Automated static deobfuscation in the context of Reverse Engineering

Sebastian Porst (sebastian.porst@zynamics.com)
Christian Ketterer (cketti@gmail.com)

SOURCE Barcelona 2009

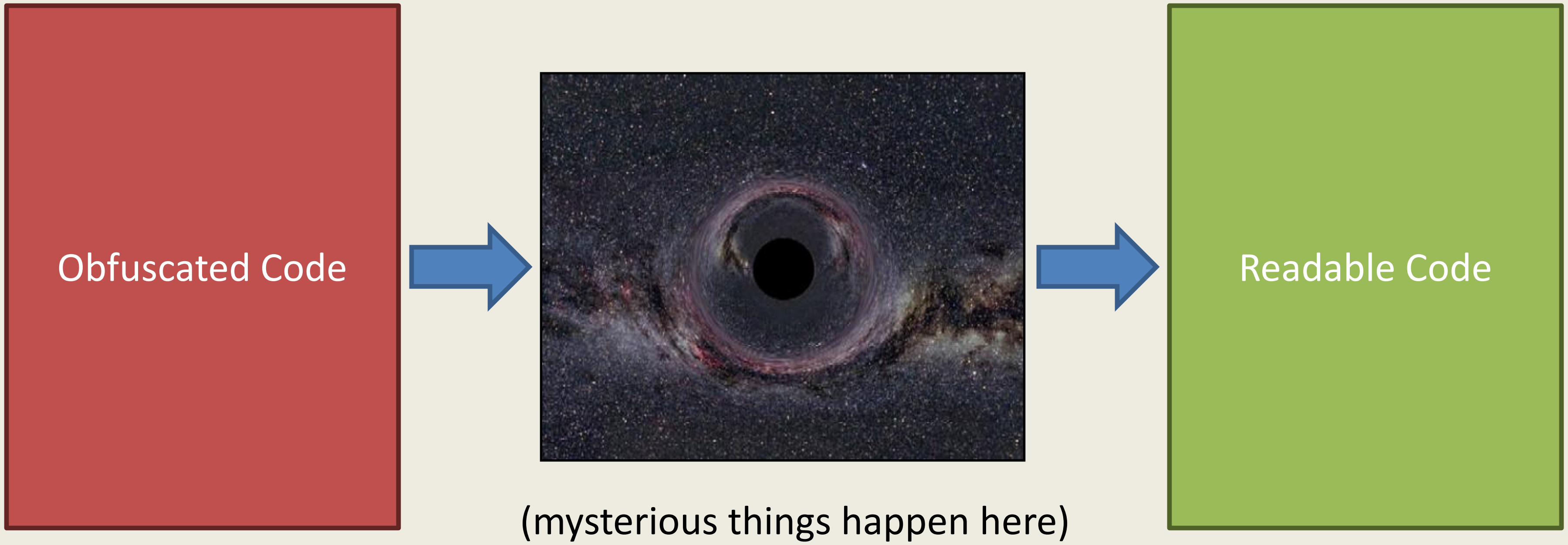
Sebastian

- zynamics GmbH
- Lead Developer
  - BinNavi
  - REIL/MonoREIL

Christian

- Student
- University of Karlsruhe
- Deobfuscation

This talk



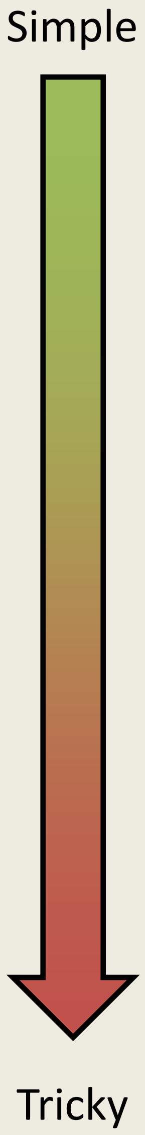
(mysterious things happen here)

20% 40% 40%

Motivation

- Combat common obfuscation techniques
- Can it be done?
- Will it produce useful results?
- Can it be integrated into our technology stack?

Examples of Obfuscation



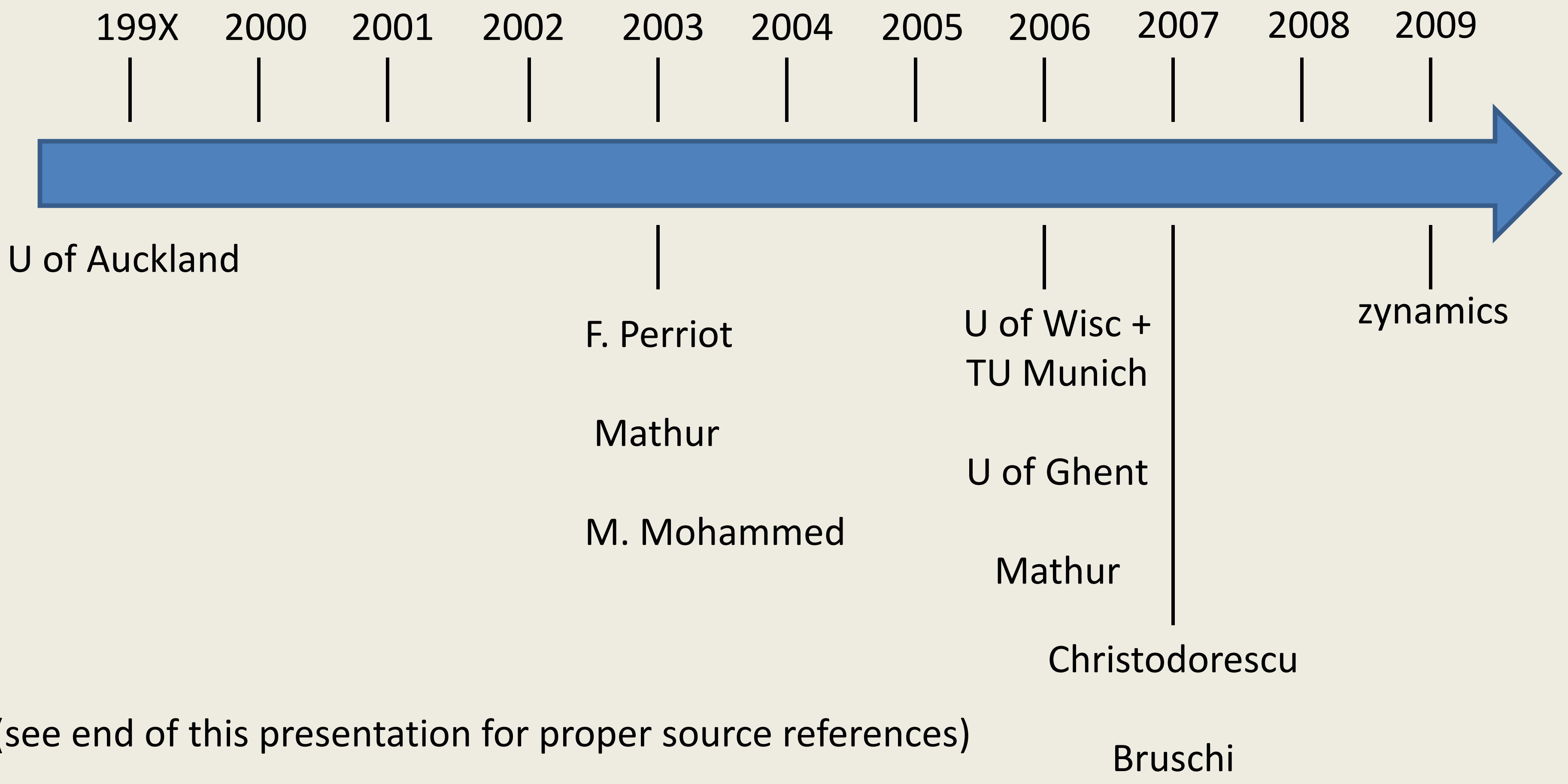
- Jump chains
- Splitting calculations
- Garbage code insertion
- Predictable branches

