Scalable framework for binary code comparison*

Hayk Aslanyan, Arutyun Avetisyan, Mariam Arutunian, Grigor Keropyan, Shamil Kurmangaleev, Vahagn Vardanyan
(ISP RAS)

Ivannikov ISPRAS Open Conference 2017

hayk@ispras.ru

*The paper is supported by RFBR grant 15-07-07541

Problem definition

Develop a tool for **binary file comparison**

Requirements:

- High accuracy
  - Overcome instruction reordering
  - Overcome register name changing
- Ability to analyze binaries from different architectures (x86, x86-64, ARM, MIPS, PPC)
- Scalable (ability to analyze large binary files)

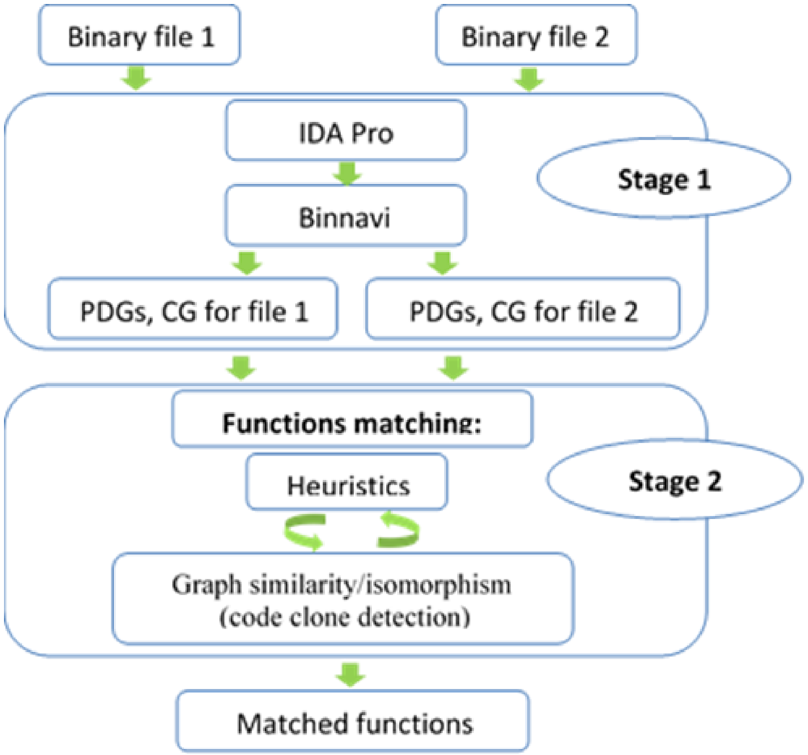
Use Cases

- Detect programmatic changes between two binaries
- Find old versions of statically linked libraries to prevent using well-known bugs
- Protection of author rights

