Shattering the monolith:

automatic detection of inlined functions

Ali Rahbar, Elias Bachaalany, Ali Pezeshk

Agenda

- Who we are
- Function inlining
- Forced inlining and reverse engineering
- Detecting inlined functions(high level)
- Detecting inlined function in binary
- Visualization
- Refactoring (Binary rewriting)

Who

- Ali Rahbar : Microsoft TWC
- Elias Bachaalany: Microsoft TWC
- Ali Pezeshk: Microsoft TWC

Background

Function Inlining

- Embedding the function body in place of a function call
- Used as an optimization by compilers
- In C and C++ users can request inlining

```
inline int max(int a, int b)
{
  return (a > b) ? a : b;
}
```

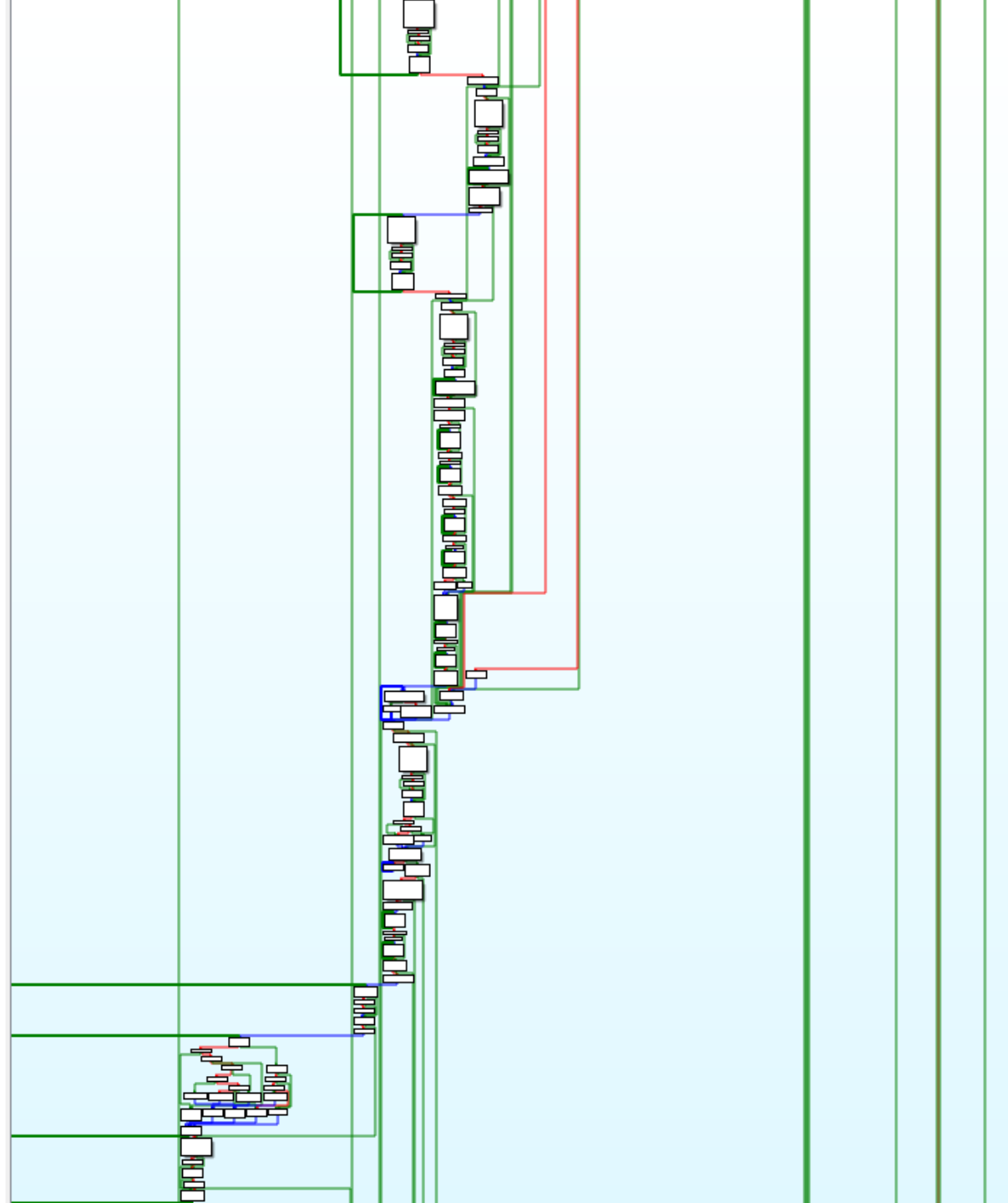
- Could be forced in various compilers:

  Gcc :**__attribute__((always_inline))**

  MSVC :**__forceinline**

Forced inlining and reverse engineering

- Simple control flow obfuscation
- Increases code size
- Increases complexity of analysis
- Identification of equivalent code is not trivial



Algorithm

Problem

- We don't want to re-analyze a code snippet only to find out that we had already seen that before
- Recognizing similar code by hand can become difficult when looking at a large number of inlined functions
- Complex navigation and visualization
- Dirty after decompilation

Solution

- A tool that can
  - Automatically detect inlined functions
  - Match equivalent inlined functions
  - Simplify visualization and interaction
  - Rewrite the binary to outline inlined functions

How to detect inlined functions

Simplifying the problem:

- Only multiple instance of an inlined function are important to detect
- Multiple equivalent code sequence could be potentially an inlined function

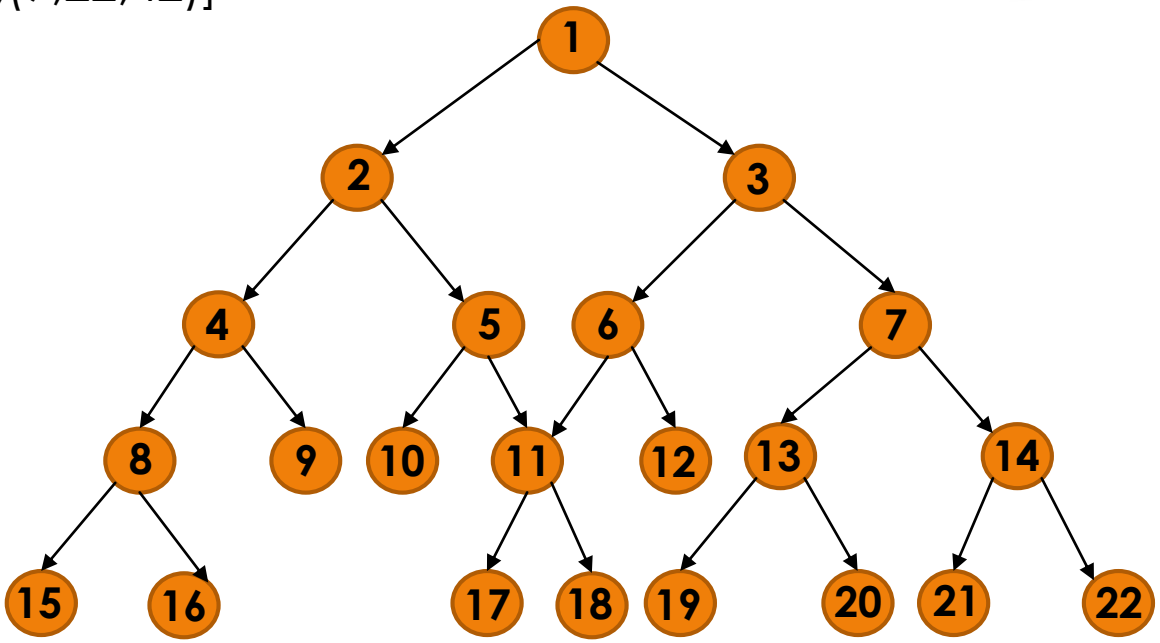
We need a way to detect multiple equivalent code sequences

High level algorithm

1) Build the CFG and break the program into blocks
2) Compare all blocks to construct a list of equivalence
3) For each pair of equivalent blocks, try to construct/find the biggest equivalent subgraphs

Example

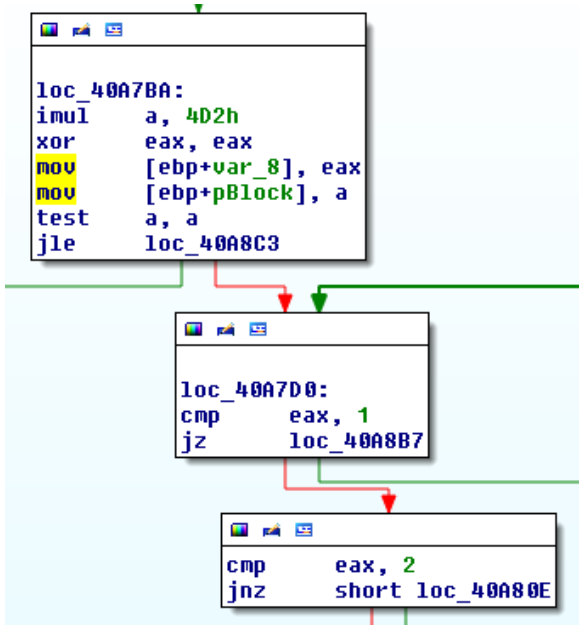
[(2,7,26),(5,13,33),(4,14,34),(9,22,42)]



