The (Long) Journey To A Multi-Architecture Disassembler

Nicolas Falliere, **Joan Calvet**, Cedric Lucas
PNF Software

REcon Montréal 2019

Who Are We?

- PNF Software: small company, founded in 2013
- Main activity: developing JEB decompiler

Nicolas Falliere

Architect, designer, developer, reverser & janitor.

Joan Calvet

Developer, reverse engineer.

Cedric Lucas

Software engineer.

You?

We are hiring

PNF Software In Three Dates

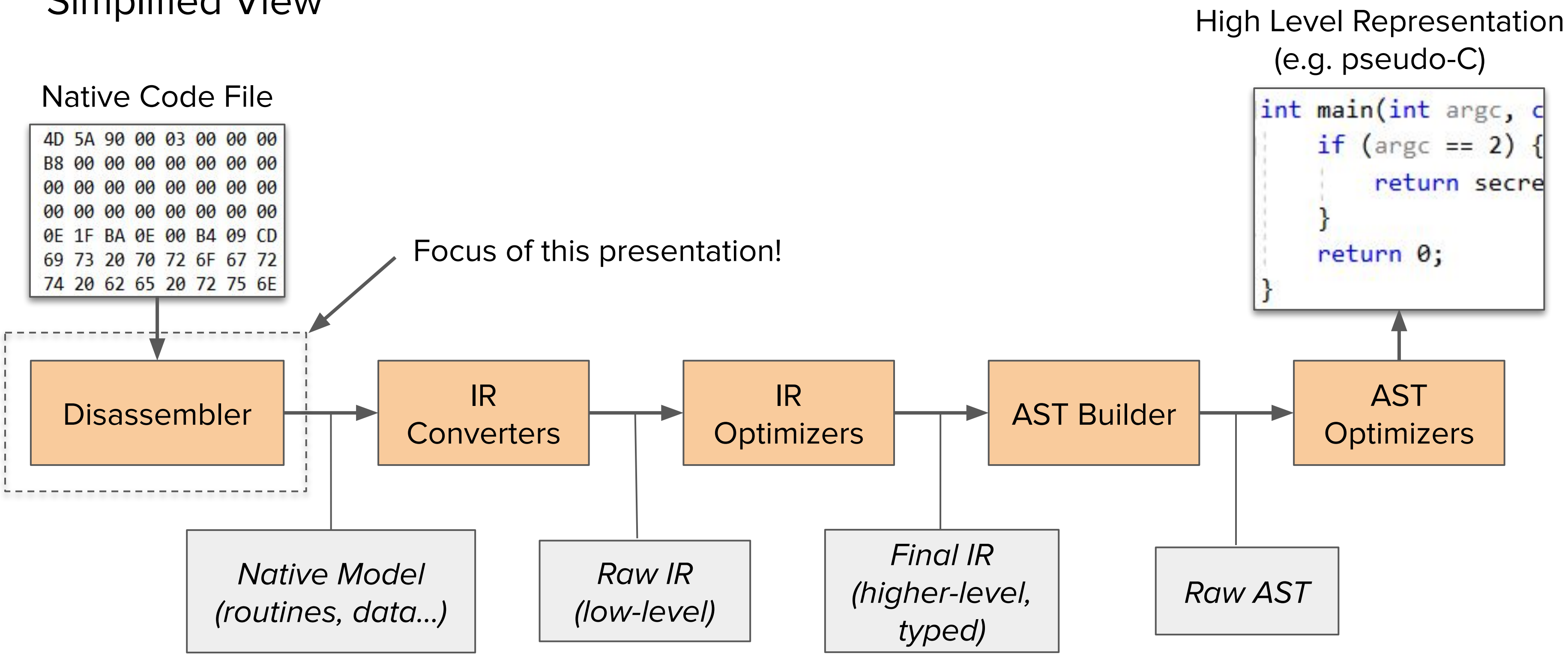
- **2012**: JEB 1.0 release
  Decompiles Android apps into Java code. Interactive UI. Scripting.

- **2014**: JEB 2.0 release
  Decompiles Windows/Linux executables (x86, ARM, MIPS,...) into C code

- **2018**: JEB 3.0 release
  Decompiles non-native platforms (Ethereum, WebAssembly)

JEB's Native Decompilation Pipeline

Simplified View



JEB’s Disassembler (1)

Informal Definitions

- Static analysis producing an assembly-based view of a whole binary, representing what can possibly happen at runtime
    - Note: we are not talking merely about individual machine instructions translation

JEB's Disassembler (1)

Informal Definitions

- Static analysis producing an assembly-based view of a whole binary, representing what can possibly happen at runtime
    - Note: we are not talking merely about individual machine instructions translation
- Intended to be a “safe” analysis, i.e. avoiding the following mistakes (ordered by seriousness):
    - Data considered as code (domino effect with wrong cross-references, branches, etc)
    - Code considered as data
    - Missed code and data

JEB's Disassembler (2)

Why Do We Need It?

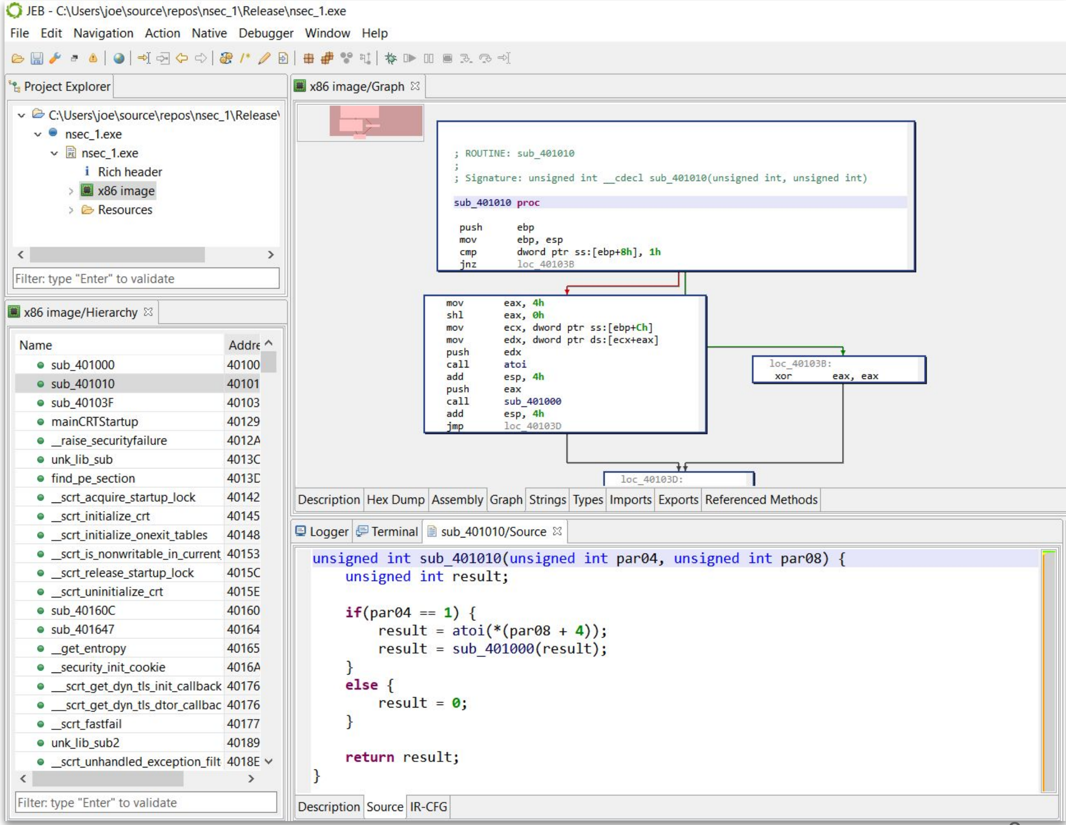
- The disassembler provides foundations for JEB's decompilation pipeline, in particular:
  - Code versus data separation
  - Control flow
  - Data flow
  - Abstractions: routines, control flow graphs, basic blocks...

- It is also useful for manual analysis (especially when decompilation fails...)

JEB's Disassembler (3)

Development Context

- Developed in-house over the last four years (in Java, like the rest of JEB)
- Most of the logic is architecture independant (except for instruction parsing and some heuristics)
- Can also parse non-native code (e.g. Ethereum, WebAssembly)



This Presentation's Intent

- **Describe what happen in a "real world" disassembler**, i.e. go further than the definition of disassemblers in textbooks, where they are usually divided into two families:
    - Linear disassemblers: sequential disassembly of all bytes belonging to code areas
        - Problem: data mixed with code
    - Recursive disassemblers: disassemble bytes by following the control flow
        - Problem: following control flow is hard

This Presentation's Intent

- **Describe what happen in a "real world" disassembler**, i.e. go further than the definition of disassemblers in textbooks, where they are usually divided into two families:
    - Linear disassemblers: sequential disassembly of all bytes belonging to code area
        - Problem: data mixed with code
    - Recursive disassemblers: disassemble bytes by following the control flow
        - Problem: following control flow is hard
- Hopefully, it will convince you (if needed) that disassembling remains a complex and interesting problem, even in an era of decompilers

Outline

1. Introduce a simple and naive disassembler algorithm

2. Show some limitations of the simple algorithm

3. Present the way we overcome these limitations in JEB

Disclaimer

This is a research talk (*not* a sales talk), showing work-in-progress and not intended to present the best ever solution to disassembling.

1. Manually Disassembling a Toy Example

Setting The Scene

Visual Studio x86 Compiler
(no optimizations, no inlining, no symbols)

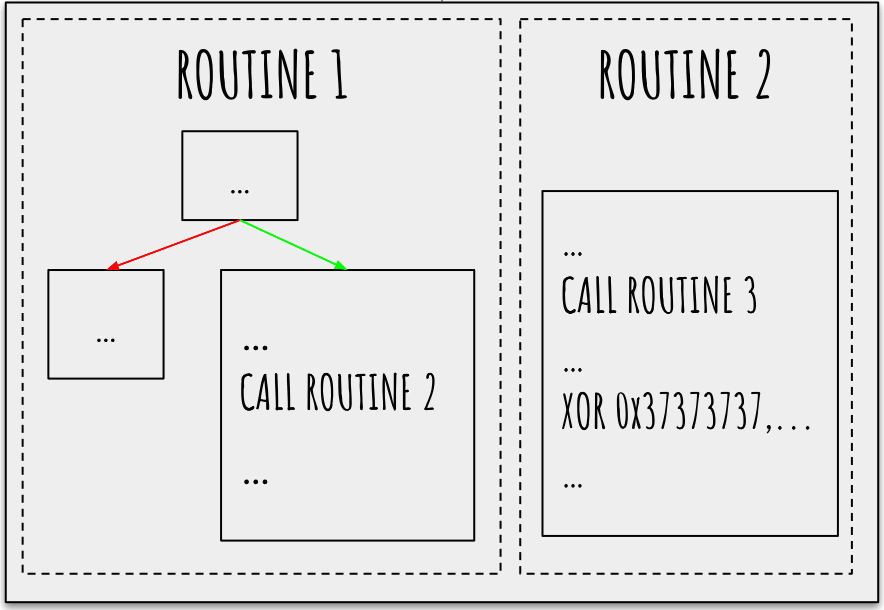
```
000  4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00
010  B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00
020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
030  00 00 00 00 00 00 00 00 00 00 00 00 F8 00 00 00
040  0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68
050  69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F
060  74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20
```

secret.exe

"Ideal" Disassembler

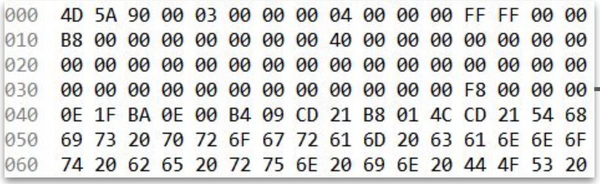
```c
int main(int argc, char** argv) {
    if (argc == 2) {
        return secret_algo(argv[1]);
    }
    return 0;
}
```

secret.c

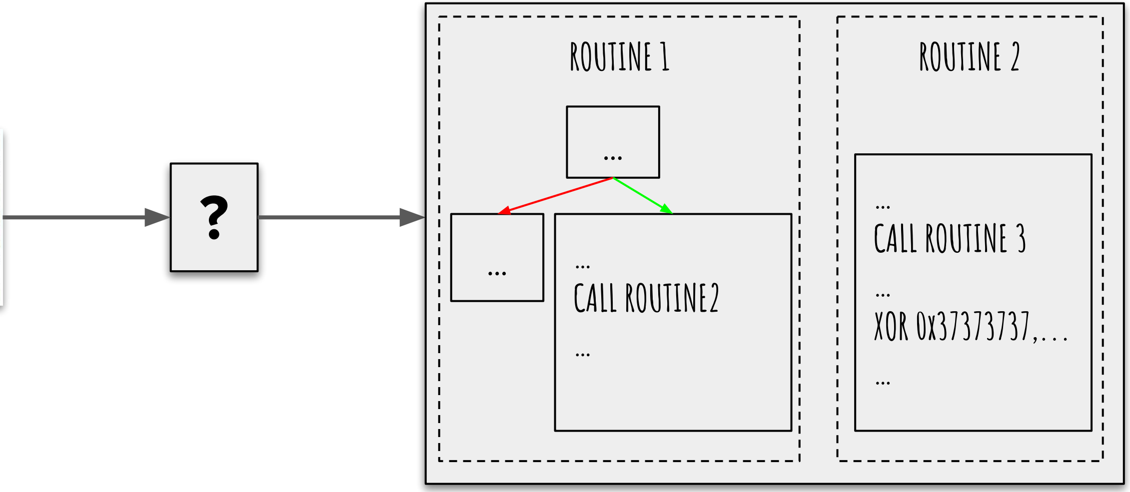


EXPECTED OUTPUT SKETCH

How To Get There?



