Obfuscation (Part 2)

Paolo Falcarin

Ca' Foscari University of Venice

Department of Enviromental Sciences, Informatics and Statistics

paolo.falcarin@unive.it

CM0626 – Software Security

CM0631-2 – Software Security

11 March 2024

Outline

Ca' Foscari University of Venice

- Data Obfuscation
- Method Obfuscation
- Anti-Disassembly
- Self-Modifying Code
- Dynamic JIT Obfuscation
- Implicit Flow

Data Obfuscation

Data Splitting / Data Merging

Ca' Foscari University of Venice

- Data Splitting distributes the information of one variable into several new variables.
  - For example, a boolean variable can be split into two boolean variables, and performing logical operations on them can get the original value.
- Data Merging aggregates several variables into one variable.
  - Collberg et al. (1998) demonstrated an example that merges two 32-bit integers into one 64-bit integer.
  - Ertaul and Venkatesh (2005) proposed another method that packs several variables into one space with discrete logarithms.

Data Proceduralization

- Data proceduralization substitutes static data with procedure calls.
- Collberg et al. proposed to substitute strings with a function which can produce all strings by specifying particular parameter values.
- Drape et al. (2004) proposed to encode numerical data with two inverse functions f and g:
  - to assign a value v to a variable i, we assign it to an injected variable j as j = f (v).
  - to use i, we invoke g(j) instead.

Data Encoding

Ca' Foscari University of Venice

- Data encoding encodes data with mathematical functions or ciphers.
- Ertaul and Venkatesh (2005) proposed to encode strings with ciphers and employ discrete logarithms to pack words.
- Fukushima et al. (2008) proposed to encode the clear numbers with xor operations and then decrypt the computation result before output
- Kovacheva (2013) proposed to encrypt strings with the RC4 cipher and then decrypt them during runtime

Array Transformations

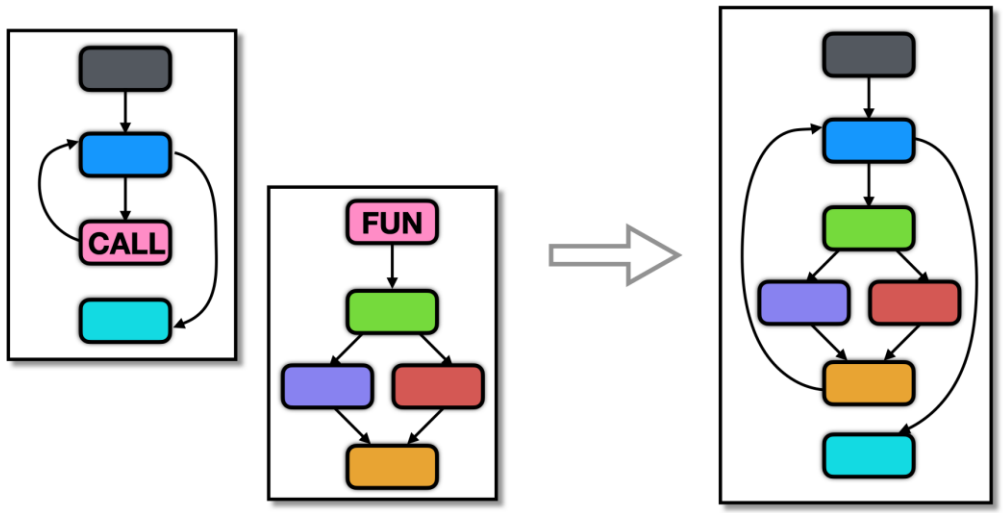
- To obfuscate arrays, Collberg et al. (1998) discussed several transformations, such as:
  - splitting one array into several subarrays
  - merging several arrays into one array
  - folding an array to increase its dimension
  - flattening an array to reduce the dimension
- Ertaul and Venkatesh (2005) suggested transforming the array indices with composite functions.

Method Obfuscation

Method Inline

Ca' Foscari University of Venice

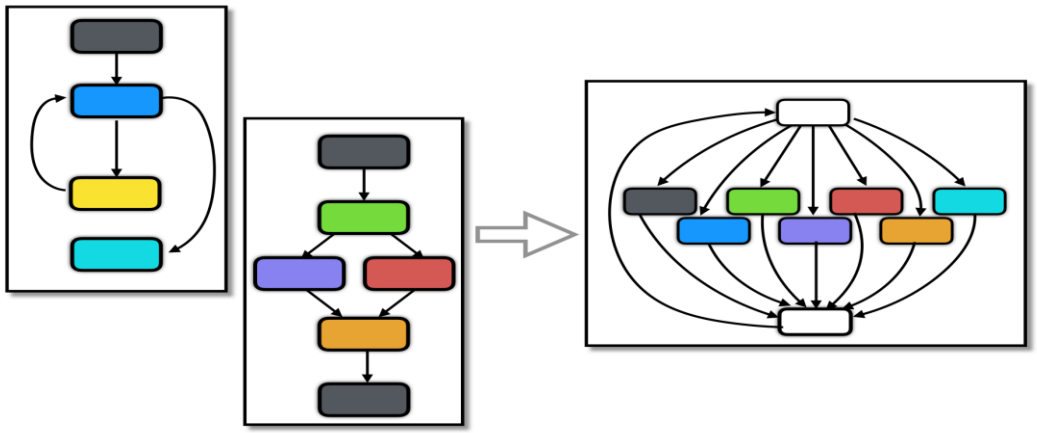
- A method is an independent procedure that can be called by other instructions of the program.
- Method inline replaces the original procedural call with the function body itself.



Method Merge

Ca' Foscari University of Venice

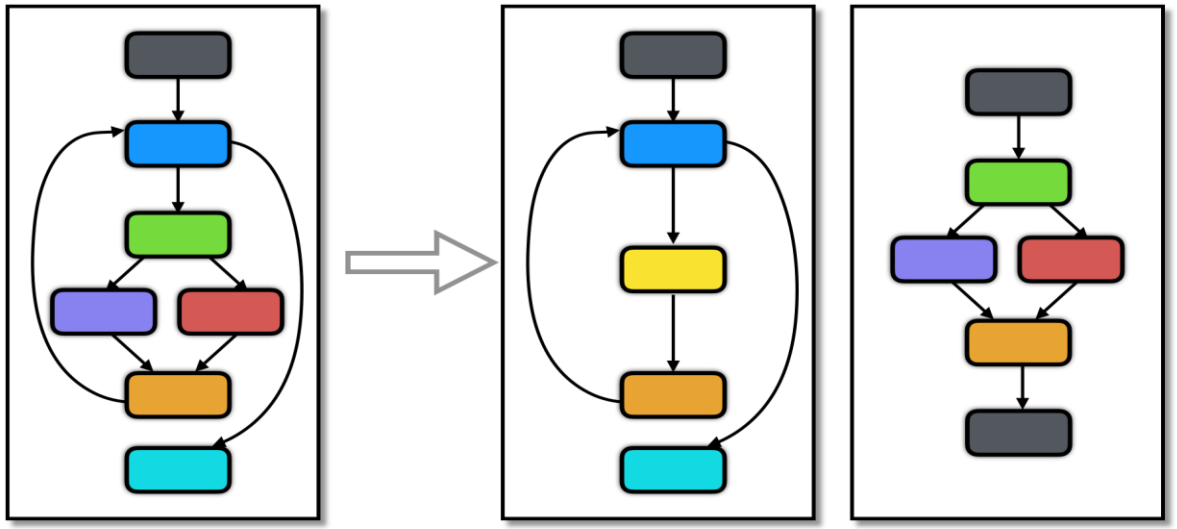
- Merge multiple functions into one: an extra formal argument is added to allow call sites to call any of the functions.
- This transformation is useful as a precursor to virtualization: if you want to virtualize both foo and bar, first merge them together, then virtualize the result.
- The transformation merges the argument list and the local variables of the functions, thereby tying them together.



Method Split (Outline)

Ca' Foscari University of Venice

- It outlines pieces of a function into their own functions.
- This transformation can be used to break a large, virtualized, function into smaller, less noticeable, pieces.
- In Tigress, four different splitting methods are supported.
- The order in which they are tried can affect the genuineness of the resulting code.



Method Split (Tigress)

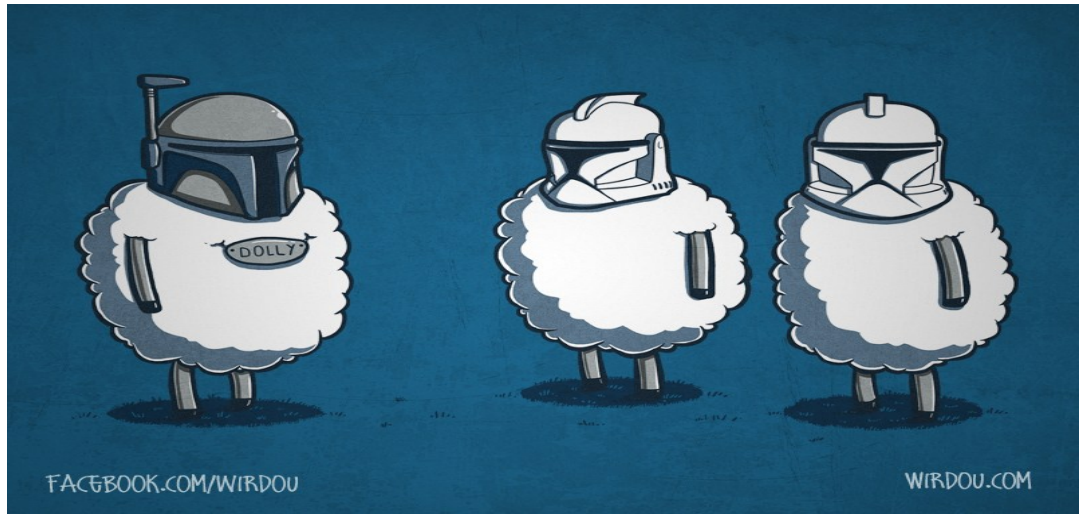
Ca' Foscari University of Venice

Option	Arguments	Description
--Transform	Split	Outline pieces of a function
--SplitKinds	top, block, deep, recursive, level, inside	• Comma-separated list specifying the order in which different split methods are attempted. Default=top,block,deep,recursive. top = split the top-level list of statements into two functions funcname_split_1 and funcname_split_2. • block = split a basic block (list of assignment and call statements) into two functions. • deep = split out a nested control structure of at least height>2 into its own function funcname_split_1. • recursive = same as block, but calls to split functions are also allowed to be split out. • level = split out a statement at a level specified by --SplitLevel. • inside = split out a statement at the innermost nesting level.
--SplitCount	*INTSPEC*	How many times to attempt the split. Default=1.
--SplitName	string	If --SplitName=name is givent, the split out functions will be named *prefix_name_number*, otherwise they will be named *prefix_originalName_split_number*. Default=none.
--SplitLevel	*INTSPEC*	Levels which could be split out when specifying --SplitKinds=level. Default=1.