$subgraph_1\{(2,5),(2,4),(4,9)\}$ == $subgraph_2\{(7,13),(7,14),(14,22)\}$

What are blocks

- Blocks are “basic block” created from the control flow
- A sequence of instruction that do not modify the control flow are in the same block
- Any change in the control flow, starts a new block



Block comparison (1)

- Can't simply compare the bytes in two blocks
- Registers change(register allocation):

```
mov     eax, [ebp+var_24]
cmp     eax, 1
jz      loc_40CE6B
cmp     eax, 2
jnz     loc_40CDC1
add     a, eax
mov     eax, 0AAA000ABh
imul    a
add     edx, a
sar     edx, 0Fh
mov     edi, edx
shr     edi, 1Fh
add     edi, edx
imul    edi, 7Bh
xor     esi, esi
test    edi, edi
jle     loc_40CE6B
```

VS

```
mov     edi, [ebp+pBlock]
cmp     edi, 1
jz      loc_40BCDD
cmp     edi, 2
jnz     short loc_40BC3D
add     a, edi
mov     eax, 0AAA000ABh
imul    a
add     edx, a
sar     edx, 0Fh
mov     edi, edx
shr     edi, 1Fh
add     edi, edx
imul    edi, 7Bh
xor     esi, esi
test    edi, edi
jle     loc_40BCD5
```

Block comparison (2)

- Instructions are reordered(instruction scheduling):

```
ADDS            R1, R3, #2
LDR.W           R3, =0xAAA000AB
SMULL.W         R3, R2, R1, R3
ADDS            R3, R2, R1
ASRS            R3, R3, #0xF
ADD.W           R3, R3, R3,LSR#31
MOVS            R2, #0x7B
MUL.W           R5, R3, R2
MOVS            R4, #0
CMP             R5, #0
BLE             loc_40A256
```

VS

```
LDR             R3, =0xAAA000AB
ADDS            R1, R2, #2
SMULL.W         R3, R2, R1, R3
ADDS            R3, R2, R1
ASRS            R3, R3, #0xF
ADD.W           R3, R3, R3,LSR#31
MOVS            R2, #0x7B
MUL.W           R5, R3, R2
MOVS            R4, #0
CMP             R5, #0
BLE             loc 40A68E
```

Block comparison (3)

- Small variations:

```
MOV             R2, R1
MOVS            R3, #1
BFI.W           R2, R3, #0, #0xC
ORR.W           R3, R2, #0x400
STR             R2, [R4]
STR             R3, [R4,#4]
ORR.W           R3, R2, #0x800
STR             R3, [R4,#8]
ORR.W           R3, R2, #0xC00
STR             R3, [R4,#0xC]
```

VS

```
MOVS            R3, #1
BFI.W           R1, R3, #0, #0xC
ORR.W           R3, R1, #0x400
STR             R1, [R0]
STR             R3, [R0,#4]
ORR.W           R3, R1, #0x800
STR             R3, [R0,#8]
ORR.W           R3, R1, #0xC00
STR             R3, [R0,#0xC]
```

Features

1) Digest of sequence of instruction types in the block:

   An ordered representation of instruction types

2) Digest of the set of instructions and operand types in the block:

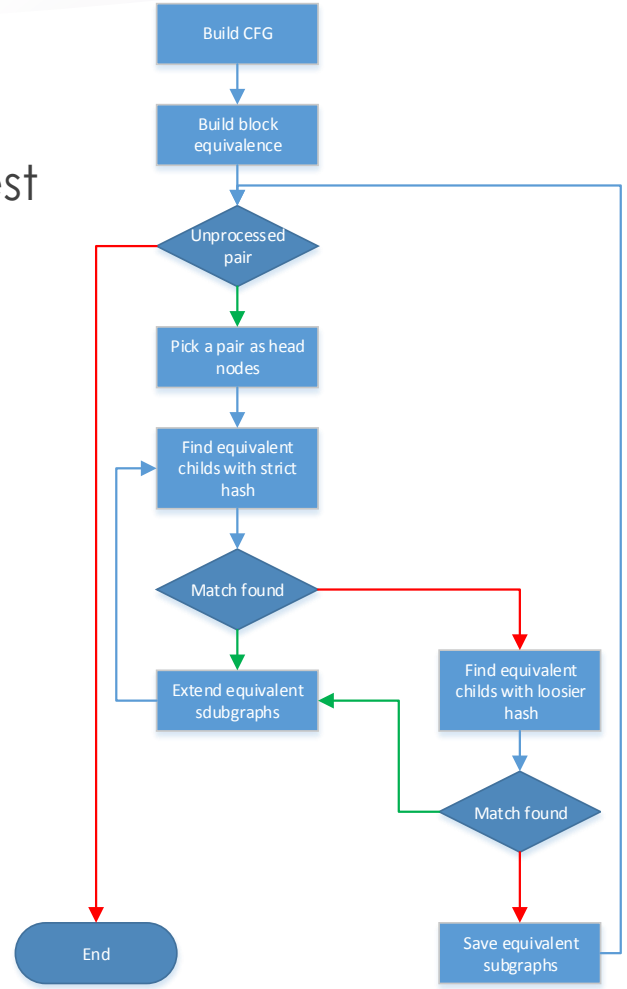
   An order agnostic combination of instruction types and operand types

   (Small Prime Product)

3) Digest of frequency of instruction types and operands types

Algorithm

- Build a CFG
- Use digest2 to calculate block equivalence of all blocks
- Take each pair of equivalent blocks as head nodes
- Try to construct the longest equivalent subgraphs with the strictest digest
- Switch to fuzzier digest if no match is found



DEMO

IDA Plugin Details

BBGrouper

- Python library that contains a set of algorithms
  - Basic Block abstraction layer (BB_types.py)
  - A set of utility functions (BB_util.py)
  - An IDA support library (BB_ida.py)

GraphSlick (1)

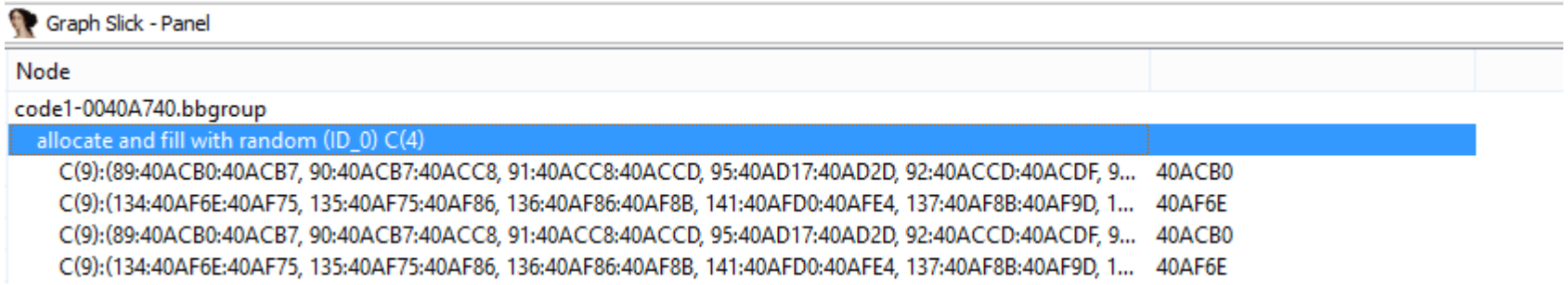
- GraphSlick is an IDA Pro plugin
- Analyze and visualize
- Result can be interactively modified by the user
- The algorithm is processor agnostic

GraphSlick (2)

- The UI represents the results using:
  - A chooser window (aka “GS Panel”): displays the BB Analyze() function call results in a list
  - A user graph window (aka “GS View”):
    - displays a user graph containing the matching results
    - Allows interactivity to manipulate the grouping results
    - Allows coloring and navigating through the results