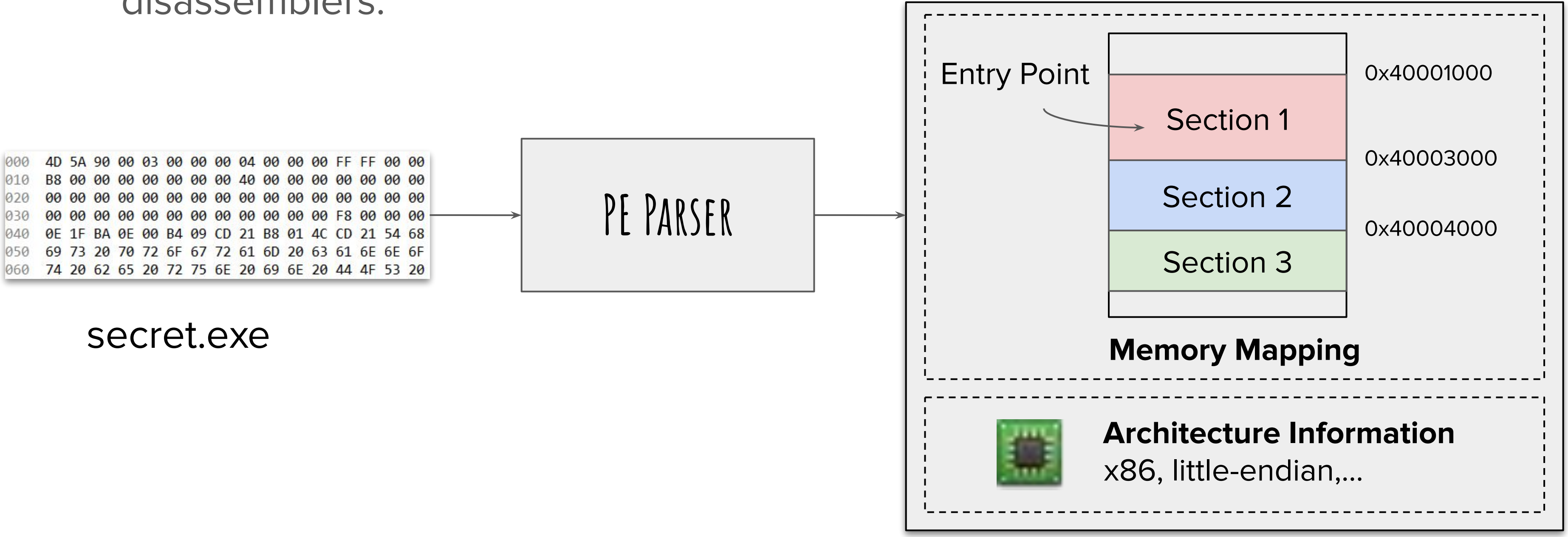
secret.exe



First, we need some prerequisites...

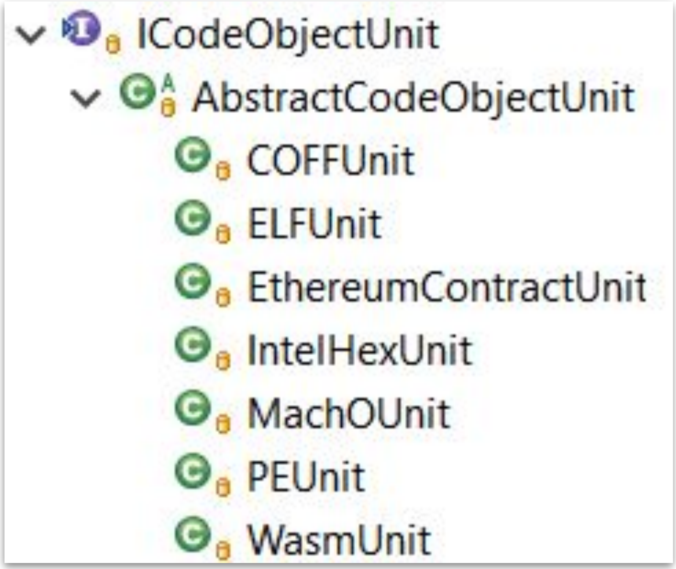
Prerequisite 1: Executable File Parsers

- Executable files formats (PE, ELF, Mach-O,...) provide necessary information for disassemblers:

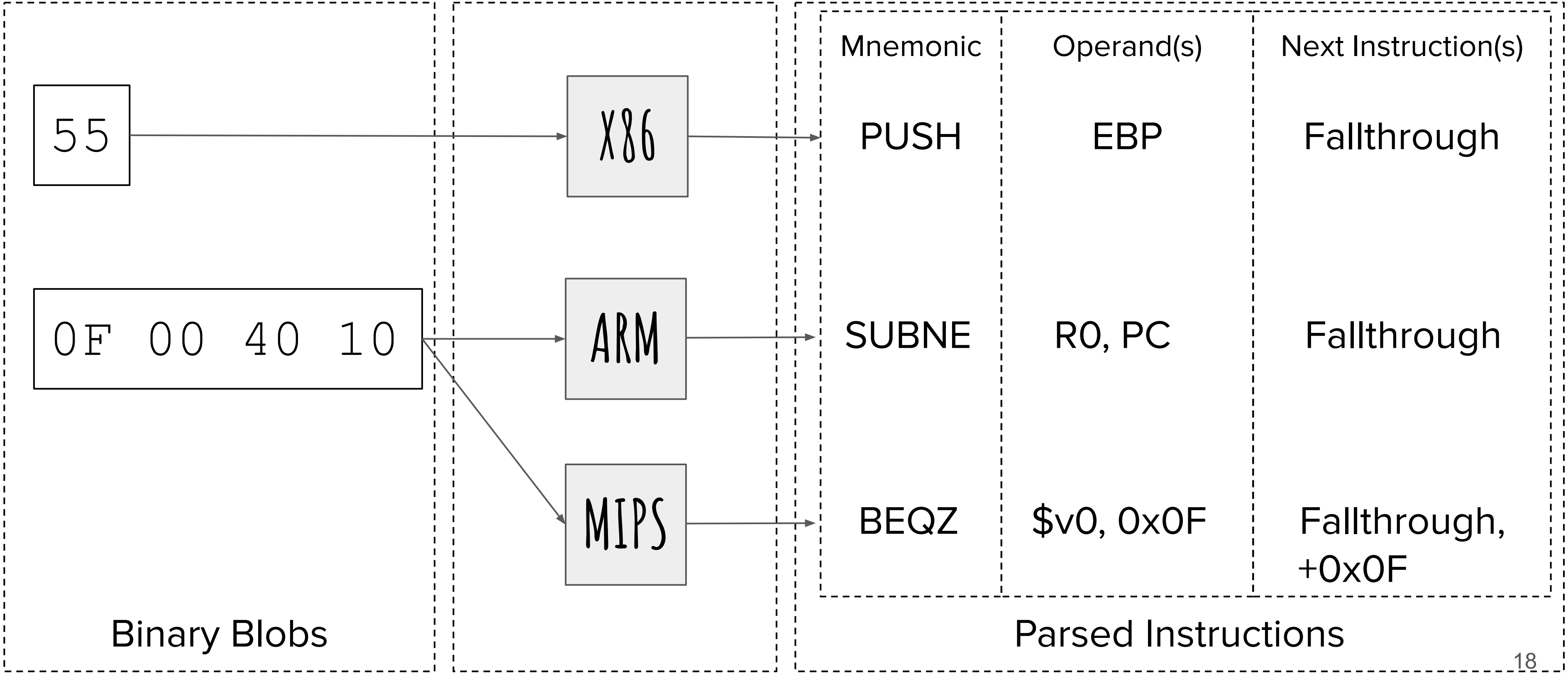


Interlude: Executable File Parsers in JEB

- Implement *ICodeObjectUnit*, an interface providing access to the parsed information (memory mapping, symbols,...) in a unified manner:

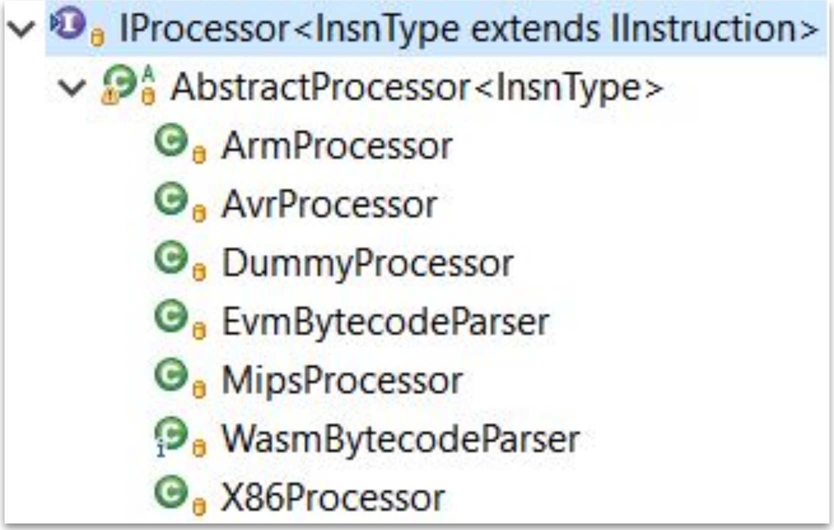


Prerequisite 2: Instruction Disassemblers



Interlude: Instruction Disassemblers in JEB

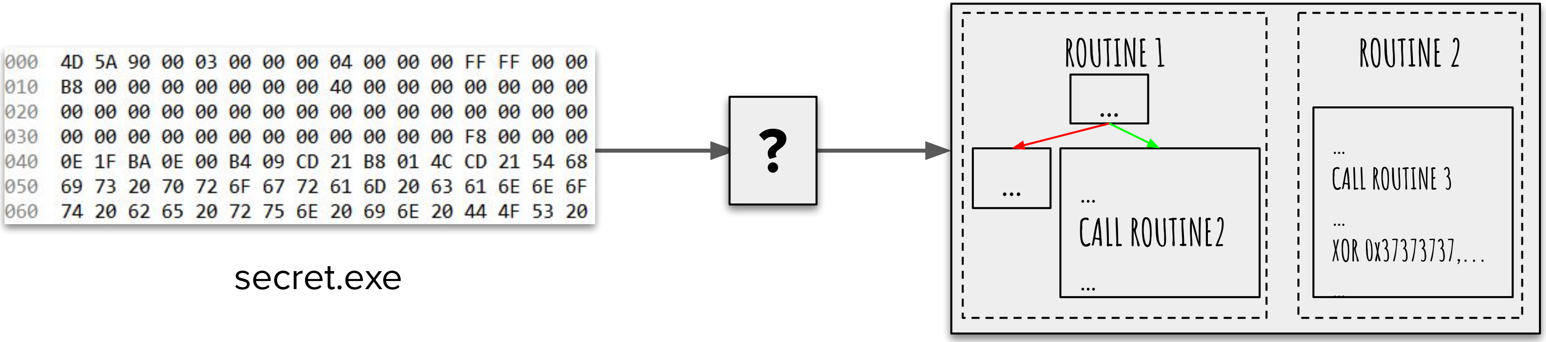
- Implement interface *IProcessor*:



- Parsed native instructions all implement *IInstruction* interface, allowing us to process them generically

Back To Our Toy Example

- Let’s assume we have a PE parser and a x86 instruction disassembler
- How to produce secret.exe disassembly?



secret.exe

Picking a First “Intuitive” Strategy

- Could we use linear disassembly?

Picking a First “Intuitive” Strategy

- Could we use linear disassembly?

- Visual Studio often puts data in `.text` section on x86
  - For example: PE data directories, import/export tables, jumptables…

Picking a First “Intuitive” Strategy

- Could we use linear disassembly?

- Visual Studio often puts data in `.text` section on x86
  - For example: PE data directories, import/export tables, jumptables…

- Therefore, recursive disassembly might be more suitable: start from the entry-point and follow the code… let’s try!

Input Memory Mapping

Disassembler Data Structures

EP

00401000	55	8B	EC	8B	45	08	33	05
00401008	80	48	41	00	5D	C3	CC	CC
00401010	55	8B	EC	83	7D	08	01	75
00401018	22	B8	04	00	00	00	C1	E0
00401020	00	8B	4D	0C	8B	14	01	52
00401028	E8	A3	24	00	00	83	C4	04
00401030	50	E8	CA	FF	FF	FF	83	C4
00401038	04	EB	02	33	C0	5D	C3	3B
00401040	0D	04	40	41	00	F2	75	02
00401048	F2	C3	F2	E9	79	02	00	00
00401050	56	6A	01	E8	FD	25	00	00
00401058	E8	9E	06	00	00	50	E8	6A
00401060	2F	00	00	E8	F7	30	00	00
00401068	8B	F0	E8	85	06	00	00	6A
00401070	01	89	06	E8	16	04	00	00
00401078	83	C4	0C	5E	84	C0	74	73
00401080	DB	E2	E8	A3	08	00	00	68

Input Memory Mapping

00401000	55	8B	EC	8B	45	08	33	05
00401008	80	48	41	00	5D	C3	CC	CC
00401010	55							
00401011	8B	EC	83	7D	08	01	75	
00401018	22	B8	04	00	00	00	C1	E0
00401020	00	8B	4D	0C	8B	14	01	52
00401028	E8	A3	24	00	00	83	C4	04
00401030	50	E8	CA	FF	FF	FF	83	C4
00401038	04	EB	02	33	C0	5D	C3	3B
00401040	0D	04	40	41	00	F2	75	02
00401048	F2	C3	F2	E9	79	02	00	00
00401050	56	6A	01	E8	FD	25	00	00
00401058	E8	9E	06	00	00	50	E8	6A
00401060	2F	00	00	E8	F7	30	00	00
00401068	8B	F0	E8	85	06	00	00	6A
00401070	01	89	06	E8	16	04	00	00
00401078	83	C4	0C	5E	84	C0	74	73

