Scalable Clone Detection on Low-Level Codebases

Davide Pizzolotto

Advisors: Katsuro Inoue, Yoshiki Higo

Table of Contents

Table of Contents

[1] Code Clone Detection in Rust Intermediate Representation.
Davide Pizzolotto and Makoto Matsushita and Katsuro Inoue.
In IPSJ/SIGSE, 2022-SE-211(26), 1-7 (2022-07-21), 2188-8825.

[2] Blanker: A Refactor-Oriented Cloned Source Code Normalizer.
Davide Pizzolotto and Katsuro Inoue.
In Proceedings of the 14th IEEE International Workshop on Software Clones, IWSC 2020, London, ON, Canada, February 18, 2020

[3] BinCC: Scalable Function Similarity Detection in Multiple Cross-Architectural Binaries.
Davide Pizzolotto and Katsuro Inoue.
IEEE Access, 2022, Volume 10, Pages 124491-124506.

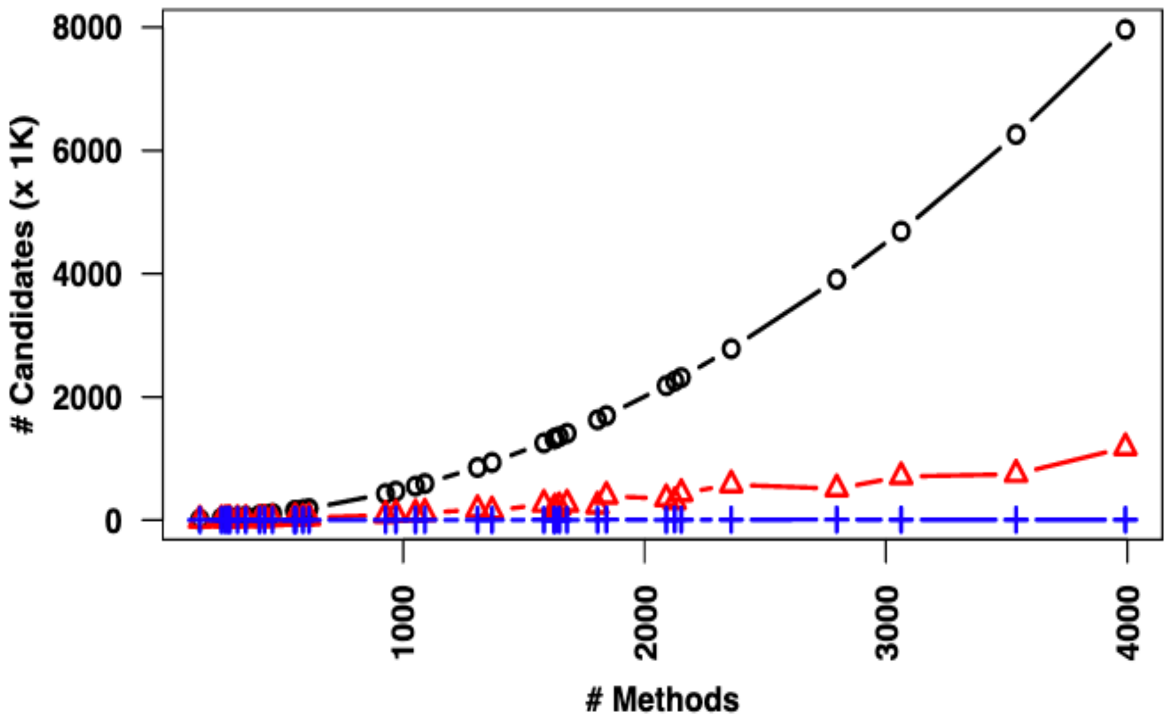
[4] Identifying Compiler and Optimization Options from Binary Code using Deep Learning Approaches.
Davide Pizzolotto and Katsuro Inoue.
In Proceedings of the 36th IEEE International Conference on Software Maintenance and Evolution, ICSME 2020, Adelaide, Australia, September 28 - October 2, 2020.

[5] Identifying Compiler and Optimization Level in Binary Code From Multiple Architectures.
Davide Pizzolotto and Katsuro Inoue.
IEEE Access, 2021, Volume 9, Pages 163461-163475.

Introduction

- **Copying and reusing** portions of code has become a common practice
- Copying code often generates **Code Clones**
- Clones create **maintainability problems**: fixing a bug in a clone snippet requires fixing the same bug in all clones

Introduction



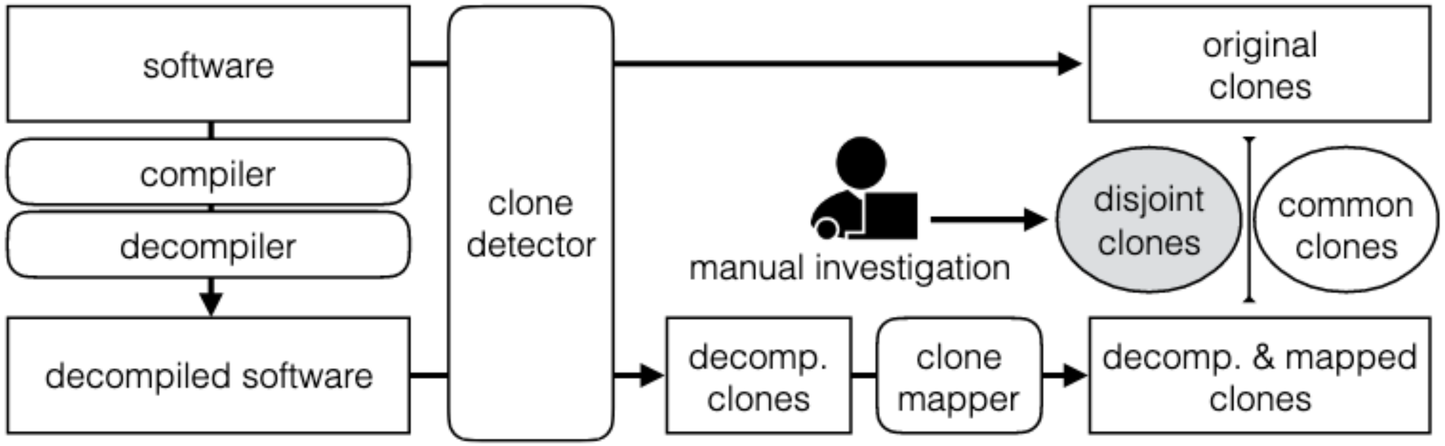
Comparison reduction in the SourcererCC tool [1]

- In recent years, several tools to detect code clones have been developed
- Several techniques have been developed to **reduce** the amount of **comparisons** done.

[1] Sajnani, Hitesh, et al. "Sourcerercc: Scaling code clone detection to big-code." *Proceedings of the 38th International Conference on Software Engineering*. 2016.

Introduction

- Moreover, also the quality of code clones have improved
- Ragkhitwetsagul et al. experimented with **compilation-decompilation** to normalize code and reduce differences between clones [1]

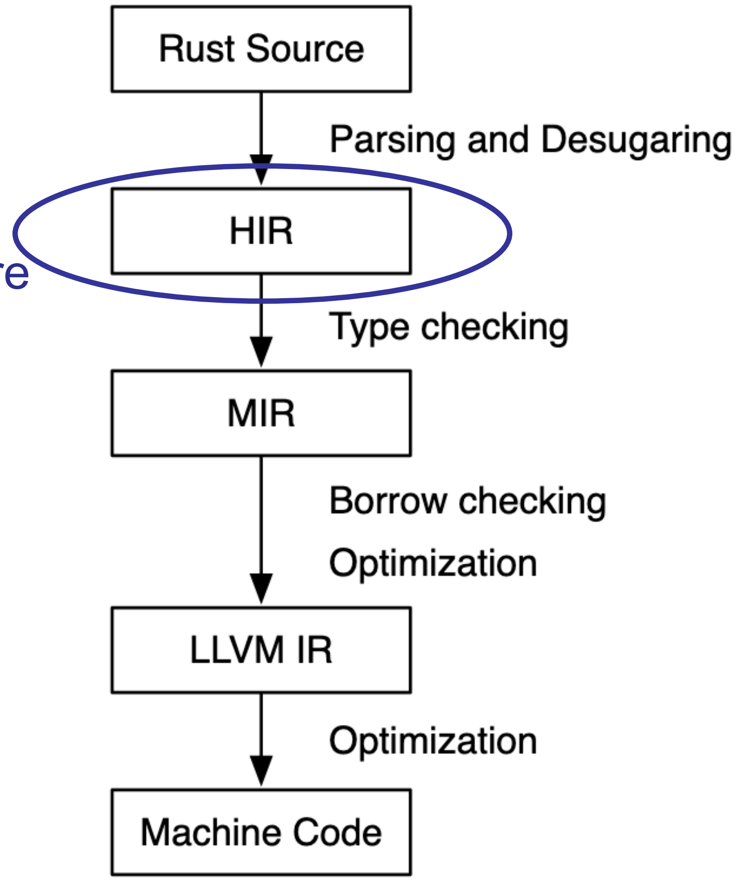


Setup of Ragkhitwetsagul et al. to detect clones using a compiler

[1] Ragkhitwetsagul, Chaiyong, and Jens Krinke. "Using compilation/decompilation to enhance clone detection." *2017 IEEE 11th International Workshop on Software Clones (IWSC)*. IEEE, 2017.