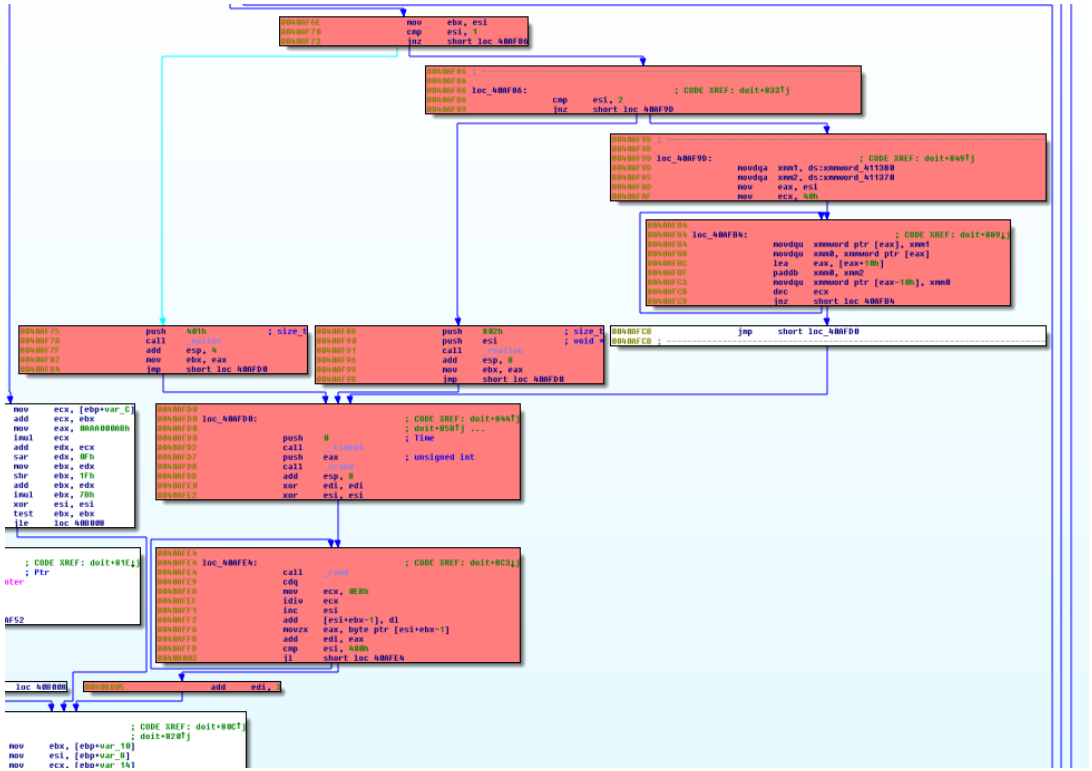
GraphSlick – Panel (1)

- The GS panel shows all the nodes and under which parent group they fall
- Each parent group contains various group each containing a set of nodes that make up similar code pattern in various program location
- "allocate and fill with random" parent group has four groups
  - Each group is composed of 9 basic blocks



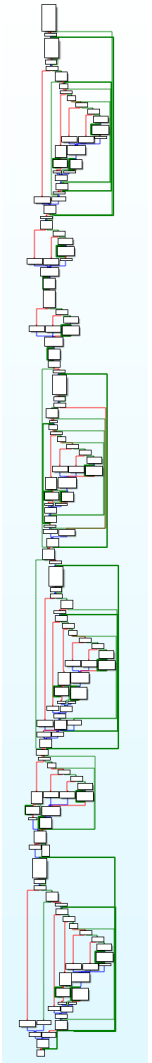
GraphSlick – Panel (2)

- These are the 9 nodes that make up “allocate and fill with random” code logic
- It is composed of 9 basic blocks
- These 9 blocks could have been an inlined function
- Or code that has been copied and pasted

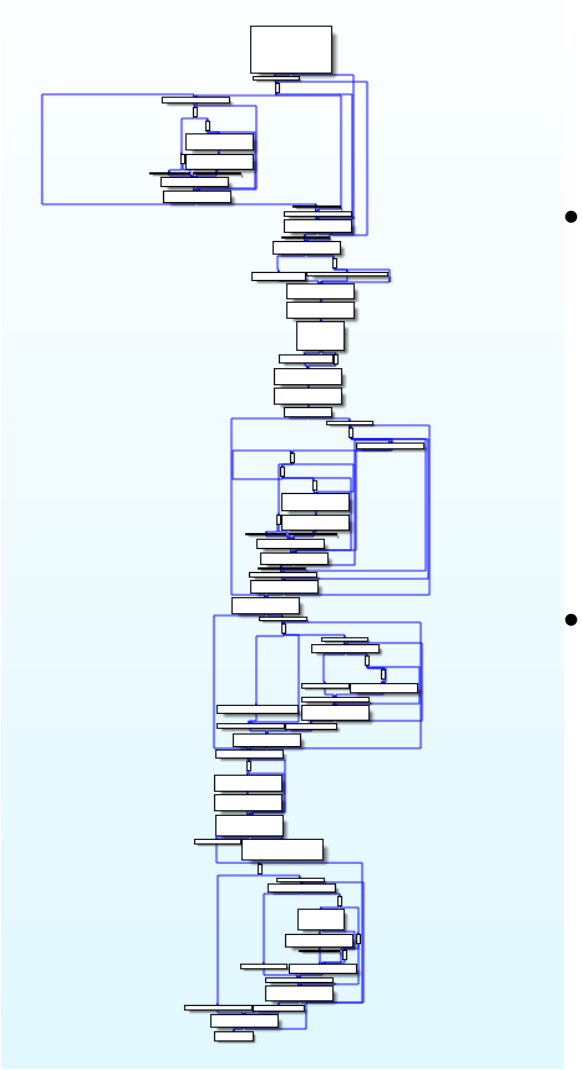


GraphSlick – Before and after

Before



After

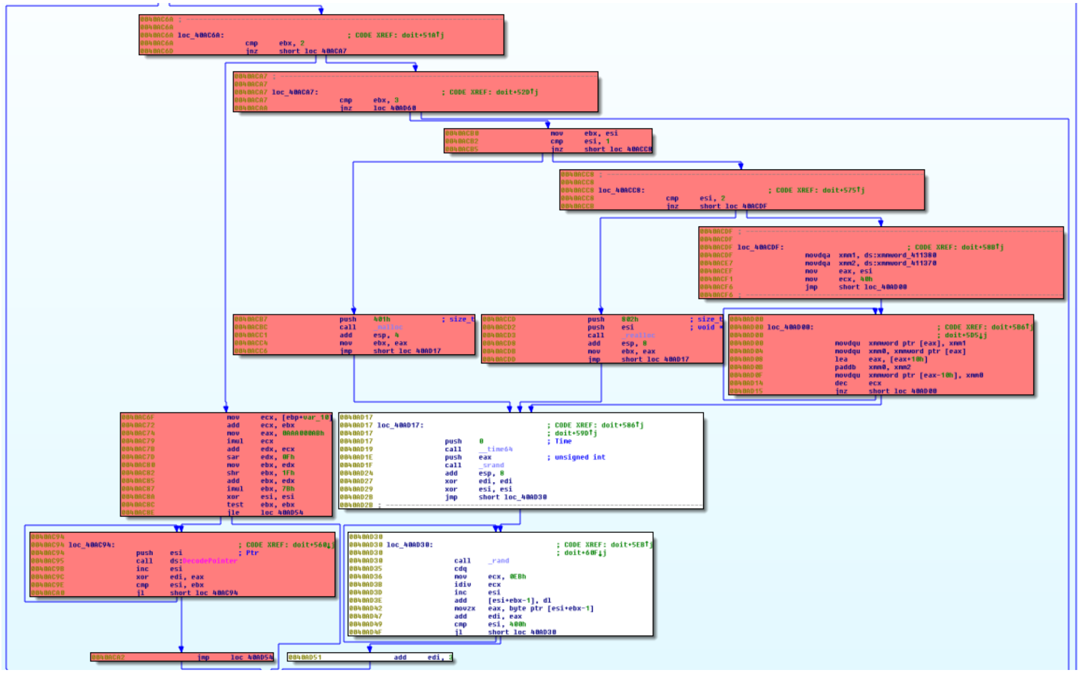


- In this example, we automatically identified similar subgraphs and grouped them under a single parent group
- Note how the graph is much more simpler

GraphSlick – Automatic grouping on x86

SG_2 (ID_2) C(6)

