Code deobfuscation by optimization

Branko Spasojević
branko.spasojevic@infigo.hr

INFIGO Information security

Overview

- Why?
- Project goal?
- How?
  - Disassembly
  - Instruction semantics
  - Optimizations
  - Assembling
- Demo!
- Questions?

INFIGO information security

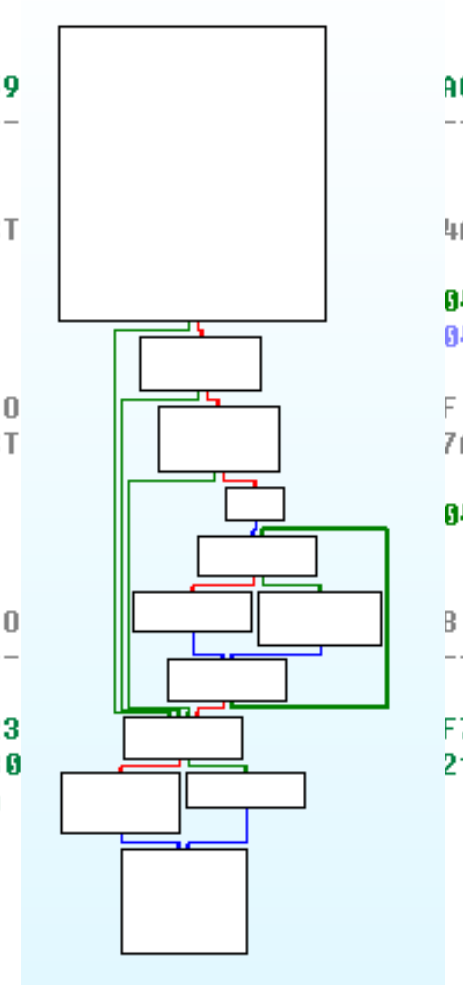
Why?

- To name a few
  - X86 is complex
    - 2 books, ~1600 pages of instructions
  - Obfuscated code = complexity++
  - Disassembly is not always pretty
  - Debugging can help mitigate some problems but not all

Why?

- Can you use graph view?

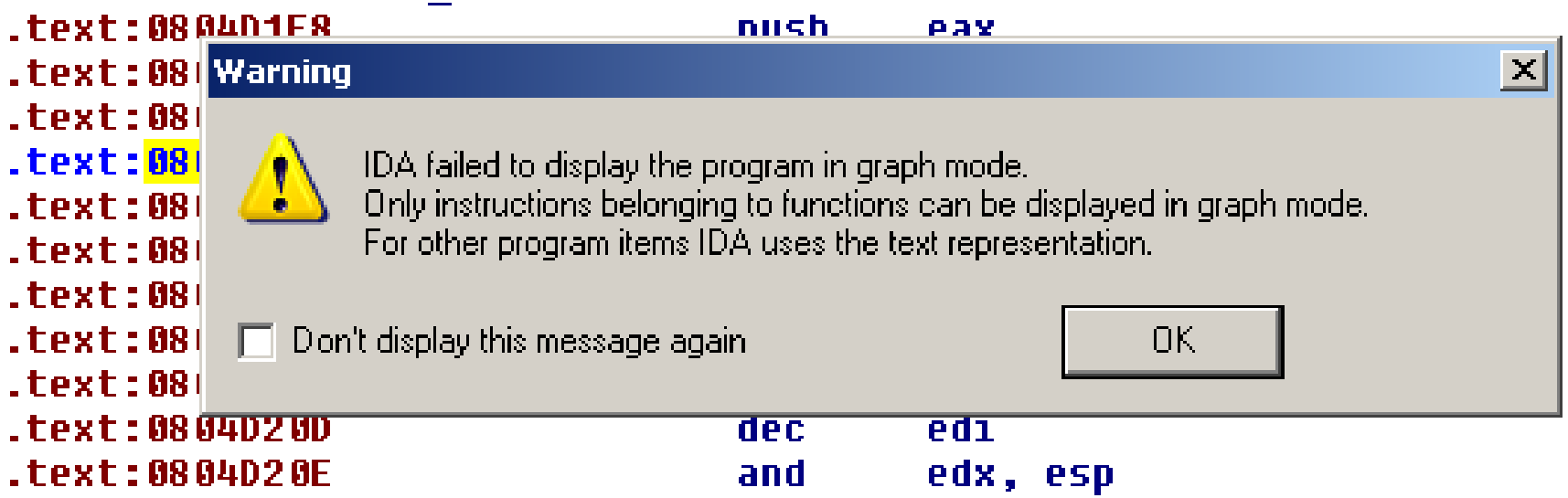
```
.text:0804B065          db 7Bh,
.text:0804B068          dd 71C59                                AC58h
.text:0804B074 ; ---------------                                ----------------------------------------
.text:0804B074          push
.text:0804B075          dec
.text:0804B076 ; START OF FUNCT                                 4AF
.text:0804B076
.text:0804B076 loc_804B076:                                     04C4AF-3C6↓j
.text:0804B076                                                  04C4AF-3CB↓o
.text:0804B076          mov
.text:0804B076 ; END OF FUNCTIO                                 F
.text:0804B07D ; START OF FUNCT                                 7AB
.text:0804B07D
.text:0804B07D loc_804B07D:                                     0497AB+28A9↓j
.text:0804B07D          push
.text:0804B082          retn
.text:0804B082 ; END OF FUNCTIO                                 B
.text:0804B082 ; ---------------                                ----------------------------------------
.text:0804B083          db 0Eh
.text:0804B084          dd 0DD23                                F7DAh, 0C9FC458Bh, 2A6AE9C3h, 92120000h
.text:0804B084          dd 48630                                2144h, 2461F6DCh, 42C3799Bh
.text:0804B0B0          db 0CCh
```