- Self-modifying code
- Control-flow flattening
- Opaque predicates
- Code parallelization
- Virtual Machines
- …

Our Deobfuscation Approach

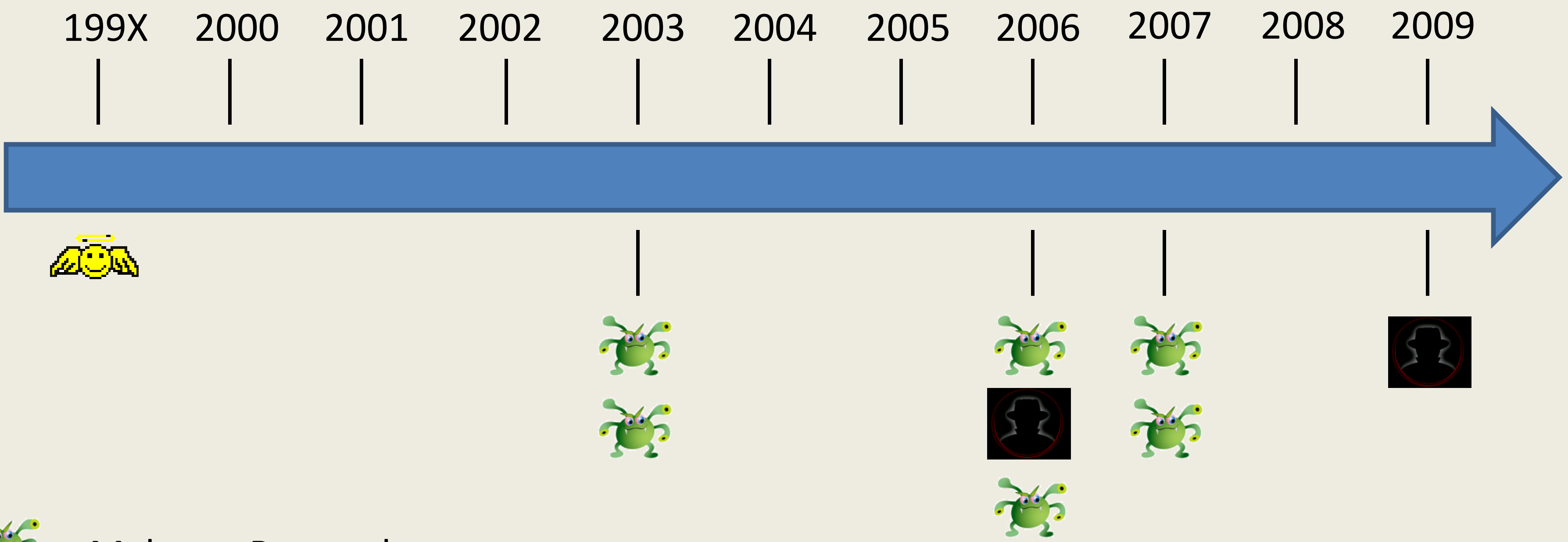
I. Copy ancient algorithms from compiler theory books
II. Translate obfuscated assembly code to REIL
III. Run algorithms on REIL code
IV. Profit (?)

We‘re late in the game ...



(see end of this presentation for proper source references)

... but



Malware Research

Defensive Reverse Engineering

Offensive Reverse Engineering

REIL

- Reverse Engineering Intermediate Language
- Specifically designed for Reverse Engineering
- Design Goal: As simple as possible, but not simpler
- In use since 2007

Uses of REIL

Register Tracking: Helps Reverse Engineers follow data flow through code
(Never officially presented)

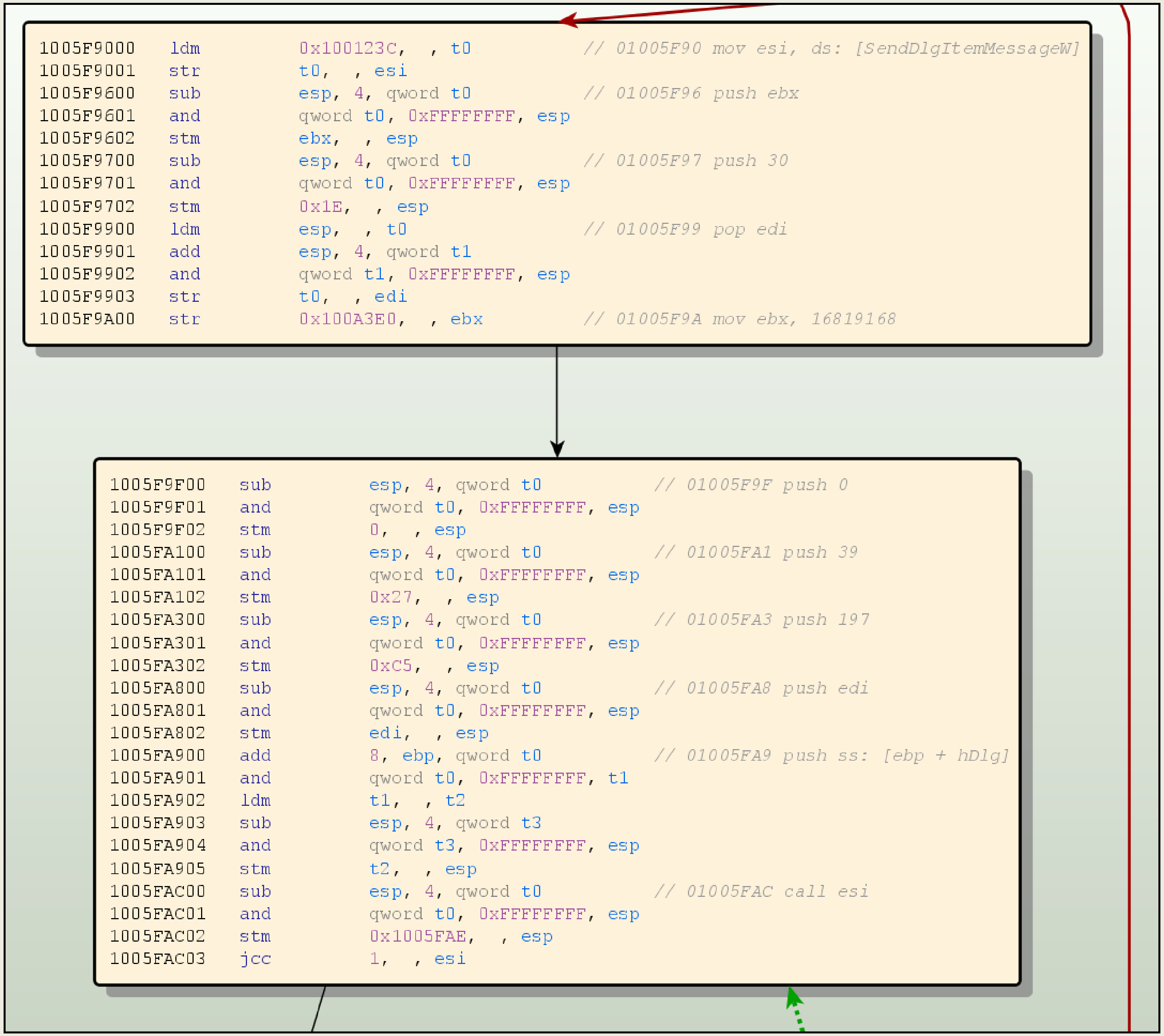
Index Underflow Detection: Automatically find negative array accesses
(CanSecWest 2009, Vancouver)