Disassembler Data Structures

```
PUSH EBP
```

Input Memory Mapping

00401000	55 8B EC 8B 45 08 33 05
00401008	80 48 41 00 5D C3 CC CC
00401010	55
00401011	8B EC
00401013	83 7D 08 01 75
00401018	22 B8 04 00 00 00 C1 E0
00401020	00 8B 4D 0C 8B 14 01 52
00401028	E8 A3 24 00 00 83 C4 04
00401030	50 E8 CA FF FF FF 83 C4
00401038	04 EB 02 33 C0 5D C3 3B
00401040	0D 04 40 41 00 F2 75 02
00401048	F2 C3 F2 E9 79 02 00 00
00401050	56 6A 01 E8 FD 25 00 00
00401058	E8 9E 06 00 00 50 E8 6A
00401060	2F 00 00 E8 F7 30 00 00
00401068	8B F0 E8 85 06 00 00 6A
00401070	01 89 06 E8 16 04 00 00

Disassembler Data Structures

```
PUSH EBP
MOV EBP, ESP
```

Input Memory Mapping

```
00401000  55 8B EC 8B 45 08 33 05
00401008  80 48 41 00 5D C3 CC CC
00401010  55
00401011  8B EC
00401013  83 7D 08 01
00401017  75
00401018  22 B8 04 00 00 00 C1 E0
00401020  00 8B 4D 0C 8B 14 01 52
00401028  E8 A3 24 00 00 83 C4 04
00401030  50 E8 CA FF FF FF 83 C4
00401038  04 EB 02 33 C0 5D C3 3B
00401040  0D 04 40 41 00 F2 75 02
00401048  F2 C3 F2 E9 79 02 00 00
00401050  56 6A 01 E8 FD 25 00 00
00401058  E8 9E 06 00 00 50 E8 6A
00401060  2F 00 00 E8 F7 30 00 00
00401068  8B F0 E8 85 06 00 00 6A
```

Disassembler Data Structures

```
PUSH EBP
MOV EBP, ESP
CMP [EBP+8], 1
```

Input Memory Mapping

00401000	55 8B EC 8B 45 08 33 05
00401008	80 48 41 00 5D C3 CC CC
00401010	55
00401011	8B EC
00401013	83 7D 08 01
00401017	75 22
00401019	B8 04 00 00 00 C1 E0
00401020	00 8B 4D 0C 8B 14 01 52
00401028	E8 A3 24 00 00 83 C4 04
00401030	50 E8 CA FF FF FF 83 C4
00401038	04 EB 02 33 C0 5D C3 3B
00401040	0D 04 40 41 00 F2 75 02
00401048	F2 C3 F2 E9 79 02 00 00
00401050	56 6A 01 E8 FD 25 00 00
00401058	E8 9E 06 00 00 50 E8 6A
00401060	2F 00 00 E8 F7 30 00 00

Disassembler Data Structures

```
PUSH EBP
MOV EBP, ESP
CMP [EBP+8], 1
JNZ 0x40103B
```

Addresses To Analyze Later

`0x40103B`

Intra Routine

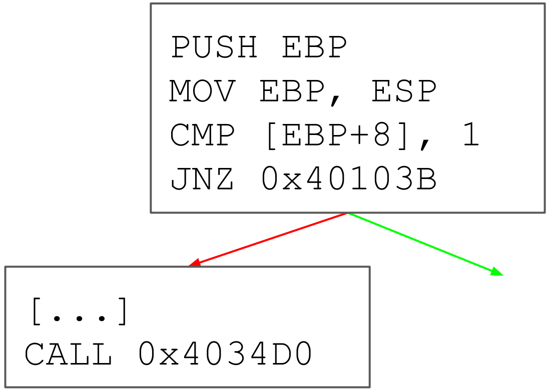
Conditional branch: end of block, continue analyzing fallthrough, store target for later analysis

Fast Forward…

Input Memory Mapping

```
00401000  55 8B EC 8B 45 08 33 05
00401008  80 48 41 00 5D C3 CC CC
00401010  55
00401011  8B EC
00401013  83 7D 08 01
00401017  75 22
00401019  B8 04 00 00 00
0040101E  C1 E0 00
00401021  8B 4D 0C
00401024  8B 14 01
00401027  52
00401028  E8 A3 24 00 00
0040102D  83 C4 04
00401030  50 E8 CA FF FF FF 83 C4
00401038  04 EB 02
0040103B  33 C0
0040103D  5D
```

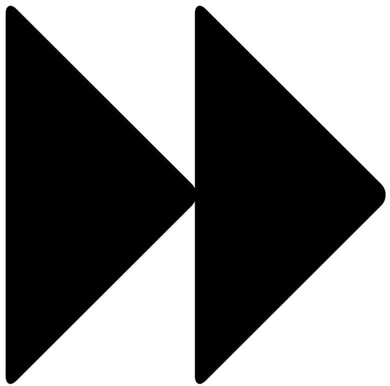
Disassembler Data Structures



Addresses To Analyze Later

Others Routines	Intra Routine
0x4034D0	0x40103B

Routine call: continue analyzing fallthrough, store target for later analysis

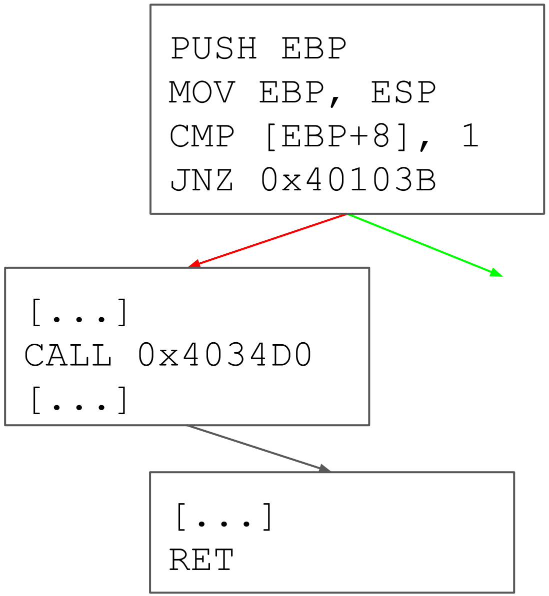


Fast Forward...

Input Memory Mapping

```
00401011   8B EC
00401013   83 7D 08 01
00401017   75 22
00401019   B8 04 00 00 00
0040101E   C1 E0 00
00401021   8B 4D 0C
00401024   8B 14 01
00401027   52
00401028   E8 A3 24 00 00
0040102D   83 C4 04
00401030   50
00401031   E8 CA FF FF FF
00401036   83 C4 04
00401039   EB 02
0040103B   33 C0
0040103D   5D
0040103E   C3
```

Disassembler Data Structures



Addresses To Analyze Later

Others Routines	Intra Routine
0x4034D0	**0x40103B**

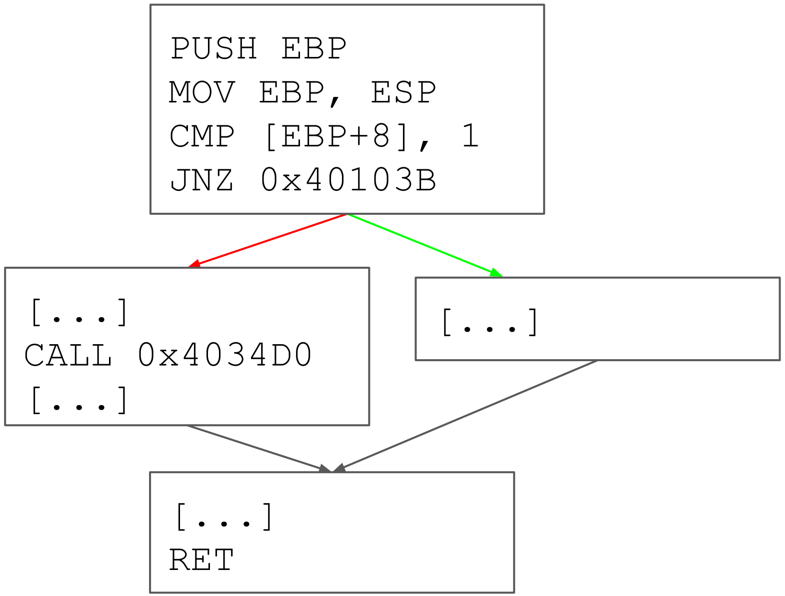
RET instruction: end of block, pop next address to analyze

Fast Forward…

Input Memory Mapping

004034D0	8B	FF	55	8B	EC	51	8B	45
004034D8	08	6A	01	6A	0A	51	51	8B
004034E0	CC	6A	00	83	61	04	00	89
004034E8	01	E8	77	FC	FF	FF	83	C4
004034F0	14	8B	E5	5D	C3	8B	FF	55
004034F8	8B	EC	51	53	56	57	E8	DF
00403500	1A	00	00	8B	F0	85	F6	0F
00403508	84	39	01	00	00	8B	16	33
00403510	DB	8B	CA	8D	82	90	00	00
00403518	00	3B	D0	74	0E	8B	7D	08
00403520	39	39	74	09	83	C1	0C	3B
00403528	C8	75	F5	8B	CB	85	C9	0F
00403530	84	11	01	00	00	8B	79	08
00403538	85	FF	0F	84	06	01	00	00
00403540	83	FF	05	75	0B	33	C0	89
00403548	59	08	40	E9	F8	00	00	00
00403550	83	FF	01	75	08	83	C8	FF

Disassembler Data Structures



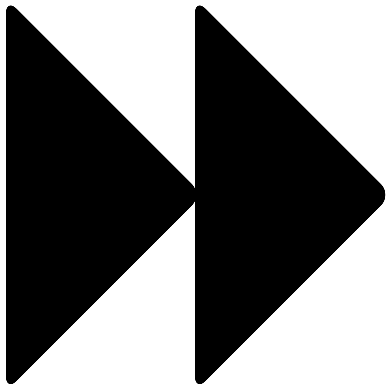
Addresses To Analyze Later

`0x4034D0`

Others Routines

Intra Routine

No more addresses to analyze: current CFG is finished. Analyze next routine.



Fast Forward...

End Result

Routine 1 - starts at 0x401010

```
PUSH EBP
MOV EBP, ESP
CMP [EBP+8], 1
JNZ 0x40103B
```

```
MOV EAX, 4
SHL EAX, 0
MOV ECX, [EBP+CH]
MOV EDX, [ECX+EAX]
PUSH EDX
CALL 0x4034D0
ADD ESP, 4
JMP LOC_401044
```

```
XOR EAX, EAX
```

```
POP EBP
RET
```

Routine 2 - starts at 0x4034D0

```
PUSH EBP
MOV EBP, ESP
MOV EAX, [EBP+8]
PUSH EAX
CALL 0x4034F5
ADD ESP, 4
XOR EAX, [0x414880]
POP EBP
RET
```

In the end, we produced the expected disassembly with a simple recursive algorithm!

The magic was in the instruction disassembler…

So, it’s not *that* hard to disassemble whole executables?

How does this algorithm generalize to others Visual Studio executables? To others compilers? To others architectures?

2. Some Questionable Assumptions We Made

More Or Less Consciously

Assumption 1: CALL Always Return To Caller

***"Routine call: <u>continue analyzing fallthrough</u>,
store target for later analysis"***

Counter-Example: Non Returning Calls

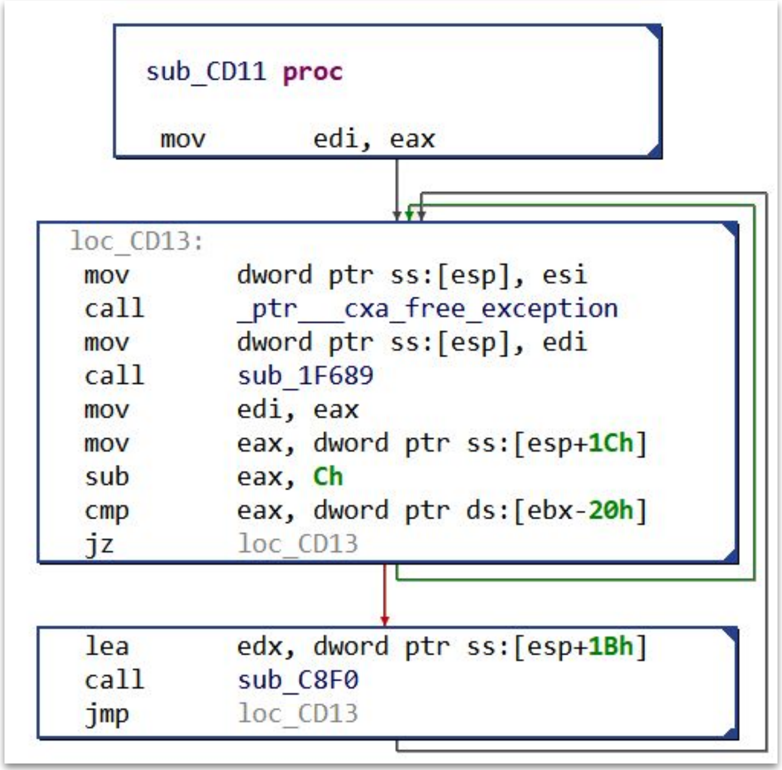
Exiting APIs - Visual Studio CRT

```
00403EEB    pop     ecx
00403EEC    push    dword ptr ss:[ebp+8]
00403EEF    call    dword ptr ds:[ptr_KERNEL32.dll!ExitProcess]
00403EF5    int3
```

```
WINBASEAPI DECLSPEC_NORETURN VOID WINAPI ExitProcess(_In_ UINT uExitCode);
```

Counter-Example: Non Returning Calls

Infinitely Looping Routines - GCC 4.9



Head-Scratching With Non Returning Routines (1)

- We need to identify non returning CALLs, otherwise our CFG will be incorrect

Head-Scratching With Non Returning Routines (1)

- We need to identify non returning CALLs, otherwise our CFG will be incorrect

- **Non returning external APIs** can be identified from their names, if we have full prototypes with non-returning attribute

Head-Scratching With Non Returning Routines (1)

- We need to identify non returning CALLs, otherwise our CFG will be incorrect

- **Non returning external APIs** can be identified from their names, if we have full prototypes with non-returning attribute

- **Non returning internal routines** (e.g. wrappers around non returning APIs) can be identified by analyzing their CFG
  - If CFG has no returning blocks, then it is a non-returning routine

Head-Scratching With Non Returning Routines (2)

- We can only know an internal routine is non returning *after* having analyzed it
  - What if we are on a CALL, and we do not have analyzed the target yet?