Method Clone

Ca' Foscari University of Venice

- If a method is heavily invoked, we can create replications of the method and randomly call one of them.
- To confuse adversarial interpretation, each version of the replication should be unique somehow, such as by adopting different obfuscation transformations or different signatures



Method Proxy

- This approach creates proxy methods to confuse reverse engineering.
- For example, creating the proxies as public static methods with randomized identifiers.
- There can be several distinct proxies for the same method (Dalla Preda and Maggi 2017).
- The approach is very useful when the method signatures cannot be changed.

Encode External

Ca' Foscari University of Venice

- The goal of this transformation is to hide calls to external functions, such as system calls or calls to standard library functions.
- Tigress just replace direct calls with indirect ones, and load the address of the functions using dlsym().
- Example:
  - The syscall.c file below makes two system calls getpid and gettimeofday

```
#include<stdio.h>
#include<unistd.h>
#include<sys/time.h>
void tigress_init() {}
 int main () {
 tigress_init();
int x = getpid();
printf("%i\n", x);
struct timeval tv;
int y = gettimeofday(&tv, NULL);
printf ("%ld.%06ld\n", tv.tv_sec, tv.tv_usec);
}
```

Encode External (Tigress script)

Ca' Foscari University of Venice

- The InitEncodeExternal transformation uses dlsym() to load the system calls we want to hide by name,
- the EncodeLiterals transformation hides these names in a function we call STRINGENCODER,
- the Virtualize transformation hides what's going on in the STRINGENCODER function,
- and finally, the EncodeExternal transformation replaces the direct calls to the system calls in main() with indirect ones.

```
> tigress -ldl \
    --Environment=x86_64:Linux:Gcc:4.6  \
    --Transform=InitEncodeExternal \
      --Functions=tigress_init \
      --InitEncodeExternalSymbols=getpid,gettimeofday  \
    --Transform=EncodeLiterals \
      --Functions=tigress_init \
      --EncodeLiteralsKinds=string \
      --EncodeLiteralsEncoderName=STRINGENCODER \
    --Transform=Virtualize \
      --Functions=STRINGENCODER \
    --Transform=EncodeExternal \
      --Functions=main \
      --EncodeExternalSymbols=getpid,gettimeofday  \
    --out=syscall_out.c syscall.c
> gcc -o syscall_out syscall_out.c -ldl
```

… The Resulting code

Ca' Foscari University of Venice

```
void *_externalFunctionPtrArray[2];
void tigress_init(void) {
  STRINGENCODER(0, encodeStrings_litStr0);
  _externalFunctionPtrArray[0] = dlsym((void *)-3, encodeStrings_litStr0);
  STRINGENCODER(1, encodeStrings_litStr1);
  _externalFunctionPtrArray[1] = dlsym((void *)-3, encodeStrings_litStr1);
}
void STRINGENCODER(int n , char str[] ) {
  STRINGENCODER_$sp[0] = STRINGENCODER_$stack[0];
  STRINGENCODER_$pc[0] = STRINGENCODER_$array[0];
  while (1) {
    switch (*(STRINGENCODER_$pc[0])) {
    case STRINGENCODER__store_char$left_STA_0$right_STA_1:
    (STRINGENCODER_$pc[0]) ++;
    *((char *)(STRINGENCODER_$sp[0] + 0)->_void_star) = (STRINGENCODER_$sp[0] + -1)->_char;
    STRINGENCODER_$sp[0] += -2;
    break;    ...
}
```

```
int main( ) {
  int x,y;
  struct timeval tv ;
  ...
  tigress_init();
  x = ((pid_t (*)(void))_externalFunctionPtrArray[1])();
  printf((char const *)"%i\n", x);
  y = ((int (*)(struct timeval * __restrict   , void * __restrict ))
            _externalFunctionPtrArray[0])((struct timeval *)(& tv),
             (void *)((void *)0));
  printf((char const *)"%ld.%06ld\n", tv.tv_sec, tv.tv_usec);
}
```

References

- Collberg C, Thomborson C, Low D (1997) A taxonomy of obfuscating transformations, Technical report. The University of Auckland
- Collberg C, Thomborson C, Low D (1998) Manufacturing cheap, resilient, and stealthy opaque constructs. In: POPL. https://doi.org/10.1145/268946.
- Collberg C, Thomborson C, Low D (1998) Breaking abstractions and unstructuring data structures. In: IEEE International Conference on Computer Languages. https://doi.org/10.1109/iccl.1998.674154
- Fukushima K, Kiyomoto S, Tanaka T, Sakurai K (2008) Analysis of program obfuscation schemes with variable encoding technique. Trans Fundam Electron IEICE Commun Comput Sci. https://doi.org/10.1093/ietfec/e91-a.1.316
- Ertaul L, Venkatesh S (2005) Novel obfuscation algorithms for software security. In: International Conference on Software Engineering Research and Practice (SERP).
- Kovacheva A (2013) Efficient code obfuscation for android. In: International Conference on Advances in Information Technology. Springer. https://doi.org/10.1007/978-3-319-03783-7_10
- Dalla Preda M, Maggi F (2017) Testing android malware detectors against code obfuscation: a systematization of knowledge and unified methodology. Journal Comput Virology Hacking Tech. https://doi.org/10.1007/s11416-016-0282-2
- Srivastava A., Lanzi A., Giffin J., Balzarotti D. (2011) Operating System Interface Obfuscation and the Revealing of Hidden Operations. In: Holz T., Bos H. (eds) Detection of Intrusions and Malware, and Vulnerability Assessment. DIMVA 2011. Lecture Notes in Computer Science, vol 6739. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-22424-9_13

Anti-Disassembly Obfuscations

Anti-Disassembly Obfuscations

Ca' Foscari University of Venice

- Insert Bogus Dead Code
- Insert Irregular Code
- Encode Branches - Branch Functions

Insert Bogus Dead Code

Insert unreachable bogus instructions:

```
if (PF)
 asm(".byte 0x55 0x23 0xff…");
```

This kind of lightweight obfuscation is common in malware.

Example of Dead Code

Ca' Foscari University of Venice

```
uint32 Skypes_hash_function () {
addr_t addr =(addr_t)((uint32)addr ^(uint32)addr);
addr = (addr_t)((uint32) addr + 0 x688E5C);
uint32 hash = 0x320E83 ^ 0x1C4C4 ;
int bound = hash + 0 xFFCC5AFD ;
do {
   uint32 data = *((addr_t)((uint32)addr + 0x10));
   goto b1; asm volatile (".byte 0x19");
   b1: hash = hash ^ data ; addr -= 1; bound--;
} while(bound !=0);
   goto b2; asm volatile (".byte 0x73");
b2: goto b3; asm volatile (".word 0xC8528417,…");
b3: hash -= 0x4C49F346;
return hash;
```

- Unconditional jumps (goto) make the actual Control Flow
- Code defined immediately after an unconditional jump is always unreachable

Insert Irregular Code

Ca' Foscari University of Venice

- Insert uncompleted instructions after unconditional jumps.
- In this way, the uncompleted instructions are unreachable as junk codes.
- If a disassembler cannot handle such uncompleted instructions, they will have troubles when separating instructions.
- Example:
  - In Java bytecode, modify bytecodes directly by employing reserved keywords to name variables and functions.
  - This is possible because only the frontend performs the validation check of identifiers.
  - The resulting modified program can still run correctly, but it would cause troubles for decompilation tools.

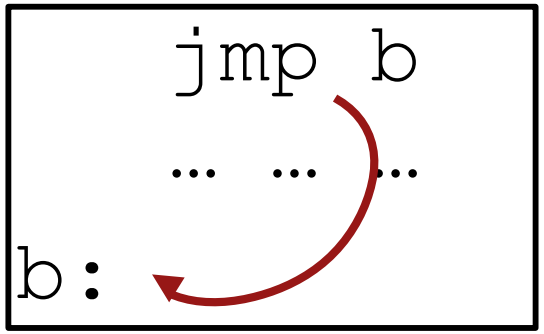
Branch Functions

- The goal of these transformations is to make it harder for automatic analysis tools (such as disassemblers) to determine the target of branches.
- A branch can be realized with:
  - Conditional Jumps (je, jne, jg, ja, …)
  - Unconditional Jumps (goto, jmp)
  - Function Calls (call , ret)

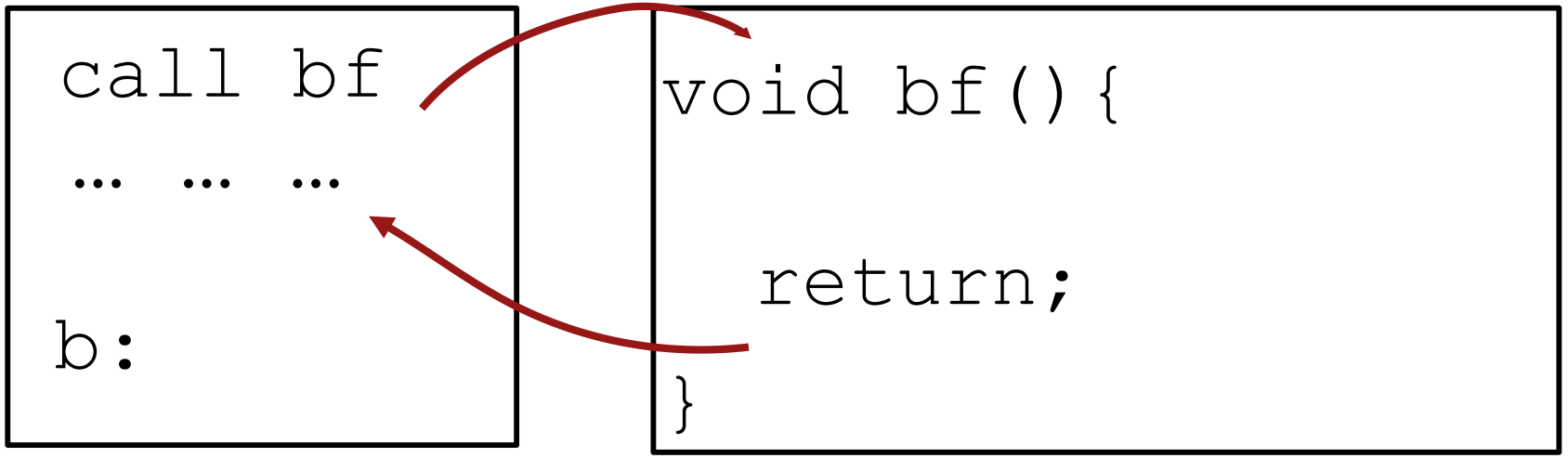
Branch Functions

- Unconditional Jump

  to b address (or label)