Automated Deobfuscation: Make obfuscated code more readable
(SOURCE Barcelona 2009, Barcelona)

ROP Gadget Generator: Automatically generates return-oriented shellcode
(Work in progress; scheduled for Q1/2010)

The REIL Instruction Set

Arithmetical	Bitwise	Data Transfer	Logical	Other
ADD	AND	STR	BISZ	NOP
SUB	OR	LDM	JCC	UNDEF
MUL	XOR	STM		UNKN
DIV				
MOD				
BSH				



```
1005F9000   ldm       0x100123C,  , t0            // 01005F90 mov esi, ds: [SendDlgItemMessageW]
1005F9001   str       t0,  , esi
1005F9600   sub       esp, 4, qword t0            // 01005F96 push ebx
1005F9601   and       qword t0, 0xFFFFFFFF, esp
1005F9602   stm       ebx,  , esp
1005F9700   sub       esp, 4, qword t0            // 01005F97 push 30
1005F9701   and       qword t0, 0xFFFFFFFF, esp
1005F9702   stm       0x1E,  , esp
1005F9900   ldm       esp,  , t0                  // 01005F99 pop edi
1005F9901   add       esp, 4, qword t1
1005F9902   and       qword t1, 0xFFFFFFFF, esp
1005F9903   str       t0,  , edi
1005F9A00   str       0x100A3E0,  , ebx           // 01005F9A mov ebx, 16819168
```

```
1005F9F00   sub       esp, 4, qword t0            // 01005F9F push 0
1005F9F01   and       qword t0, 0xFFFFFFFF, esp
1005F9F02   stm       0,  , esp
1005FA100   sub       esp, 4, qword t0            // 01005FA1 push 39
1005FA101   and       qword t0, 0xFFFFFFFF, esp
1005FA102   stm       0x27,  , esp
1005FA300   sub       esp, 4, qword t0            // 01005FA3 push 197
1005FA301   and       qword t0, 0xFFFFFFFF, esp
1005FA302   stm       0xC5,  , esp
1005FA800   sub       esp, 4, qword t0            // 01005FA8 push edi
1005FA801   and       qword t0, 0xFFFFFFFF, esp
1005FA802   stm       edi,  , esp
1005FA900   add       8, ebp, qword t0            // 01005FA9 push ss: [ebp + hDlg]
1005FA901   and       qword t0, 0xFFFFFFFF, t1
1005FA902   ldm       t1,  , t2
1005FA903   sub       esp, 4, qword t3
1005FA904   and       qword t3, 0xFFFFFFFF, esp
1005FA905   stm       t2,  , esp
1005FAC00   sub       esp, 4, qword t0            // 01005FAC call esi
1005FAC01   and       qword t0, 0xFFFFFFFF, esp
1005FAC02   stm       0x1005FAE,  , esp
1005FAC03   jcc       1,  , esi
```

Why REIL?

- Simplifies input code
- Makes effects obvious
- Makes algorithms platform-independent

MonoREIL

- Monotone Framework for REIL
- Based on Abstract Interpretation
- Used to write static code analysis algorithms

http://www.flickr.com/photos/wedrrc/3586908193/

Why MonoREIL?

- In General: Makes complicated algorithms simple (trade brain effort for runtime)
- Deobfuscator: Wrong choice really, but we wanted more real-life test cases for MonoREIL

Building the Deobfuscator

- Java
- BinNavi Plugin
- REIL + MonoREIL

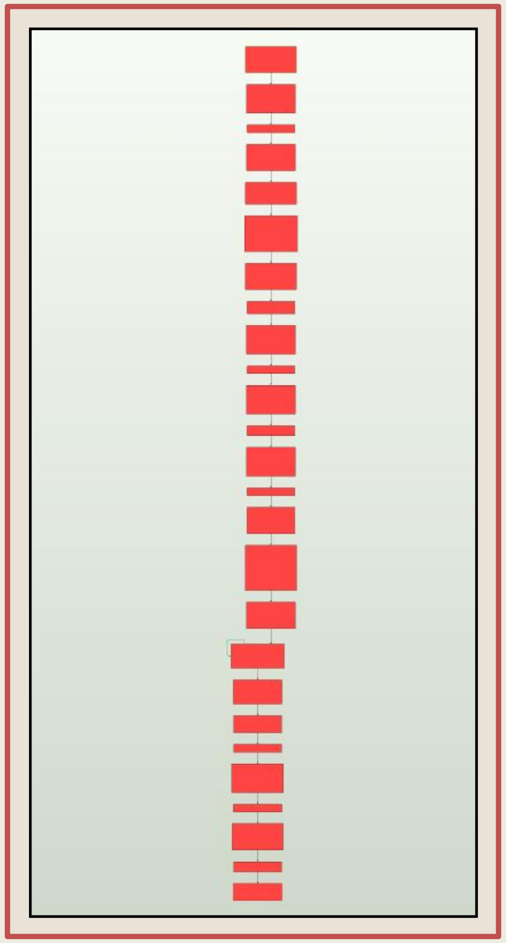
http://www.flickr.com/photos/mattimattila/3602654187/