C(11):(84:40AC6A:40AC6F, 85:40AC6F:40AC94, 88:40ACA7:40ACB0, 86:40AC94:40ACA2, 89:40ACB0:40ACB7, 87...	40AC6A
C(11):(106:40ADAC:40ADB1, 107:40ADB1:40ADD3, 110:40ADE0:40ADE9, 108:40ADD3:40ADDB, 111:40ADE9:40A...	40ADAC
C(11):(84:40AC6A:40AC6F, 85:40AC6F:40AC94, 88:40ACA7:40ACB0, 86:40AC94:40ACA2, 89:40ACB0:40ACB7, 87...	40AC6A
C(11):(106:40ADAC:40ADB1, 107:40ADB1:40ADD3, 110:40ADE0:40ADE9, 108:40ADD3:40ADDB, 111:40ADE9:40A...	40ADAC
C(11):(84:40AC6A:40AC6F, 85:40AC6F:40AC94, 88:40ACA7:40ACB0, 86:40AC94:40ACA2, 89:40ACB0:40ACB7, 87...	40AC6A
C(11):(106:40ADAC:40ADB1, 107:40ADB1:40ADD3, 110:40ADE0:40ADE9, 108:40ADD3:40ADDB, 111:40ADE9:40A...	40ADAC

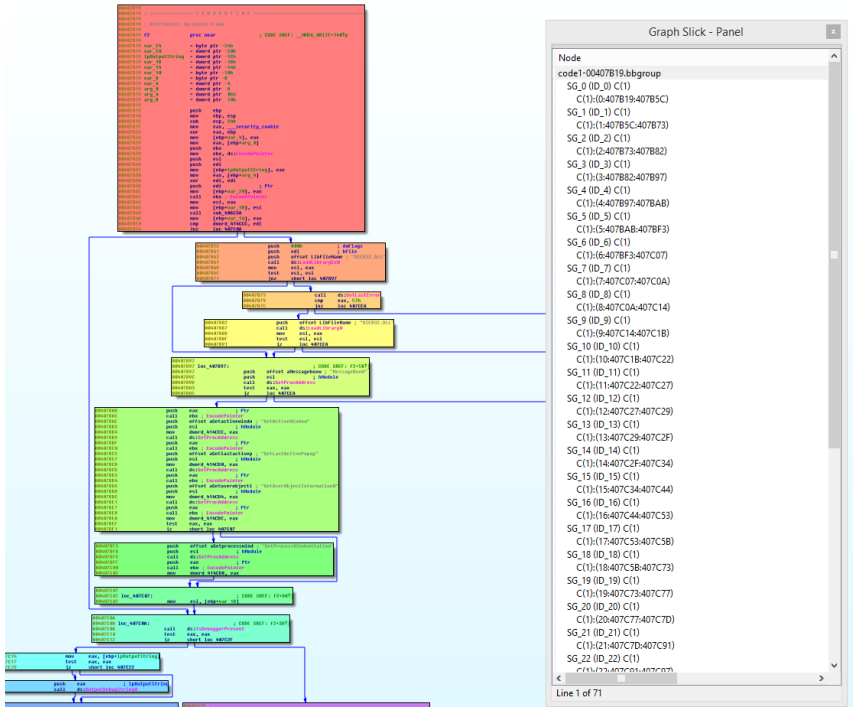


In this example, we can see how the GraphSlick panel:

- Created one parent group with 6 sub groups
- Each sub-group is composed of 11 basic blocks

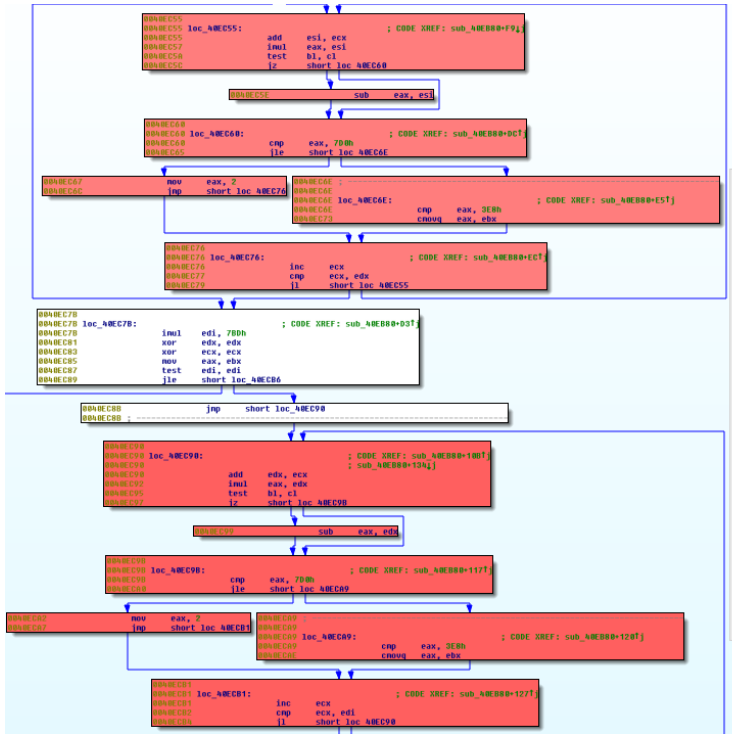
GraphSlick – Automatic coloring

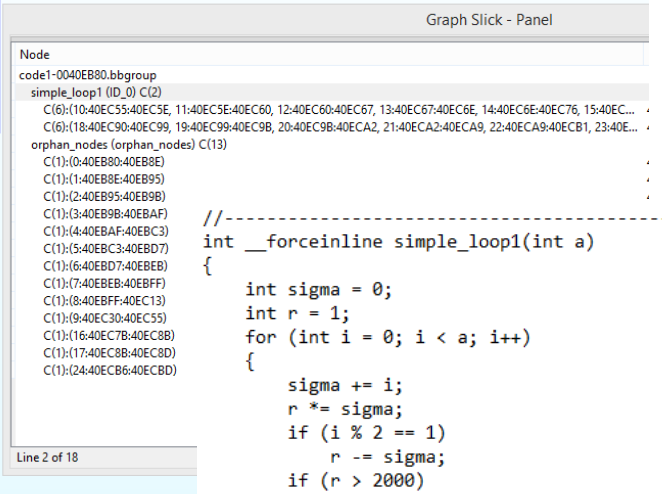
- Here we see a function with no similar basic blocks
- Each basic block is its own parent block
- Each parent block is automatically assigned a distinct color



GraphSlick – Automatic color shades

- The “doit” function has the body of “simple_loop1” inserted twice
- The GS panel detected 6 basic blocks per inlined function call
- All the groups belonging to the same parent group have the same color but with different shade
- The other “Orphan nodes” are just regular blocks






```
//----------------------------------------------------------
int __forceinline simple_loop1(int a)
{
    int sigma = 0;
    int r = 1;
    for (int i = 0; i < a; i++)
    {
        sigma += i;
        r *= sigma;
        if (i % 2 == 1)
            r -= sigma;
        if (r > 2000)
            r = 2;
        else if (r > 1000)
            r = 1;
    }

    return sigma;
}

//----------------------------------------------------------
int doit(int a)
{
    int r = simple_loop1(a * 10);
    r += simple_loop1(a * 1981);
    return r;
}
```