Introduction

Check for clones here



- We tried to remove the decompilation step of previous works, and applied code cloning on the **compiler intermediate code**
- We also manually implemented compiler transformations
- Results were good, but too much effort is required to map the transformed code to the original one

We have to go deeper!

```
00000000  3b 00 91 df 96 f6 33 73  7f f1 de 13 a2 8a 45 30
00000010  f6 01 ff e2 52 43 15 4e  1c a9 bf 9a 1c 41 8b 40
00000020  fa 14 30 24 2f ed bc 00  7d 46 4c 32 03 f2 ba 69
00000030  dd f5 28 87 84 20 61 f5  c9 3a 54 c2 98 9e c1 11
00000040  20 df 23 16 22 64 71 90  c1 2c 7c 1e 68 0e e2 28
00000050  66 b8 d2 05 2e e7 75 11  1b c8 4e 4c d4 9b 4a 8b
00000060  69 75 fb de 05 b3 4f f2  dc 26 04 4a 02 2a 2c 56
00000070  55 ef 93 07 e6 a3 2f 01  4a d9 75 3d b8 2b 13 f1
00000080  a3 30 7d c5 e2 0f 69 16  03 21 51 0e b5 d5 08 98
00000090  3e ca c5 22 5f b0 d4 3d  2e 78 11 92 99 66 24 5a
000000a0  56 96 74 41 cd 41 91 d4  02 65 ca 20 3e 1c a4 c1
000000b0  c9 b6 e9 aa 89 89 40 e4  66 c4 d4 3f 49 85 e5 66
000000c0  56 82 93 f9 94 87 15 9c  2f 46 08 30 01 79 28 e3
000000d0  41 e7 29 24 ad 21 0a 4b  e0 79 ea 7f fd 4b ec 10
000000e0  a9 b8 23 96 69 17 a9 4e  8b 13 0d 5c 4c 28 28 f2
000000f0  ae e7 6e d8 e8 54 7e 15  da 51 2d 38 00 5f 59 26
```

- Going closer to machine level complicates everything: no more variables, comments, optimized code everywhere…
- Source clone detectors fails
- Binary clone detectors are limited to pairwise comparison and slow

We have to go deeper!

```
4889fe
488d0da60601.
48c7c2ffffff.
31ff
e998faffff
```

```
4889fe
488d0d36f900.
48c7c2ffffff.
31ff
e998faffff
```

We have to go deeper!

quotearg in /bin/ls

```
4889fe            mov rsi, rdi
488d0da60601.     lea rcx, obj.default_quoting_options
48c7c2ffffff.     mov rdx, 0xffffffffffffffff
31ff              xor edi, edi
e998faffff        jmp sym.quotearg_n_options
```

quotearg in /bin/mv

```
4889fe            mov rsi, rdi
488d0d36f900.     lea rcx, obj.default_quoting_options
48c7c2ffffff.     mov rdx, 0xffffffffffffffff
31ff              xor edi, edi
e998faffff        jmp sym.quotearg_n_options
```

Goals

- A **fast** clone detection in binary code
- Comparing **multiple files** at once
- Ability to scale up to hundred of megabytes
- **Cross-architecture** compatibility

Table of Contents

Part 1: Improving Source Clone detection

Part 2: Scalable Binary Clone detection

Binary clone detection – Main Problem

- Our idea is to compare the **program flow**, called Control Flow Graph (CFG) and find similar ones.
- CFGs encode the program's logic and can be extracted from the binary code.
- However, comparing CFGs requires **exponential time**!
- The worst case require 2^{10000} comparisons.
- The average case require 2^{40} comparisons.

Binary clone detection – Our solution

- Our solution is to convert the program flow into **high level structures**, and compare with **hashing**

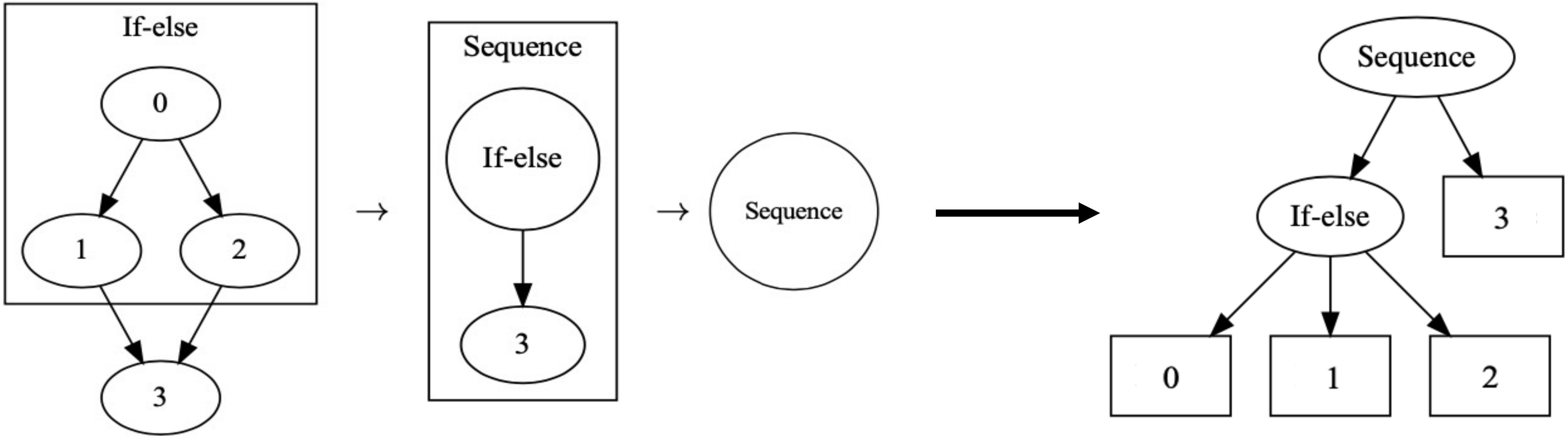
```
void test(int input) {
    printf("hello ");
  if (input == 0) {
    printf("then");
  } else {
    printf("else");
  }
    printf("!!!");
}
```

→

```
63: fcn.00000019 ();
bp: 1 (vars 1, args 0)
sp: 0 (vars 0, args 0)
rg: 0 (vars 0, args 0)
0x00000019      837dfc00        cmp dword [var_4h], 0
0x0000001d  ~   0f8513000000    jne 0x36
;-- mach0_segment64_0:
;-- mach0_cmd_0:
0x00000020      0000            add byte [rax], al
0x00000022      00              invalid
----------- true: 0x00000036  false: 0x00000023
0x00000023      488d3d350000.   lea rdi, [0x0000005f]
0x0000002a      b000            mov al, 0
0x0000002c      e800000000      call 0x31
0x00000031      e90e000000      jmp 0x44
----------- true: 0x00000044
0x00000036      488d3d270000.   lea rdi, [0x00000064]
0x0000003d      b000            mov al, 0
0x0000003f      e800000000      call 0x44
----------- true: 0x00000044
0x00000044      488d3d1e0000.   lea rdi, [0x00000069]
0x0000004b      b000            mov al, 0
0x0000004d      e800000000      call 0x52
0x00000052      4883c410        add rsp, 0x10
0x00000056      5d              pop rbp
0x00000057      c3              ret
```