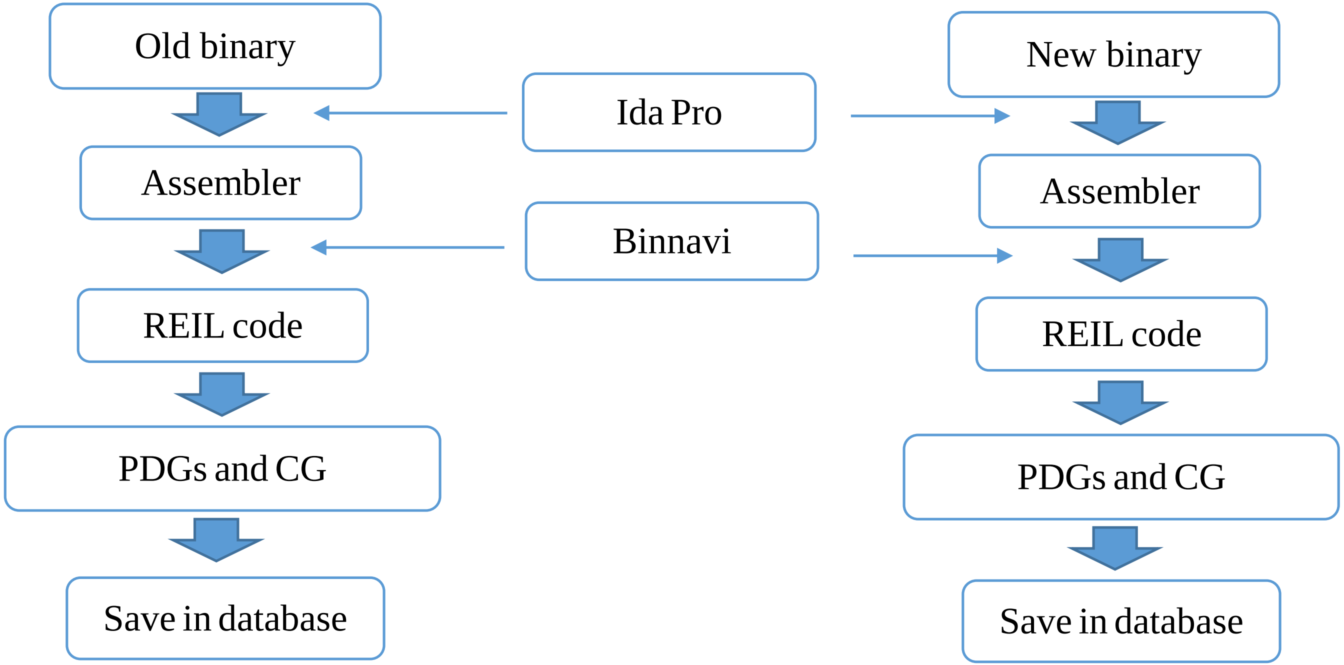
Related work

- BMAT
- BinDiff
- DarunGrim2
- BinHunt

Tool architecture



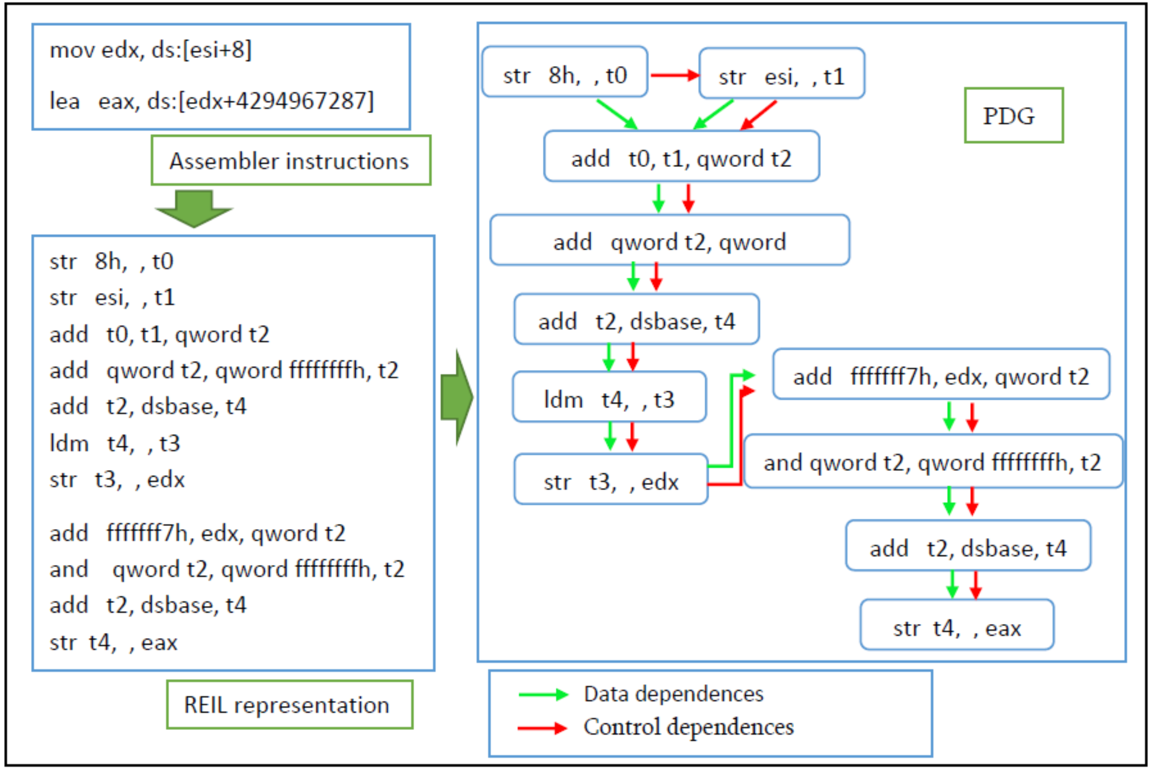
Stage 1


Old binary
Assembler
REIL code
PDGs and CG
Save in database
Ida Pro
Binnavi
New binary
Assembler
REIL code
PDGs and CG
Save in database

REIL (Reverse Engineering Intermediate language)

- Platform independent
- 17 simple instructions (and, add, ldm, stm…)
- No side effects

Program Dependence Graph (PDG, example)



Stage 2

Heuristics

+

Detect similarity based on maximum subgraph detection

MD-index

$$e \in Edges(G) \quad \left\{ \begin{array}{c} z0 \; = \; topologicalorder(src(e)), \\ z1 \; = \; indegree(src(e)), \\ z2 \; = \; outdegree\big(src(e)\big), \\ z3 \; = \; indegree(dest(e)), \\ z4 \; = \; outdegree(dest(e)) \end{array} \right\}$$

hash(e) $= z_0 + z_1\sqrt{2} + z_2\sqrt{3} + z_3\sqrt{5} + z_4\sqrt{7}$

$$\sum_{e \in Edges(Graph)} \frac{1}{\sqrt{hash(e)}}$$

Stage 2 (Heuristics)

1. Match functions based on hash of the original raw function bytes
2. Match CGs edges based on their source and target function's PDGs MD indices
3. Match functions based on a hash of the CG edges with MD-index calculation of destination and source vertices neighbors
4. Match functions based on a MD-index hash of the PDGs edges