INFIGO information security

Why?

- Now what?



INFIGO information security

Why?

- No public/opensource tool for deobfuscation
- No framework to analyze instruction semantics
- Fun thing to do?
- To speed up things
- Reuse code for some other projects

Project goal?

- Rewrite code to fix disassembly representation problems
- Build framework to analyze instruction tainting and semantics
  - Extend it for automatic deobfuscation
  - Expose API
    - Ease development of heuristic deobfuscation rules and code transformations
    - Experiment with code transformations

How? - Disassembly

- Main disassembly unit is a function
- Function representation should have all instructions visible
- Problems (for reversers)
  - Basic block scattering
    - Not a real problem for disassembler
  - Fake paths in conditional jumps lead to broken disassembly (opaque predicates)
  - Instruction overlapping
    - Not a real problem for disassembler

How? - Disassembly

- JCC path leads to broken disassembly
  - Replace it with `RET`, add comment to instruction and continue
  - This way code can be transformed to a function

```
.text:0804AFE3 9C              pushf
.text:0804AFE4 F9              stc
.text:0804AFE5 0F 82 28+       jb  loc_804D813
.text:0804AFEB 21 57 CF        and [edi-31h], edx
.text:0804AFEE FD              std
.text:0804AFEF 50              push eax
.text:0804AFF0 46              inc esi
.text:0804AFF1 FB              sti
.text:0804AFF2 C3              retn
.text:0804AFF2              sub_804AFE0 endp ; sp-analysis failed

.text:0804AFF4 18 42 15        sbb [edx+15h], al
```

INFIGO information security

How? - Disassembly

- Instruction overlapping hides code paths
  - Disassembly graph should contain all instructions

```
                                ..
:00001000 58                    pop eax     ; 0a04b0d7
:00001001 8D 40 0A              lea eax, [eax+0Ah] ; 0a04b0d8
:00001004 EB 04                 jmp short loc_100A ; 0a04b0db
:00001006                     ; optimized:00001006
:0000100A
:0000100A                     loc_100A:     ; CODE XREF: sub_1000+4↑j
:0000100A FF E0                 jmp eax     ; 0a04b0dc
```

How? - Disassembly

- Function representation
  - Graph
    - One graph represents one function
    - Nodes in graph represent instructions
    - Edges represent control flow
    - IDA disassembly engine used for parsing opcodes
    - Depth first search for path exploring

How? - Disassembly

- Nodes represent Instructions
  - Instruction contains the following information:
    - OriginEA, Mnemonic, Disassembly, Prefix, Operands, Opcode, Operand types...
- Instruction information populated from two sources:
  - Information from IDA API
    - `GetMnem()`, `GetOpnd()`, `GetOpType()`
  - Information derived from `GetDisasm()` API

IDA – Side story

- Mnemonics differ
  - `GetMnem()` != `GetDisasm()`
    - `GetMnem()` returns basic mnemonic e.g. `STOS`
    - `GetDisasm()` returns mnemonic variant `STOSD`
- `GetMnem()` = "`xlat`"
  - `GetOpnd()` = "" but `GetOpType()` = 1

XLAT/XLATB--Table Look-up Translation

Opcode	Instruction	Description
D7	XLAT *m8*	Set AL to memory byte DS:[(E)BX + unsigned AL]
D7	XLATB	Set AL to memory byte DS:[(E)BX + unsigned AL]