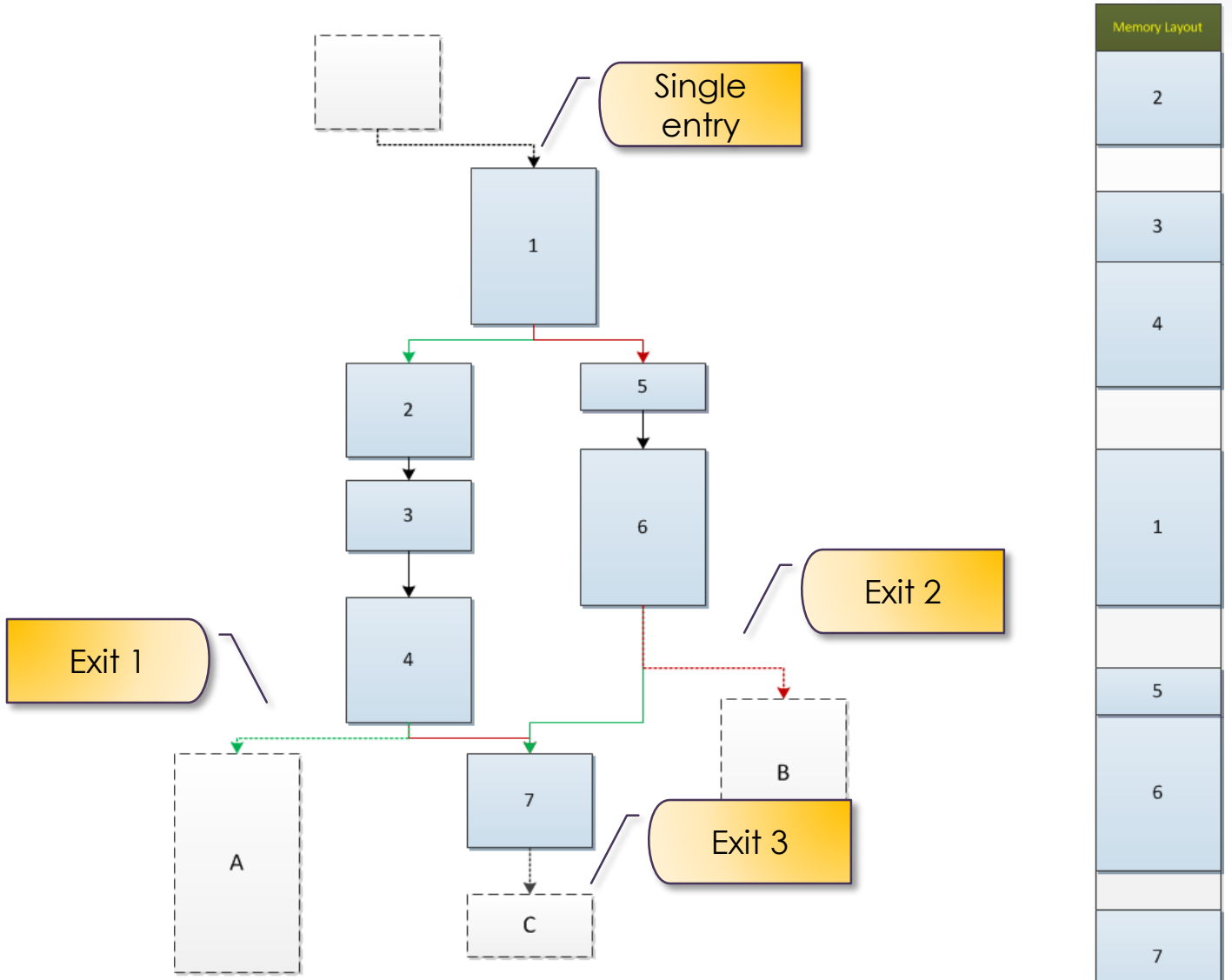
Refactoring

Refactoring overview

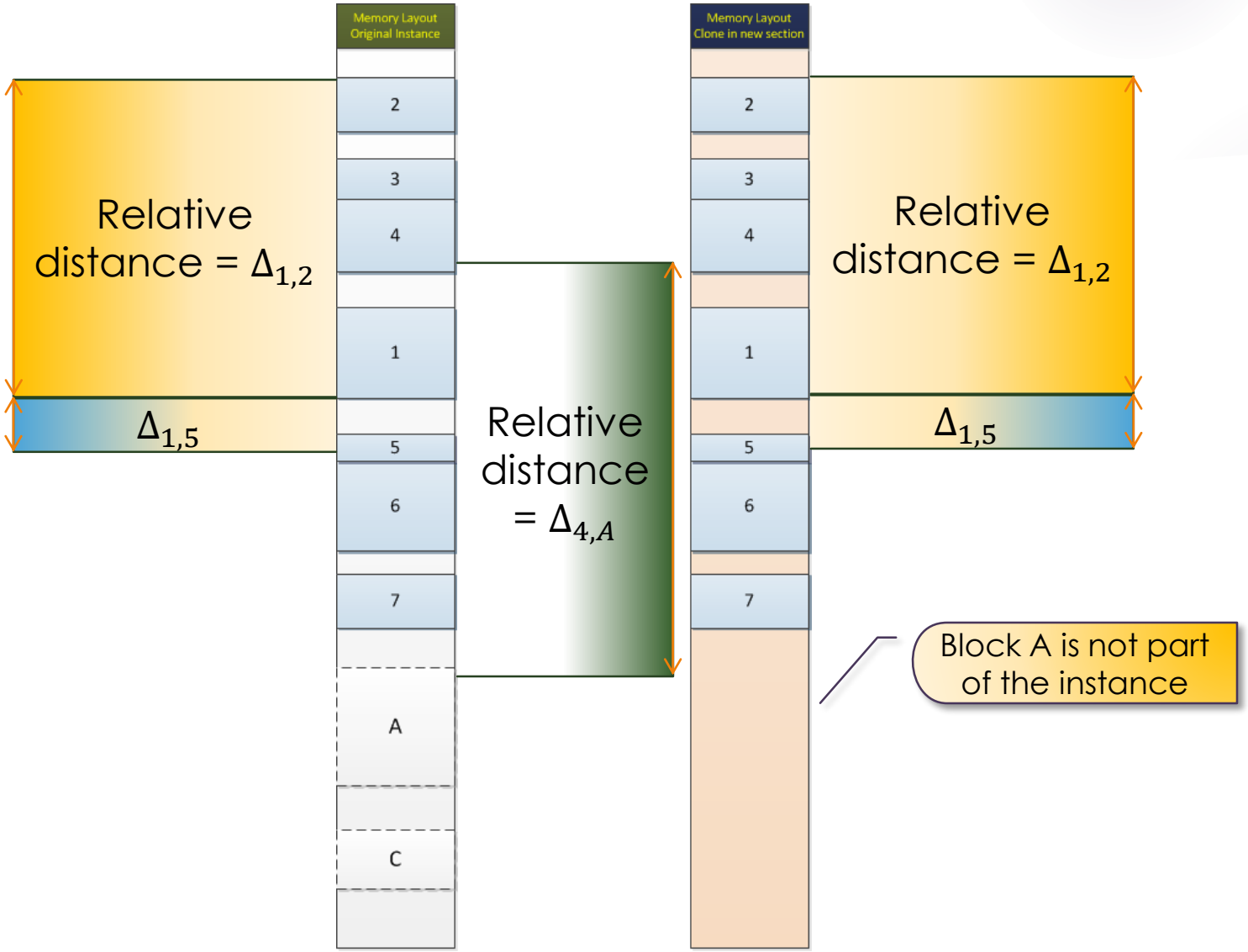
- Add a sufficiently large section to the binary
- Script will
  - Move snippets of code for inlined function instances found to the new section
  - Fix up the moved code for calls, jumps/branches, and returns
  - Patch the original location to do a call in place of the inlined function instance
  - Patch after the call to handle return value to stitch back the original code flow

Note that the following description is for ARM

Glance at an inlined function instance



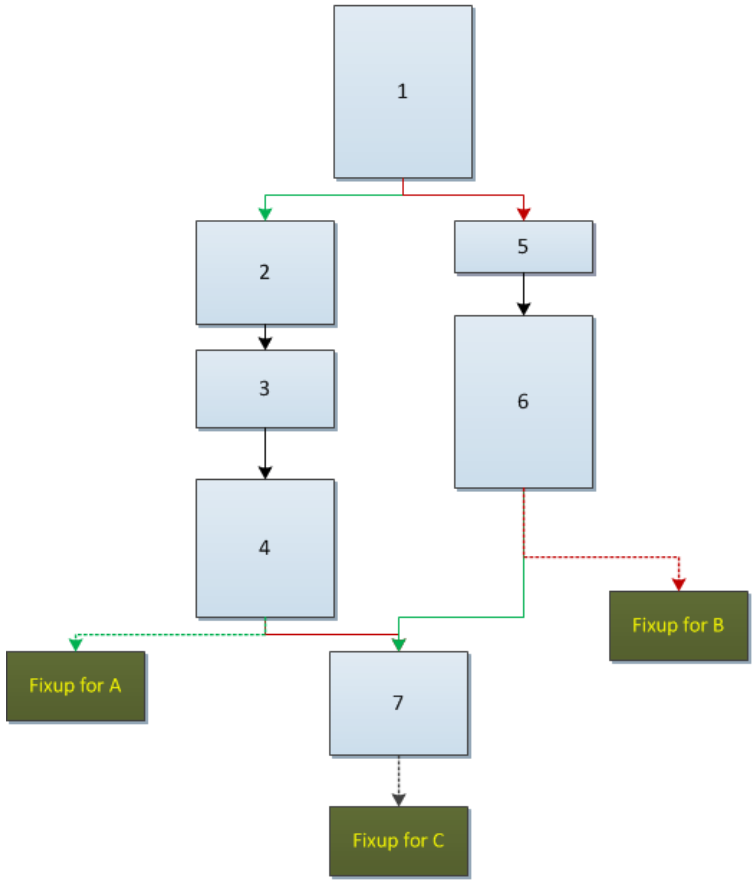
Moving an inlined function instance (1)