Binary clone detection – Our solution

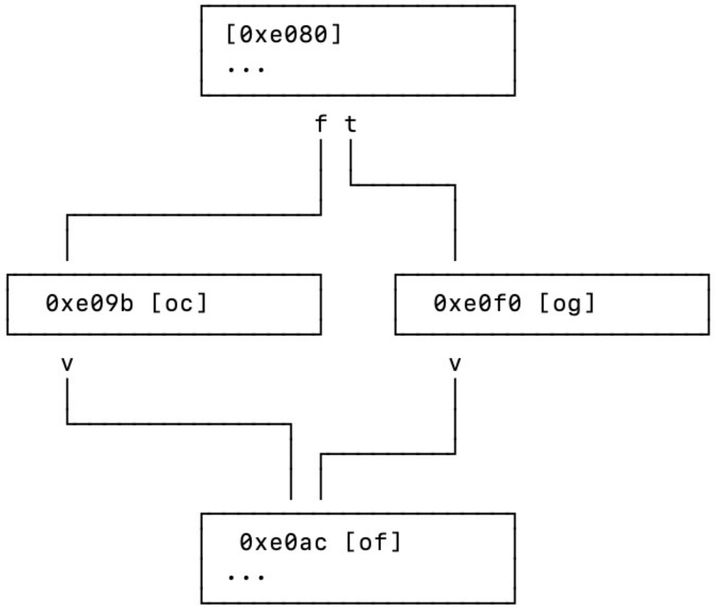
- Our solution is to convert the program flow into **high level structures**, and compare with **hashing**



Disasm and CFG

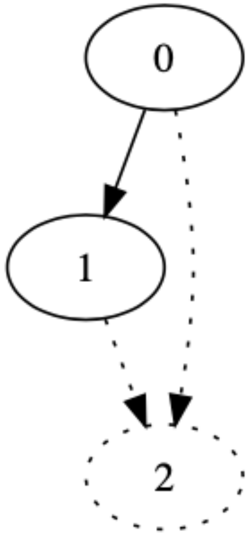
We use existing tools to disassemble and build the Control Flow Graph (CFG)

```
0x0000e080  4155 4989 f541 5455 4889 fd53 4883 ec08
0x0000e090  4883 faff ba05 0000 0074 5548 8d35 8d01
0x0000e0a0  0100 31ff e847 61ff ff49 89c4 4889 eebf
0x0000e0b0  0100 0000 e827 8900 004c 89ea be08 0000
0x0000e0c0  0031 ff48 89c3 e825 8100 0048 83c4 0849
0x0000e0d0  89d8 4c89 e25b 4889 c15d 31f6 415c 31ff
0x0000e0e0  31c0 415d e937 64ff ff0f 1f80 0000 0000
0x0000e0f0  488d 351d 0101 0031 ffe8 f260 ffff 4989
0x0000e100  c4eb a966 662e 0f1f 8400 0000 0000 6690
0x0000e110  4157 4156 4155 4531 ed41 5449 89d4 ba05
0x0000e120  0000 0055 4889 f548 8d35 1e01 0100 5348
0x0000e130  89fb 4883 ec18 4c8b 35c3 7401 0048 897c
0x0000e140  2408 31ff e8a7 60ff ff4c 89f6 4c8d 350e
0x0000e150  0101 0048 89c7 e8e5 61ff ff4c 8b3b 31db
0x0000e160  4d85 ff75 44e9 8600 0000 660f 1f44 0000
0x0000e170  4c89 ff49 89ed e885 8800 0048 8b3d 7e74
0x0000e180  0100 4c89 f2be 0100 0000 4889 c131 c0e8
0x0000e190  5c64 ffff 488b 4424 0848 83c3 014c 01e5
0x0000e1a0  4c8b 3cd8 4d85 ff74 4748 85db 74c2 4c89
0x0000e1b0  e248 89ee 4c89 efe8 6461 ffff 85c0 75b0
0x0000e1c0  4c89 ffe8 3888 0000 488b 3d31 7401 00be
0x0000e1d0  0100 0000 488d 158e 0001 0048 89c1 31c0
0x0000e1e0  e80b 64ff ffeb ad66 0f1f 8400 0000 0000
```



Reconstruction

We check for 11 different high-level **patterns** defining a CFG, finding them with a **set of rules**



If-else rules: ✖

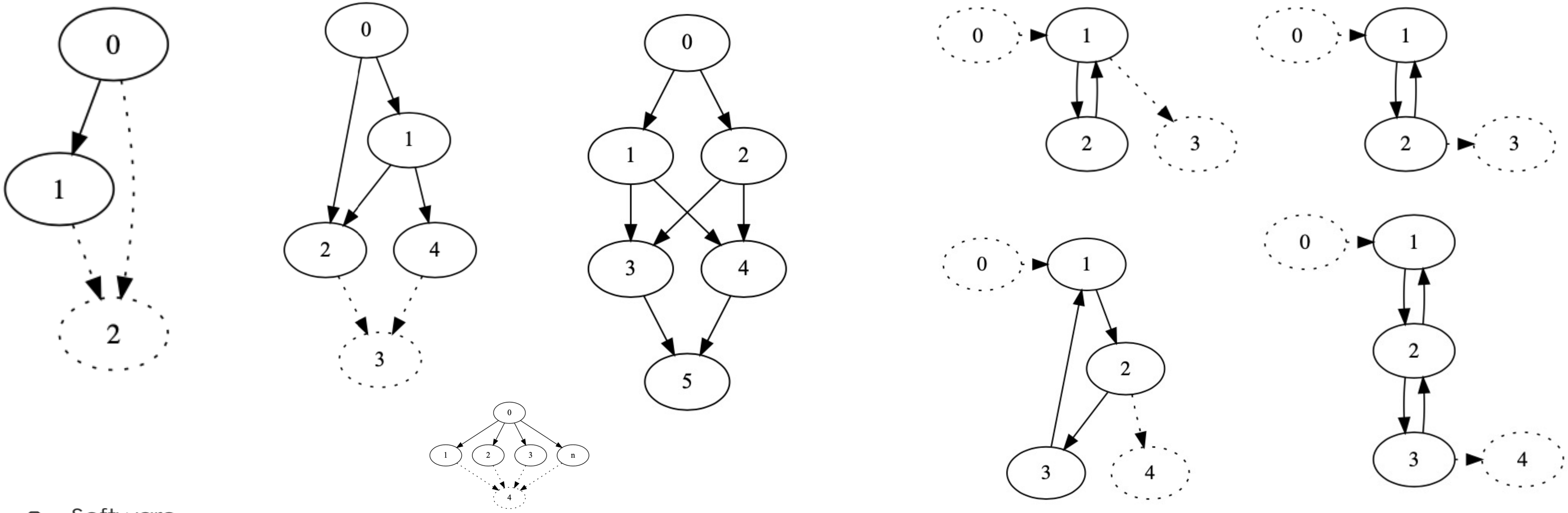
Sequence rules: ✖

While rules: ✖

If-then rules: ○
- node has two children, 1 and 2
- child 1 has one parent and one child
- child 2 is the son of 0 and child 1