How? – Disassembly - Functions

- Function abstracted as a Class
  - Function = Basic Blocks + CFG
  - CFG stored as two graphs
    - References from location (GetRefsFrom)
    - References to location (GetRefsTo)
  - Some of the exposed functions are: `GetRefsFrom()`, `GetRefsTo()`, `DFSFalseTraverseBlocks()`, ...
  - CFG optimizations mainly operate on Function class

How? – Disassembly – Basic Blocks

- Basic Block implemented as linked list in Function Class
  - Each entry is an Instruction Class instance
  - Instruction stores relevant instruction data:
    - Prefix, mnemonic, operands, types, values, comments...
- Stores instruction information from two sources:
  - IDA `GetOp*()` functions
  - Parsing of `GetDisasm()` string, regex style ☺

How? – Instruction Semantics

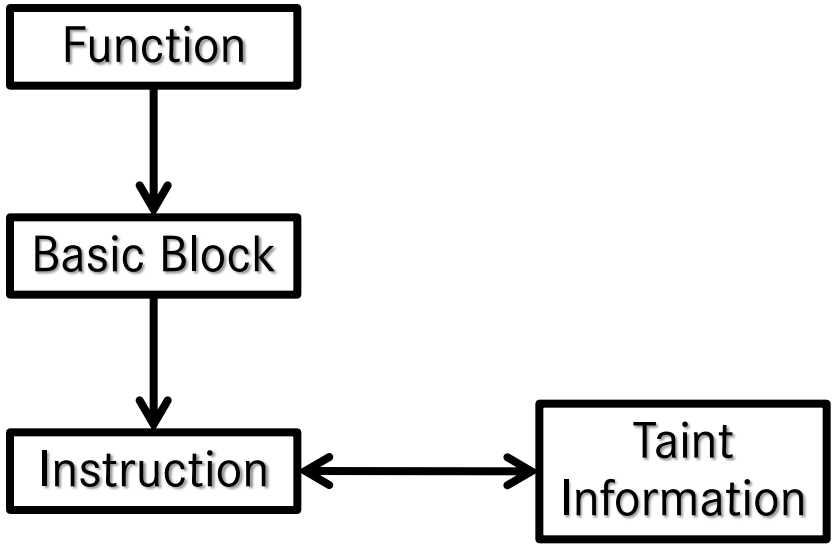
- Semantics?
- Operands:
  - Visible, hidden, flags
  - What you see: `IMUL ECX`
  - What you get: `EDX:EAX = EAX * ECX` + oszapc
- 695 different mnemonics (not including different opcodes and prefix combinations)
- MazeGen's XML (ref.x86asm.net) saves the day
  - Read the docs, many useful fields and attributes

How? – Instruction Semantics

- Implemented in TaintInstr Class
  - Contains information about:
    - Source and destination operands (displayed and hidden)
    - Flag modification
    - Side effects (e.g. `ESP+4` for `POP`)
    - Ring association (`LLDT`…)
- BlockTainting Class automates process on blocks
- Tainting information necessary to perform safe optimizations

How? – Overall

- Function
  - CFG information
- Basic Blocks
  - Instruction grouping
- Instruction
  - Opcode information
- Taint information
  - Operands information



How? – Optimizations

- We have foundation to analyze code
- It's time to exploit some algorithms
- Four main types of optimizations:
  - CFG reductions
    - JCC reduction
    - JMP merging
  - Dead code removal
  - Heuristic rules
  - Constant propagation and folding (TODO)