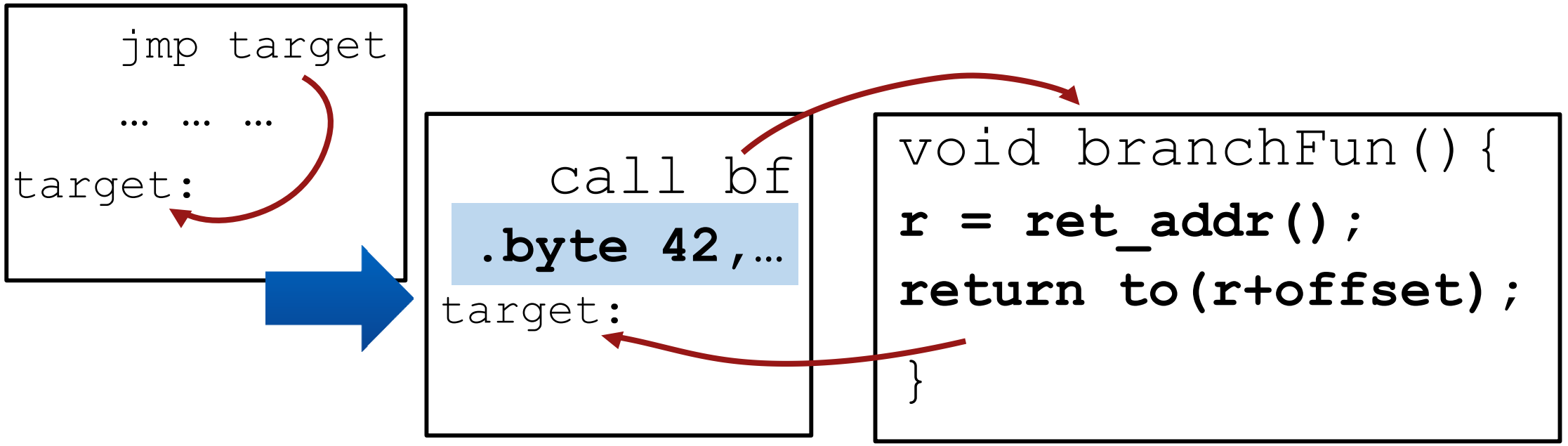
- Function call



Branch Functions

Ca' Foscari University of Venice

- An unconditional Jump to a target address can be transformed to a function call returning to an address plus an offset
- The space between the call and the actual return address can be filled with dead code
- The offset is the number of bytes from the return address to the target address



References

Ca' Foscari University of Venice

- The Branch Function transformation implements a simplistic version of Linn and Debray's Obfuscation of Executable Code to Improve Resistance to Static Disassembly, Linn and Debray's algorithm replaces direct jumps with calls to a special *branch function* which sets the return address to the target of the original branch, and then returns.
- There are many attacks published on branch functions, including Static Disassembly of Obfuscated Binaries by Christopher Kruegel, William Robertson, Fredrik Valeur and Giovanni Vigna, and Deobfuscation: Reverse engineering obfuscated code by Sharath Udupah, Saumya Debray, and Matias Madou.
- Kevin A. Roundy and Barton P. Miller's survey paper Binary-code obfuscations in prevalent packer tools is a good source of information on techniques used by current obfuscation tools.

Example

Ca' Foscari University of Venice

- How does Ghidra do with Branch Functions and Control Flow Flattening?

```
tigress --Environment=x86_64:Linux:Gcc:4.6 \
   --Transform=InitBranchFuns \
      --InitBranchFunsCount=1 \
      --InitBranchFunsObfuscate=false \
   --Transform=AntiBranchAnalysis \
      --AntiBranchAnalysisKinds=branchFuns \
      --AntiBranchAnalysisObfuscateBranchFunCall=false \
      --Functions=play \
   --out=player-bf.c player.c
gcc player-bf.c -o player-bf
```

Example

Ca' Foscari University of Venice

```
1000009c0 - _play
undefined _play()
       undefined          AL:1           <RETURN>
       undefined8         Stack[-0x10]:8 local_10
       undefined1         Stack[-0x78]:1 local_78
       undefined4         Stack[-0x7c]:4 local_7c
       undefined8         Stack[-0x88]:8 local_88
       undefined4         Stack[-0x8c]:4 local_8c
       undefined8         Stack[-0xb0]:8 local_b0
       undefined8         Stack[-0x118... local_118
             _play
...09c0 PUSH  RBP
...09c1 MOV   RBP,RSP
...09c4 SUB   RSP,0x130
...09cb LEA   RAX,[PTR_LAB_100001070]
...09d2 MOV   ECX,0x60
...09d7 MOV   R8D,ECX
...09da MOV   R9,qword ptr [->__got::___...
...09e1 MOV   R9=>->__got::___stack_chk_...
...09e4 MOV   qword ptr [RBP + local_10]...
...09e8 MOV   dword ptr [RBP + local_7c]...
...09eb MOV   qword ptr [RBP + local_88]...
...09ef MOV   dword ptr [local_8c + RBP]...
...09f5 LEA   RSI=>local_78,[RBP + -0x70]
...09f9 MOV   RDI,RSI
...09fc MOV   RSI=>PTR_LAB_100001070,RAX
...09ff MOV   RDX,R8
...0a02 CALL  __stubs::_memcpy
...0a07 MOV   qword ptr [local_b0 + RBP]...
...0a12 MOV   RAX,qword ptr [local_b0 + ...
...0a19 SUB   RAX,0x0
...0a1d MOV   RAX,qword ptr [RBP + RAX*0...
...0a22 MOV   qword ptr [local_118 + RBP...
...0a29 JMP   LAB_100000d98
```

```
100000d98 - LAB_100000d98
             LAB_100000d98
...0d98 MOV   RAX,qword ptr [local_118 +...
...0d9f JMP   RAX
```

```
1
2  void _play(void)
3
4  {
5    undefined local_78 [88];
6    code *UNRECOVERED_JUMPTABLE;
7    undefined8 local_10;
8
9    local_10 = *(undefined8 *)___stack_chk_guard;
10   _memcpy(local_78,&PTR_LAB_100001070,0x60);
11                     /* WARNING: Could not recover jumptable at 0x000100000d9f. Too many branches */
12                     /* WARNING: Treating indirect jump as call */
13   (*UNRECOVERED_JUMPTABLE)();
14   return;
15 }
16
```

Exercise

Ca' Foscari University of Venice

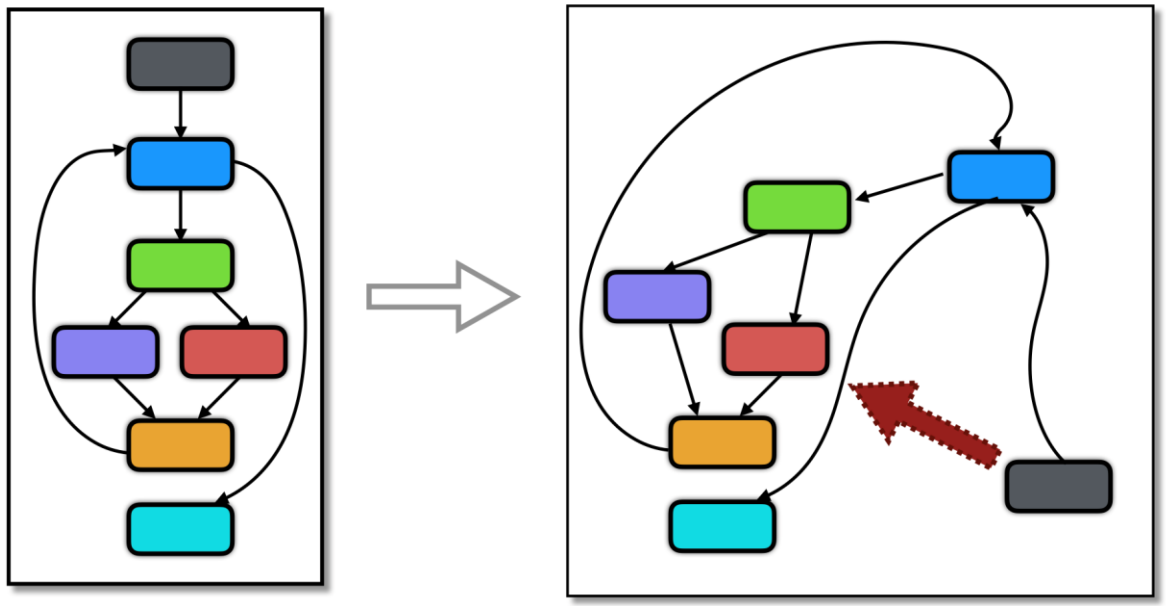
```
************************************************************
* 7) Branch Functions
************************************************************
tigress --Environment=x86_64:Linux:Gcc:4.6 \
                    --Transform=InitBranchFuns \
                       --InitBranchFunsCount=1 \
  --Transform=AntiBranchAnalysis \
     --AntiBranchAnalysisKinds=branchFuns \
     --Functions=fib \
  --out=fib7.c fib.c
```

Self-Modifying Code

Abnormal Programs

Ca' Foscari University of Venice

- In a "normal" program, the code segment does not change.
- But, the programs in this course are not normal !
- Programs which change the code segment are called self-modifying.