Head-Scratching With Non Returning Routines (2)

- We can only know an internal routine is non returning *after* having analyzed it
    - What if we are on a CALL, and we do not have analyzed the target yet?

- We could stop analyzing the caller, and analyze the callee first, but:
    - Maintaining the caller’s state could be tricky; possible state explosions with chain of calls
    - Often, we do not know where the callee is!

The JEB Way

- **For external APIs**: custom C parser to build type libraries from compiler/SDK header files
  - Type libraries provide non-return attributes for standard APIs

The JEB Way

- **For external APIs**: custom C parser to build type libraries from compiler/SDK header files
  - Type libraries provide non-return attributes for standard APIs

- **For internal routines**:
  - Identify simple cases at the time of the CALL (e.g. trampolines to non-returning API)
  - Otherwise, terminate caller's analysis assuming callee returns, then analyze callee; if callee found non-returning, re-analyze its callers
    - Tricky: the first caller analysis, without knowing non-returning callees, can be hard to "undo"

Assumption 2: Routine Control Flow Graphs Are Distincts

"No more addresses to analyze:
<u>current CFG is finished</u>."

```
; ROUTINE: __startTwoArgErrorHandling
; (Routine has gaps: 40BB07h-40BB10h)

__startTwoArgErrorHandling proc

push      ebp
mov       ebp, esp
add       esp, E0h
mov       dword ptr ss:[ebp-20h], eax
mov       eax, dword ptr ss:[ebp+18h]
mov       dword ptr ss:[ebp-10h], eax
mov       eax, dword ptr ss:[ebp+1Ch]
mov       dword ptr ss:[ebp-Ch], eax
jmp       loc_40BB10
```

```
; ROUTINE: __startOneArgErrorHandling

__startOneArgErrorHandling proc

push      ebp
mov       ebp, esp
add       esp, E0h
mov       dword ptr ss:[ebp-20h], eax
fstp      qword ptr ss:[ebp-8h]
mov       dword ptr ss:[ebp-1Ch], ecx
mov       eax, dword ptr ss:[ebp+10h]
mov       ecx, dword ptr ss:[ebp+14h]
mov       dword ptr ss:[ebp-18h], eax
mov       dword ptr ss:[ebp-14h], ecx
lea       eax, dword ptr ss:[ebp+8h]
lea       ecx, dword ptr ss:[ebp-20h]
push      eax
push      ecx
push      edx
call      sub_40BF85
add       esp, Ch
fld       qword ptr ss:[ebp-8h]
cmp       word ptr ss:[ebp+8h], 27Fh
jz        loc_40BB41
```

Counter-Example: Routines Sharing Code in Visual Studio 2017 CRT

Head-Scratching On Shared Code

2

```
; ROUTINE: __startTwoArgErrorHandling
; (Routine has gaps: 40BB07h-40BB10h)

__startTwoArgErrorHandling proc

push      ebp
mov       ebp, esp
add       esp, E0h
mov       dword ptr ss:[ebp-20h], eax
mov       eax, dword ptr ss:[ebp+18h]
mov       dword ptr ss:[ebp-10h], eax
mov       eax, dword ptr ss:[ebp+1Ch]
mov       dword ptr ss:[ebp-Ch], eax
jmp       loc_40BB10
```

1

```
; ROUTINE: __startOneArgErrorHandling

__startOneArgErrorHandling proc

push      ebp
mov       ebp, esp
add       esp, E0h
mov       dword ptr ss:[ebp-20h], eax
fstp      qword ptr ss:[ebp-8h]
mov       dword ptr ss:[ebp-1Ch], ecx
mov       eax, dword ptr ss:[ebp+10h]
mov       ecx, dword ptr ss:[ebp+14h]
mov       dword ptr ss:[ebp-18h], eax
mov       dword ptr ss:[ebp-14h], ecx
lea       eax, dword ptr ss:[ebp+8h]
lea       ecx, dword ptr ss:[ebp-20h]
push      eax
push      ecx
push      edx
call      sub_40BF85
add       esp, Ch
fld       qword ptr ss:[ebp-8h]
cmp       word ptr ss:[ebp+8h], 27Fh
jz        loc_40BB41
```

Let's say, we parse 1 first

Then, we parse 2 and found a **branch *within* an existing basic block**!

Do we duplicate instructions into another basic block, or do we split the existing basic block?

Head-Scratching On Shared Code

- Basic block = series of instructions executed in a row
  - Foundation for later analysis: basic blocks can be processed without dealing with possible control flow changes
  - Exception: exceptions (haha)

Head-Scratching On Shared Code

- Basic block = series of instructions executed in a row
    - Foundation for later analysis: basic blocks can be processed without dealing with possible control flow changes
    - Exception: exceptions (haha)

- Duplicating instructions means having different possible basic blocks at the same address
    - Later analysis would become harder to write, as we would need to check all possible basic blocks
    - Likely not a good idea!

The JEB Way

- During disassembling we build “skeletons” basic blocks (simple containers for instructions, easily modifiable), and split them when we found a branch coming directly within a block

The JEB Way

- During disassembling we build “skeletons” basic blocks (simple containers for instructions, easily modifiable), and split them when we found a branch coming directly within a block

- Once disassembly is finished, we build final control flow graphs with proper basic blocks, which means:
  - An address belongs to at most one basick block
  - Basic blocks can be shared among routines

The JEB Way: End Result

```
; ROUTINE: __startTwoArgErrorHandling
; (Routine has gaps: 40BB07h-40BB10h)

__startTwoArgErrorHandling proc

push    ebp
mov     ebp, esp
add     esp, E0h
mov     dword ptr ss:[ebp-20h], eax
mov     eax, dword ptr ss:[ebp+18h]
mov     dword ptr ss:[ebp-10h], eax
mov     eax, dword ptr ss:[ebp+1Ch]
mov     dword ptr ss:[ebp-Ch], eax
jmp     loc_40BB10
```

```
; ROUTINE: __startOneArgErrorHandling

__startOneArgErrorHandling proc

push    ebp
mov     ebp, esp
add     esp, E0h
mov     dword ptr ss:[ebp-20h], eax
```

```
fstp    qword ptr ss:[ebp-8h]
mov     dword ptr ss:[ebp-1Ch], ecx
mov     eax, dword ptr ss:[ebp+10h]
mov     ecx, dword ptr ss:[ebp+14h]
mov     dword ptr ss:[ebp-18h], eax
mov     dword ptr ss:[ebp-14h], ecx
lea     eax, dword ptr ss:[ebp+8h]
lea     ecx, dword ptr ss:[ebp-20h]
push    eax
push    ecx
push    edx
call    sub_40BF85
add     esp, Ch
fld     qword ptr ss:[ebp-8h]
cmp     word ptr ss:[ebp+8h], 27Fh
jz      loc_40BB41
```

Assumption 3: Branch Instructions Immediately End Basic-Blocks

"Conditional branch: end of block, continue analyzing fallthrough, store target for later analysis"

Counter-Example: MIPS Branch Delay Slots

```
LW      $gp, 10h($sp)
BEQ     $v0, $s5, loc_408CBC
NOP
LUI     $v0, 41h
ADDIU   $v0, $v0, 6550h
BEQZ    $v0, loc_400328
LI      $v0, 1h
...
```

Branch delay slots (**always** executed)

Conditional branch (if $v0 == $s5)

Conditional branch (if $v0 == 0)

Head-Scratching With Delay Slots (1)

- Where do we cut the basic block?

```
...
LUI      $v0, 41h
ADDIU    $v0, $v0, 6550h
BEQZ     $v0, loc_400328
LI       $v0, 1h
...
```

Head-Scratching With Delay Slots (1)

- Where do we cut the basic block?

- A basic block = a series of instructions executed successively, i.e. **the delay slot belongs to the same block as the branch**

```
...
LUI         $v0, 41h
ADDIU       $v0, $v0, 6550h
BEQZ        $v0, loc_400328
LI          $v0, 1h
...
```

Head-Scratching With Delay Slots (1)

- Where do we cut the basic block?
- A basic block = a series of instructions executed successively, i.e. **the delay slot belongs to the same block as the branch**
- But... branch instructions *in the middle* of basic blocks breaks one of the most common assumption on control flow graphs!
- Can we avoid that?

```
...
LUI        $v0, 41h
ADDIU      $v0, $v0, 6550h
BEQZ       $v0, loc_400328
LI         $v0, 1h
...
```

Head-Scratching With Delay Slots (2)

- A CFG is just a representation, what if we simply revert instructions' order in the graph?

```
...
LUI         $v0, 41h
ADDIU       $v0, $v0, 6550h
BEQZ        $v0, loc_400328
LI          $v0, 1h
...
```

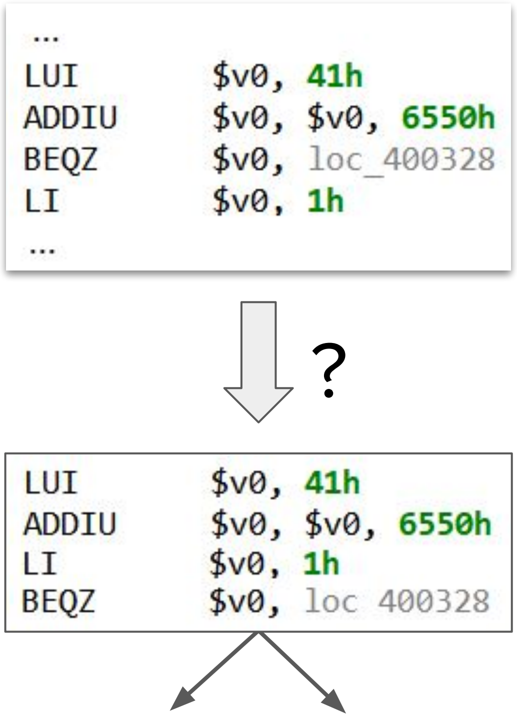
?

```
LUI         $v0, 41h
ADDIU       $v0, $v0, 6550h
LI          $v0, 1h
BEQZ        $v0, loc_400328
```

Head-Scratching With Delay Slots (2)

- A CFG is just a representation, what if we simply revert instructions' order in the graph?

- Order of expression evaluation would then be broken
    - Branch condition must be evaluated first
    - Here according to the graph `$v0` is set to 1 *before* being used as the branch condition => not a conditional branch anymore



Head-Scratching With Delay Slots (3)

- What if we group the branch and its delay slot into one artificial instruction?

```
...
LUI       $v0, 41h
ADDIU     $v0, $v0, 6550h
BEQZ      $v0, loc_400328
LI        $v0, 1h
...
```

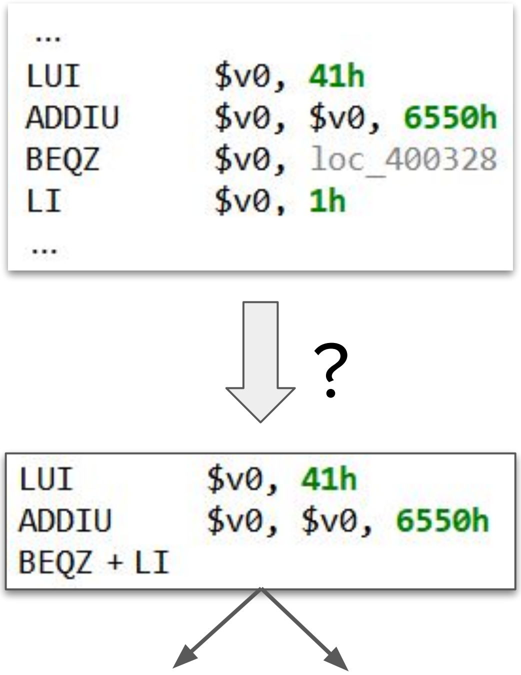
?

```
LUI       $v0, 41h
ADDIU     $v0, $v0, 6550h
BEQZ + LI
```

Head-Scratching With Delay Slots (3)

- What if we group the branch and its delay slot into one artificial instruction?

- It is actually legal to have a branch directly on the delay slot instruction, which we cannot not handle with that representation...



The JEB Way

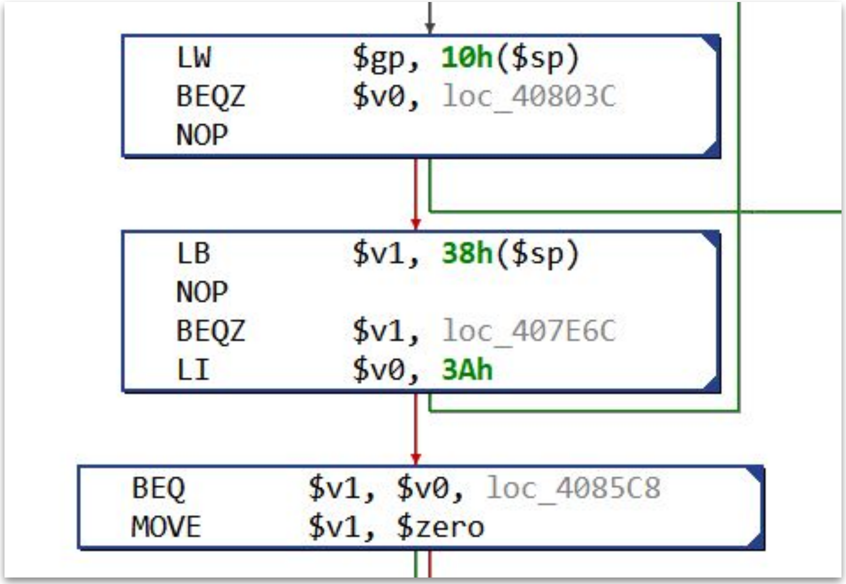
- Branch instructions are allowed in the middle of basic blocks

- Instruction disassemblers provide number of delay slot(s) for branch instructions:

int	getDelaySlotCount () Get the number of instructions in the delay slot.