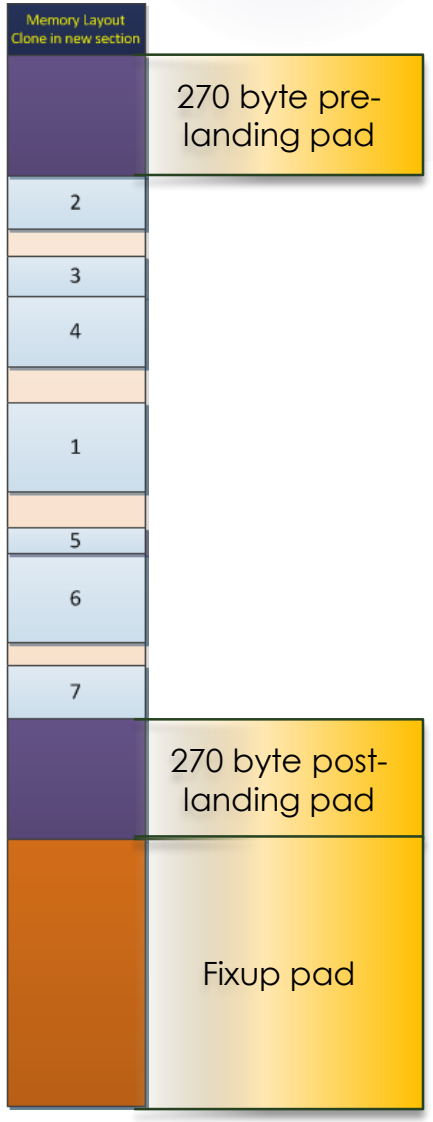


Fixing branches/jumps

- Gaps are preserved when creating the clone
  - Relative branches/jumps that fall back into the instance are good to go
  - Calls outside the instance are good to go as well (absolute address)
  - Any absolute jumps/branches into the instance need to be patched
  - Relative jumps may need to be patched as well

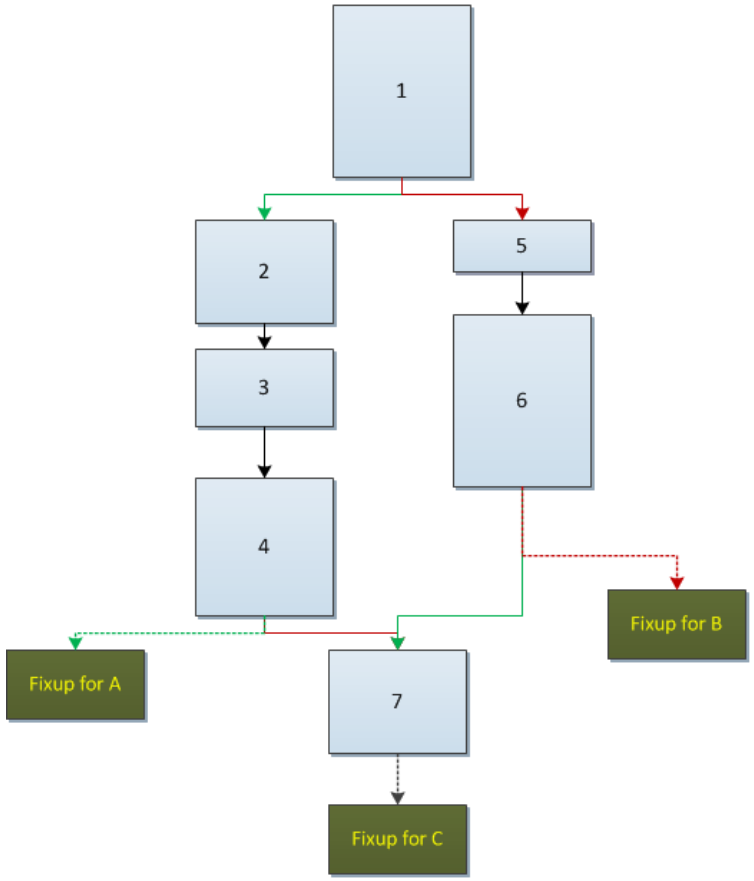
Moving an inlined function instance (2): Envelope

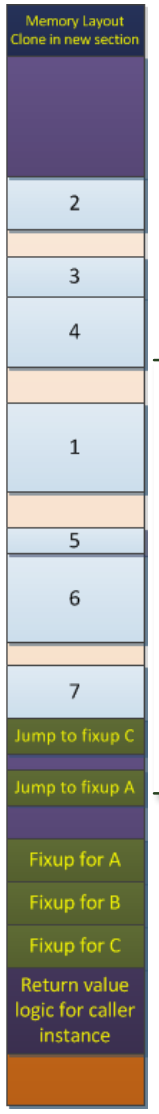




Relative address fixups (1)

- If jump/branch destination falls in a gap or the pre/post landing pads we just add a jump to the fix up at that location
- This helps handling thumb mode 2-byte relative jumps/branches, since the fixup pad can be farther than 258 bytes away and the instruction size won't allow larger distance specified otherwise



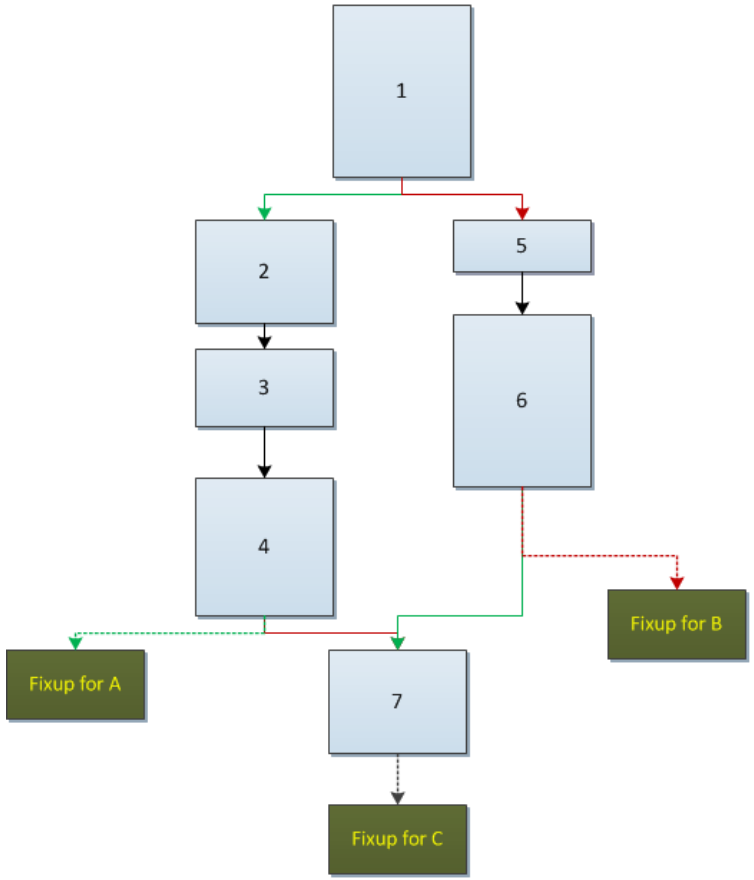


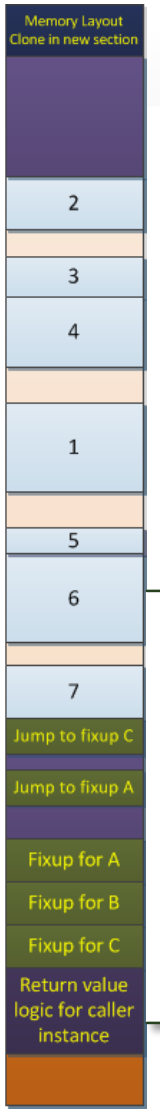
Relative distance = $\Delta_{4,A}$

Happens to fall in the post-landing pad

Relative address fixups (2)

- Depending on $\Delta_{6,B}$ we need to patch the branch/jump instruction to go to the fixup location in the fixup pad