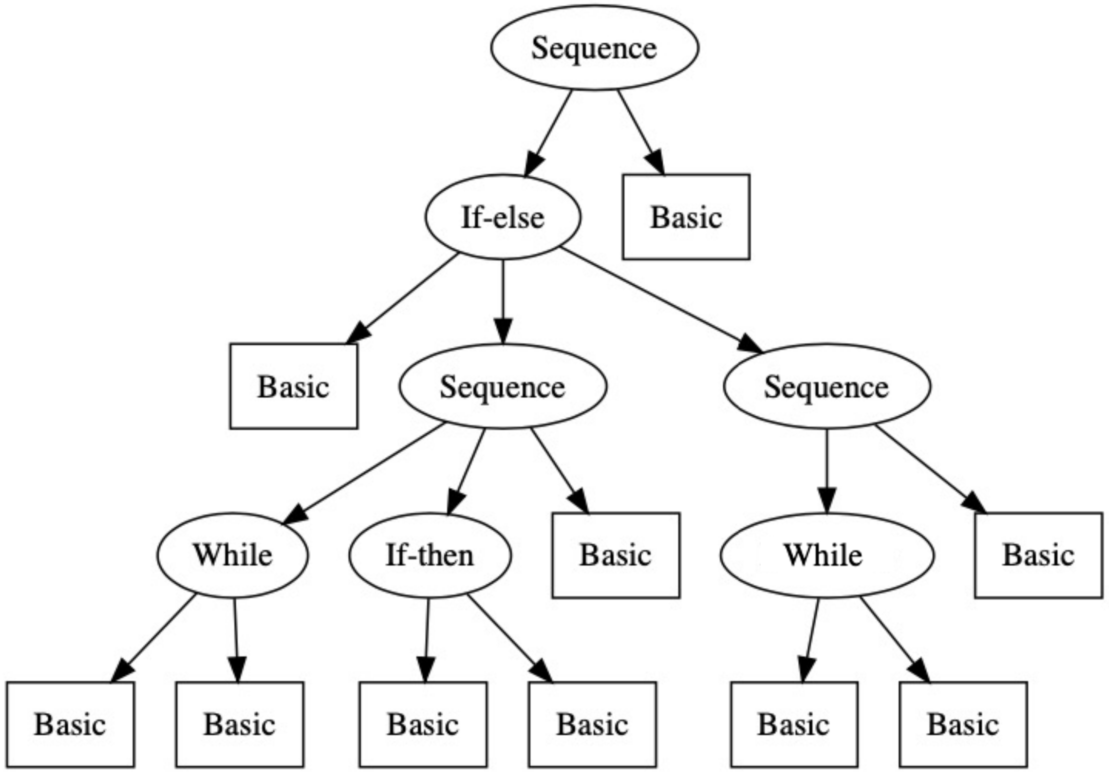
Reconstruction

We check for 11 different high-level **patterns** defining a CFG, finding them with a **set of rules**



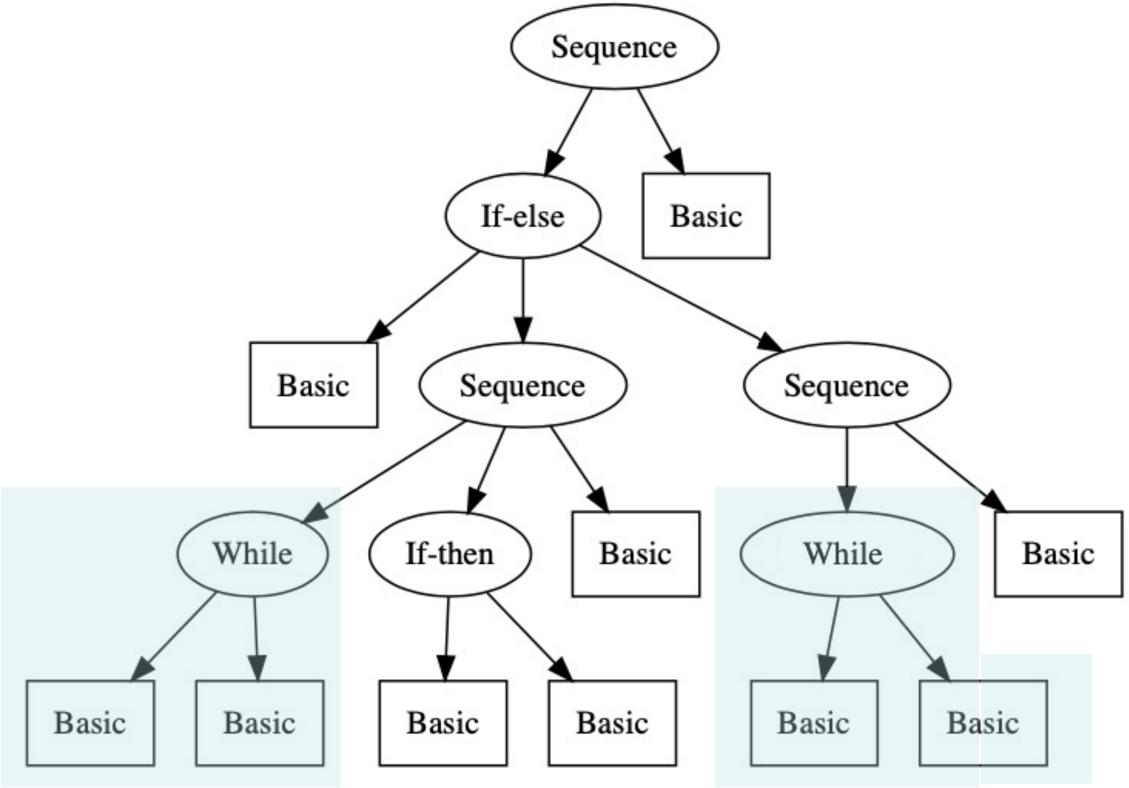
Comparison

Finally, we compare using **hashing**, and check for **semantic consistency** using **cosine similarity**.

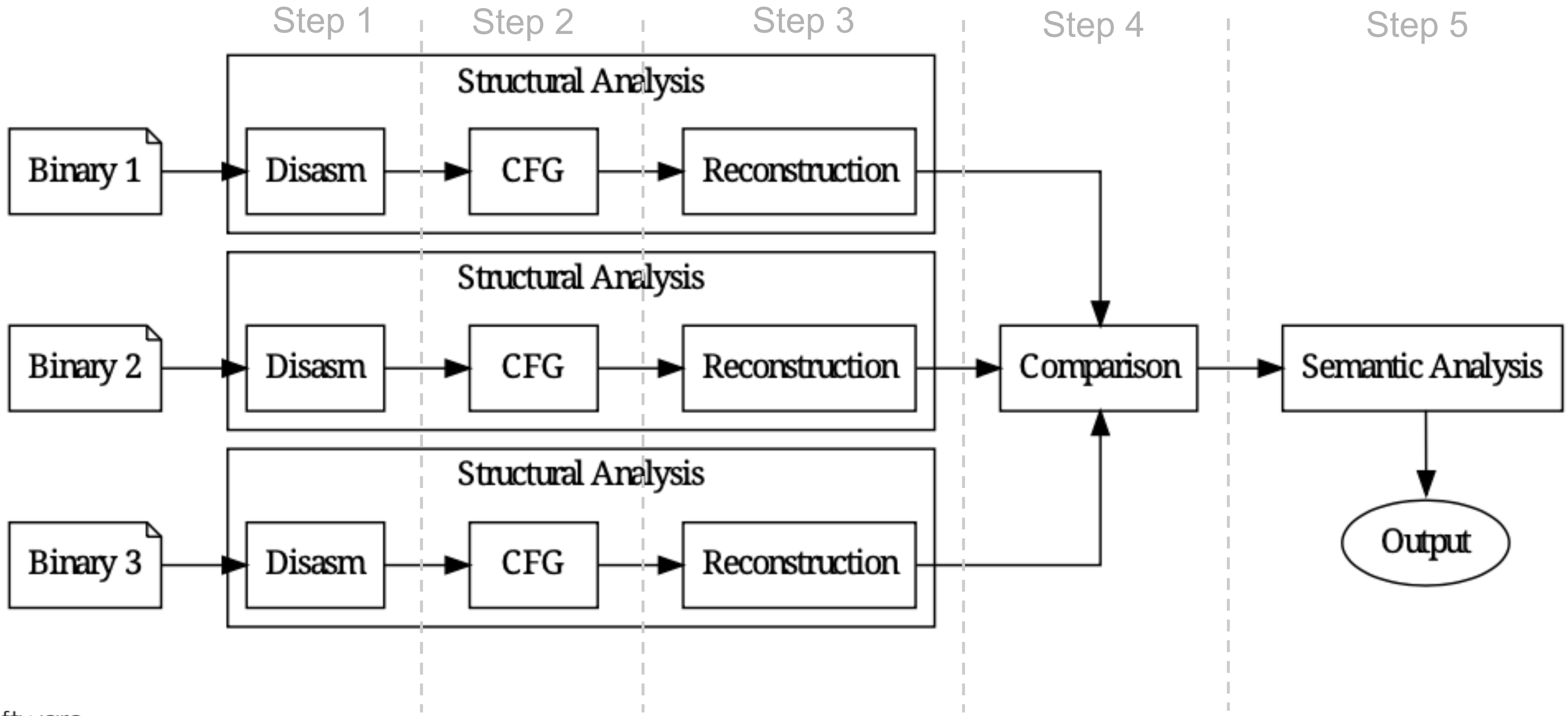


Comparison

Finally, we compare using **hashing**, and check for **semantic consistency** using **cosine similarity**.