Block Merging

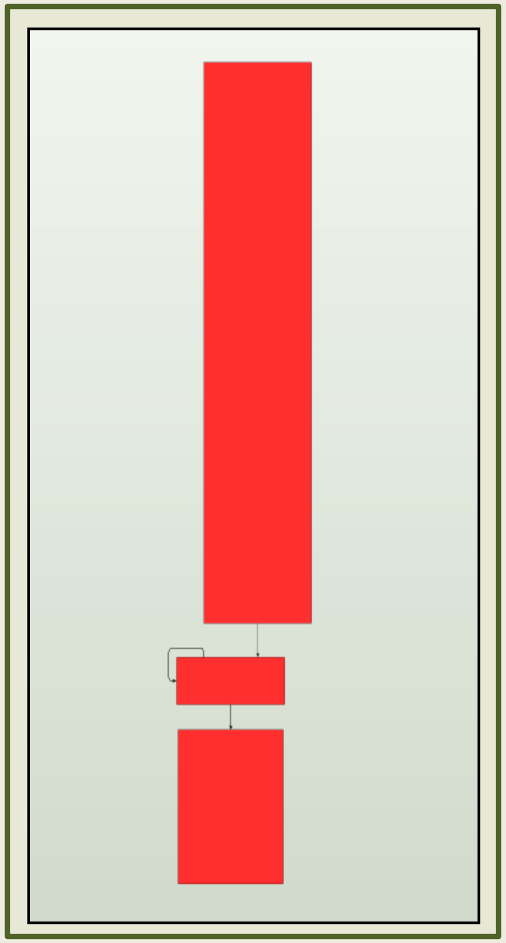
- Long chains of basic blocks ending with unconditional jumps
- Confusing to follow in text-based disassemblers
- Advantage of higher abstraction level in BinNavi
  - Block merging is purely cosmetic

Block Merging

Before



After

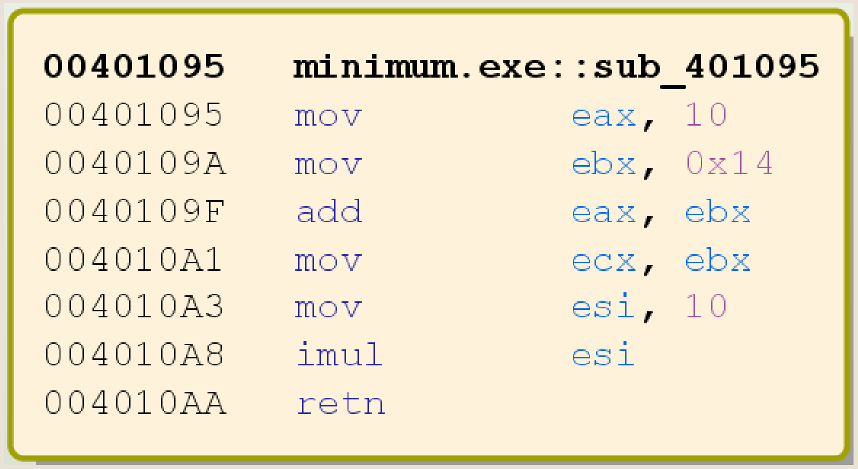


Constant Propagation and Folding

- Two different concepts
- One algorithm in our implementation
- Partial evaluation of the input code

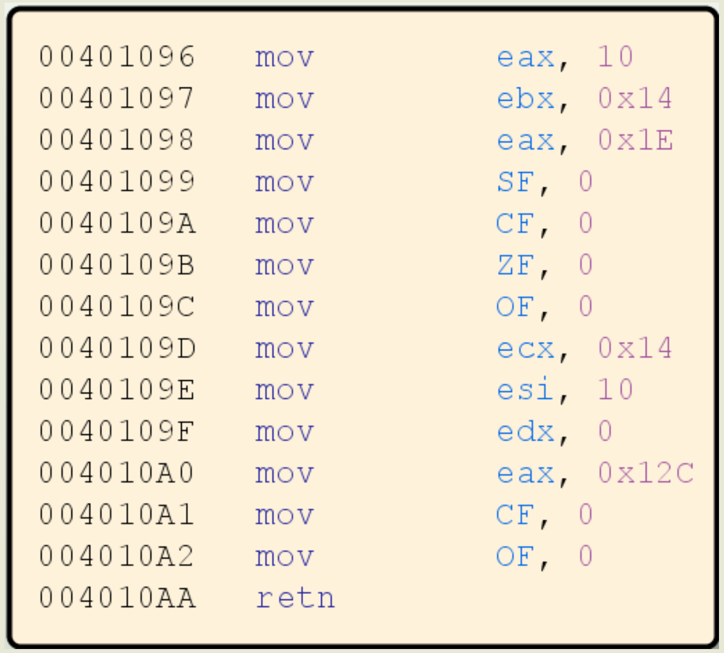
Constant Propagation and Folding

Before



```
00401095    minimum.exe::sub_401095
00401095    mov           eax, 10
0040109A    mov           ebx, 0x14
0040109F    add           eax, ebx
004010A1    mov           ecx, ebx
004010A3    mov           esi, 10
004010A8    imul          esi
004010AA    retn
```

After



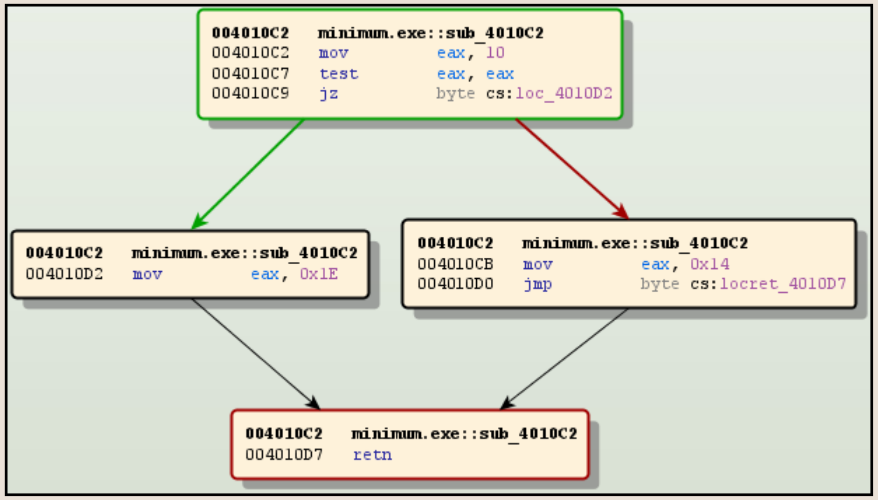
```
00401096    mov           eax, 10
00401097    mov           ebx, 0x14
00401098    mov           eax, 0x1E
00401099    mov           SF, 0
0040109A    mov           CF, 0
0040109B    mov           ZF, 0
0040109C    mov           OF, 0
0040109D    mov           ecx, 0x14
0040109E    mov           esi, 10
0040109F    mov           edx, 0
004010A0    mov           eax, 0x12C
004010A1    mov           CF, 0
004010A2    mov           OF, 0
004010AA    retn
```

Dead Branch Elimination

- Removes branches that are never executed
  - Turns conditional jumps into unconditional jumps
  - Removes code from unreachable branch
- Requires constant propagation/folding