INFIGO information security

How? – Optimizations - CFG

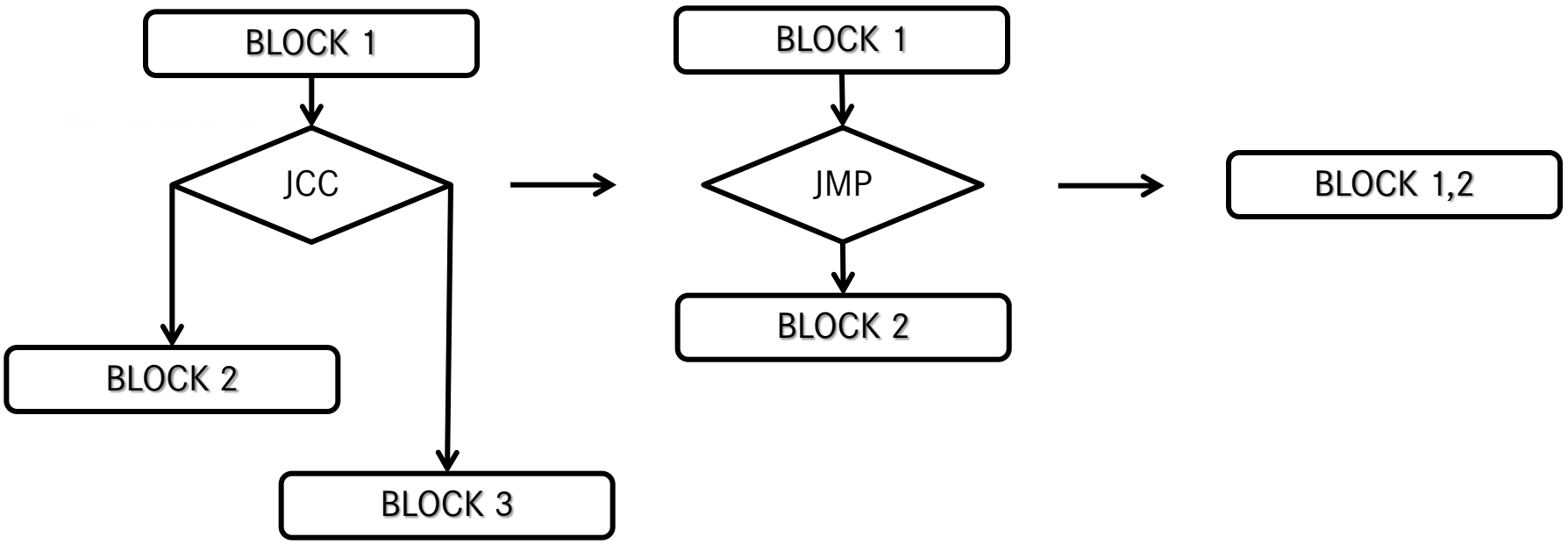
- JCC reductions
  - JCC path depends on flags status
  - Use tainting information to detect constant flags
  - Replace JCC with `JMP`
  - e.g. `AND`, clears OF and CF flags
    - [JO, JNO, JC, JB, ...] all take single path
- Results in smaller graphs and better/more precise disassembly
- Removes fake paths that break creation of functions and mess up disassembly

How? – Optimizations - CFG

- JMP merging
  - If current block ends in `JMP` and
  - next block has only single reference then merge them
- Increases block size, reduces CFG complexity
- Code optimizations are block based so merging can influence a lot the final code quality

How? – Optimizations - CFG

- Two staged CFG optimization:



How? – Optimizations – Dead code

- Dead code
  - Every instruction whose execution doesn't modify programs final state or control flow
  - Every instruction of a block in which ALL taints get overwritten before being used
- Removing
  - If instruction taints memory -> leave it
  - If instruction changes control flow -> leave it
  - For every instruction in a block
    - Get instruction taints (modified data)
      - If all instruction taints are tainted again before getting used, remove instruction and continue

How? – Optimizations – Rule based

- There is obfuscation which bothers you and isn't automagically removed?
- Adding rule based optimization is easy?

```
def RET2JMP(self, bb):
    instr = bb[-1]
    if instr.GetMnem().lower().find("ret") >= 0:
        for (ref, path) in self.function.GetRefsFrom(instr.GetOriginEA()):
            if ref != None:
                instr.SetMnem("jmp")
                instr.SetComment("-replaced[RET]")
                instr.SetDisasm("jmp %08xh" % ref)
                instr.SetIsModified(True)
                find_push = bb[-2]
                if find_push.GetMnem().lower() == "push":
                    self.function.RemoveInstruction(find_push.GetOriginEA(), bb[0].GetOriginEA())
```

INFIGO information security

How? – Assembling

- `idaapi.Assemble()`?
- "…, we do not support it. It is very limited and can handle only some trivial instructions. We do not have plans to improve or modify it." Ilfak
- Sensitive to syntax
- Remember `GetMnem()`?
- IDA before 5.5 can't assemble JCC easily…
- BUT, it can handle most instructions if you play nice (Batch(1) is your friend)

DEMO TIME!

Conclusion

- It can remove static obfuscations
- You can feed it data from disassembler for better results
  - Tool chaining!
- Work in progress
  - It has bugs :D send samples and will fix them
  - Got ideas? Share them.
- You can extend, improve, contribute!
- Shouts: n00ne, bzdrnja, tox, haarp, MazeGen, RolfRolles, all gnoblets, reddit/RE

Thank you for your attention!
Questions?

http://code.google.com/p/optimice

INFIGO Information security