Overview

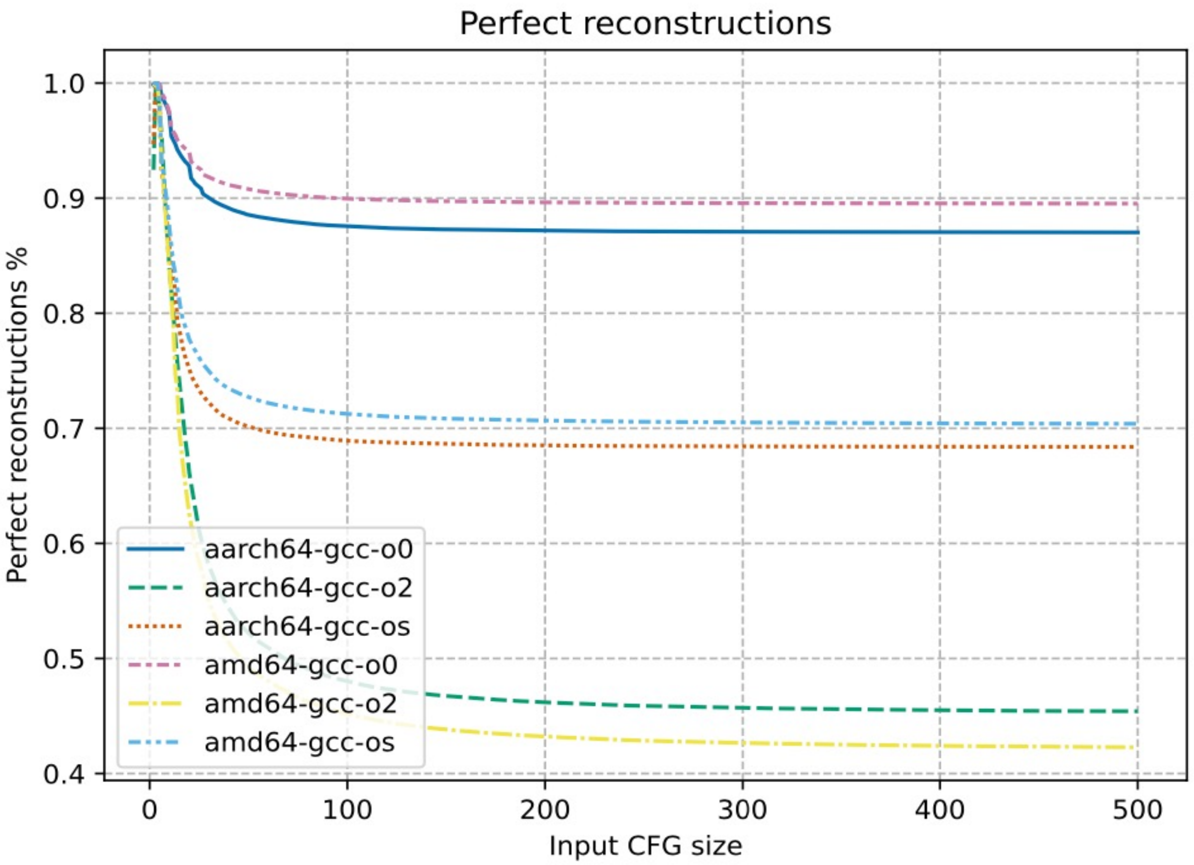

Step 1
Step 2
Step 3
Step 4
Step 5
Structural Analysis
Binary 1
Disasm
CFG
Reconstruction
Structural Analysis
Binary 2
Disasm
CFG
Reconstruction
Comparison
Semantic Analysis
Structural Analysis
Binary 3
Disasm
CFG
Reconstruction
Output

Research Questions

- **RQ1:** Can we convert every CFG to a tree?
- **RQ2:** How accurate is our clone detection?
- RQ3: Can we detect library usage in a real-world application?
- **RQ4:** How performant is our tool varying the input size?

RQ1

RQ1: Can we convert every CFG to a tree?



No. In highly optimized code our approach can convert only 50% of the functions

RQ2

How accurate is our clone detection?

θ	Same architecture (x86_64) Min. Cosine Sim. 0.98	0.99	0.999
2	0.8494 / 0.9746	0.8702 / 0.9708	0.8752 / 0.7690
3	0.9148 / 0.9114	0.9280 / 0.9084	0.9306 / 0.9235
4	0.9178 / 0.9143	0.9345 / 0.9145	0.9363 / 0.9039
5	0.9627 / 0.8593	0.9876 / 0.8646	0.9901 / 0.7964
6	0.9658 / 0.8905	0.9888 / 0.7599	0.9907 / 0.8004

θ	Cross architecture (x86_64, aarch64) Min. Cosine Sim. 0.98	0.99	0.999
2	0.7311 / 0.5071	0.7334 / 0.5087	0.7516 / 0.4645
3	0.7241 / 0.5402	0.7368 / 0.5476	0.7449 / 0.5250
4	0.7691 / 0.5504	0.7679 / 0.5647	0.7720 / 0.5402
5	0.7204 / 0.5398	0.7240 / 0.5145	0.7320 / 0.5263
6	0.7410 / 0.4922	0.7430 / 0.4908	0.7500 / 0.4888

Using structural analysis and semantic analysis

RQ2

How accurate is our clone detection?

	Same architecture (x86_64)					Cross architecture (x86_64, aarch64)				
Min.Cosine Sim.	TP	FP	FN	Precision	Recall	TP	FP	FN	Precision	Recall
0.95	93287	252978	7360	0.2694	0.9269	218512	867087	209742	0.2013	0.5102
0.98	12256	14320	2090	0.4612	0.8543	39171	41092	41092	0.3710	0.4880
0.99	7928	2648	307	0.7496	0.9627	16626	11555	17958	0.5900	0.4807
0.999	8922	1163	3724	0.8847	0.7055	17406	3701	28343	0.8247	0.3805

Using semantic analysis only

RQ2

Precision comparison with BinDiff and DeepBinDiff

bin	BinCC (ours)	BinDiff	DeepBinDiff
dir	0.9593 (172)	0.9333 (105)	0.9368 (95)
ls	0.9593 (172)	0.9333 (105)	0.9167 (96)
mv	0.9704 (169)	0.9739 (115)	0.9245 (106)
cp	0.9652 (144)	0.9783 (92)	0.8295 (88)
sort	0.9923 (131)	0.9670 (91)	0.9157 (83)
du	0.9574 (188)	0.9937 (159)	0.9933 (150)
csplit	0.9574 (94)	0.9254 (67)	0.9194 (62)
expr	0.9489 (98)	0.9677 (62)	0.9062 (64)
nl	0.9444 (90)	0.9672 (61)	0.9828 (58)
ptx	0.9266 (109)	0.9444 (72)	0.9254 (67)
split	0.9375 (96)	0.9538 (65)	0.9831 (59)
mean	0.9562	0.9580	0.9303

