub ecx, eax
mov eax, ecx
dec eax
and ecx, 0
jmp 0x401922
```

Outputs:
eax_2 = symbol1-symbol2-1;
ebx_1 = symbol 1;
ecx_2 = 0x0;
edx_1 = symbol1 + 0xA

Constraint solver

Symbolic inputs to basic block 2:
ebx_0 = symbol3;
ecx_0 = symbol4;

```
lea eax, [ebx]
mov edx, 0x000A
nop
nop
add eax, edx
xchg eax, eax
not ecx
add ecx, ebx
lea edx, [ebx]
xor ebx, ebx
jmp 0x401A22
```

Outputs:
ecx_2 = symbol3-symbol4-1;
edx_1 = symbol 3;
ebx_1 = 0x0;
eax_3 = symbol4 + 0xA

Block-Centric Limitation (1)

```
1: void BitCount1 (unsigned int n)
2: {
3:   unsigned int count = 0;
4:   for (count = 0; n; n >>= 1)
5:      count += n & 1 ;
6:   printf ("%d", count);
7: }
```

(a)

```
1: void BitCount2 (unsigned int n)
2: {
3:   unsigned int count = 0;
4:   while (n != 0 ) {
5:       n = n & (n-1);
6:       count++;
7:    }
8:   printf ("%d", count);
9: }
```

(b)

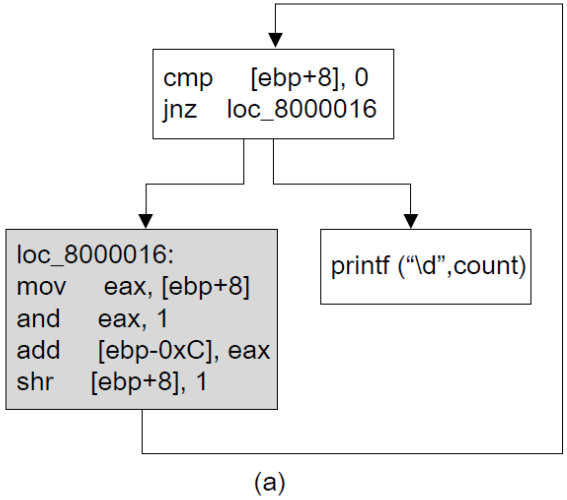
```
1: void BitCount3 (unsigned int n)
2: {
3:   n = (n & (0x55555555)) +
          ((n >> 1) & (0x55555555));
4:   n = (n & (0x33333333)) +
          ((n >> 2) & (0x33333333));
5:   n = (n & (0x0f0f0f0f)) +
          ((n >> 4) & (0x0f0f0f0f));
6:   n = (n & (0x00ff00ff)) +
          ((n >> 8) & (0x00ff00ff));
7:   n = (n & (0x0000ffff)) +
          ((n >> 16) & (0x0000ffff));
8:   printf ("%d", n);
9: }
```

(c)

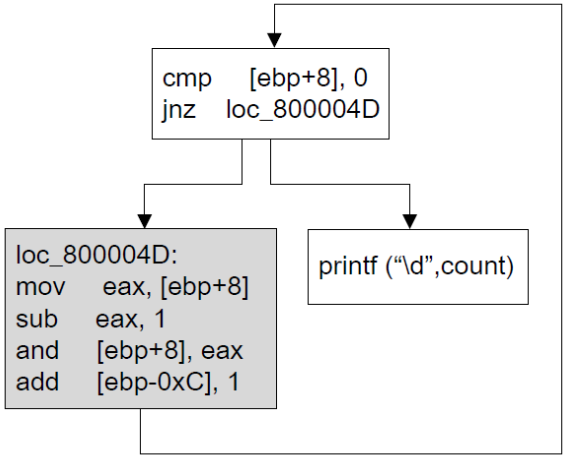
Figure 3: Semantic equivalence spreads across basic blocks.