Dead Branch Elimination

Before



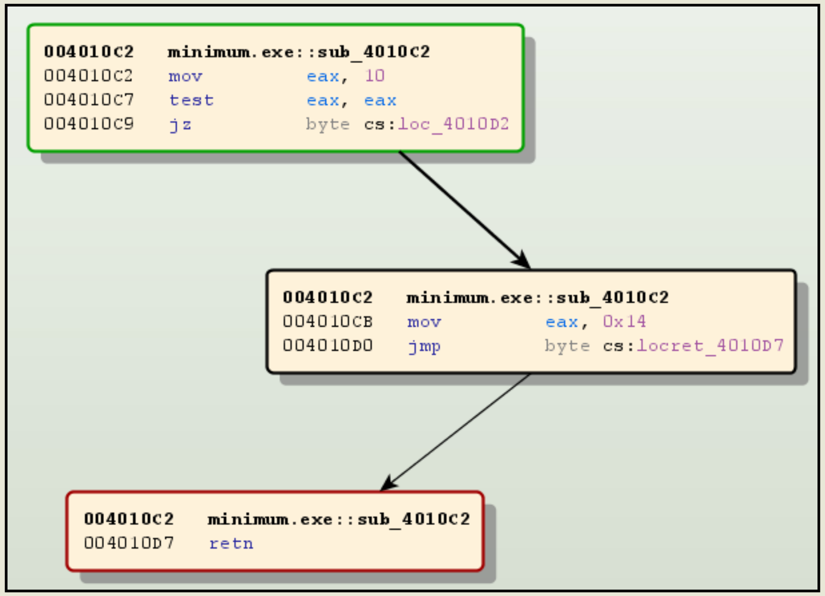
```
004010C2   minimum.exe::sub_4010C2
004010C2   mov       eax, 10
004010C7   test      eax, eax
004010C9   jz        byte cs:loc_4010D2
```

```
004010C2   minimum.exe::sub_4010C2
004010D2   mov       eax, 0x1E
```

```
004010C2   minimum.exe::sub_4010C2
004010CB   mov       eax, 0x14
004010D0   jmp       byte cs:locret_4010D7
```

```
004010C2   minimum.exe::sub_4010C2
004010D7   retn
```

After



```
004010C2   minimum.exe::sub_4010C2
004010C2   mov       eax, 10
004010C7   test      eax, eax
004010C9   jz        byte cs:loc_4010D2
```

```
004010C2   minimum.exe::sub_4010C2
004010CB   mov       eax, 0x14
004010D0   jmp       byte cs:locret_4010D7
```

```
004010C2   minimum.exe::sub_4010C2
004010D7   retn
```

Dead Code Elimination

- Removes code that computes unused values
- Gets rid of inserted garbage code
- Cleans up after constant propagation/folding

Dead Code Elimination

Before

```
004010AB    minimum.exe::sub_4010AB
004010AB    mov            eax, 10
004010B0    mov            ecx, 5
004010B5    mov            edx, 0x14
004010BA    mov            ecx, 10
004010BF    imul           ecx
004010C1    retn
```

After

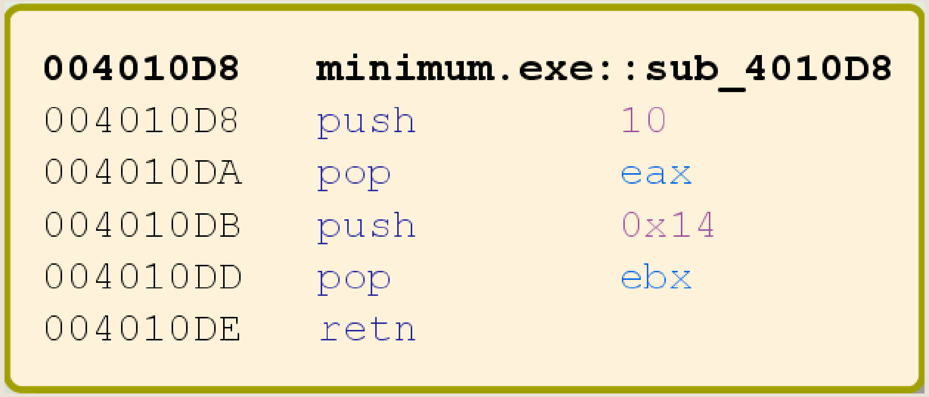
```
004010AC    mov          ecx, 10
004010AD    mov          eax, 0x64
004010AE    mov          edx, 0
004010AF    mov          CF, 0
004010B0    mov          OF, 0
004010C1    retn
```

Dead Store Elimination

- Comparable to dead code elimination
- Removes useless memory write accesses
- Limited to stack access in our implementation
- Only platform-specific part of our optimizer

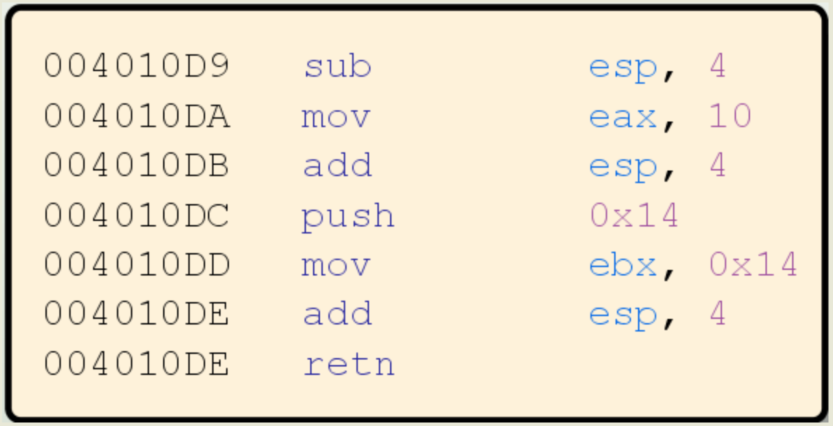
Dead Store Elimination

Before



```
004010D8    minimum.exe::sub_4010D8
004010D8    push        10
004010DA    pop         eax
004010DB    push        0x14
004010DD    pop         ebx
004010DE    retn
```

After



```
004010D9    sub         esp, 4
004010DA    mov         eax, 10
004010DB    add         esp, 4
004010DC    push        0x14
004010DD    mov         ebx, 0x14
004010DE    add         esp, 4
004010DE    retn
```

Suddenly it dawned us:

Deobfuscation for RE brings new problems which do not exist in other areas

Let‘s get some help

Perfect Deobfuscation Oracle

Problem: Side effects

```
push 10
pop eax
```

```
mov eax, 10
```

Removed code was used

- in a CRC32 integrity check
- as key of a decryption routine
- as part of an anti-debug check
- ...