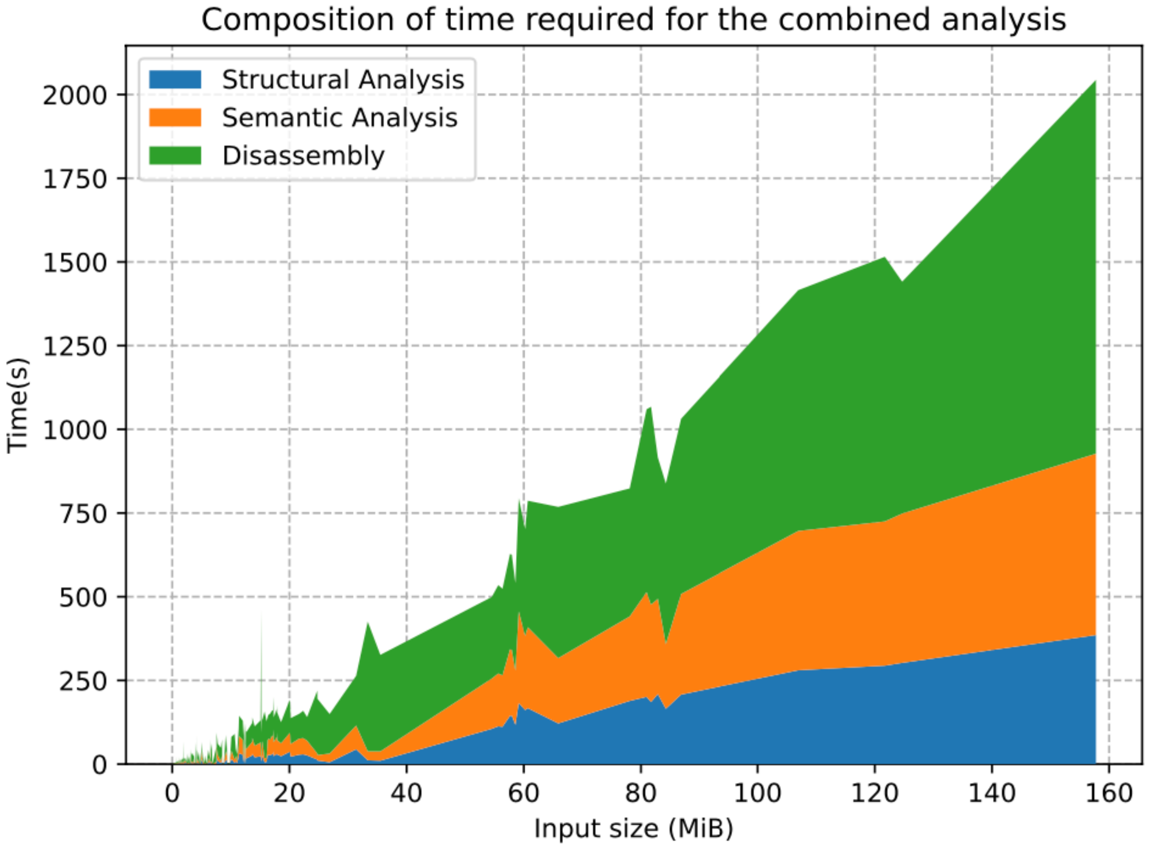
RQ2

Speed comparison with DeepBinDiff

bin	size (KiB)	BinCC (ours)	DeepBinDiff
dir	1081	3.40s	1409s
ls	1081	3.18s	1432s
mv	1045	3.43s	1999s
cp	978	3.22s	1740s
sort	952	2.90s	1216s
du	921	2.80s	2043s
csplit	682	2.55s	659s
expr	673	2.44s	652s
nl	642	2.43s	527s
ptx	749	2.58s	771s
split	701	2.43s	702s

RQ4

How performant is our tool varying the input size?



Limitations

- Our approach is **faster** and **more scalable** than the competition, while obtaining the **same accuracy**
- However, like the rest of the binary clone detectors, it is limited by different compilers and optimization flags in the analyzed files

Table of Contents

Compiler detection

Checking just the PE/ELF header is not always sufficient

```
String dump of section '.comment':
  [     0]  Linker: LLD 10.0.0
  [    13]  GCC: (GNU) 9.2.0
  [    25]  clang version 10.0.0
```

Compiler detection - Overview

- Deep Learning based approach
- Trained on 76k different binary files and more than 24M functions
- Real-time performance, several microseconds for each batch
- Detecting O0/O1/O2/O3/Os optimization levels

Input Type

Without disassembly

4889442418	mov qword [var_18h], rax
31c0	xor eax, eax
4885ff	test rdi, rdi
7423	je 0xd03c
488b4208	mov rax, qword [rdx + 0x8]
48893424	mov qword [rsp], rsi
4889e6	mov rsi, rsp
4889442408	mov qword [rsp + 0x8], rax
488b02	mov rax, qword [rdx]
4889442410	mov qword [rsp + 0x10], rax
e85a0e0000	call fcn.0000de90
4885c0	test rax, rax
0f95c0	setne al

Better for long input sequences

With disassembly

4889442418	mov qword [var_18h], rax
31c0	xor eax, eax
4885ff	test rdi, rdi
7423	je 0xd03c
488b4208	mov rax, qword [rdx + 0x8]
48893424	mov qword [rsp], rsi
4889e6	mov rsi, rsp
4889442408	mov qword [rsp + 0x8], rax
488b02	mov rax, qword [rdx]
4889442410	mov qword [rsp + 0x10], rax
e85a0e0000	call fcn.0000de90
4885c0	test rax, rax
0f95c0	setne al

Better for (very) short input sequences

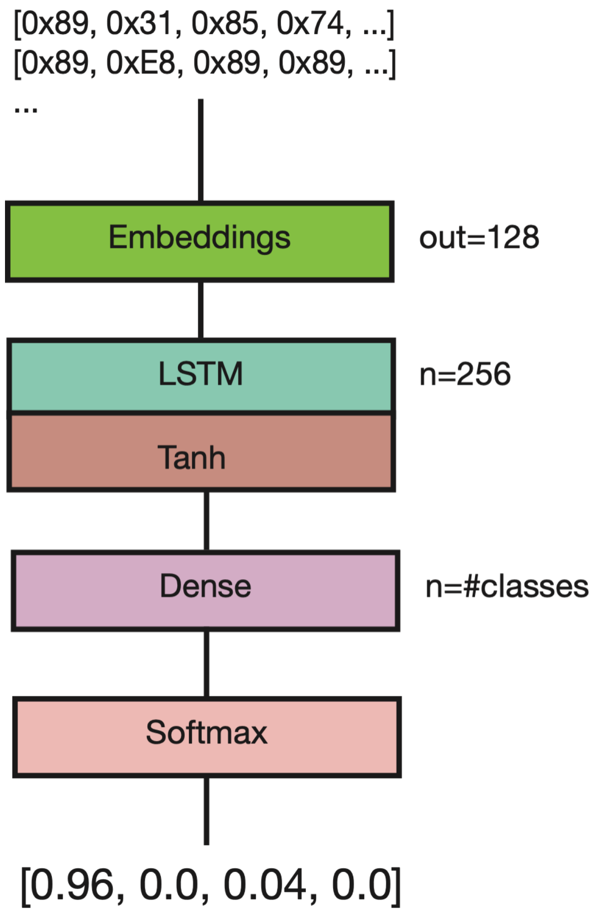
Adding variation

We also **padded** and shifted the input to teach the network how to work with **small sequences**

f0	25	14	de	af	8c	85	c3
85	bf	5b	cf	e0	f2	63	0b
92	af	97	0b	06	84	1d	5d
e3	14	bc	ac	a8	de	21	e7
73	11	27	9a	ff	4f	d9	73
03	d6	ce	de	8b	0d	af	46
74	37	35	f2	49	c3	e5	69
8c	47	4a	57	d2	cf	7e	46

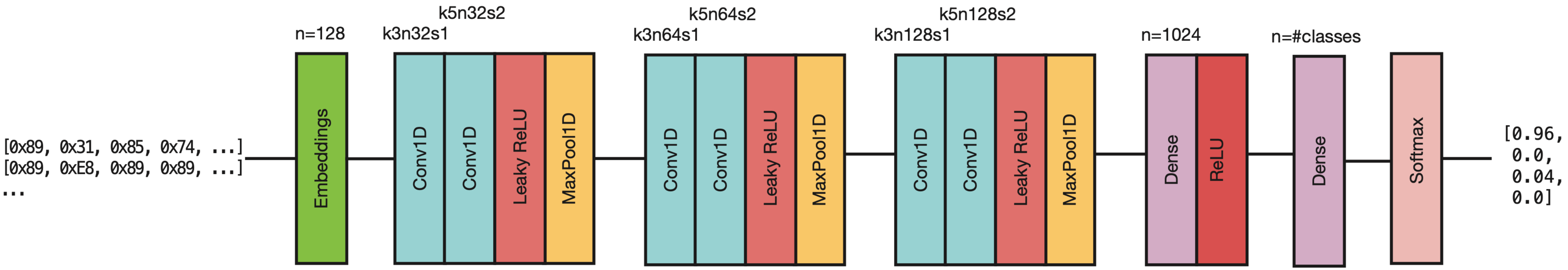
00	f0	25	14	de	af	8c	85
00	00	00	00	85	bf	5b	cf
00	00	00	92	af	97	0b	06
00	00	00	00	00	00	e3	14
00	00	00	00	00	73	11	27
00	00	03	d6	ce	de	8b	0d
00	00	00	74	37	35	f2	49
00	8c	47	4a	57	d2	cf	7e

LSTM network



- Slow training (hours)
- Slow inference (hundredth of milliseconds)
- High accuracy for very small inputs (less than 50 bytes)