5. Match functions based on a hash of the PDGs nodes (considers data dependencies between PDG instructions to group nodes, computes hash for every group and combines them to final hash)
6. Match functions based on based on a prime signature matching of original PDGs instructions (assign prime number to each instruction and then compute product of assigned primes for entire function)

Stage 2 (Maximum common subgraph)

1. For each matched pair of CGs vertices consider their predecessors (successors) : P1 (S1) and P2 (S2)

2. For all pairs of vertices from P1 (S1) and P2 (S2) detect maximum common subgraphs and construct matrix from matched parts

3. Apply Hungarian algorithm on the matrix for finding the best correspondence of PDGs

4. Repeat 1-3 steps until there are not considered pairs of matched vertices

Result demonstration

File : ifret.o

```
ifret.o
 Function name : f(void)$0
* 1) 0 : push_[qword_rbp]
* 2) 1 : mov_[qword_rbp,_qword_rsp]
* 3) 4 : mov_[ss:_[rbp_+_var_C],_6]
* 5) 11 : mov_[ss:_[rbp_+_var_8],_7]
* 4) 18 : mov_[edx,_ss:_[rbp_+_var_8]]
* 7) 21 : mov_[eax,_ss:_[rbp_+_var_C]]
* 6) 24 : add_[eax,_edx]
* 9) 26 : mov_[ss:_[rbp_+_var_4],_eax]
* 8) 29 : mov_[eax,_ss:_[rbp_+_var_4]]
* 10) 32 : pop_[qword_rbp]
* 11) 33 : retn_[]
```