Block-centric binary diffing methods fail to match these three cases

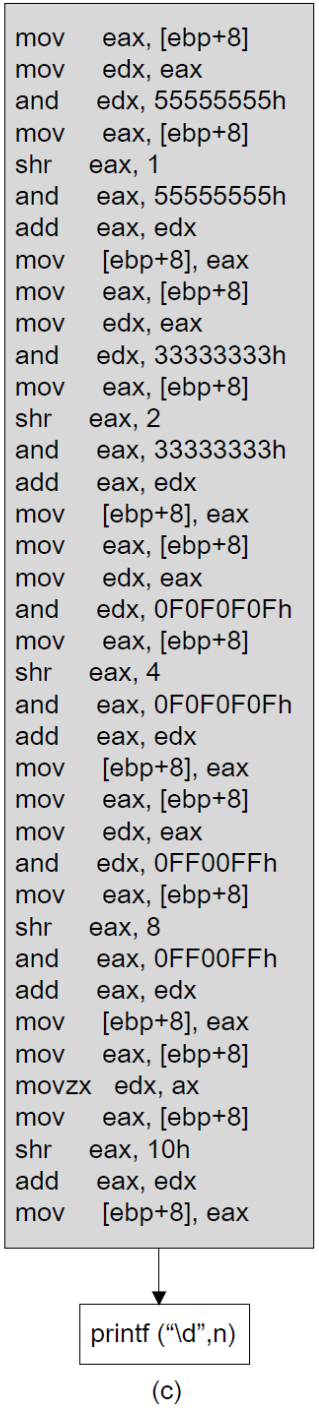




(a)



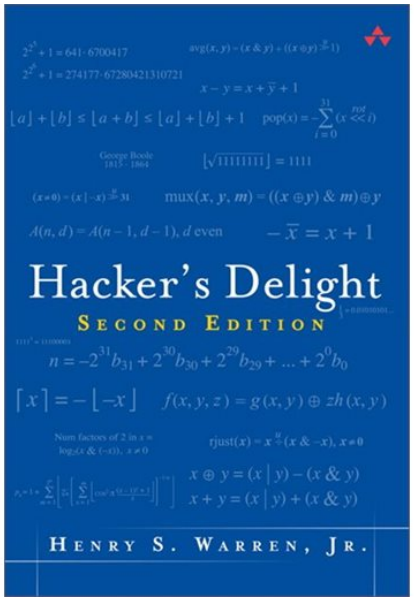
(b)



(c)

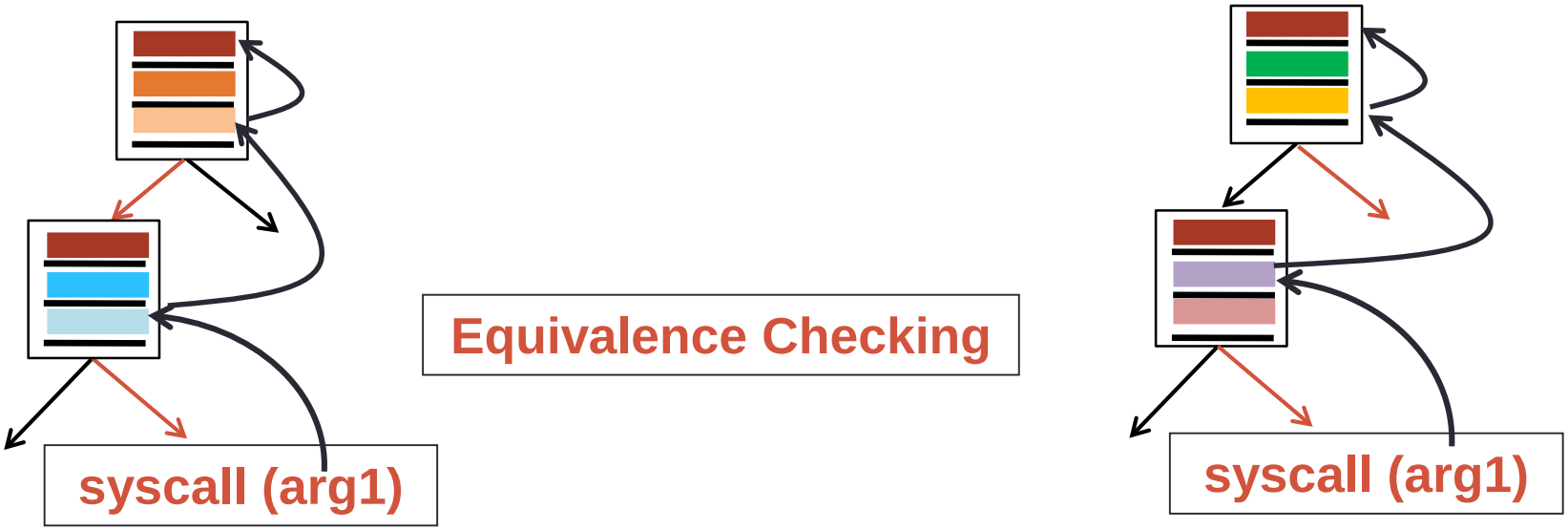
Block-Centric Limitation (2)

- Compiler optimizations
  - loop unrolling and function inline
- Return-oriented programming obfuscation
  - The chain of ROP gadgets will result in a set of small basic blocks
- Different implementations of the same algorithm
  - More examples in Hacker's Delight
- Control flow obfuscation
  - opaque predicates and control flow flattening
- Virtualization obfuscation
  - The decode-dispatch loop generates a sequence of basic blocks to interpret one x86 instruction.