Example of End Result (JEB’s CFG)

```
LW      $gp, 10h($sp)
BEQZ    $v0, loc_40803C
NOP
LB      $v1, 38h($sp)
NOP
BEQZ    $v1, loc_407E6C
LI      $v0, 3Ah
BEQ     $v1, $v0, loc_4085C8
MOVE    $v1, $zero
```



Assumption 4: The Instruction Set Remains The Same

(never said anything about that, but that was taken for granted, right?)

Counter-Example: ARM/Thumb Switch

Thumb ISA

```
00010320 04 B4          PUSH    {R2}
00010322 01 B4          PUSH    {R0}
00010324 DF F8 10 C0    LDR     R12, gvar_10338
00010328 4D F8 04 CD    STR     R12, [SP, #-4h]!
0001032C 03 48          LDR     R0, gvar_1033C
0001032E 04 4B          LDR     R3, gvar_10340
00010330 FF F7 DE EF    BLX     _ptr___libc_start_main
00010334 FF F7 E8 EF    BLX     _ptr_abort
```

BLX: Branch with Link and **eXchange instruction set**

ARM and Thumb are *different* instruction sets sharing the *same* encoding space

Both can be in the same executable...

ARM ISA

```
000102F0                _ptr___libc_start_main proc
000102F0
000102F0 00 C6 8F E2    ADR     R12, loc_102F8
000102F4 10 CA 8C E2    ADD     R12, R12, #10000h
000102F8 18 FD BC E5    LDR     PC, [R12, #D18h]!
000102F8
000102F8                _ptr___libc_start_main endp
```

Head-Scratching With Instruction Set Switching

- Instruction disassemblers must handle all possible instruction sets for a given architecture

Head-Scratching With Instruction Set Switching

- Instruction disassemblers must handle all possible instruction sets for a given architecture

- **Knowing an address is code (and not data) is not enough**, we need to know the actual instruction set

- This information can come from various sources:
  - Address is called in a certain way (e.g. ARM's BLX)
  - Address is referenced in a certain way (e.g. ELF ARM symbol with address LSB set to 1)
  - Address has a specific alignment
  - ...

The JEB Way

- JEB's instruction disassemblers can handle different instructions sets, and switch from one to another based on information provided by the disassembler

- On unknown code addresses, the disassembler heuristically orders instruction sets by likelihood, and tries all of them until it gets a "correct" result (i.e. a proper routine)

Assumption 5: Control Flow Can Always Be Followed

"Routine call: continue analyzing fallthrough address, <u>store target for later analysis</u>"

Counter-Example: Jumptables

VS2017 x86 Compact Switch

```
switch (parameter) {
  case 1:
    return_value = 1;
    break;
  case 2:
    return_value = 2;
    break;

  [...REDACTED...]

  case 400:
    return_value = 400;
    break;
  default:
    return_value = 0;
    break;
}
```

Counter-Example: Jumptables

VS2017 x86 Compact Switch

```
dec     ecx
cmp     ecx, 18Eh
jnbe    loc_401A1E
```

```
jmp     dword ptr ds:[4*ecx+401A24h]
```

```
loc_401A1E:
  xor     eax, eax
  ret
```

```
00401A24  dd 4010C4h
00401A28  dd 4010CAh
00401A2C  dd 4010D0h
00401A30  dd 4010D6h
00401A34  dd 4010DCh
00401A38  dd 4010E2h
00401A3C  dd 4010E8h
00401A40  dd 4010EEh
```

Discovering control flow here means finding ECX's possible values

How To Find Possible Values For Indirect Operands?

(i.e. register or memory operands)

- We need these values (in particular) to follow indirect branches
- For a better control flow we need a better data flow (and vice versa...)
- What if... we try to answer that in a syntactic manner?

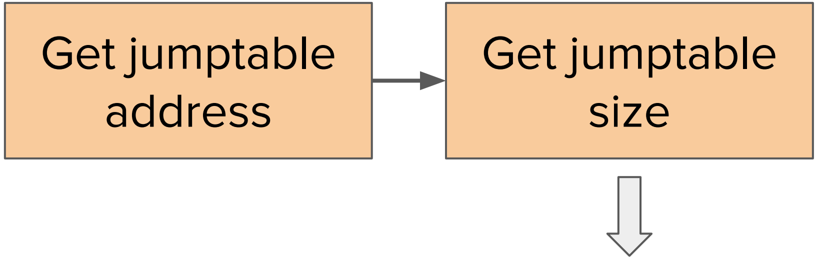
The JEB Way: Pattern Matching For Visual Studio x86 Jumptables

The JEB Way: Pattern Matching For Visual Studio x86 Jumptables

Get jumptable address

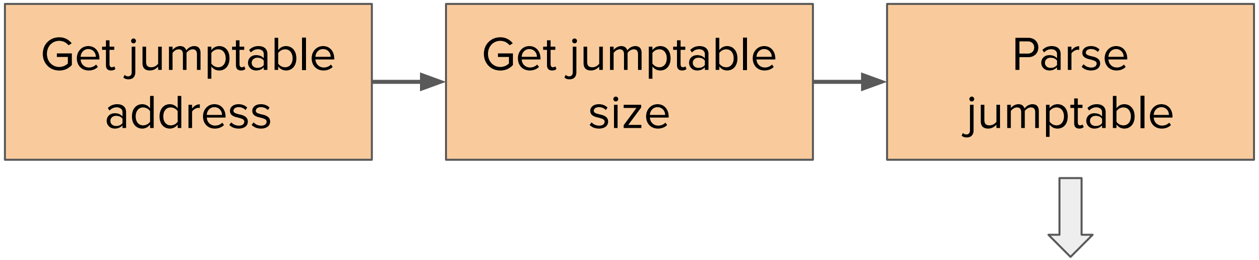
```
// check if the given instruction follows the pattern 'jmp [4*R+T]' and return T if yes
if(insn.getMnemonic().equals("jmp")) {
    IX86Operand operand = insn.getOperands()[0];
    if(operand instanceof X86OperandCMA) {
        X86OperandCMA operand_ = (X86OperandCMA)operand;
        if(operand_.getMemoryScale() == 4 && operand_.getMemoryIndexRegister() != -1
                && operand_.getMemoryDisplacement() != 0) {
            return operand_.getMemoryDisplacement();
        }
    }
}
```

The JEB Way: Pattern Matching For Visual Studio x86 Jumptables



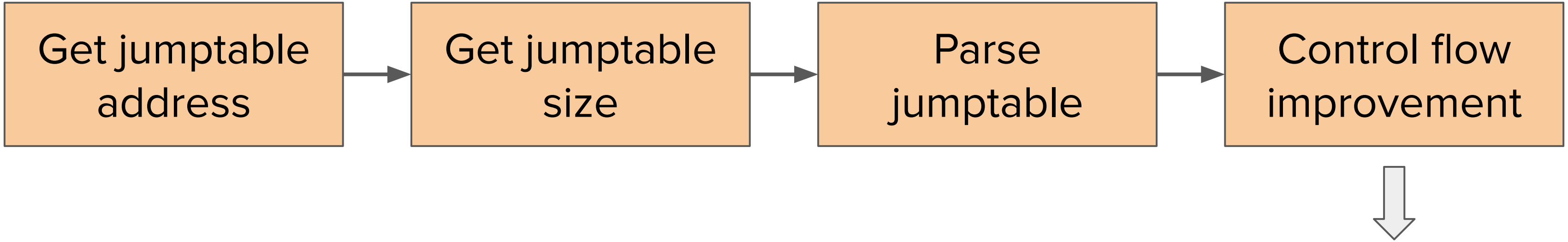

```java
// check if unique previous block ends with JA/JNBE preceded by CMP
if(!parentBlockLastIns.getMnemonic().equals("ja")
        && !parentBlockLastIns.getMnemonic().equals("jnbe")) {
    return NativeCodeAnalyzerExtensionResult.ignore();
}
if(!parentBlockBeforeLastIns.getMnemonic().equals("cmp")) {
    return NativeCodeAnalyzerExtensionResult.ignore();
}
IX86Operand operand = parentBlockBeforeLastIns.getOperands()[1];
if(operand.getOperandType() != IInstructionOperandGeneric.TYPE_IMM) {
    return NativeCodeAnalyzerExtensionResult.ignore();
}
numberOfEntries = operand.getOperandValue() + 1; // !!
```

The JEB Way: Pattern Matching For Visual Studio x86 Jumptables




```java
// main jumptable parsing
JumpTableInformation mainTableInfo = new JumpTableInformation(possibleJumpTableAddress, 4);
long curAddress = possibleJumpTableAddress;
for(int i = 0; i < numberOfEntries; i++) {
    Long caseHandler = gca.getMemory().readInt(curAddress) & 0xFFFFFFFFL;
    SwitchCaseInformation caseInfo = new SwitchCaseInformation();
    caseInfo.setCaseHandler(gca.getProcessor().createEntryPoint(caseHandler));
    caseInfo.setJumpTableEntryAddress(curAddress);
    caseInfo.setJumpTableEntrySize(4);
    switchInfo.addCase(caseInfo);
    curAddress += 4;
}
```

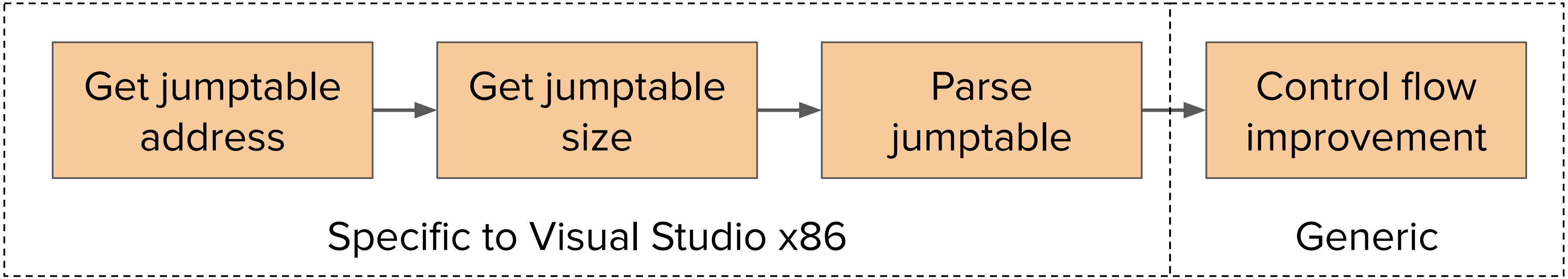
The JEB Way: Pattern Matching For Visual Studio x86 Jumptables




```
// 1. analyze each case handler -- if not already done -- and
// connect them to 'switch' instruction
for(SwitchCaseInformation caseInfo: switchInfo.getCases()) {
    // ...[REDACTED]...
}

// 2. define jumptables, as arrays of addresses
for(JumpTableInformation jumpTableInfo: jumpTableInfos) {
  // ...[REDACTED]...
}
```

The JEB Way: Pattern Matching For Visual Studio x86 Jumptables



(More on how we integrate such compiler-specific logic into the disassembler later)

Computing control flow with pattern matching... really?

Syntactic solutions might be acceptable, though inelegant, when the target code is *very* common, because:

- Development effort is (usually) limited
- Parsing performances are good

But obviously syntactic solutions cannot scale in the context of a multi-architecture disassembler...

Moar Jumptables Examples
Case 1: ARM GCC

```
CMP         R3, #95
LDRLS       PC, [PC, R3, LSL #2] ; if R3 <= 95, PC = JUMPTABLE[R3 * 4]
B           default_case         ; otherwise goto default
; ---------------------------------------------------------------------
dd offset loc_92F4               ; JUMPTABLE (96 entries)
dd offset default_case
dd offset default_case
dd offset default_case
dd offset default_case
dd offset default_case
dd offset default_case
dd offset default_case
```

Moar Jumptables Examples

Case 2: ARM GCC -fPIC

```
CMP         R0, #17
ADDLS       PC, PC, R0, LSL #2  ; if R0 <= 17, PC = PC + (R0 << 2)
B           default_case        ; otherwise goto default
B           loc_8A72C
B           loc_8A72C
B           loc_8A7A8
B           default_case
```

Moar Jumptables Examples

Case 3: ARM Thumb GCC

```
ADDS        R0, R1, #0
BLX         switchu8
; ---------------------------------------------------------------
db 5                          ; JUMPTABLE's number of entries
db ECh                        ; JUMPTABLE
db 6Ch
db 5Dh
db 1Eh
db 6
db 4                          ; default offset
```

```
LDRB        R12, [LR, #-1]
CMP         R0, R12
LDRBCC      R0, [LR, R0]
LDRBCS      R0, [LR, R12]
ADD         R12, LR, R0, LSL #1
BX          R12
```

- These are only a few of the existing ARM jumptables implementations, syntactic solutions seem therefore a long journey to go…

- And there are cases that are just out of reach!

Entry Point

```
00400260  MOVE    $zero, $ra
00400264  BAL     loc_40026C
00400268  NOP
0040026C  LUI     $gp, 6
00400270  ADDIU   $gp, $gp, 10E4h
00400274  ADDU    $gp, $gp, $ra
00400278  MOVE    $ra, $zero
0040027C  LW      $a0, 823Ch($gp)
00400280  LW      $a1, 0($sp)
00400284  ADDIU   $a2, $sp, 4
00400288  LI      $at, FFF8h
0040028C  AND     $sp, $sp, $at
00400290  ADDIU   $sp, $sp, FFE0h
00400294  LW      $a3, 8360h($gp)
00400298  LW      $t0, 81F0h($gp)
0040029C  NOP
004002A0  SW      $t0, 10h($sp)
004002A4  SW      $v0, 14h($sp)
004002A8  SW      $sp, 18h($sp)
004002AC  LW      $t9, 825Ch($gp)
004002B0  NOP
004002B4  JALR    $t9            → ?
004002B8  NOP
```

MIPS Position-Independent Code (System V ABI)

Syntactic methods clearly not suitable here!

How To Find Possible Values For Indirect Operands?