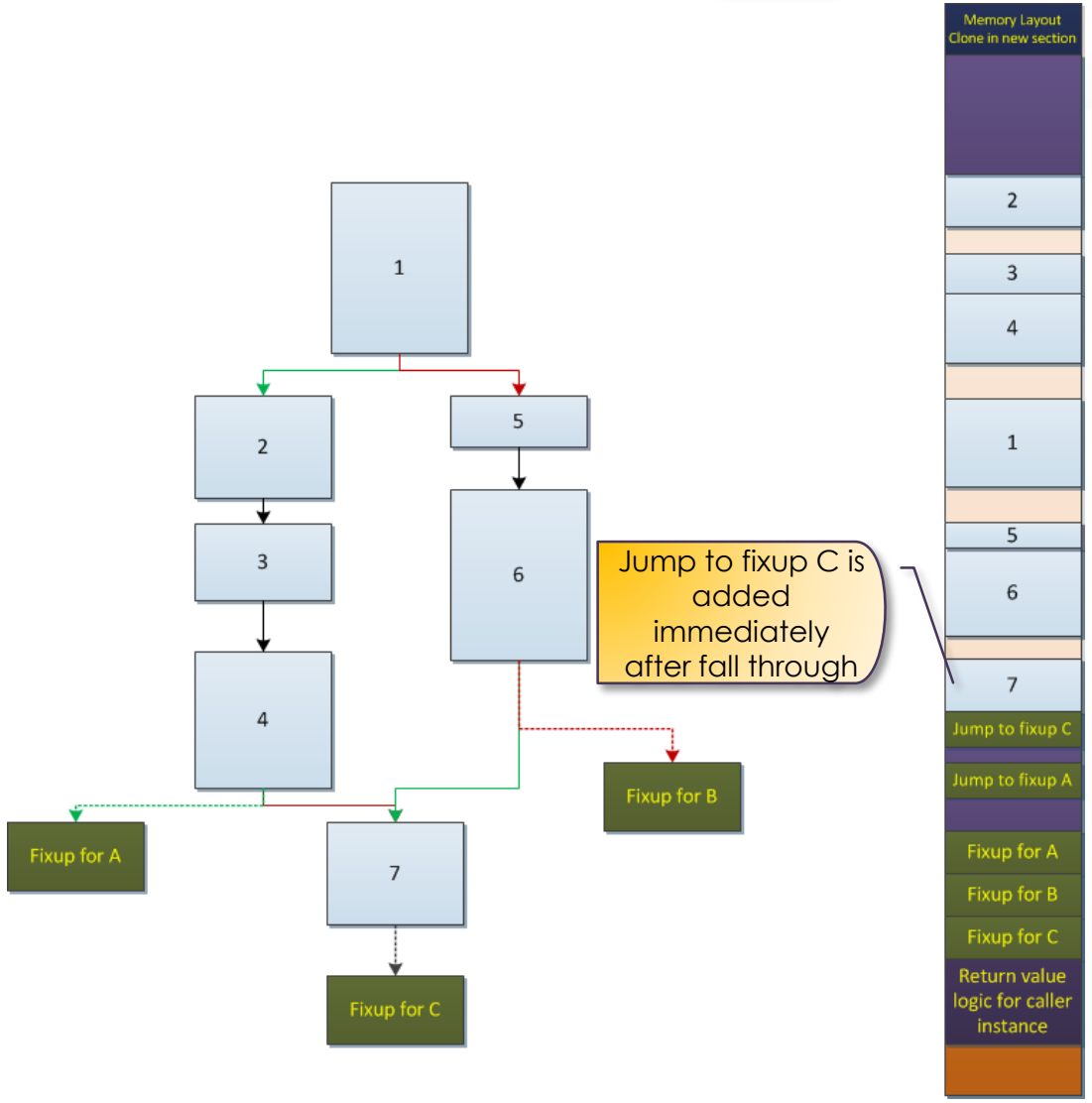


Relative distance = $\Delta_{6,B}$

Happens to fall outisde the post-landing pad
So here the branch/jump needs to be patched

Fixing the fall through to outside of instance

- Fall through location is the instruction address right after end of block 7, which happens to fall outside the instance
- We put a jump to the fixup for C at this location (there's either a gap here or we're in the post-landing pad)



Fixing up flow to outside of instance

- The fixup block for each branch/jump to outside the instance sets a constructed unique return value and returns

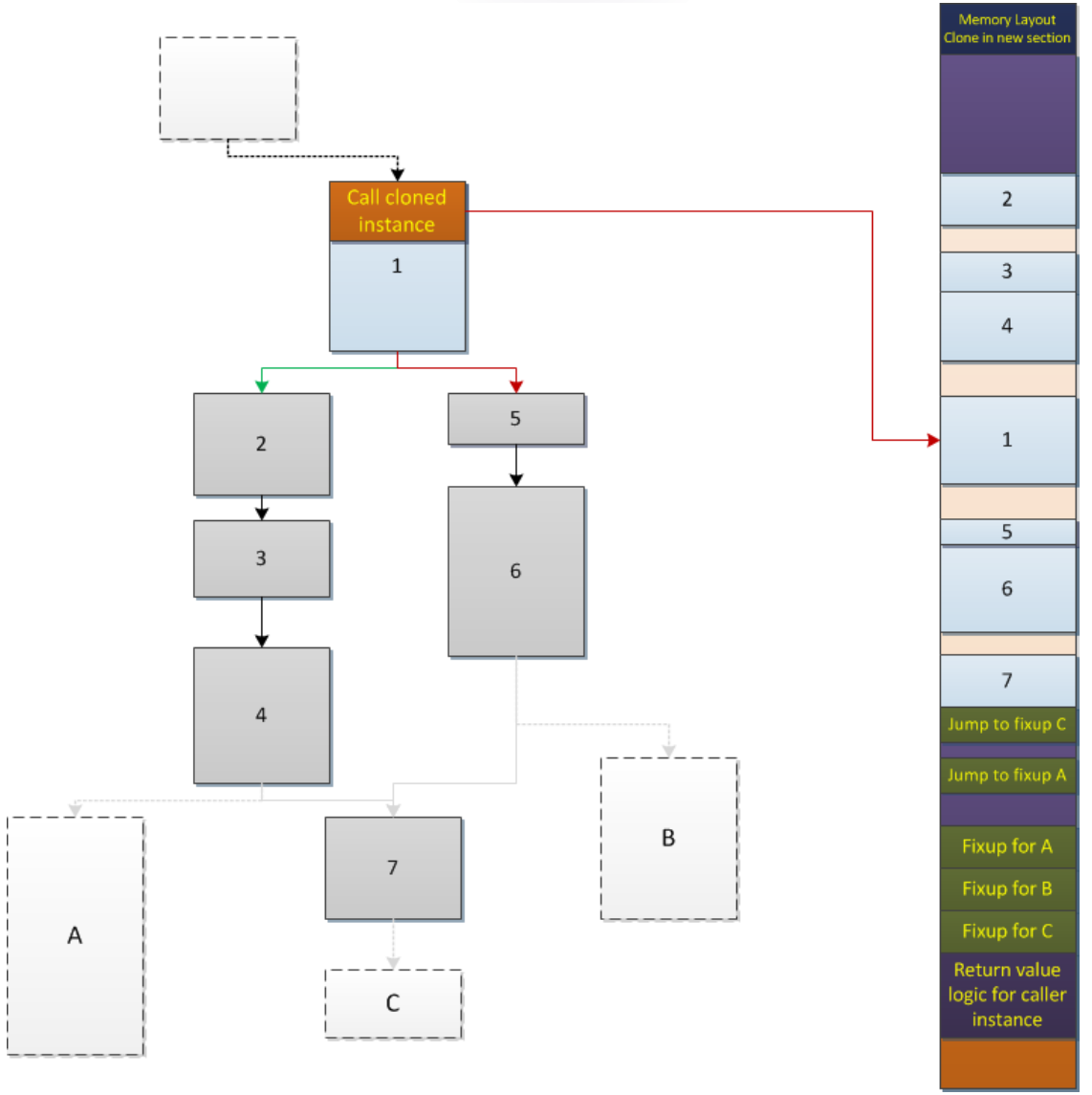
- For instance fixup for A looks like this

```
MOV.W           R0, #1; return value for fixup A
MOV             PC, LR
```

- Then at call location we can use the return value and stich back the flow to the original form with a decision tree

Patching the call site

- We need at least 4 bytes for the call, so we put restrictions on the size of the first block of the instance