Problem: Code Blowup

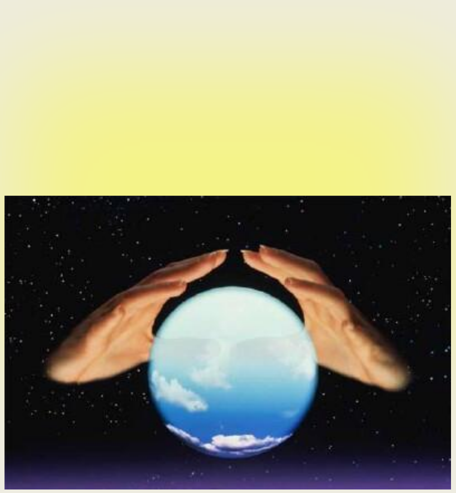
```
mov eax, 10
add eax, 10
```

```
mov eax, 20
clc
...
```

Good luck setting
- AF
- CF
- OF
- PF
- ZF

Problem: Moving addresses

```
0000: jmp ecx
0002: push 10
0003: pop eax
```

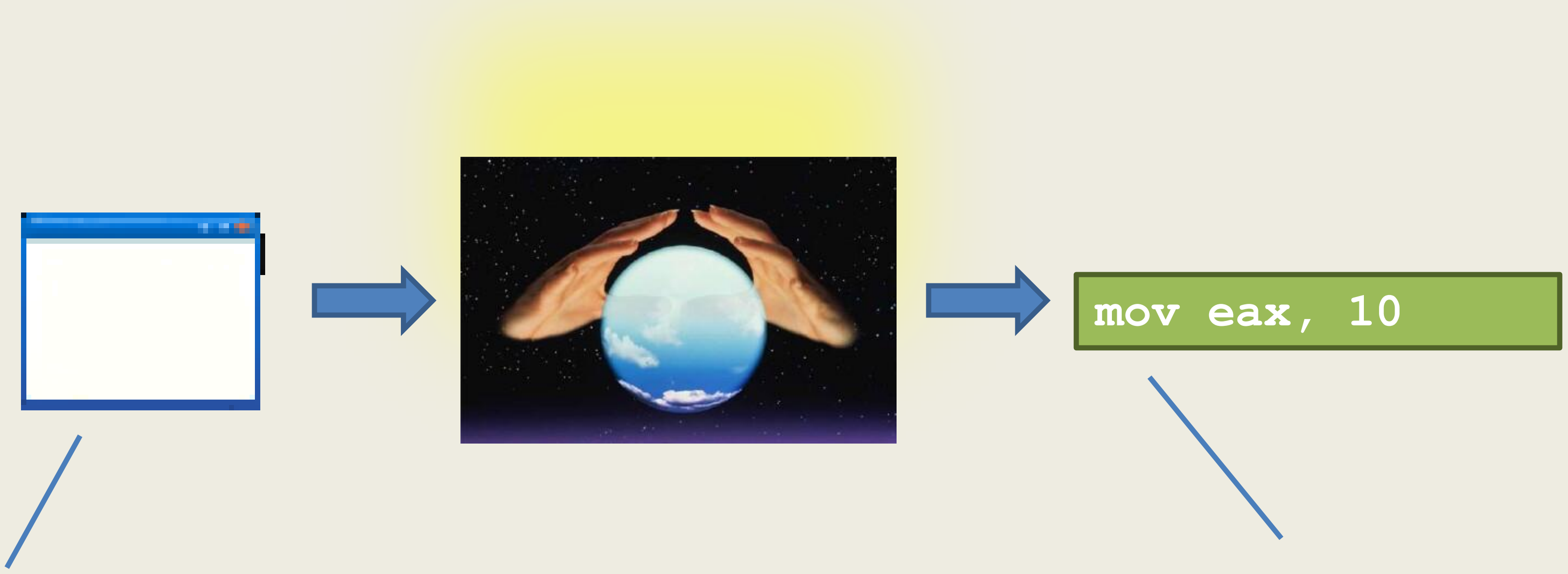


```
0000: jmp ecx
0002: mov eax, 10
```

ecx is 0003 but static analysis can not know this

we just missed the pop instruction

Problem: Inability to debug



```
mov eax, 10
```

Executable Input File

Deobfuscated list of Instructions but no executable file

The only way to solve all* problems:

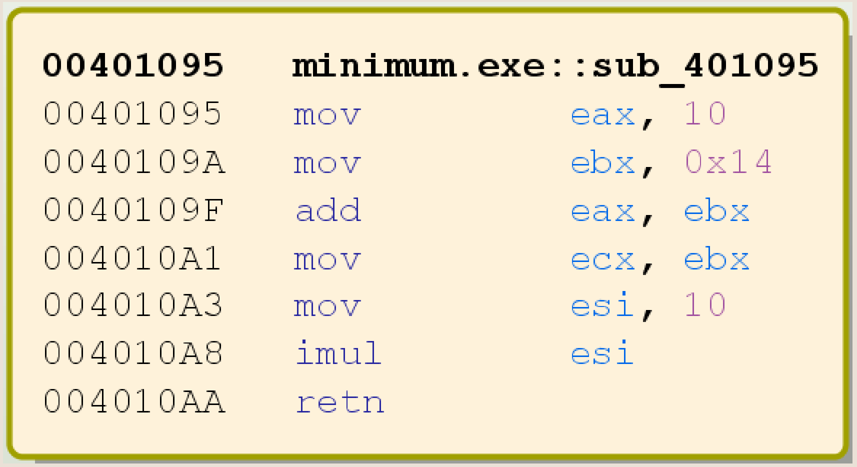
A full-blown native code compiler with an integrated optimizer

Too much work, maybe we can approximate ...

* except for the side-effects issue

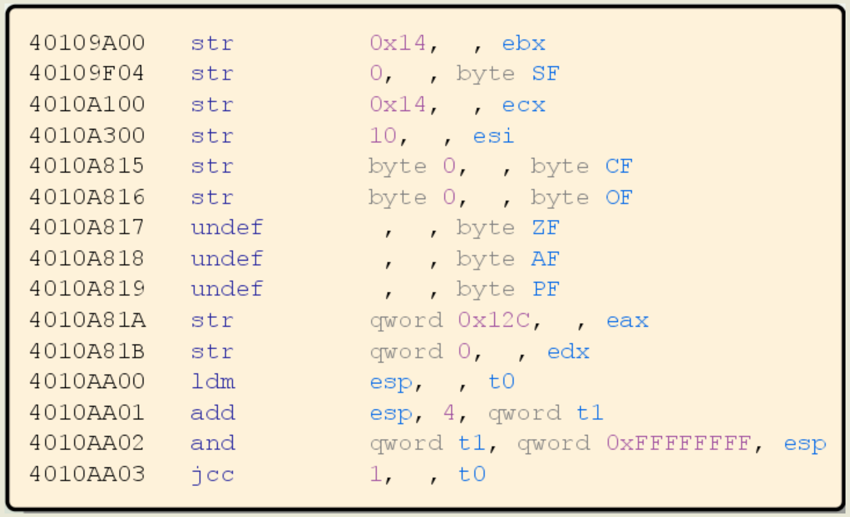
Only generate optimized REIL code

Before



```
00401095    minimum.exe::sub_401095
00401095    mov          eax, 10
0040109A    mov          ebx, 0x14
0040109F    add          eax, ebx
004010A1    mov          ecx, ebx
004010A3    mov          esi, 10
004010A8    imul         esi
004010AA    retn
```

After



```
40109A00   str      0x14,  , ebx
40109F04   str      0,  , byte SF
4010A100   str      0x14,  , ecx
4010A300   str      10,  , esi
4010A815   str      byte 0,  , byte CF
4010A816   str      byte 0,  , byte OF
4010A817   undef     ,  , byte ZF
4010A818   undef     ,  , byte AF
4010A819   undef     ,  , byte PF
4010A81A   str      qword 0x12C,  , eax
4010A81B   str      qword 0,  , edx
4010AA00   ldm      esp,  , t0
4010AA01   add      esp, 4, qword t1
4010AA02   and      qword t1, qword 0xFFFFFFFF, esp
4010AA03   jcc      1,  , t0
```