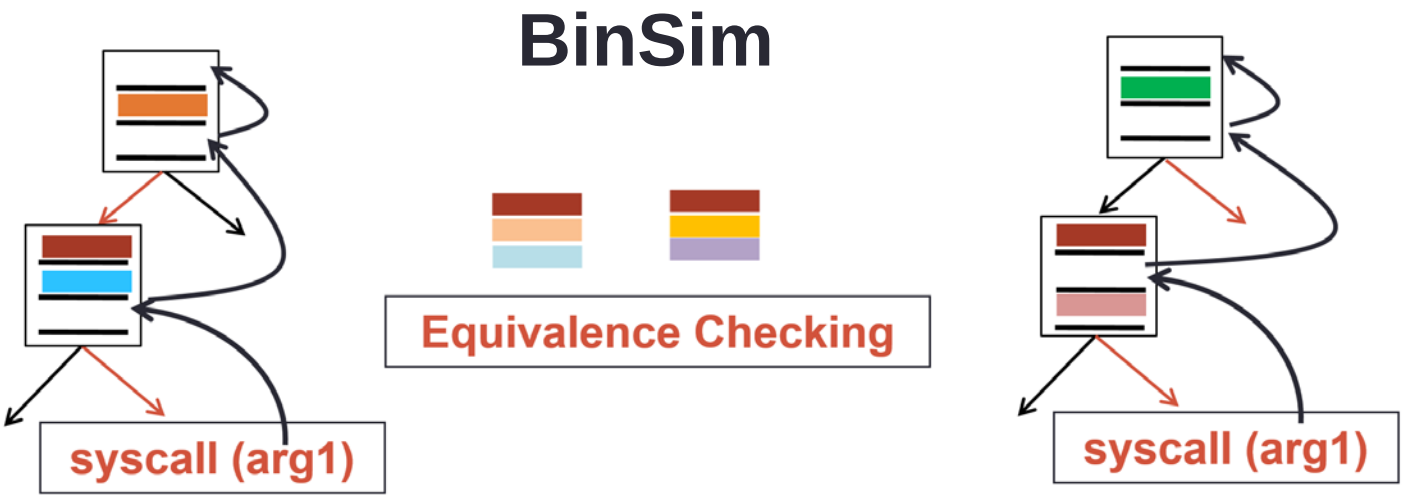
BinSim Overview

- System Call Sliced Segments + Equivalence Checking

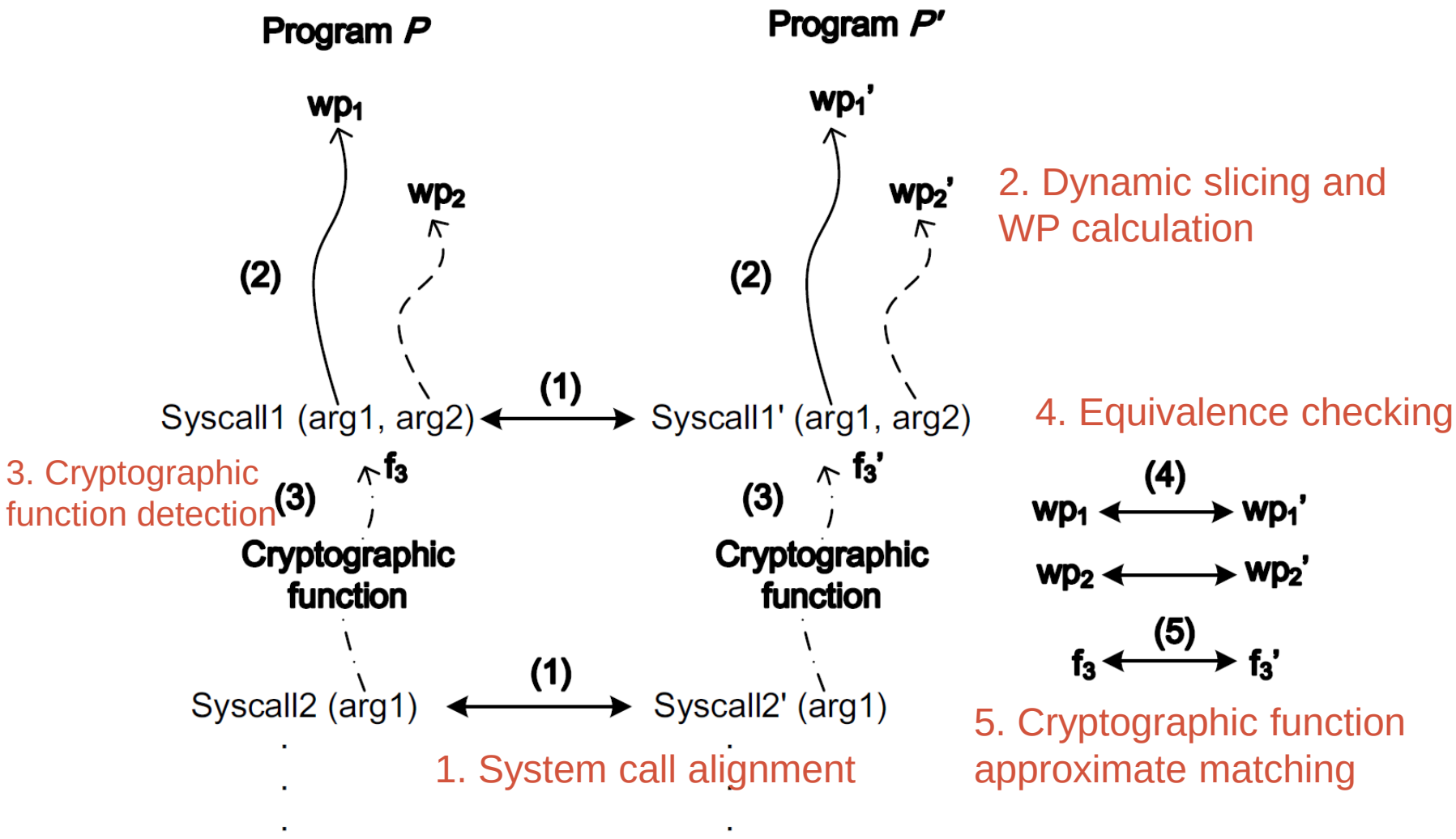


BinSim Advantages

- Whether two matched system calls are ***conditional equivalent***
- System call sliced segments bypass the boundary of a basic block
  - more likely to address "**Block-centric**" limitation



BinSim Core Method



BinSim Architecture

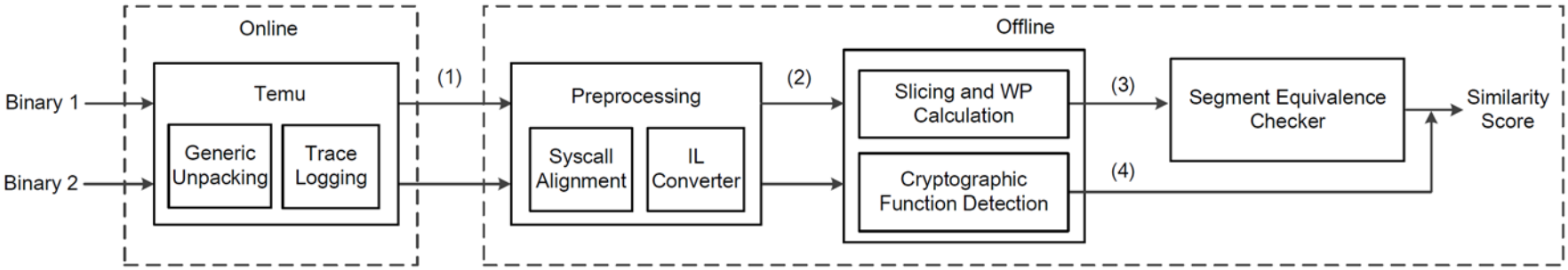


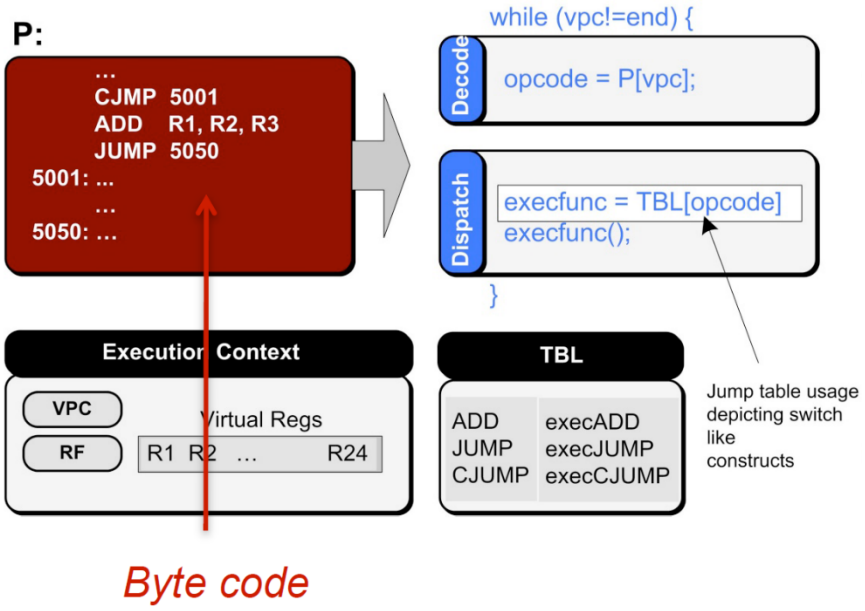
Figure 6: Schematic overview of BinSim. The output for each processing: (1) unpacked code, instruction log, memory log, and system call sequences; (2) IL traces and initial matched system call pairs; (3) weakest preconditions of system call sliced segments; (4) identified cryptographic functions.

System Call Alignment

- We rely on MalGene [CCS'15], an advanced bioinformatics-inspired approach to perform system call sequence alignment
- Remove system call noises
  - multi-tag taint analysis
  - consider the parameter semantics
  - fake dependency: *xor eax, eax; NtClose(eax)*

Dynamic Slicing Binary Code (1)