Back To The Original Question (And Let's Forget About Syntactic Solutions)

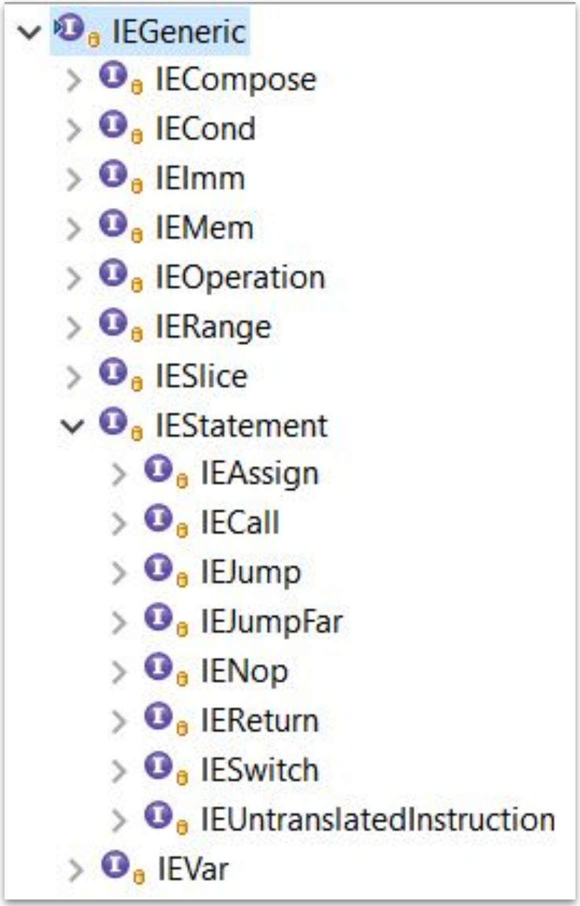
- Could we simulate routines execution, such that we would know the machine state (registers and memory) between each instruction?
    - Need native instructions' semantics to update the state (that's the hard part)
    - Spoiler alert: not always doable in a safe way due to unknown inputs, but could solve simple cases in a "generic" way (i.e. more generic than the syntactic ones)

Interlude: JEB's Intermediate Language

Definition

Low-level imperative language, made of *expressions*:

- 16 different expressions
- Expressions can contain sub-expressions (accessed through a tree-like structure)
- Highest-level expressions are *statements*



Interlude: JEB's Intermediate Language

Example

s32:**var1**[0:8[= s8:**var2** + s8:**var3**

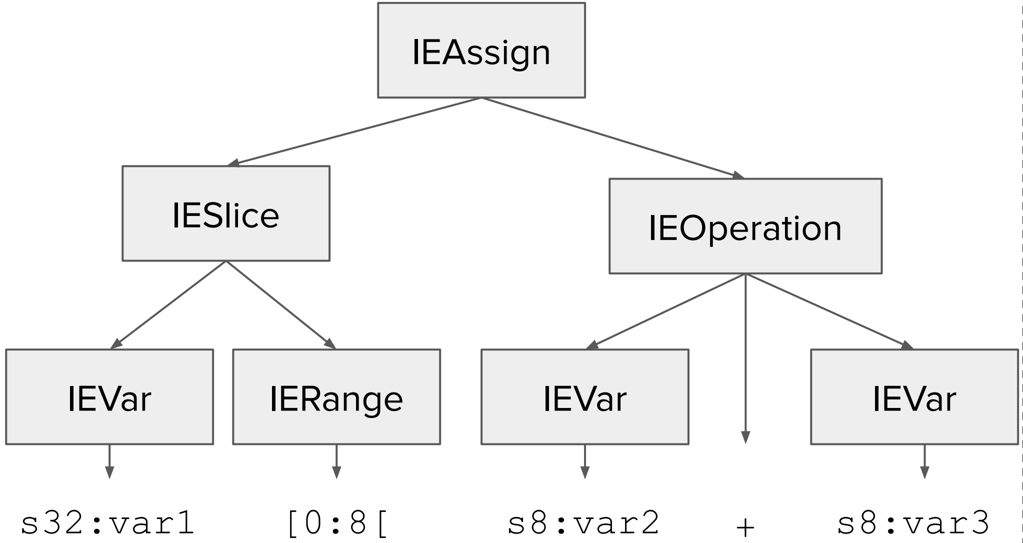
Textual Representation

Interlude: JEB's Intermediate Language

Example

s32:**var1**[0:8[= s8:**var2** + s8:**var3**

Textual Representation



Object Representation

Interlude: JEB's Intermediate Language

Main Purpose

The IL's primary purpose is to allow expressing native instruction's semantics:

```
xor eax, dword ds:[10000h]
```

X86 Instruction

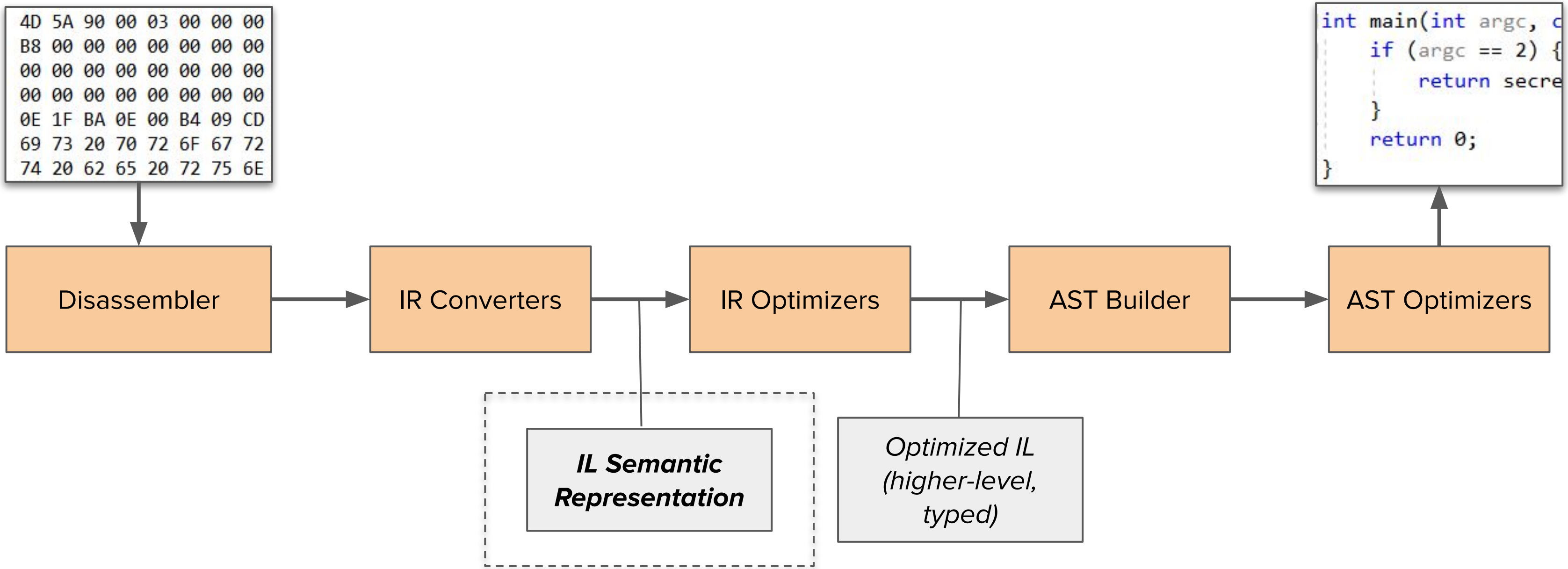
```
s32:_eax = (s32:_eax ^ 32<s16:_ds>[i32:10000h])
s1:_zf = (s32:_eax ? i1:0 : i1:1)
s1:_sf = s32:_eax[31:32[
s1:_pf = PARITY(s32:_eax[0:8[)
s1:_of = i1:0
s1:_cf = i1:0
```

Semantic Representation in JEB's IL

Interlude: JEB's Intermediate Language

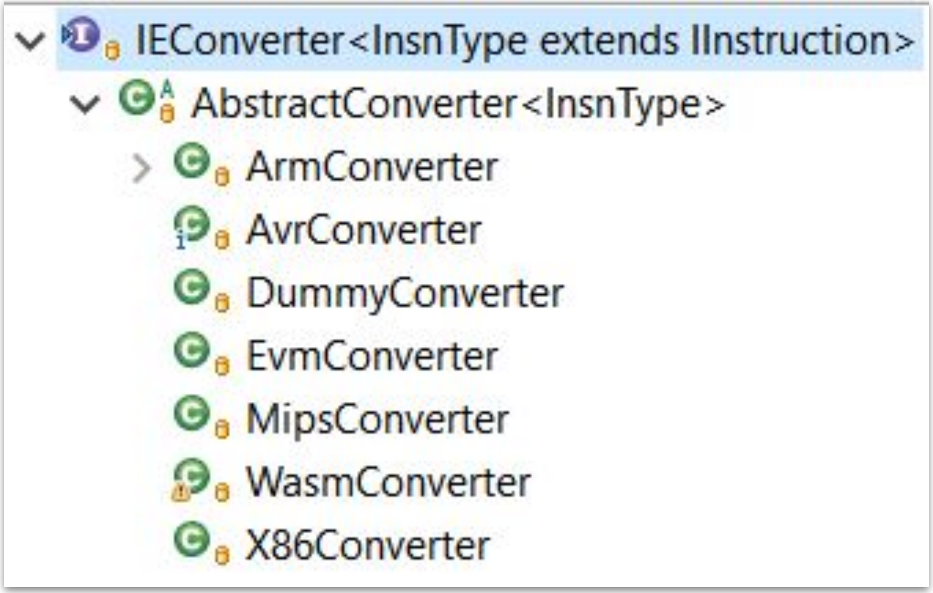
Main Purpose

- IL semantic representation is the foundation for JEB's decompilation pipeline:



Interlude: JEB's Intermediate Language
Main Purpose

- Heavy lifting is done by native-to-IL converters:



Having access to native-to-IL converters allows us to implement the simulation at the IL level:

- We only have to handle the 16 IL expressions (rather than all native instructions)
- All architectures for which we have a native-to-IL converter will benefit from it
- Drawback: the performance cost of IL conversion

The JEB Way : Intermediate Language Simulation

1. Convert each native routine into its equivalent semantic representation in JEB’s IL
    - Produces CFG of IL statements (no optimizations at this point)
    - For example, here is `secret.exe main()` first IL basic block:

```
s32:_esp = (s32:_esp - i32:00000004h)
32<s16:_ss>[s32:_esp] = s32:_ebp
s32:_ebp = s32:_esp
32<s16:_ds>[i32:004152A0h] = i32:00401000h
s1:_zf = ((32<s16:_ss>[(s32:_ebp + i32:00000008h)] - i32:00000002h) ? i1:0 : i1:1)
s1:_sf = (32<s16:_ss>[(s32:_ebp + i32:00000008h)] - i32:00000002h)[31:32[
s1:_pf = PARITY((32<s16:_ss>[(s32:_ebp + i32:00000008h)] - i32:00000002h)[0:8[)
s1:_cf = (32<s16:_ss>[(s32:_ebp + i32:00000008h)] <u i32:00000002h)
s1:_af = ((32<s16:_ss>[(s32:_ebp + i32:00000008h)] ^ i32:00000002h) ^
s32:_eip = (s1:_zf ? i32:00401033h : i32:0040104Dh)
```

The JEB Way : Intermediate Language Simulation

2. Simulate IL routine to build the machine state at each instruction:

   ○ Start from a clean state (pseudo-realistic values in registers, and allocated stack memory)

   ○ Implement IL expression evaluation:

```
/**
 * Evaluate the IRE.
 *
 * @param state IR state (input and output)
 * @return the concrete evaluation
 * @throws Exception force client code to wrap calls for safe execution
 */
IEImm evaluate(IEState state) throws Exception;
```

The JEB Way : Intermediate Language Simulation

```
@Override
public EImm evaluate(IEState state) throws Exception {
    EImm a = op1.evaluate(state);
    EImm b = op2 == null ? null: op2.evaluate(state);

    switch(optype) {
    case ADD:
        a = a._add(b);
        break;
    case SUB:
        a = a._sub(b);
        break;
    case MUL_S:
        a = a._mul(b);
        break;
    case MUL_U:
        a = a._mulU(b);
```

IEOperation evaluation (excerpt)

The JEB Way : Intermediate Language Simulation

- Routine simulator follows the control flow from the entry-point and evaluates all encountered statements

The JEB Way : Intermediate Language Simulation

- Routine simulator follows the control flow from the entry-point and evaluates all encountered statements

- Simulator is intended to be “safe”, i.e. to provide only trustable values:

    - When IL expression evaluation fails (because of missing information, e.g. an uninitialized memory slot), we abort the routine simulation

    - On subroutine calls, we consider all registers spoiled (i.e. now containing an unknown value)

The JEB Way : Intermediate Language Simulation

3. Report the values found during simulation to the disassembler

The JEB Way:
MIPS Position-Independent Code
(before IL simulation)

```
00400260  MOVE   $zero, $ra
00400264  BAL    loc_40026C
00400268  NOP
0040026C  LUI    $gp, 6
00400270  ADDIU  $gp, $gp, 10E4h
00400274  ADDU   $gp, $gp, $ra
00400278  MOVE   $ra, $zero
0040027C  LW     $a0, 823Ch($gp)
00400280  LW     $a1, 0($sp)
00400284  ADDIU  $a2, $sp, 4
00400288  LI     $at, FFF8h
0040028C  AND    $sp, $sp, $at
00400290  ADDIU  $sp, $sp, FFE0h
00400294  LW     $a3, 8360h($gp)
00400298  LW     $t0, 81F0h($gp)
0040029C  NOP
004002A0  SW     $t0, 10h($sp)
004002A4  SW     $v0, 14h($sp)
004002A8  SW     $sp, 18h($sp)
004002AC  LW     $t9, 825Ch($gp)
004002B0  NOP
004002B4  JALR   $t9
004002B8  NOP
```