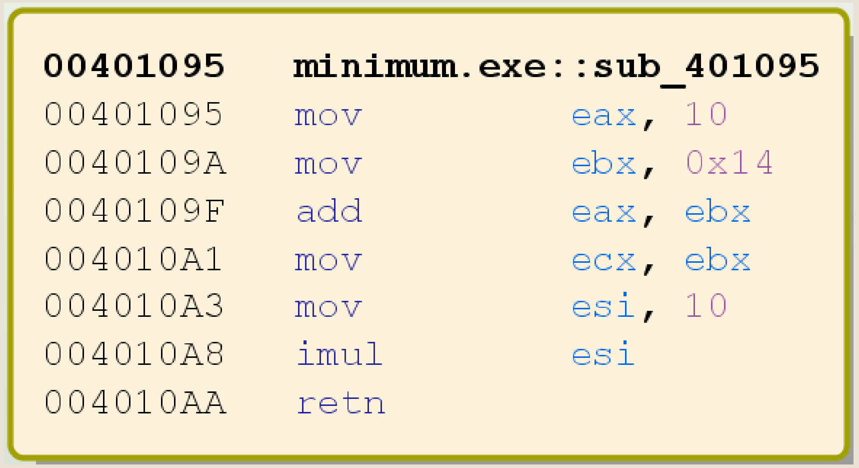
Only generate optimized REIL code

- Produces excellent input for other analysis algorithms
- Code blow-up solved
- Keeps address/instruction mapping
- Code can not be debugged natively but interpreted

- Side effects problem remains
- Pretty much unreadable for human reverse engineers

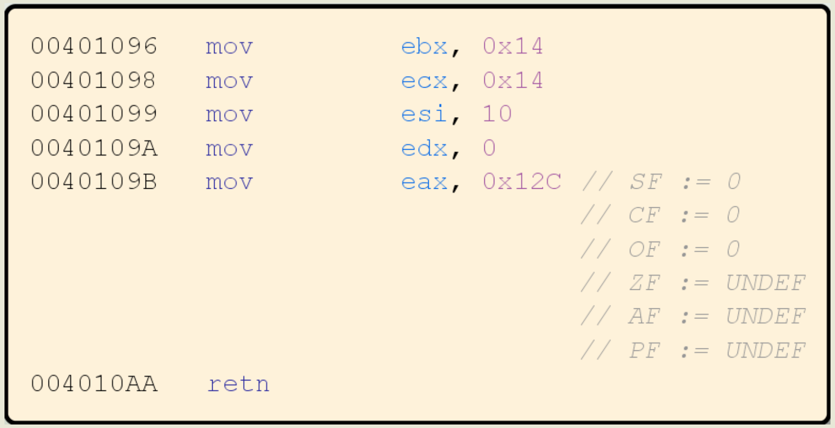
Effect comments

Before



```
00401095    minimum.exe::sub_401095
00401095    mov          eax, 10
0040109A    mov          ebx, 0x14
0040109F    add          eax, ebx
004010A1    mov          ecx, ebx
004010A3    mov          esi, 10
004010A8    imul         esi
004010AA    retn
```

After



```
00401096    mov          ebx, 0x14
00401098    mov          ecx, 0x14
00401099    mov          esi, 10
0040109A    mov          edx, 0
0040109B    mov          eax, 0x12C  // SF := 0
                                     // CF := 0
                                     // OF := 0
                                     // ZF := UNDEF
                                     // AF := UNDEF
                                     // PF := UNDEF
004010AA    retn
```

Effect comments

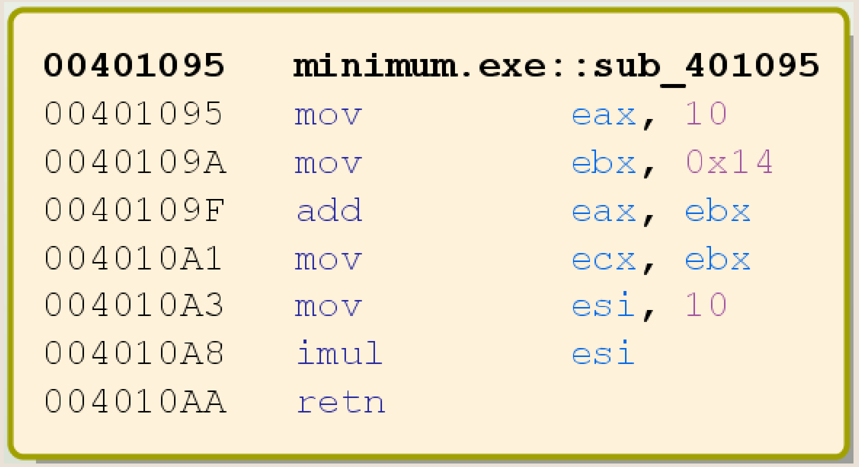
- Results can easily be used by human reverse engineers
- Code blow-up solved

- Side effects problem remains
- Address mapping problem
- Code can not be debugged
- Comments have semantic meaning

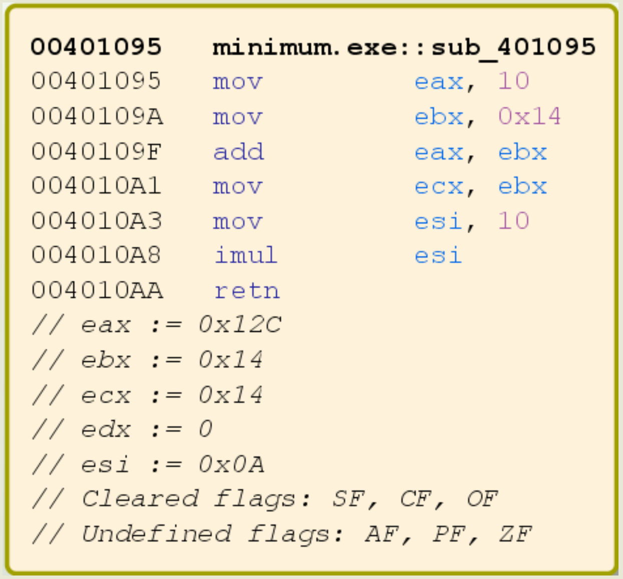
Extract formulas from code

Before

After



```
00401095    minimum.exe::sub_401095
00401095    mov             eax, 10
0040109A    mov             ebx, 0x14
0040109F    add             eax, ebx
004010A1    mov             ecx, ebx
004010A3    mov             esi, 10
004010A8    imul            esi
004010AA    retn
```



```
00401095    minimum.exe::sub_401095
00401095    mov             eax, 10
0040109A    mov             ebx, 0x14
0040109F    add             eax, ebx
004010A1    mov             ecx, ebx
004010A3    mov             esi, 10
004010A8    imul            esi
004010AA    retn
// eax  := 0x12C
// ebx  := 0x14
// ecx  := 0x14
// edx  := 0
// esi  := 0x0A
// Cleared flags: SF, CF, OF
// Undefined flags: AF, PF, ZF
```