File : ifretFIXED.o

```
ifretFIXED.o
 Function name : fret(void)$0
* 1) 0 : push_[qword_rbp]
* 2) 1 : mov_[qword_rbp,_qword_rsp]
* 3) 4 : mov_[ss:_[rbp_+_var_C],_4]
* 5) 11 : mov_[ss:_[rbp_+_var_8],_7]
* 4) 18 : mov_[edx,_ss:_[rbp_+_var_C]]
* 7) 21 : mov_[eax,_ss:_[rbp_+_var_8]]
* 6) 24 : add_[eax,_edx]
* 9) 26 : mov_[ss:_[rbp_+_var_4],_eax]
* 8) 29 : mov_[eax,_ss:_[rbp_+_var_C]]
    32 : cmp_[eax,_ss:_[rbp_+_var_8]]
    35 : jge_[qword_42]
    37 : mov_[eax,_ss:_[rbp_+_var_4]]
    40 : jmp_[qword_45]
    42 : mov_[eax,_ss:_[rbp_+_var_C]]
* 10) 45 : pop_[qword_rbp]
* 11) 46 : retn_[]
```

Results sequential version

Test names	Versions		Sizes (MB)		Functions' count		Matching time (sec.)	Matched pairs (count)
	old	*new*	*old*	*new*	*old*	*new*		
python	3.5.1	3.5.2	12	12	3944	3951	55	3944
php	7.0.5	7.0.6	29	29	8287	8292	99	8287
libxml2	2.9.2	2.9.3	5.4	5.4	2584	2603	20	2584
openssl	1.0.1r	1.0.1s	2.8	2.9	5395	5430	47	5395
openssl	1.0.1f	1.0.1s	2.2	2.9	5414	5430	48	5414
rsync	3.0.9	3.1.1	1.6	1.8	599	636	8	599
gcc	4.9.0	5.4.0	3.2	3.5	1094	1145	12	1094
git	2.6.0	2.9.5	9.4	9.8	3335	3471	32	3334

Tests are performed on 3.3GHz processor with 4 physical cores

Comparison with BinDiff

Test names	Versions		BinDiff results		Our results		Common part
	old	*new*	*Matched pairs*	*False positives*	*Matched pairs*	*False positives*	
python	3.5.1	3.5.2	3931	36	3944	8	3895
php	7.0.5	7.0.6	8287	16	8287	9	8271
libxml2	2.9.2	2.9.3	2581	4	2584	3	2577
openssl	1.0.1r	1.0.1s	5303	6	5395	6	5373
openssl	1.0.1f	1.0.1s	5413	108	5414	27	5305
rsync	3.0.9	3.1.1	569	148	599	79	420
gcc	4.9.0	5.4.0	1068	208	1094	79	860
git	2.6.0	2.9.5	3335	350	3334	68	2984

Comparison of binaries, which are generated from different compilers

Compiler version	Programs				
	python	openssl	postgresql	libxml	php
gcc 4.8 vs 5.4.0	88.5%	83.5%	88.9%	88.9%	89.4%
gcc 4.8 vs 7.2.0	99%	92%	99.6%	92.6%	99.6%
gcc 5.4 vs 7.2.0	88.6%	88.7%	92%	95%	89%
clang 5.0 vs gcc 4.8.0	99%	90%	99.7%	70%	99%

Result summary

- Comparison of sequential versions which are compiled with the same compile options, true positives > 95%
- Comparison of sequential versions which are compiled with O2 and O3 options, true positives > 90%
- Comparison of sequential versions which are compiled with gcc and clang (linux) with the same option, true positives > 80%
- Comparison of sequential versions which are compiled with O0 and O3 options, true positives < 30%
- Comparison of sequential versions which are compiled in windows and linux – true positives < 30%

Future work

- Detection of old versions of statically linked libraries in binaries
- Mapping of binary to source code
- Reduce false positives

Thank you!