- Dynamic slicing on the obfuscated binaries is never a textbook problem
  - Indirect memory access: *mov ebx, [4*eax+4]*
  - Implicit control transfers: *COMVcc, SETcc, REP, LOOP, CMPXCHG*
  - Decode-dispatch loop of virtualization obfuscation



Source: SHARIF, M., LANZI, A., GIFFIN, J., AND LEE, W. Automatic reverse engineering of malware emulators. (S&P'09)

Dynamic Slicing Binary Code (2)

- Our solution: split data dependencies and control dependencies tracking
  - index and value based slicing to only trace data flow
  - tracking explicit and implicit control dependencies: eflags bit
  - exception: *jecxz* jumps if register ecx is zero
  - remove the fake control dependencies caused by virtualization obfuscation code dispatcher: *5→ 3,163 → 109*

Handling Cryptographic Functions

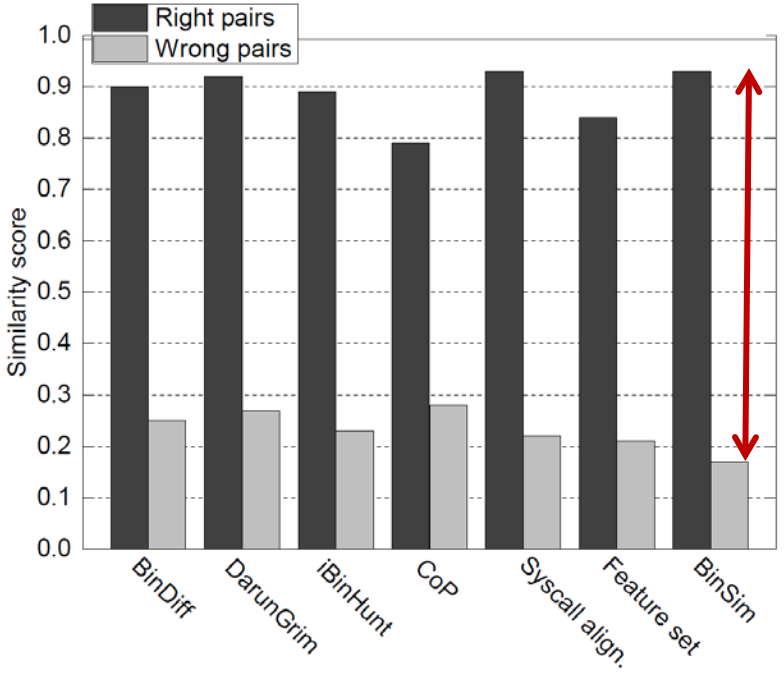


- Cryptographic functions have been a known challenge to symbolic execution
- Almost no interaction with system calls
- Inspired by Caballero et al.’s work (CCS'10), we do a “stitched symbolic execution”

Ground Truth Dataset

Table 1: Different obfuscation types and their examples.

	Type	Examples