Adding return value handling

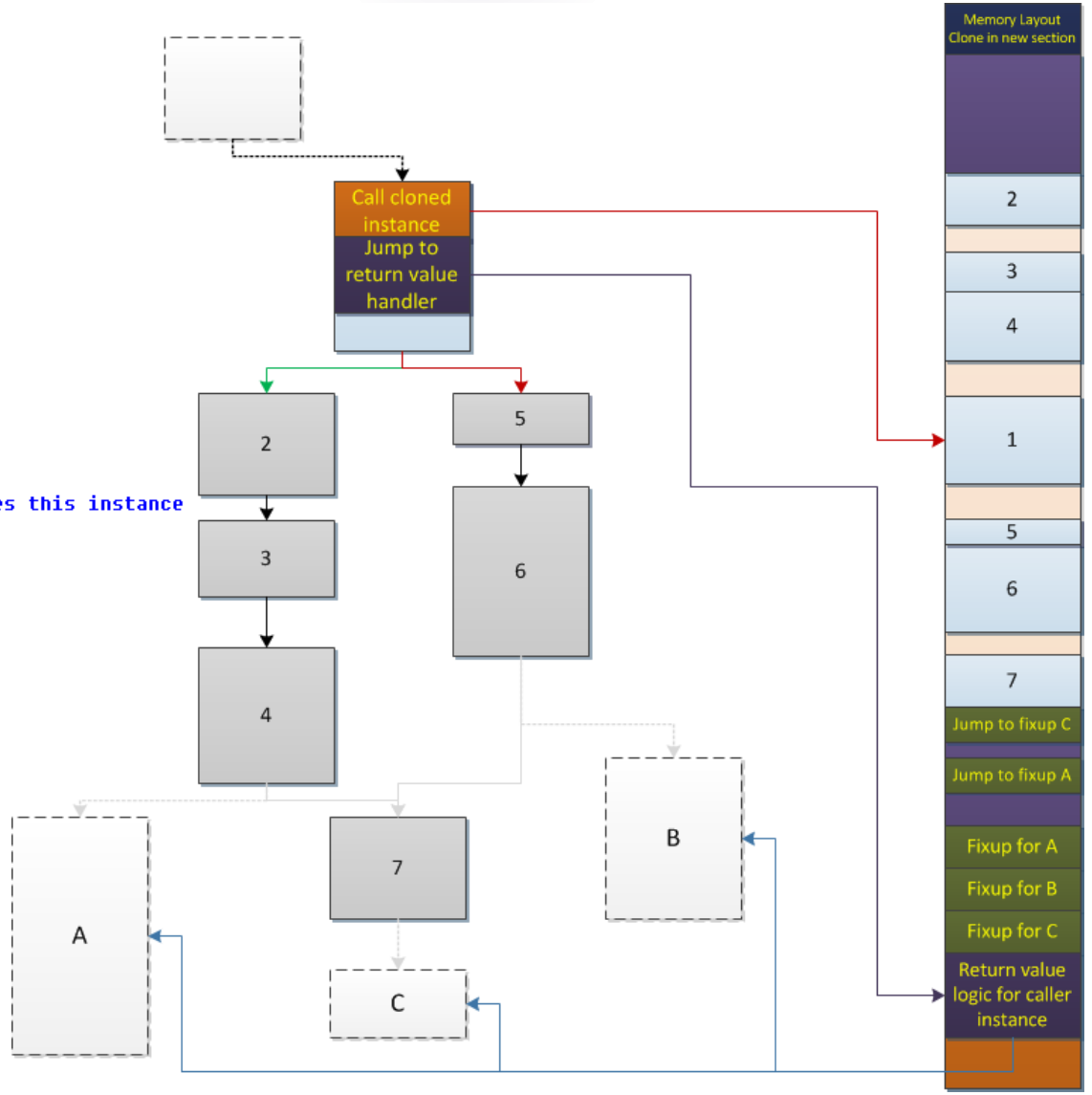
- We need another 4 bytes for the jump to the return value handler logic which we place in the fixup pad after the fixup blocks

- Sample code for the patch

```
BL      sub_9C9488 ; Call the refactored instance function
B.W     loc_9C95BA ; Stitch back code flow. Jump to the location in fixup pad that handles this instance
```

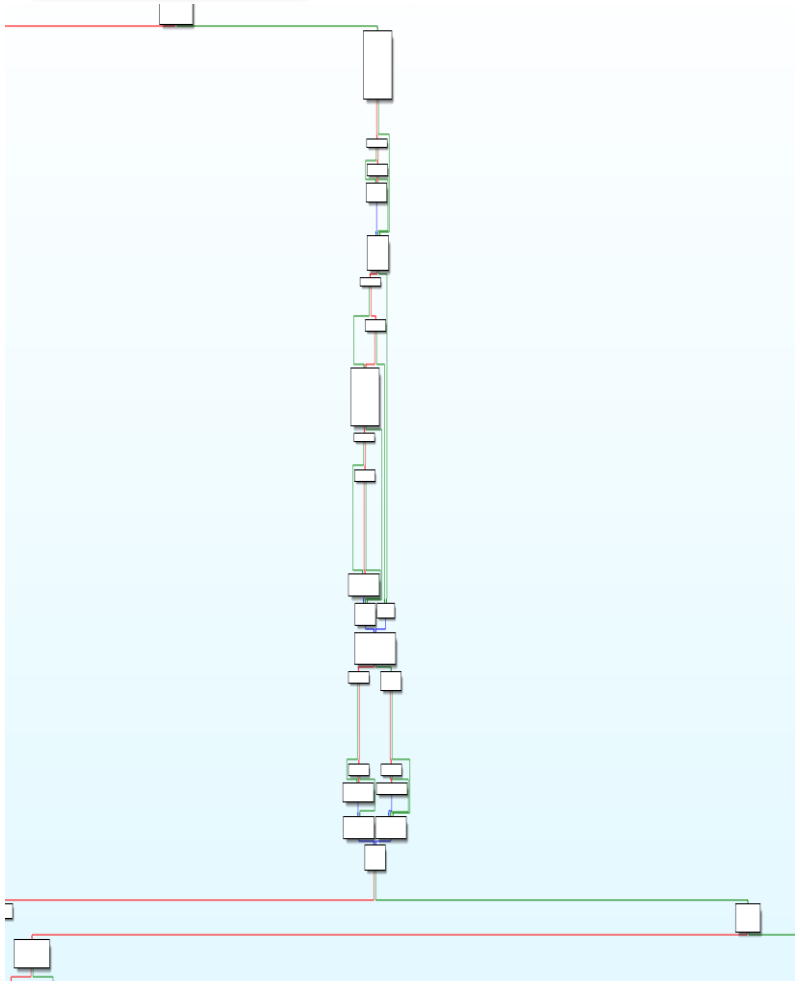
- Sample code for return value logic

```
loc_9C95BA
                MOVS            R0, R0
                CMP             R0, #1
                ITTT EQ
                MOVEQW          R0, #0x5555
                MOVTEQ.W        R0, #0x85
                BXEQ            R0 ; loc_855555
                MOV             R0, #0x85554F
                BX              R0 ; loc_85554E
```

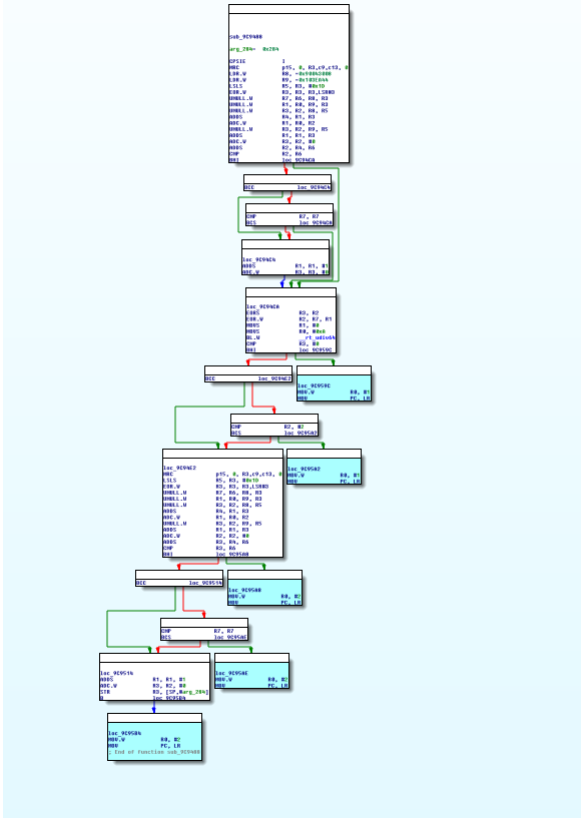


Sample

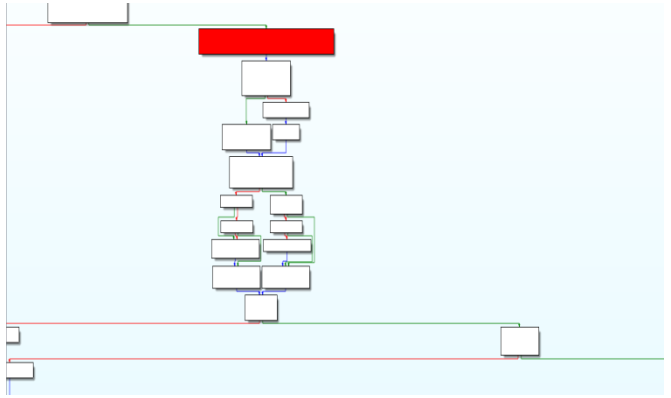
Original binary



Inlined function instance



Refactored binary



What about different register usages

- Different instances of an inlined function may use different registers and parameters may be passed to them through different registers through optimization
- We can clone each instance as described and then in decompilation have the different instances be renamed to the same function

Current implementation limitations

- ARM only
- We currently don't handle jump tables
- Some corner cases still problematic ☺

Questions