Example: Instruction Set (1)

Ca' Foscari University of Venice

Opcode	Mnemonic	Operands	Semantics
0	call	addr	function call to addr
1	calli	reg	function call to address in reg
2	brg	offset	branch to pc + offset if flags for > are set
3	inc	reg	reg ← reg + 1
4	bra	offset	branch to pc + offset
5	jmpi	reg	jump to address in reg
6	prologue		beginning of function
7	ret		return from function

Example: Instruction Set (2)

Ca' Foscari University of Venice

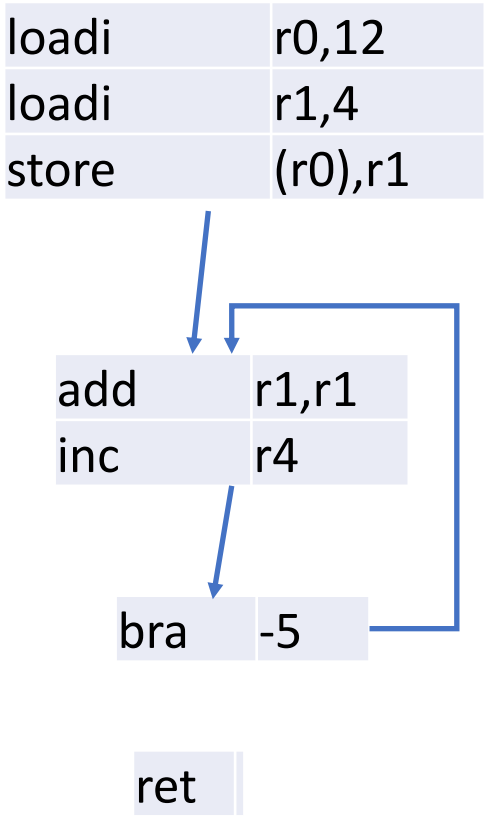
Opcode	Mnemonic	Operands	Semantics
8	load	reg1,(reg2)	reg1 ← [reg2]
9	loadi	reg, imm	reg ← imm
10	cmpi	reg, imm	compare reg and imm and set flags
11	add	reg1, reg2	reg1 ← reg1 + reg2
12	brge	offset	branch to pc + offset if flags for ≥ are set
13	breq	offset	branch to pc + offset if flags for = are set
14	store	(reg1), reg2	[reg1] ← reg2

Example: Code

Ca' Foscari University of Venice

0:	[9,0,12]	loadi	r0, 12
3:	[9,1,4]	loadi	r1, 4
6:	[14,0,1]	store	(r0), r1
9:	[11,1,1]	add	r1, r1
12:	[3,4]	inc	r4
14:	[4,-5]	bra	-5
16:	[7]	ret	

Building the CFG is simple!



The CFG isn't even connected!

The backwards branch at position 14 forms an infinite loop!

Look Again!

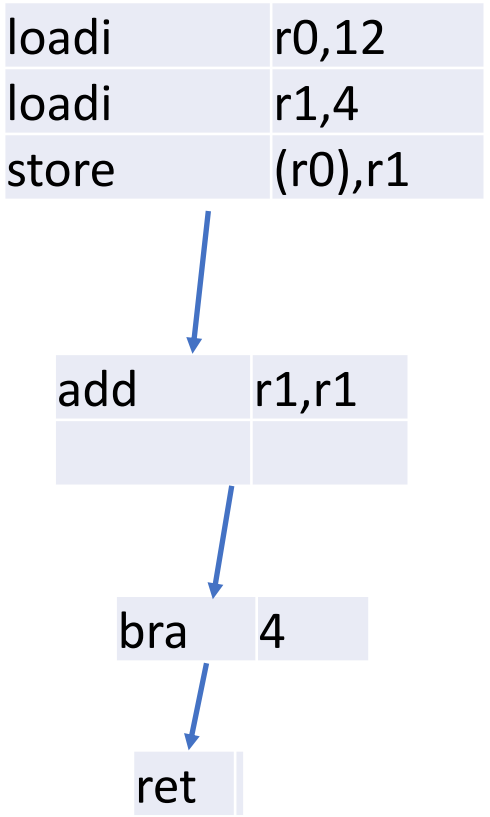
Ca' Foscari University of Venice

Address	Opcodes	Mnemonics	Operands
0:	[9, 0, 12]	loadi	r0, 12
3:	[9, 1, 4]	loadi	r1,4
6:	[14, 0, 1]	store	(r0),r1
9:	[11, 1, 1]	add	r1,r1
12:	[3, 4]	inc	r4
14:	[4,-5]	bra	-5
16:	[7]	ret	

- The `store` instruction is writing the byte 4 to position 12…
- …changing the `inc r4` instruction into a `bra 4` !

- The opcode of bra is 4…
- The operand is 4 and it jumps forward of 4 bytes to address 16

The Actual CFG



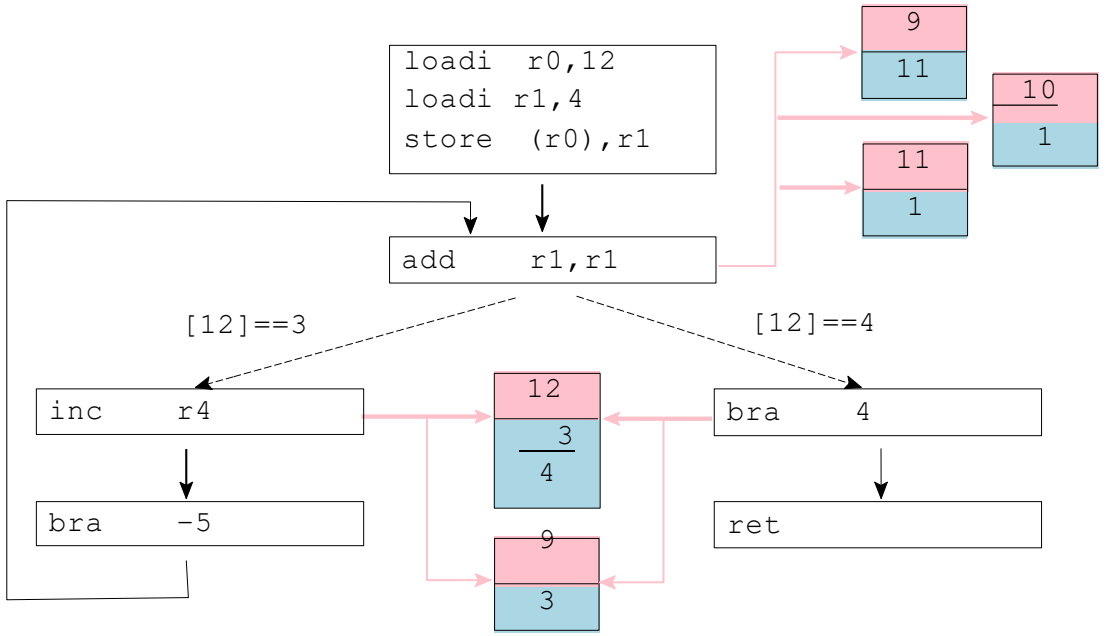
The actual control flow graph looks like this.

If the code bytes are changing at runtime, a standard control flow graph is not sufficient.

New CFG Model

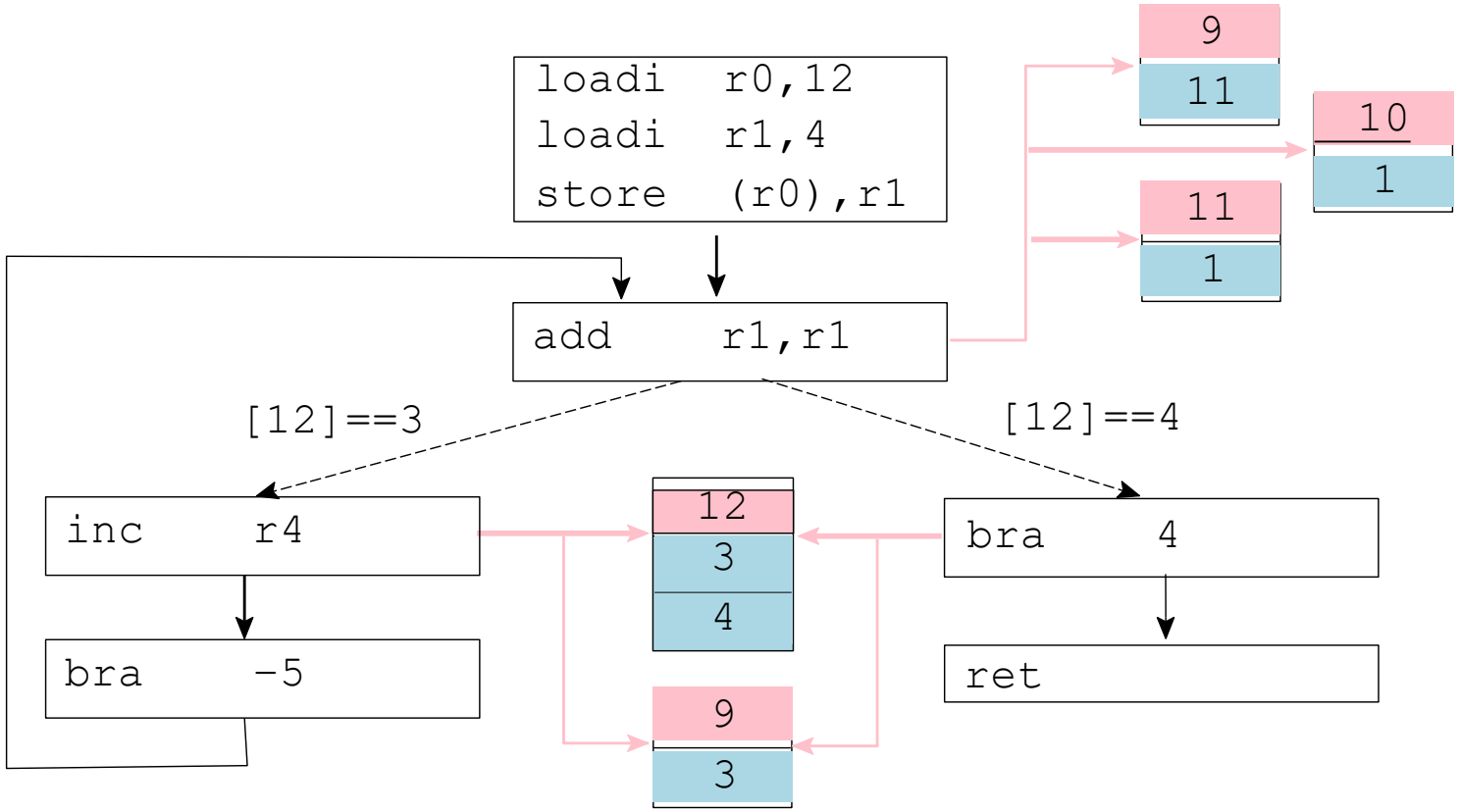
Ca' Foscari University of Venice

- Add a code byte data structure to the graph that represents all the different states each instruction can be in.
- Add conditions to the edges; only if the condition on an edge is true can control take that path.