The JEB Way:
MIPS
Position-Independent Code
(after IL simulation)

```
00400260  MOVE   $zero, $ra
00400264  BAL    loc_40026C        ; POST: $ra=loc_40026C
00400268  NOP
0040026C  LUI    $gp, 6
00400270  ADDIU  $gp, $gp, 10E4h
00400274  ADDU   $gp, $gp, $ra     ; PRE: $ra=loc_40026C
00400278  MOVE   $ra, $zero
0040027C  LW     $a0, 823Ch($gp)   ; POST: $a0=sub_409B20
00400280  LW     $a1, 0($sp)
00400284  ADDIU  $a2, $sp, 4
00400288  LI     $at, FFF8h
0040028C  AND    $sp, $sp, $at
00400290  ADDIU  $sp, $sp, FFE0h
00400294  LW     $a3, 8360h($gp)   ; POST: $a3=_init
00400298  LW     $t0, 81F0h($gp)   ; POST: $t0=sub_416530
0040029C  NOP
004002A0  SW     $t0, 10h($sp)     ; PRE: $t0=sub_416530
004002A4  SW     $v0, 14h($sp)
004002A8  SW     $sp, 18h($sp)
004002AC  LW     $t9, 825Ch($gp)   ; POST: $t9=__uClibc_main
004002B0  NOP
004002B4  JALR   $t9               ; -> __uClibc_main
004002B8  NOP
```

Interlude: Revisiting Syntactic Solutions With JEB IL

Example: x86/ARM Jumptables

```
push    ebp
mov     ebp, esp
push    ecx
mov     eax, dword ptr ss:[ebp+8]
mov     dword ptr ss:[ebp-4], eax
cmp     dword ptr ss:[ebp-4], 9
jnbe    loc_401436
mov     ecx, dword ptr ss:[ebp-4]
jmp     dword ptr ds:[4*ecx+gvar_401440]
```

```
...
if (32[(s32:_SP0 - i32:8h)] >u i32:9h)
  goto 0027
...
s32:_eip = 32[((s32:_ecx * i32:4h) + i32:00401440h)]
```

```
PUSH    {R11}
ADD     R11, SP, #0
SUB     SP, SP, #Ch
STR     R0, [R11, #-8]
LDR     R3, [R11, #-8]
CMP     R3, #9
LDRLS   PC, [PC, R3, LSL #2]
B       loc_10784
```

```
...
if (s32:_R3 >u i32:9h)
  goto 0021
...
s32:_PC = 32[((s32:_R3 * i32:4h) + i32:00010714h)]
```

Disassembly

Optimized JEB's IL

Interlude: Revisiting Syntactic Solutions With JEB IL

Example: x86/ARM Jumptables

```
push    ebp
mov     ebp, esp
push    ecx
mov     eax, dword ptr ss:[ebp+8]
mov     dword ptr ss:[ebp-4], eax
cmp     dword ptr ss:[ebp-4], 9
jnbe    loc_401436
mov     ecx, dword ptr ss:[ebp-4]
jmp     dword ptr ds:[4*ecx+gvar_401440]
```

```
...
if (32[(s32:_SP0 - i32:8h)] >u i32:9h)
  goto 0027
...
s32:_eip = 32[((s32:_ecx * i32:4h) + i32:00401440h)]
```

One pattern catches both implementations!

```
PUSH    {R11}
ADD     R11, SP, #0
SUB     SP, SP, #Ch
STR     R0, [R11, #-8]
LDR     R3, [R11, #-8]
CMP     R3, #9
LDRLS   PC, [PC, R3, LSL #2]
B       loc_10784
```

```
...
if (s32:_R3 >u i32:9h)
  goto 0021
...
s32:_PC = 32[((s32:_R3 * i32:4h) + i32:00010714h)]
```

Disassembly

Optimized JEB's IL

IL simulation provides only concrete and trustable values, and therefore does not always work.

So what if we cannot follow the control flow from `main()`? Do we have another way to find `secret_algo()`?

Distinguishing Code From Data

- In theory, intractable problem (like anything interesting in program analysis)

Distinguishing Code From Data

- In theory, intractable problem (like anything interesting in program analysis)

- In practice, it is indeed a hard problem on most architectures, because:
    - Code and data share the same memory space
    - Almost any series of bytes corresponds to a machine instruction, due to instruction sets encoding “density”

Distinguishing Code From Data

- In theory, intractable problem (like anything interesting in program analysis)

- In practice, it is indeed a hard problem on most architectures, because:
  - Code and data share the same memory space
  - Almost any series of bytes corresponds to a machine instruction, due to instruction sets encoding “density”

- But... context can help!

Distinguishing Code From Data

- Raw dump of a x86 executable compiled by Visual Studio 2017:

```
55 8B EC 8B 45 08 50 E8 D4 24 00 00 83 C4 04 33
05 80 48 41 00 5D C3 CC CC CC CC CC CC CC CC CC
55 8B EC C7 05 A0 52 41 00 00 10 40 00 83 7D 08
02 75 1A B8 04 00 00 00 C1 E0 00 8B 4D 0C 8B 14
01 52 FF 15 A0 52 41 00 83 C4 04 EB 02 33 C0 5D
C3 3B 0D 04 40 41 00 F2 75 02 F2 C3 F2 E9 79 02
```

```
push ebp
mov ebp, esp
```
(classic VS routine prologue)

```
pop ebp
ret
```
(classic VS routine epilogue)

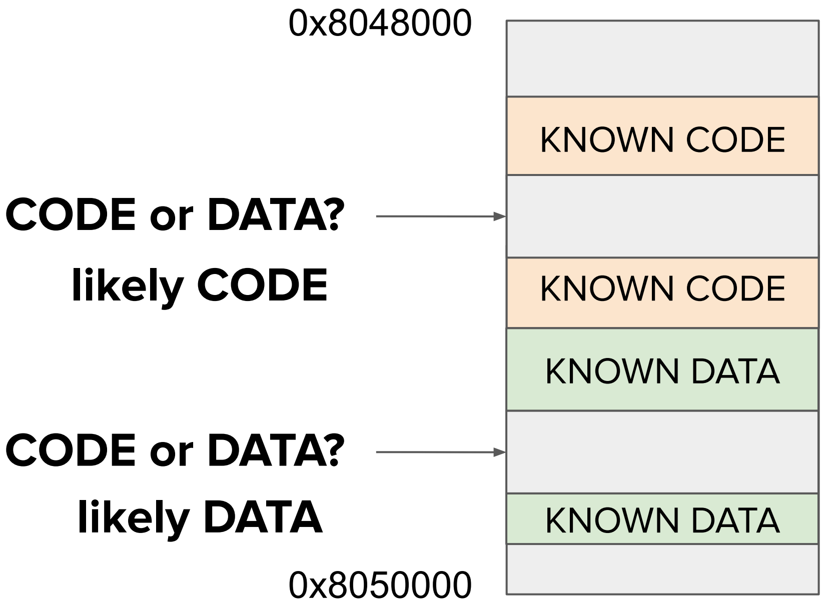
```
int 3
int 3
...
```
(classic VS code padding)

CODE or DATA?

likely CODE!

Distinguishing Code From Data

- Memory view of a x86 executable compiled by GCC 4.9:



GCC for x86 (usually) does not mix code and data!

The JEB Way: Compiler-Specific Heuristics

- Identify the compiler that served to create the target and apply specific heuristics

The JEB Way: Compiler-Specific Heuristics

- Identify the compiler that served to create the target and apply specific heuristics
  - Example: **address *A* is considered to be *likely* code** if:

compiler is gcc or clang
and **architecture is x86**
and **no obfuscations/malformations**
and ***A* is within code area**
and **bytes at *A* do not look like code padding**

The JEB Way: Compiler-Specific Heuristics

How To Integrate Them Into a Generic Disassembler? (1)

- At startup, JEB loads different “extensions” for each compiler/file format/architecture:

- INativeCodeAnalyzerExtension<InsnType extends IInstruction>
  - AbstractAnalyzerExtension<InsnType>
    - ArmAnalyzerExtension
    - ArmGccAnalyzerExtension
    - AvrAnalyzerExtension
    - DummyAnalyzerExtension
    - ELFAnalyzerExtension
    - EmptyAnalyzerExtension<InsnType>
    - EvmAnalyzerExtension
    - MipsAnalyzerExtension
    - WasmAnalyzerExtension
    - X86AnalyzerExtension
    - X86ClangAnalyzerExtension
    - X86GccAnalyzerExtension
    - X86PEAnalyzerExtension
    - X86VisualStudioAnalyzerExtension

The JEB Way: Compiler-Specific Heuristics

How To Integrate Them Into a Generic Disassembler? (2)

- Disassembler then queries loaded extensions for heuristics, for example:

getPossiblePaddingSize (long address, long addressMax)

Determine if a given memory area *looks like* (could be) starting with padding, and provides the size of the padding looking area, if any.

getPrologueLooking (long address, long addressMax)

Determine if a given memory area *looks like* (could be) the beginning of a routine.

isCandidateSwitchDispatcher (long address, InsnType insn, List<InsnType> insns)

Determine (heuristically) if the provided branching instruction (jump/call/...) could be the dispatcher of a switch-like statement.

See INativeCodeAnalyzerExtension

The JEB Way: Compiler-Specific Heuristics

What If... Heuristics Are Wrong?

- Mistakes happen: misidentified compiler, new/old compiler version, obfuscation...

The JEB Way: Compiler-Specific Heuristics

What If... Heuristics Are Wrong?

- Mistakes happen: misidentified compiler, new/old compiler version, obfuscation...

- Disassembler feedback loop:
  - Errors are logged (e.g. supposedly-code bytes cannot be disassembled, routine cannot be built,...)
  - If too many errors made, disassembler switches back to “safe mode”, where only conservative heuristics are applied

The JEB Way: Compiler-Specific Heuristics

What If... Heuristics Are Wrong?

- Mistakes happen: misidentified compiler, new/old compiler version, obfuscation...
- Disassembler feedback loop:
  - Errors are logged (e.g. supposedly-code bytes cannot be disassembled, routine cannot be built,...)
  - If too many errors made, disassembler switches back to “safe mode”, where only conservative heuristics are applied
- Last resort: user can change disassembler decisions

Assumption 6: All Code Matters

What's Up With `atoi()`?

Counter-Example: Statically Linked Library Routines

```
int secret_algo(char * arg) {
    return atoi(arg) ^ secret_key;
}
```

```
push      ebp
mov       ebp, esp
mov       eax, dword ptr ss:[ebp+8]
push      eax
call      sub_4034D0
add       esp, 4
xor       eax, dword ptr ds:[gvar_414880]
pop       ebp
ret
```

```
mov       edi, edi
push      ebp
mov       ebp, esp
push      ecx
mov       eax, dword ptr ss:[ebp+8]
push      1
push      Ah
push      ecx
push      ecx
mov       ecx, esp
push      0
and       dword ptr ds:[ecx+4], 0
mov       dword ptr ds:[ecx], eax
call      sub_403165
add       esp, 14h
mov       esp, ebp
pop       ebp
ret
```

... *[large routine]* ...

Pretty complex routine, but... it's "just" `atoi()`!

Identifying Library Routines

- One of the oldest reverse-engineering problem...

Identifying Library Routines

- One of the oldest reverse-engineering problem…
- Such identification allows users to read documentation rather than analyze code

The **atoi**() function converts the initial portion of the string pointed to by *nptr* to *int*. The behavior is the same as

```
    strtol(nptr, NULL, 10);
```

Identifying Library Routines

- One of the oldest reverse-engineering problem...
- Such identification allows users to read documentation rather than analyze code

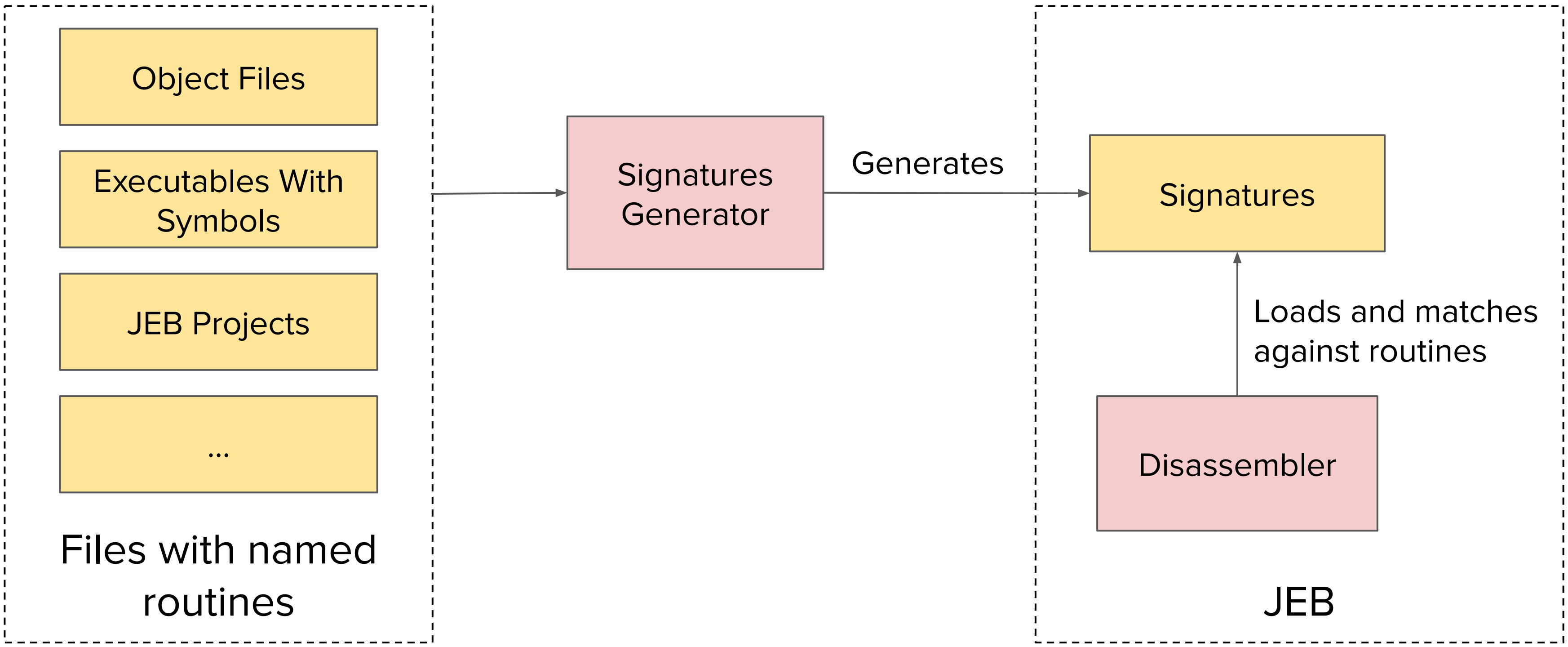
```
The atoi() function converts the initial portion of the string
pointed to by nptr to int.  The behavior is the same as

    strtol(nptr, NULL, 10);
```

- Such identification allows automatic analysis to have precise information on the routine, in particular its prototype:

```
int atoi(const char *nptr);
```

The JEB Way: Signatures (Basic) Workflow



Interlude: JEB Native Signatures (1)

- Signatures are designed as generic containers to identify an “item”, i.e. anything possibly defined in JEB native model (routines, basic blocks, data blobs, etc)

Interlude: JEB Native Signatures (1)

- Signatures are designed as generic containers to identify an “item”, i.e. anything possibly defined in JEB native model (routines, basic blocks, data blobs, etc)

- For that purpose, they contain two collections:
    - **Features**: characteristics of the item serving to identify it
        - Each feature can be matched against another
    - **Attributes**: knowledge on the item (name, origin,...)