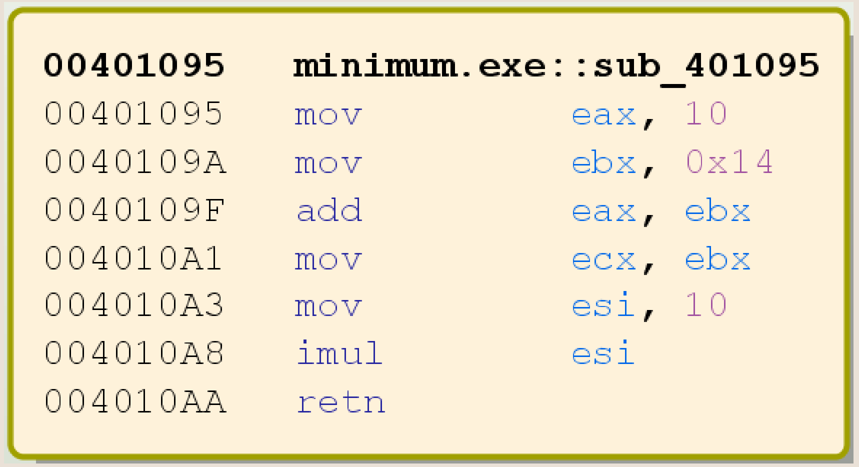
Extract formulas from code

- Results can easily be used by human reverse engineers
- No code generation necessary, only extraction of semantic information
- Solves all problems because original program remains unchanged

- Not really deobfuscation (but produces similar result?)

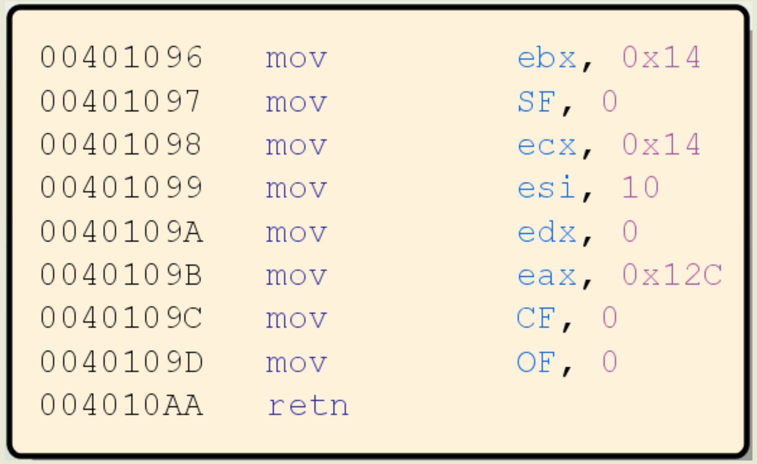
Implement a small pseudo-compiler

Before



```
00401095    minimum.exe::sub_401095
00401095    mov            eax, 10
0040109A    mov            ebx, 0x14
0040109F    add            eax, ebx
004010A1    mov            ecx, ebx
004010A3    mov            esi, 10
004010A8    imul           esi
004010AA    retn
```

After



```
00401096    mov            ebx, 0x14
00401097    mov            SF, 0
00401098    mov            ecx, 0x14
00401099    mov            esi, 10
0040109A    mov            edx, 0
0040109B    mov            eax, 0x12C
0040109C    mov            CF, 0
0040109D    mov            OF, 0
004010AA    retn
```

Implement a small pseudo-compiler

- This is what we did
- Closest thing to the real deal
- Code blow-up is solved
  - Partially
- Natively debug the output
  - not in our case
  - pseudo x86 instructions

- Side effects problem remains
- Address mapping problem remains
- Why not go for a complete compiler?

Economic value in creating a complete optimizing compiler for RE?

Not for us ☹

- Small company
- Limited market
- Wrong approach?

Alternative Approaches

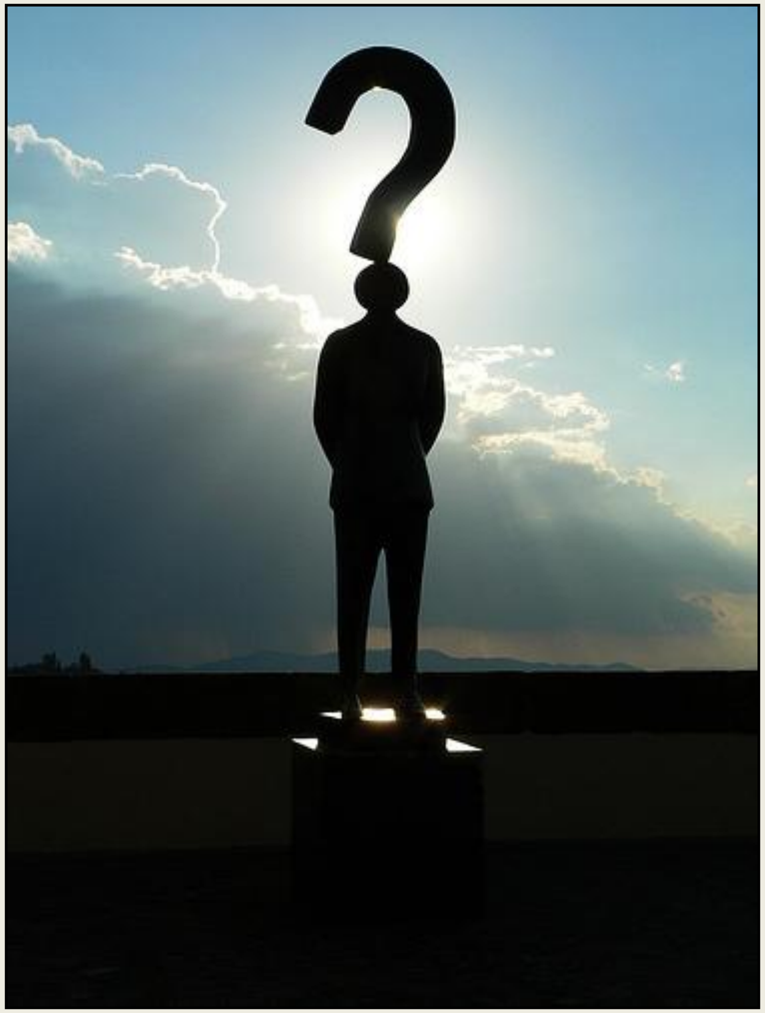
- Deobfuscator built into disassembler
- REIL-based formula extraction
- Hex-Rays Decompiler
- Code optimization and generation based on LLVM
- Emulation / Dynamic deobfuscation

Conclusion

- The concept of static deobfuscation is sound
  - Except for things like side-effects, SMC, ...
- A lot of work
- Expression reconstruction might be much easier and still produce comparable results

Related work

- A taxonomy of obfuscating transformations
- Defeating polymorphism through code optimization
- Code Normalization for Self-Mutating Malware
- Software transformations to improve malware detection
- Zeroing in on Metamorphic Computer Viruses
- …



http://www.flickr.com/photos/marcobellucci/3534516458/