New CFG Model

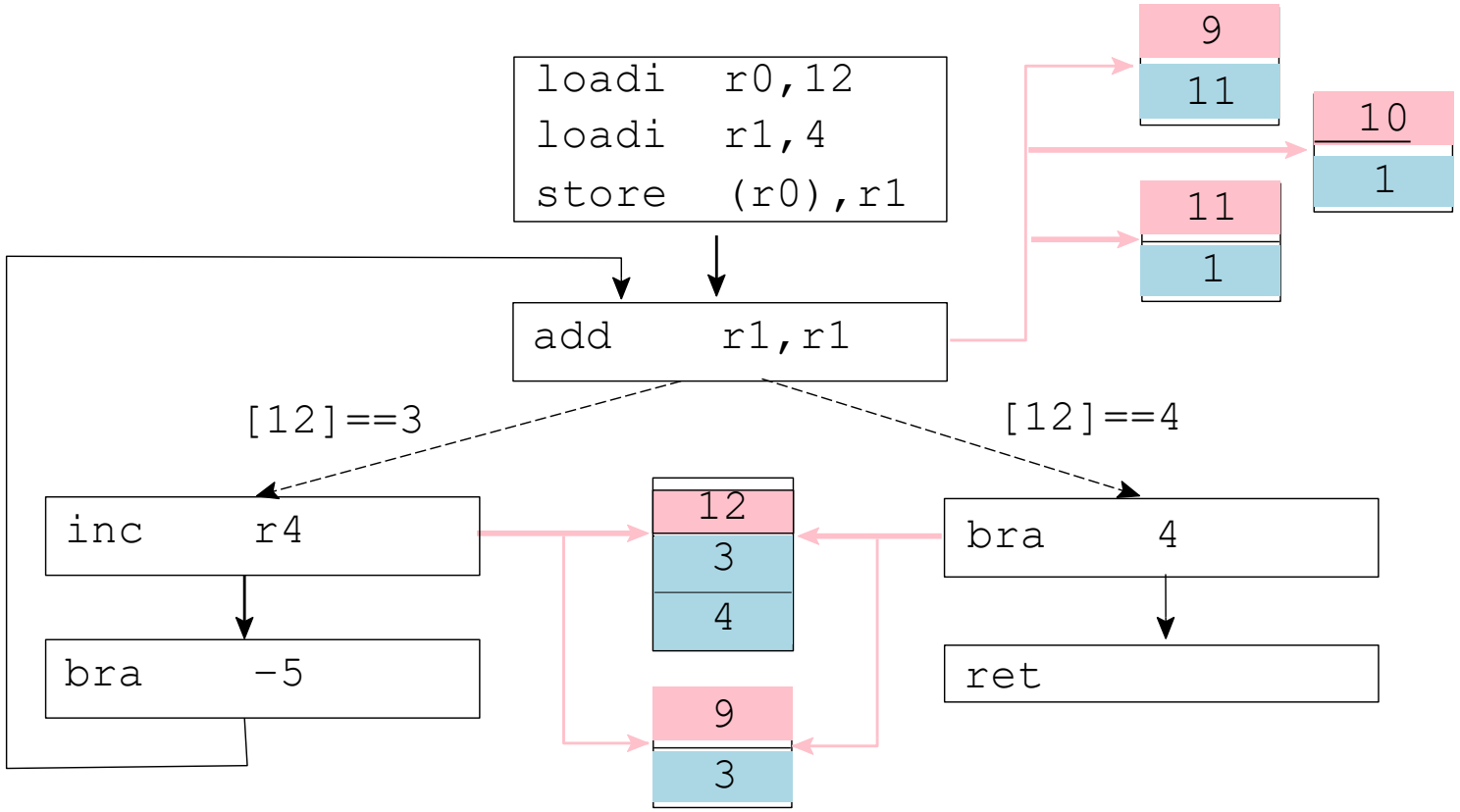
- The add instruction comprises three bytes h11, 1, 1i at addresses 9-11.
- Code byte addresses are in pink and the code bytes themselves in blue.
- At location 12, two values can be stored, 3 and 4.
- The outgoing edges from add's basic block are conditional on what is stored at 12, either 3 or 4.



New CFG Model

Ca' Foscari University of Venice

- Nice representation of a self-modifying function!
- But, hard to build in practice.
- Computer viruses are often self-modifying.

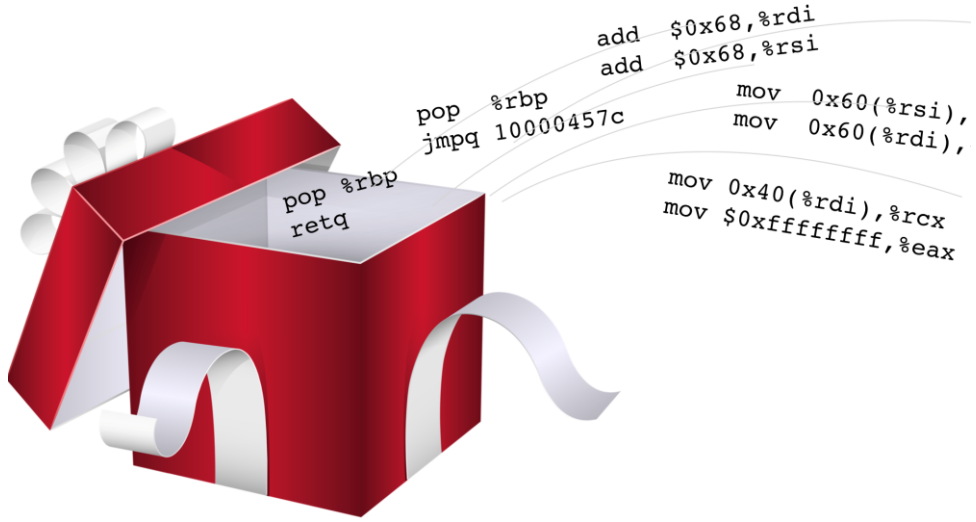


JIT Dynamic Obfuscation

JIT

Ca' Foscari University of Venice

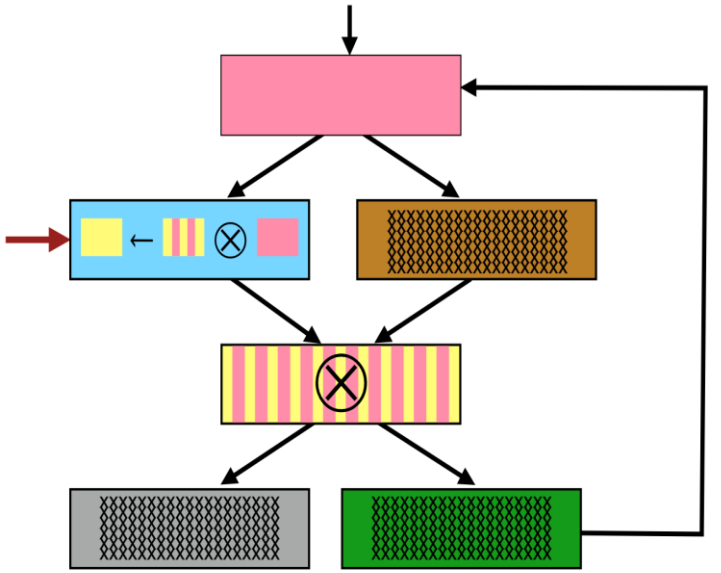
- This transformation translates a function F into a new function F' consisting of a sequence of intermediate code instructions such that, when F' is executed, F will be dynamically compiled to machine code.
  - Example of runtime code generation,
  - or just-in-time compilation,
  - or dynamic unpacking.
- Used by downloaders to install software
- ..but also packers in malware!



JIT Dynamic Obfuscation

Ca' Foscari University of Venice

- This transformation is similar to the JIT transformation, except the jitted code is continuously modified and updated at runtime.
  - Use self-modifying code
  - Keep the code in constant move
  - The code should never exist in cleartext



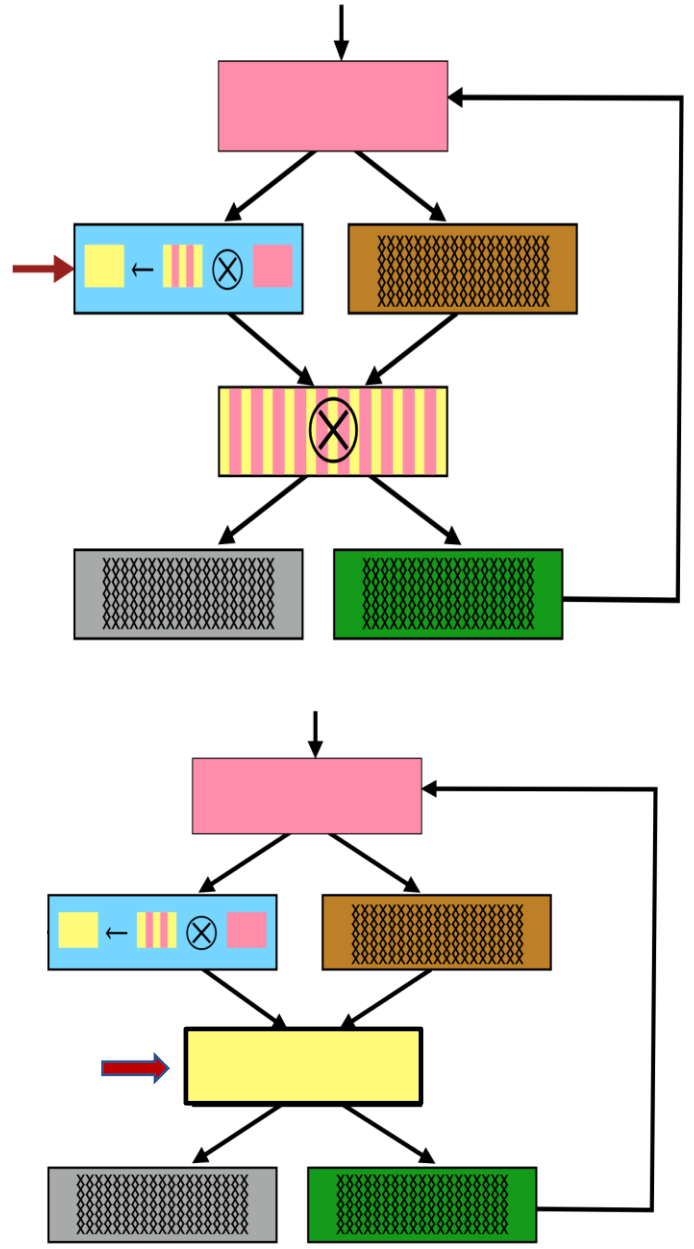
JIT Dynamic Schemes in Tigress

Ca' Foscari University of Venice

- Generalization of a set of previously published schemes:
  1. Aucsmith, Tamper Resistant Software: An Implementation, IH'96
  2. Cappaert, Preneel, et al. Towards Tamper Resistant Code Encryption Practive and Experience, ISPEC'08
  3. Madou, et al., Software protection through dynamic code mutation, WISA'05
  4. Kanzaki, Exploiting Self-Modification Mechanism for Program Protection
- Each of these schemes can be seen as a combination of two properties:
  1. the level of granularity at which the program is encoded and decoded (function-level, basic block-level, instruction-level, or byte-level)
  2. the codec that determines how a function/block should be encoded and decoded.