Interlude: JEB Native Signatures (2)

- Example: signature for `_memcpy_s` routine from Visual Studio 2008 /MT standard libraries:
  - Features:
    - Routine size: `57`
    - Routine hash: `A2D4D55381F634B0...34F4F0176D181218D`
  - Attributes:
    - Routine name: `_memcpy_s`
    - Original file name: `libcmt.lib>memcpy_s.obj`
    - Compiler name: `Microsoft Visual C++ 2008 (9.0)`

Which Features To Identify Compiler Library Routines?

- Compiler routines signatures should be **false positive free**, because:
    - JEB's decompilation pipeline is going to rely on them (e.g. by using the corresponding prototypes)
    - Users are going to rely blindly on them in the UI

Which Features To Identify Compiler Library Routines?

- Compiler routines signatures should be **false positive free**, because:
  - JEB's decompilation pipeline is going to rely on them (e.g. by using the corresponding prototypes)
  - Users are going to rely blindly on them in the UI

- **False positives happen when a matched signature does not convey the original routine behavior** (or the original prototype):
  - For example, `_memmove` and `_memcpy` have the exact same code (and therefore behavior) in VS2013 /MT libraries; identifying one as the other is ***not*** a false positive in this context

Feature: Routine Code Hash (1)

- Hash computed from the routine assembly code:
  - Using assembly rather than binary code allows to avoid generating signatures for all different endianness (e.g. on MIPS)
  - We use a custom (and simple) hash algorithm producing a 64-byte value
- Hash computation is (almost) the same for x86/ARM/MIPS thanks to IInstruction interface

Feature: Routine Code Hash (2)

- The parts of the routine code depending of its actual location are normalized before hash computation :

X86 object file snippet

```
FF 15 AE 26 00 00    call     dword ptr ds:[26AEh]
33 C9                xor      ecx, ecx
A3 B2 25 00 00       mov      dword ptr ds:[25B2h], eax
```

Normalization: abstract absolute addresses (any constant actually)

```
call [address]
xor ecx, ecx
mov [address], eax
```

More complex normalization cases exist (e.g. ARM relocations can change BL mnemonic to BLX)

It Might Not Be Enough...

- For example in the case of wrappers:

```
_mbbtombc       proc

                push    ebp
                mov     ebp, esp
                push    0
                push    dword ptr ss:[ebp+8]
                call    _mbbtombc_l
                pop     ecx
                pop     ecx
                pop     ebp
                ret

_mbbtombc       endp
```

```
_mbctombb       proc

                push    ebp
                mov     ebp, esp
                push    0
                push    dword ptr ss:[ebp+8]
                call    _mbctombb_l
                pop     ecx
                pop     ecx
                pop     ebp
                ret

_mbctombb       endp
```

Same routine code hash, but different behaviors
=> another feature: name of called routines

It Might (Still) Not Be Enough...

```
push    ebp
mov     ebp, esp
push    ebx
mov     ebx, ecx
mov     ecx, FF00h
mov     eax, ebx
and     eax, ecx
jz      loc_615C
cmp     eax, ecx
jz      loc_615C
push    1
push    dword ptr ss:[ebp+4]
call    extern?_RTC_Failure@@YAXPAXH@Z
pop     ecx
pop     ecx

mov     al, bl
pop     ebx
pop     ebp
ret
```

```
push    ebp
mov     ebp, esp
push    ebx
mov     ebx, ecx
mov     ecx, FFFFFF00h
mov     eax, ebx
and     eax, ecx
jz      loc_615C
cmp     eax, ecx
jz      loc_615C
push    1
push    dword ptr ss:[ebp+4]
call    extern?_RTC_Failure@@YAXPAXH@Z
pop     ecx
pop     ecx

mov     al, bl
pop     ebx
pop     ebp
ret
```

Same routine code hash and callee routines, but different behaviors
=> another feature: constants

Signatures Generation Strategy (1)

- Adding *all* possible features in each signature is non-optimal, because:
  - Matching performance (during disassembling) depends on the number of features in signatures
  - Some features can be useless (e.g. a large routine is likely to be uniquely identified *only* by its hash)
  - Some features can be a burden (e.g. with called routines name the callees routines have to be matched before the caller)
- Therefore, we ideally want the *minimal* set of features to identify a routine without false positives

Signatures Generation Strategy

Pragmatic Approach

- Start with a *minimal* set of features for each routine:
    - routine code hash
    - for small routines: called routine names

Signatures Generation Strategy (2)

Pragmatic Approach

- Start with a *minimal* set of features for each routine:
  - routine code hash
  - for small routines: called routine names
- Compare with the routines of the *same* library:
  - If no other routine has the same features, keep the signature as-is
  - If there exists a routine with a different name (= different behavior) and the same features: insert additional features until the collision is resolved

How To Deal With Indistinguishable Routines?

- Some routines do not have particular features to distinguish them, e.g. very small routines:

```
_get_heap_handle proc

                 mov     eax, dword ptr ds:[25B2h]
                 ret

_get_heap_handle endp
```

How To Deal With Indistinguishable Routines?

- Some routines do not have particular features to distinguish them, e.g. very small routines:

```
_get_heap_handle proc

                 mov      eax, dword ptr ds:[25B2h]
                 ret

_get_heap_handle endp
```

- But their location might help:
  - We generate signatures with “low” confidence for these routines; by default those signatures are not used during matching
  - If a memory area contains a large number of routines identified from the same library, we apply the untrusted signatures *to this particular area*

JEB Signatures Packages

- More than 200 packages at this point (mainly x86/x64 Visual Studio, ARM/ARM64 Android NDK)
  - Visual Studio was the primary target, which might explain some biases in the signatures generation process

- Based on identified compiler, packages are loaded at runtime, and each disassembled routine is tentatively matched
  - Also, users can generate their own packages

Android NDK R16 STLport
Android NDK R16 API 21 zlib
Android NDK R17 gnustl
Android NDK R17 libc++
Android NDK R17 libc
Android NDK R17 API 21 libmath
Android NDK R17 STLport
Android NDK R17 API 21 zlib
Android NDK R18 libc++
Android NDK R18 libc
Android NDK R18 API 21 libmath
Android NDK R18 API 21 zlib
Android NDK R19 libc++
Android NDK R19 libc
Android NDK R19 API 21 libmath
Android NDK R19 API 21 zlib
Microsoft Visual C++ 2008 /MD (X64)
Microsoft Visual C++ 2008 /MDd (X64)

Packages List Extract

3. Enough With Broken Assumptions, What's The Point?

Let’s Sum Up

- Originally, we successfully disassembled `secret.exe` with a simplistic recursive algorithm, but we made a lot of assumptions on the way

- Then, we showed that many of these assumptions can be broken just by looking at standard compilers’ code

- You probably have in mind a tons of others broken assumptions we made (instructions do not overlap, code does not modify itself…)
  - It’s the way many obfuscation techniques work, by breaking assumptions made by analysis tools

~~Pessimistic~~ Realistic Conclusion

There is no such thing as a disassembler able to correctly disassemble *all* programs for *all* architectures/compilers

- No interesting assumptions can hold true on so many diverse programs
- If I was still an academic, I would here pedantically mention the link with the halting problem and its generalization, the Rice's theorem

We cannot disassemble correctly all programs, but we might still be able to do “ok” on a subset of them.

What Can We Do?

- Divide the universe of programs into “groups” with the following properties:
  - There exists reliable ways to check if a program belongs to the group
  - There are non trivial assumptions holding true for the whole group

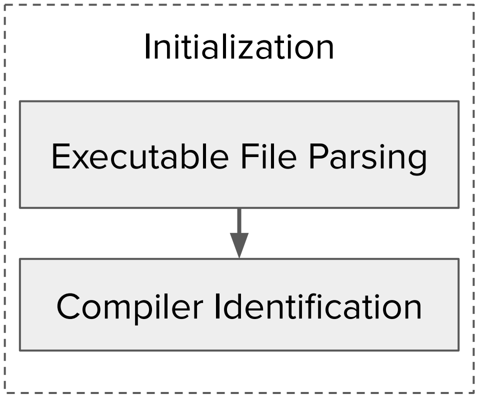
What Can We Do?

- Divide the universe of programs into “groups” with the following properties:
  - There exists reliable ways to check if a program belongs to the group
  - There are non trivial assumptions holding true for the whole group

- When a program does not belong to a known group, apply only conservative assumptions

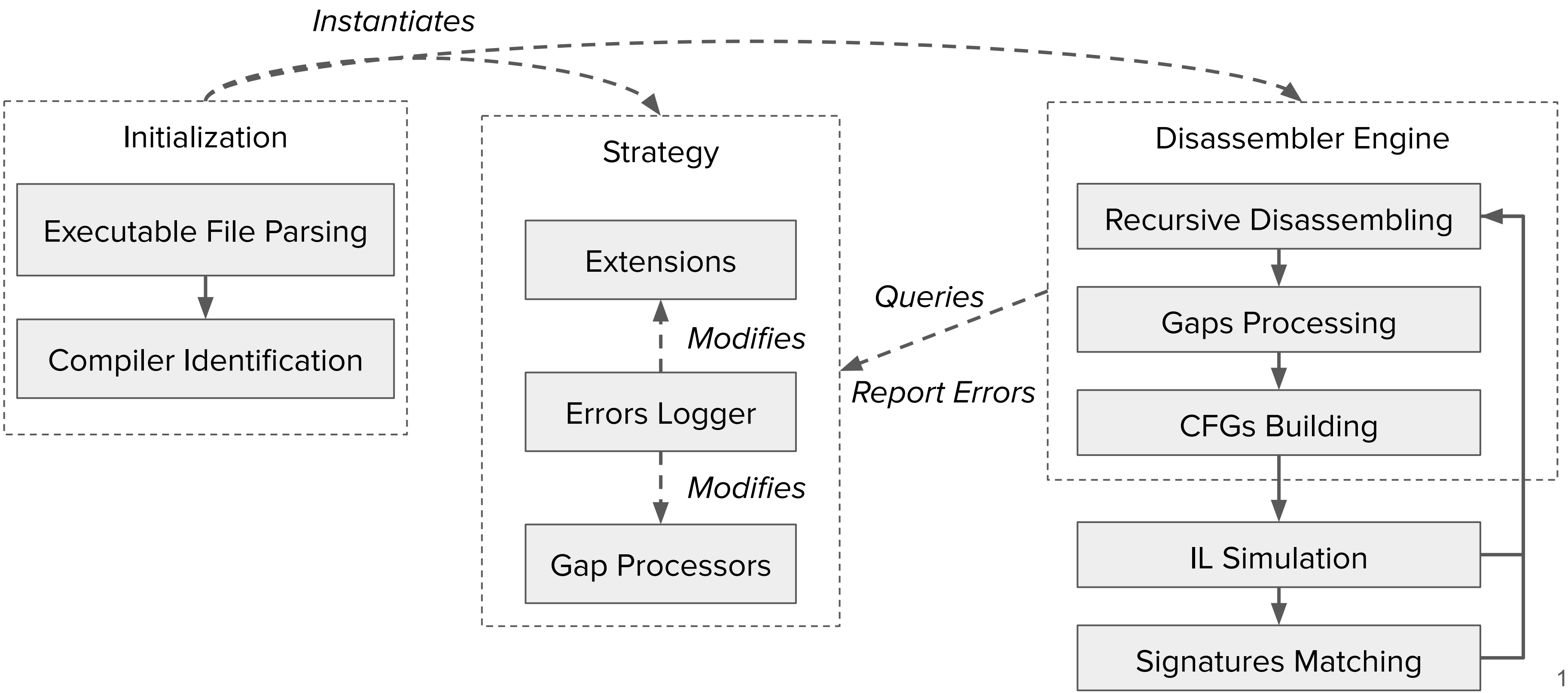
The JEB Way

(Very Simplified) Disassembler Workflow



The JEB Way

(Very Simplified) Disassembler Workflow



Final Notes

- Building “groups” of programs is easier (for developers) if:
    - The disassembler is coded in an “informative” way, i.e. it explicitly reports broken assumptions
    - The disassembler is thoroughly tested on *diverse* sample sets
    - Users report samples breaking the analysis (when they can ;-))
        - But ideally, the disassembler should provide them the way to tweak the broken assumptions

Conclusion

- Hopefully this presentation convinced you (if it was needed) that disassembly remains a complicated problem, albeit an old one

- Interestingly, many novel anti-exploitation techniques tend to make disassembly easier, because they provide hints for “code versus data”
    - ELF executable segments, Microsoft Control Flow Guard, Intel Control-flow Enforcement Technology,...

Thank you!

Contact:

- join us on slack!
- twitter @jebdec
- email us privately at support@pnfsoftware.com