1	Intra-basic-block	Register swapping, junk code, instructions substitution and reorder
2	Control flow	Loop unrolling, opaque predicates, control flow flatten, function inline
3	ROP	Synthetic benchmarks collected from the reference [79]
4	Different implementations	BitCount (Figure 3) isPowerOfTwo (Appendix Figure 12) flp2 (Appendix Figure 13)
5	Covert computation[59]	Synthetic benchmarks
6	Single-level virtualization	VMProtect [69]
7	Multi-level virtualization	Synthetic benchmarks collected from the reference [79]



- Similarity scores change from right pairs to wrong pairs.

Comparative Evaluation Results

Table 3: Absolute difference values of similarity scores under different obfuscation schemes and combinations.

Sample	Obfuscation type	BinDiff	DarunGrim	iBinHunt	CoP	Syscall alignment	Feature set	BinSim
BullMoose	6	0.58	0.56	0.39	0.61	0.08	0.10	0.08
Clibo	1+6	0.57	0.64	0.41	0.62	0.10	0.12	0.10
Branko	1+2+6	0.63	0.62	0.35	0.68	0.10	0.15	0.12
Hunatcha	2	0.40	0.42	0.19	0.30	0.12	0.17	0.12
WormLabs	1	0.10	0.12	0.03	0.03	0.08	0.12	0.05
KeyLogger	2	0.38	0.39	0.12	0.26	0.09	0.15	0.09
Sasser	1+2+6	0.62	0.62	0.42	0.58	0.12	0.18	0.10
Mydoom	1+2	0.42	0.38	0.10	0.38	0.10	0.15	0.05