JIT Dynamic (Aucsmith)

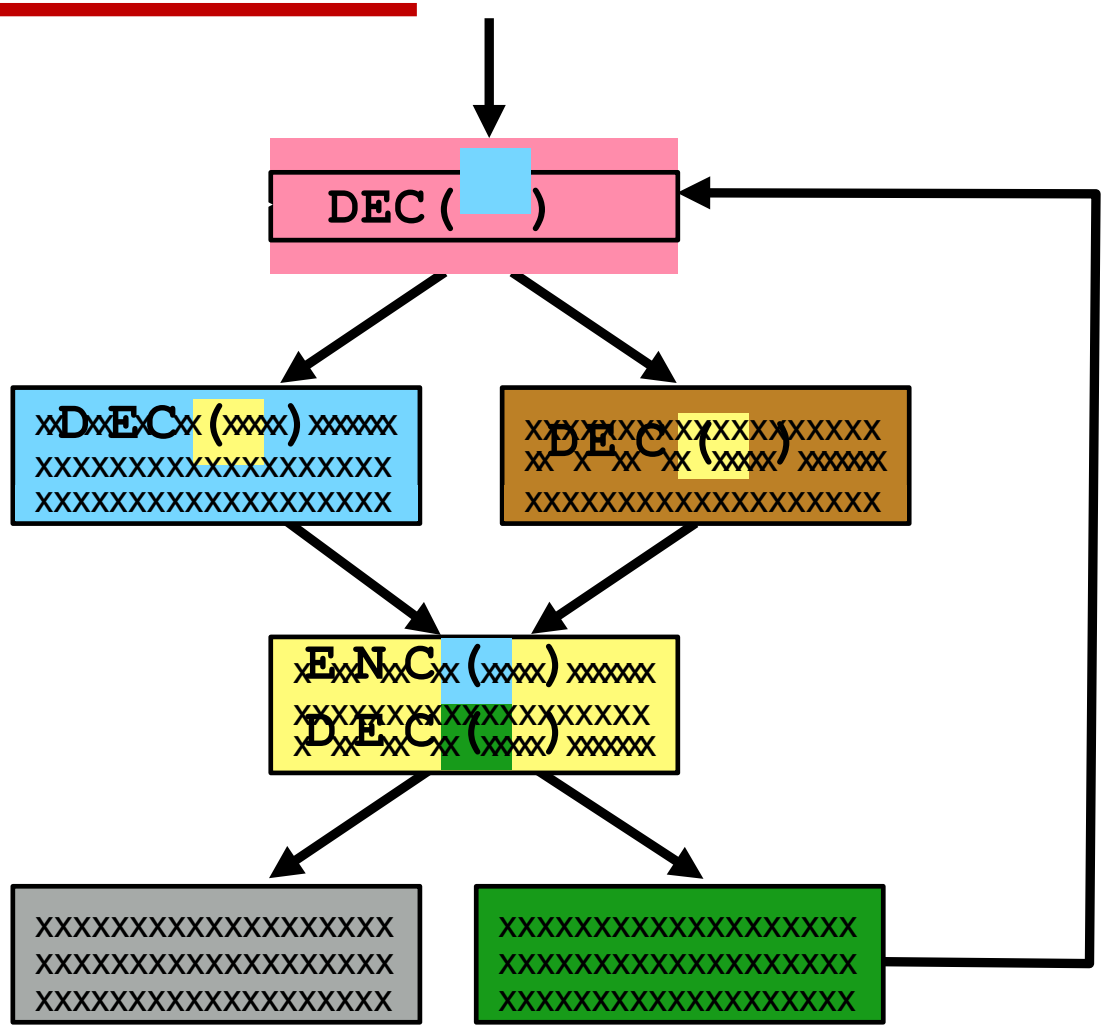
- Aucsmith's scheme works on basic blocks and encodes two blocks by xor:ing them with each other;
- Pink-and-yellow striped block is the XOR of the pink and the yellow blocks.
- Right before we want to execute the yellow block, the blue block XORs the striped block with the pink block…
- …and the yellow block appears in cleartext and we can jump to it.



JIT Dynamic (Cappaert)

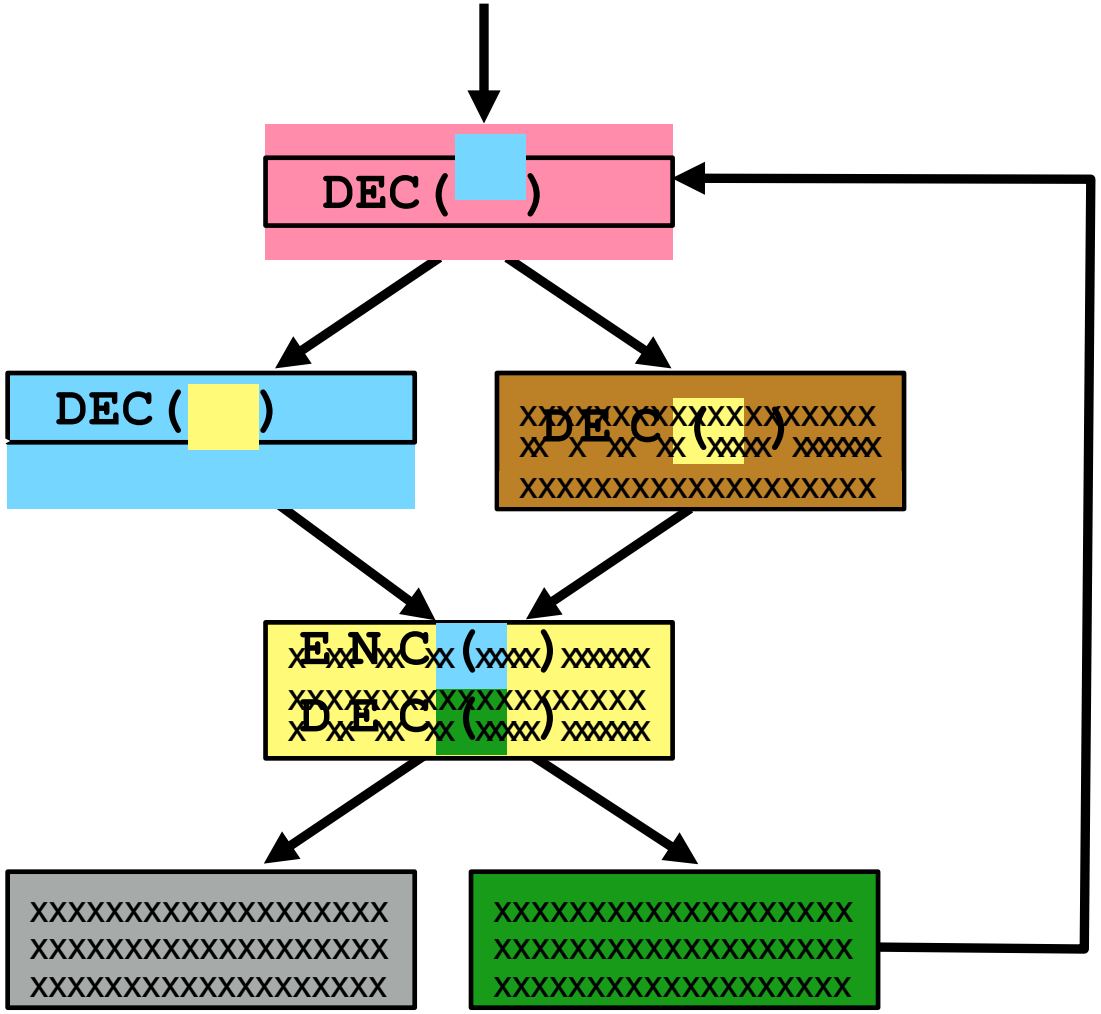
Ca' Foscari University of Venice

- Cappaert's scheme was originally designed to work at the granularity of functions and encodes by encrypting using a standard encryption algorithm (AES)
- To avoid having to embed keys directly in the executable (where they are easy to find) this scheme uses other blocks as the key.
  1. The first code block decrypts the following code block
  2. The decrypted block contains the code to decrypt the next block and to encrypt the previous block
- Ideally only one code block in clear an instant of time