CNN network



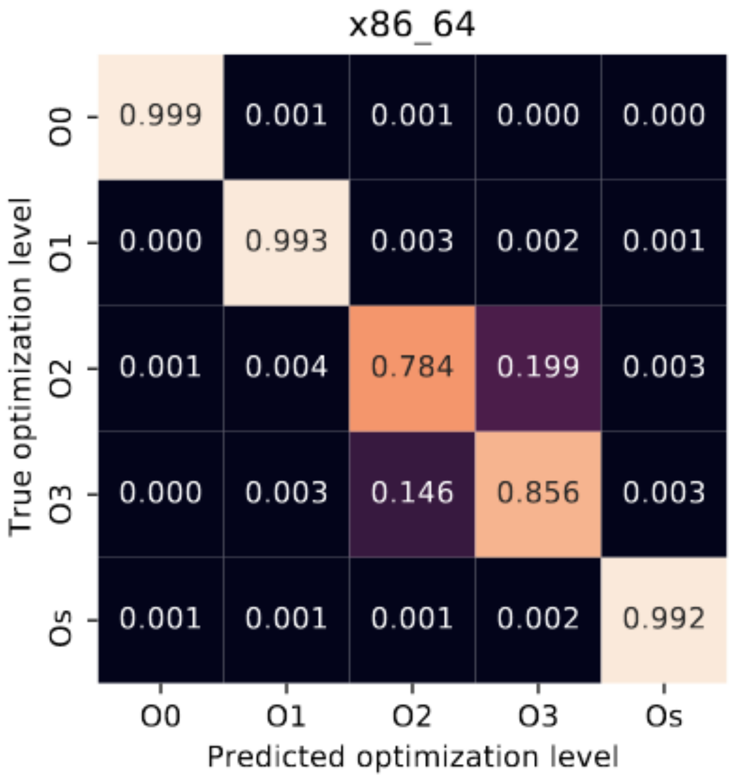
- Fast training (minutes)
- Fast inference (several microseconds)
- Lower accuracy for very small inputs, comparable to RNNs for medium sized inputs

Research Questions

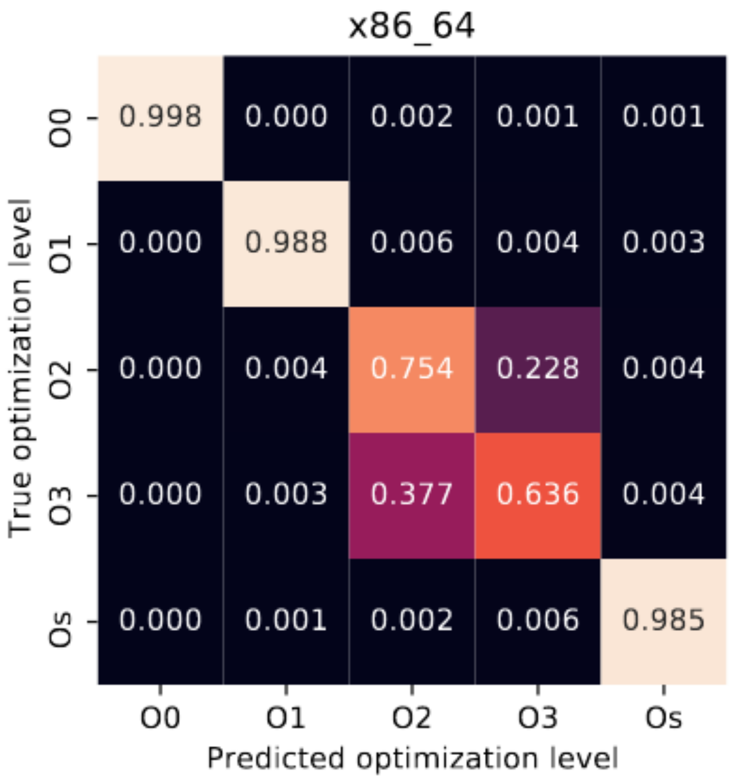
- **RQ1:** Is is better to use a LSTM or a CNN?
- **RQ2:** What is the minimum number of bytes for accurate predictions?
- RQ3: Using a disassembler increases accuracy?
- RQ4: Using padding increases accuracy?
- **RQ5:** What are the most common optimizations?

RQ1

RQ1: Is is better to use a LSTM or a CNN?