ROP	3	0.63	0.54	0.49	0.52	0.10	0.10	0.10
Different implementations	4	0.48	0.39	0.48	0.52	0.05	0.10	0.05
Covert computation	5	0.45	0.36	0.44	0.45	0.05	0.10	0.05
Multi-level virtualization	7	0.68	0.71	0.59	0.69	0.15	0.18	0.16
Average								
"Right pairs" vs. "Obfuscation pairs"		0.50	0.48	0.34	0.46	0.10	0.15	0.09
"Right pairs" vs. "Wrong pairs"		0.65	0.65	0.66	0.55	0.71	0.63	0.76

Analyzing Wild Malware Variants

- Collect 1,050 active malware samples from VirusShare on February 2017, 112 families
  - Perform intra-family pairwise comparison
  - BinSim are able to find small distances among intra-family samples.
  - 11% of malware variants are “conditionally equivalent”
  - Example: A variant of CryptoWall terminate execution if infected machine’s UI languages are former Soviet Union country languages





BinSim Limitations

- BinSim only detects the similarities/differences exhibiting during execution.
  - incomplete path coverage + environment-sensitive malware
- Attack BinSim's enhanced slicing algorithm
  - slice size explosion
- Custom cryptographic algorithm
  - evade cryptographic function approximate matching

Q & A