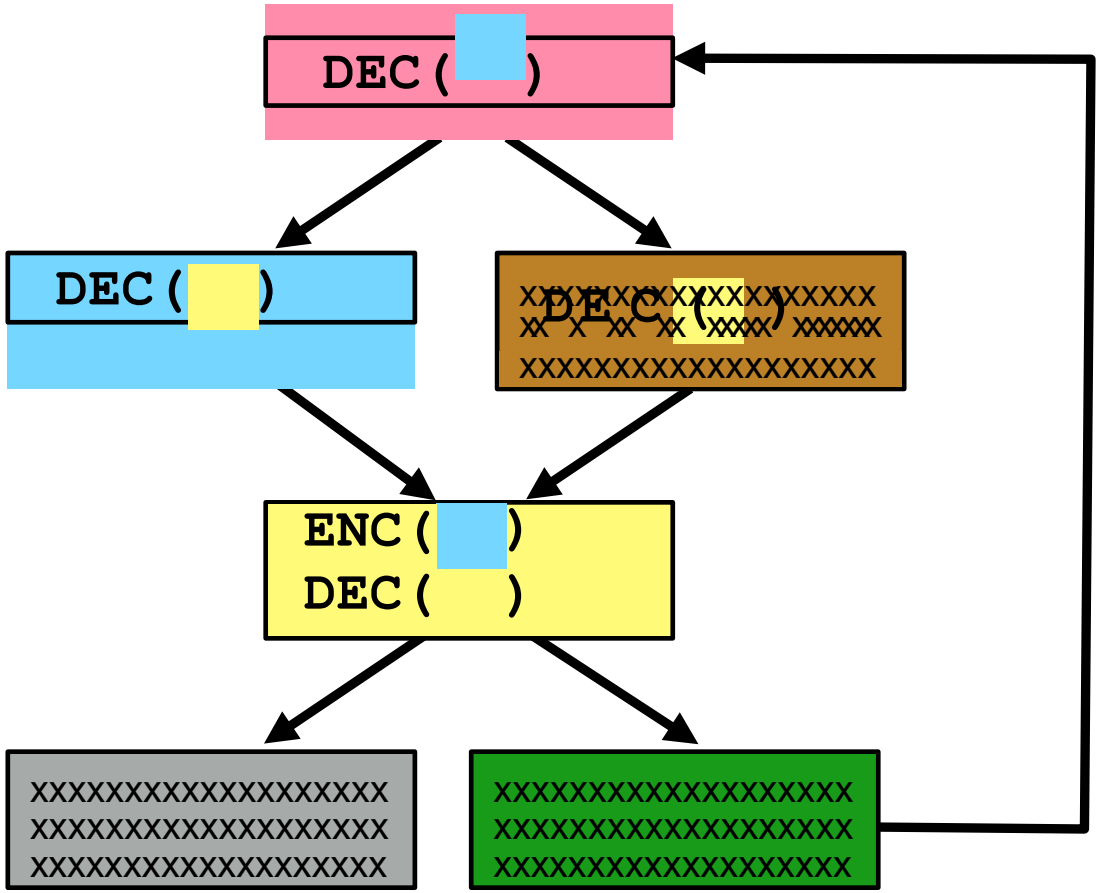
JIT Dynamic (Cappaert)

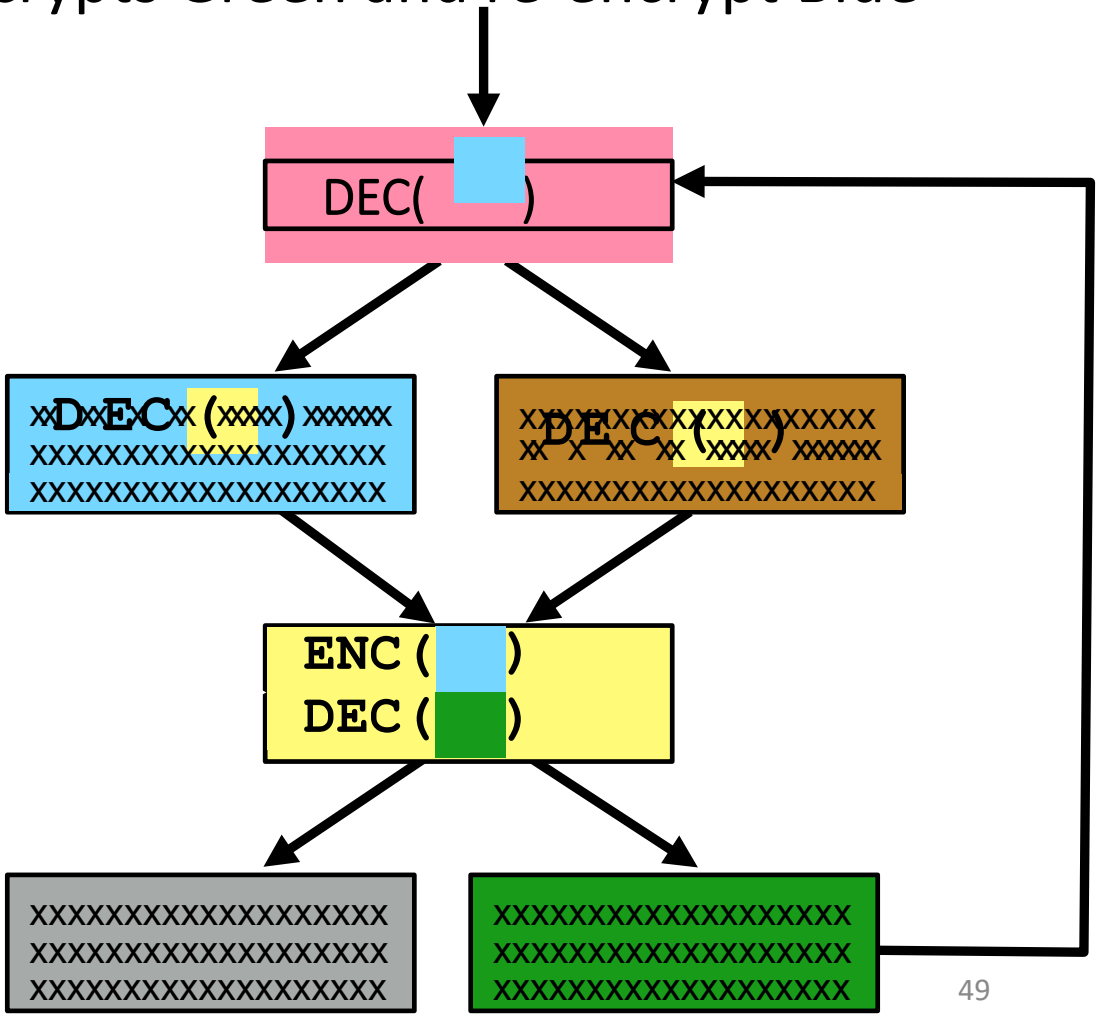
- For example: the pink block decrypts the blue block which contains the decryption routine of the yellow block



JIT Dynamic (Cappaert)

- Blue block decrypts Yellow block, which decrypts Green and re-encrypt Blue

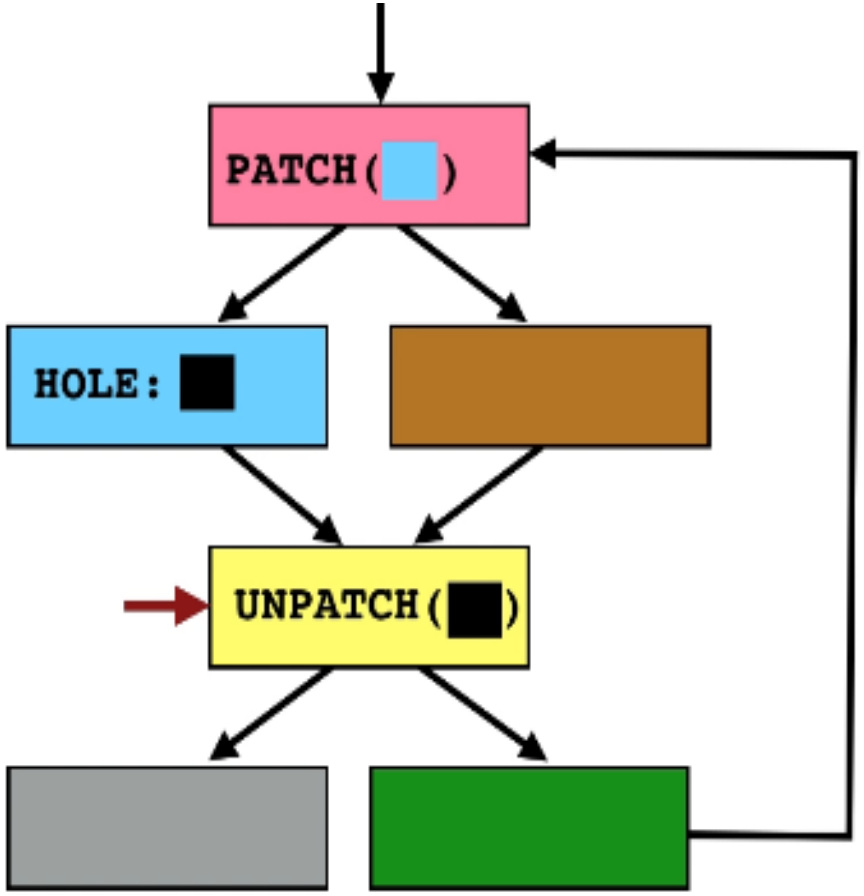




JIT Dynamic (Kanzaki)

Ca' Foscari University of Venice

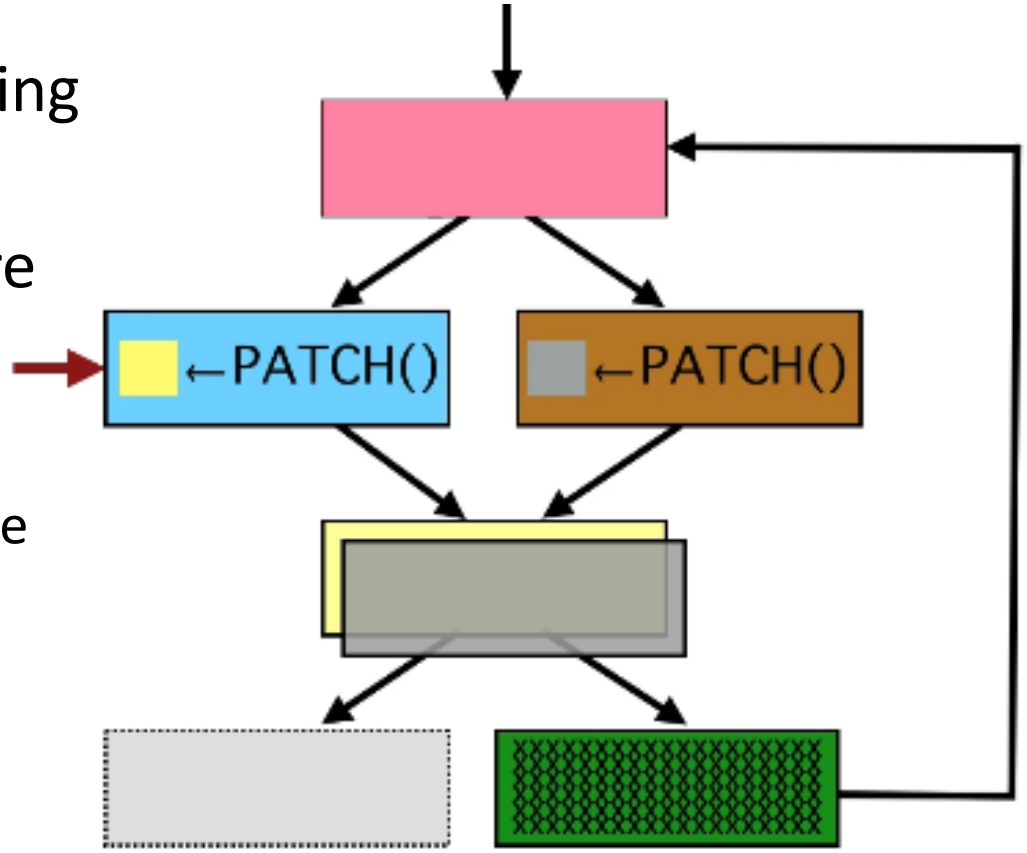
- Kanzaki's scheme works at the granularity of the raw bytes of the program and encodes/decodes by patching holes left in the code.
- Example:
  1. The pink block patches the hole in the blue block
  2. After the blue block has run, the yellow block unpatch the block by restoring the hole in the blue block