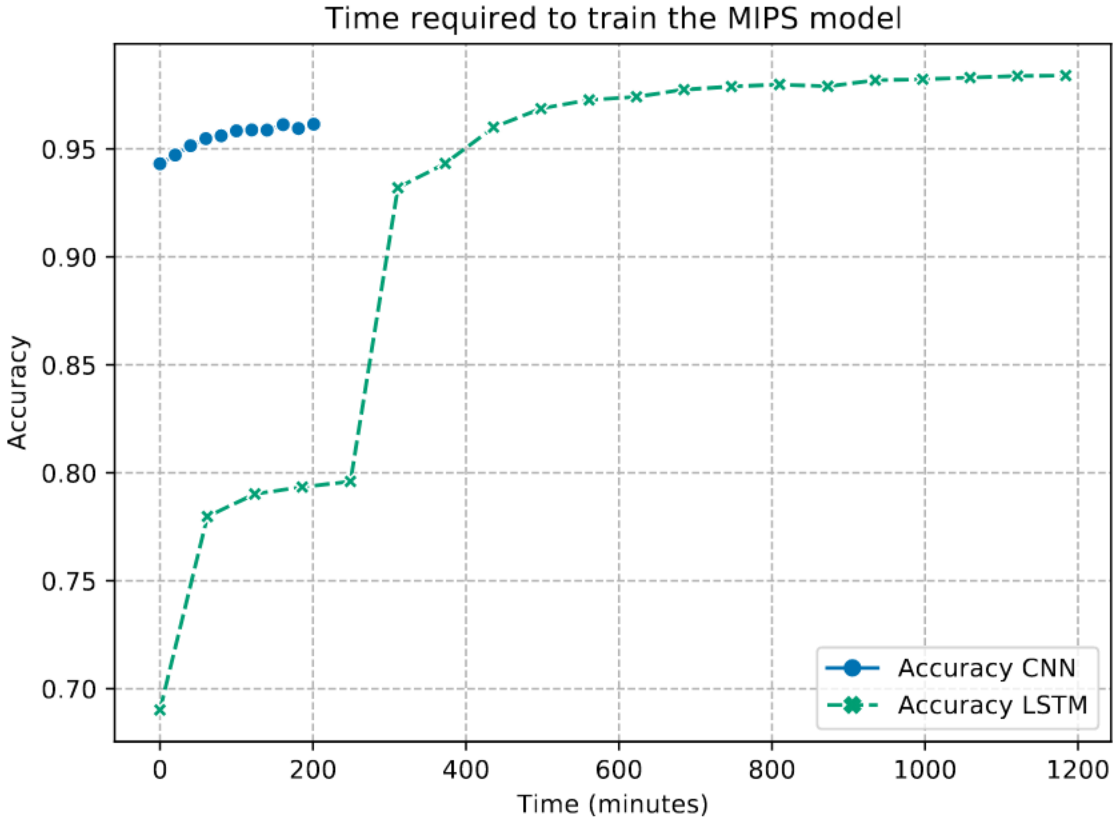
LSTM



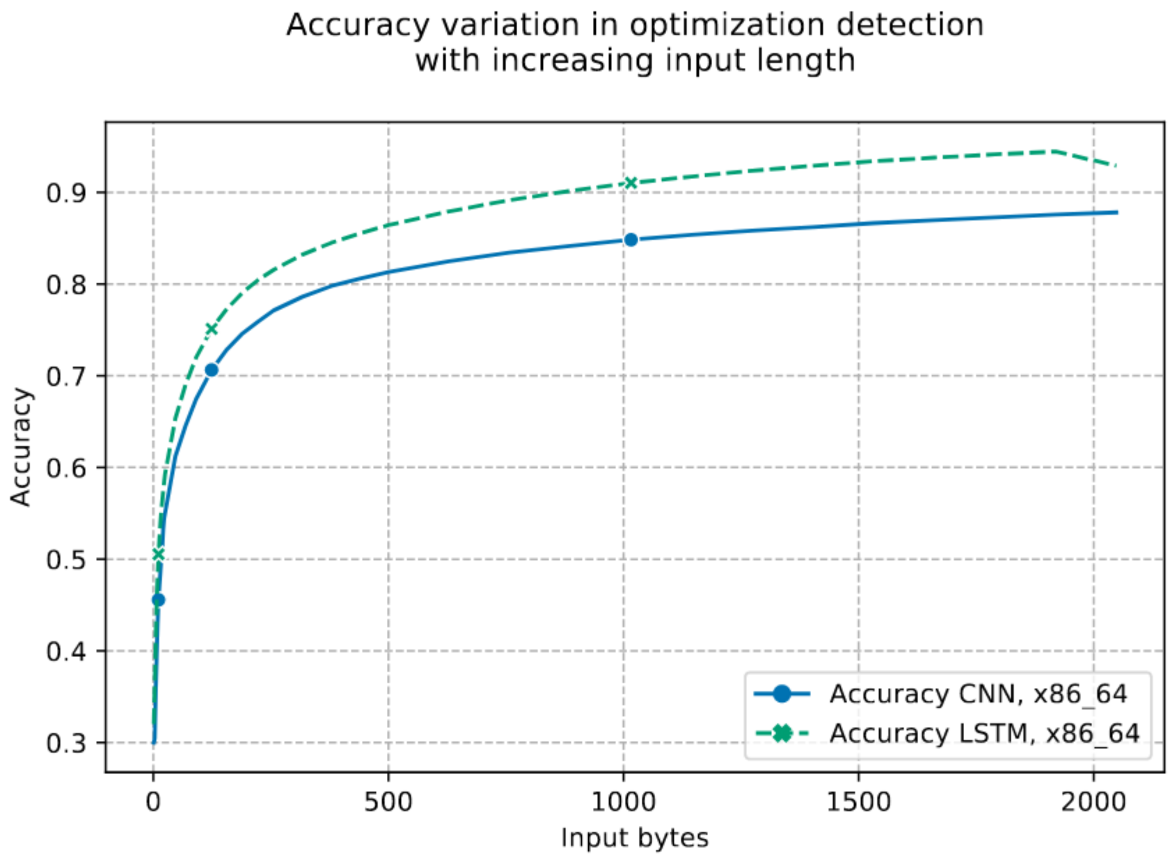
CNN

RQ1

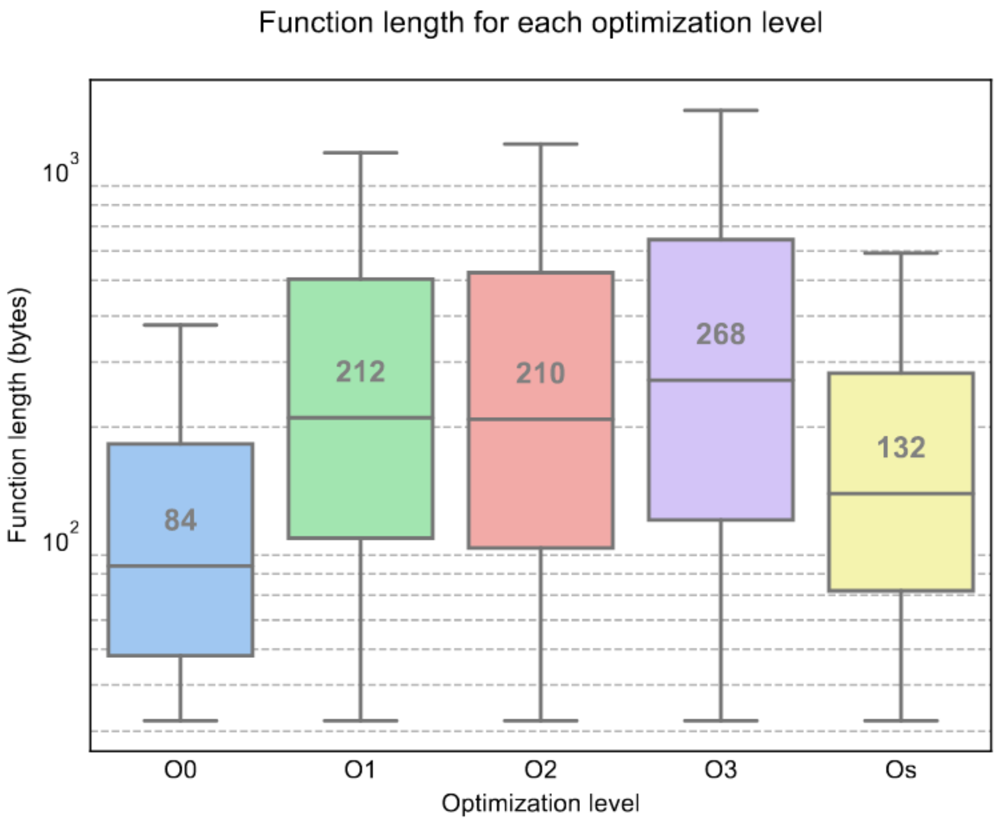
RQ1: Is is better to use a LSTM or a CNN?



RQ2



Accuracy variation in optimization detection
with increasing input length
Accuracy
0.3
0.4
0.5
0.6
0.7
0.8
0.9
0
500
1000
1500
2000
Input bytes
Accuracy CNN, x86_64
Accuracy LSTM, x86_64



Function length for each optimization level
Function length (bytes)
10^3
10^2
84
212
210
268
132
O0
O1
O2
O3
Os
Optimization level

RQ5

RQ5: What are the most common optimization levels?

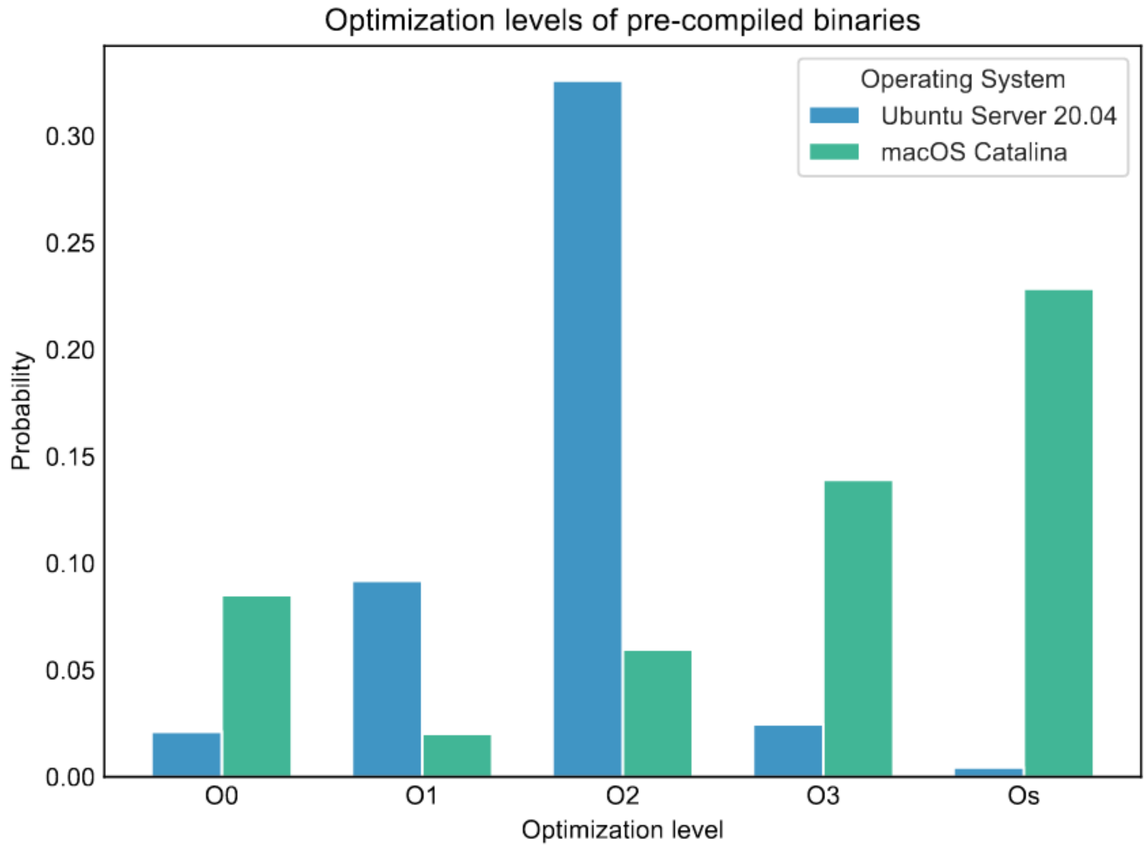


Table of Contents

Conclusion

- We presented a new approach at detecting clones in binary code
- This approach is fast and scalable, works in cross architecture and can reach the same accuracy as state-of-the-art
- Although this approach suffers when comparing differently optimized binaries, we developed an optimization detector