JIT Dynamic (Madou)

Ca' Foscari University of Venice

- Madou's scheme was originally designed to work at the function level and encodes by letting blocks occupy the same space at runtime.
- A special PATCH function restores blocks before they are needed.
- Example:
  1. The yellow and grey blocks both exist in the same memory location.
  2. The blue block decodes the yellow block by patching it; then jumps to it,
  3. Then the brown block needs to execute the gray block, so it patches it, and then jumps to it.



Dynamic Codecs in Tigress

- Tigress works at the granularity of basic blocks and it can be instantiated with arbitrary codecs (encoding/decoding schemes)
  1. Before a function starts executing, (a subset of) its basic blocks are encoded using one of the available codecs
  2. Before a block is executed it is decoded
  3. After a block has finished executing, it is eventually re-encoded.
- Note that multiple encoding schemes can be used within one function
  - Some basic blocks may be encoded using Aucsmith's scheme, some with Cappaert's scheme, etc.

Dynamic Codecs: Tigress Implementation

Ca' Foscari University of Venice

1. Convert F to jitted code, identically to the Jit transformation;
2. Construct a CFG from the jitted code;
3. Transform the CFG, making certain instructions into their own blocks which will not be encoded:
   1. make every branch instruction a separate basic block;
   2. make every instruction that references in-code data blocks a separate basic block;
   3. add a level of indirection for indirect branches.
4. Add any necessary cipher functions to the executable;
5. Decide, for every basic block, which codec it should use (if any);
6. For each basic block that should be encoded, find allowable points within the CFG where decoders/encoders could safely be inserted;
7. Insert the decoders/encoders at the chosen points;
8. Generate a file __dump__.c with the jitting code for the function, plus instructions to dump the bytes for each of the basic blocks we want to encode:

Issues with Dynamic JIT schemes

- Recursion is an issue for dynamic obfuscation.
- In general, whenever you call a function it has to be in its fully encoded form.
- This presents a problem if a function calls itself (directly or through a recursive call chain) from somewhere in the middle where some blocks are encoded and some are in cleartext.
- For simple recursive procedures Tigress detects this and
  - blocks that contain (direct or indirect) recursive calls that will not be encoded;
  - encoders/decoders are placed such that recursive calls won't leave any blocks in cleartext.

Exercise

Ca' Foscari University of Venice

```
*************************************************************
* 6) Dynamic Obfuscation
*************************************************************
tigress --Environment=x86_64:Linux:Gcc:4.6 \
   --Transform=JitDynamic \
      --Functions=fib \
         --JitDynamicCodecs=xtea \
         --JitDynamicDumpCFG=false \
         --JitDynamicBlockFraction=%50 \
         --out=fib6.c fib.c
```

Implicit Flow

Implicit Flow

Ca' Foscari University of Venice

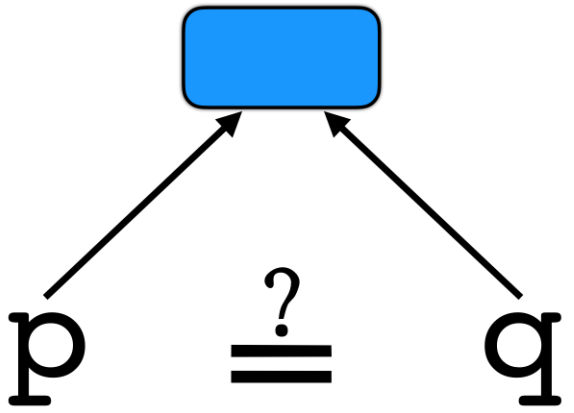
- This strategy converts explicit control flow instructions to implicit ones.
- It can hinder reverse engineers from addressing the correct control flows.
- For example, we can replace the control instructions of assembly codes (e.g., jmp and jne) with a combination of mov and other instructions which implement the same control semantics.

Anti-Alias Analysis

Ca' Foscari University of Venice

- The goal of this transformation is to disrupt static analysis tools that make use of inter-procedural alias analysis.
- The current Tigress implementation simply replace all direct function calls with indirect ones.
- A call to x = foo(n) turns into:

```
void *arr[]  = {..., & foo, ... };
int main () {
   int x = ((int (*)(int n )) arr[expr=42])(n);
}
```



Anti-Alias Analysis

Ca' Foscari University of Venice

- arr is a global array containing function addresses
- $expr^{=42}$ is an opaque expression computing the index of foo's address in arr.
- To make analysis a bit harder, we can also insert bogus elements in arr, and insert updates to these bogus elements.

```
void *arr[]  = {..., & foo, ... };
int main () {
   int x = ((int (*)(int n )) arr[expr=42])(n);
}
```

```
int main () {
  int x;
arr[expr=6] = arr[expr=42];
arr[expr=7] = &x;
}
```

- On the right, both arr[6] and arr[7] are bogus.

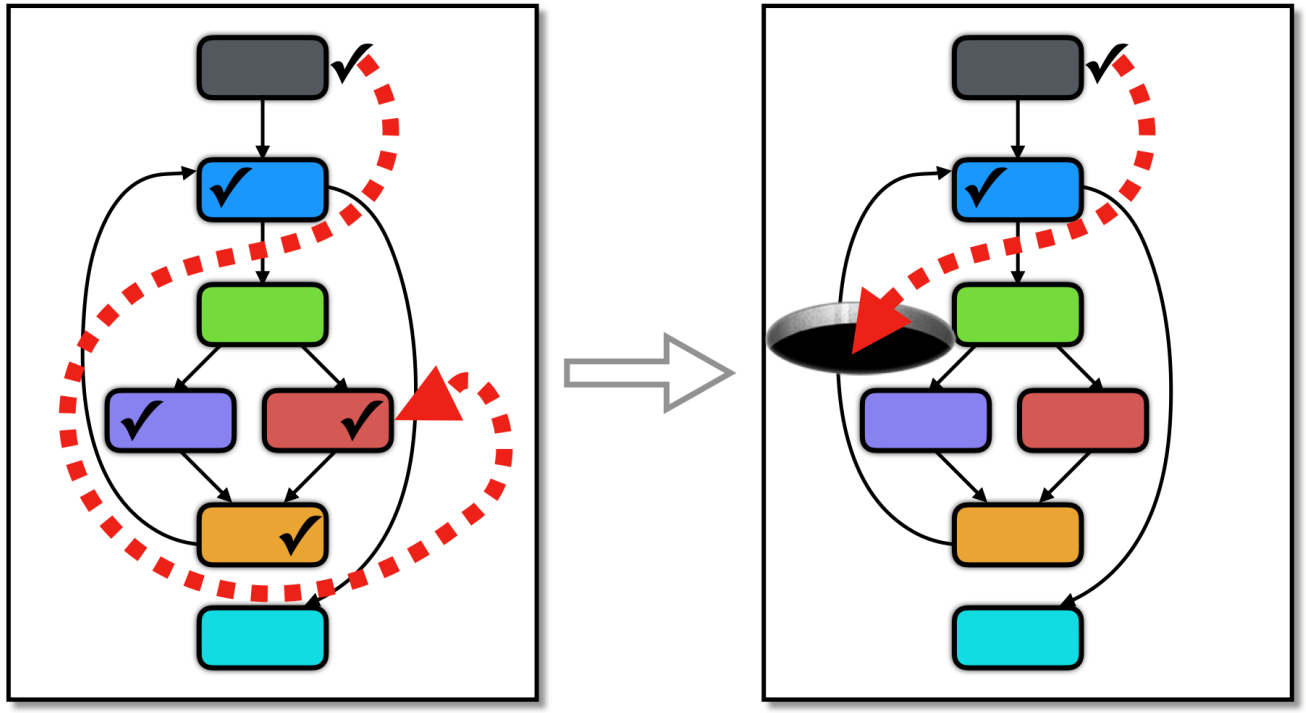
Taint Analysis

Ca' Foscari University of Venice

- Measuring the level of influence external data have on the application
- When debugging a program, one can see data moved and copied around all the time
- It can be seen as Information Flow Analysis
- Flow (x --> y) is a series of operations that use the value of an object X to derive the value of an object Y
- If the source of the value of object X is not trustworthy, then X is tainted.
- Taint is propagated to other objects using the data from X.
- Dynamic Taint analysis can be used to infer code reachability

Anti Taint Analysis

Ca' Foscari University of Venice



Anti Taint Analysis

Ca' Foscari University of Venice

- The goal of this transformation is to disrupt analysis tools that make use of dynamic taint analysis.
- Tigress use two basic ways to copy a variable using control-flow statements, rather than data-flow ones:

1. counting up to the value of the variable
2. copying it bit by bit, tested in an if-statement.

This can be done in a simple loop, an unrolled loop, or by throwing exceptions caught in a signal handler.

Anti Taint Analysis

Ca' Foscari University of Venice

```
argc_origPtr13 = (unsigned char *)(& argc);
argc_copyPtr15 = (unsigned char *)(&
argc_copy14);
size_iter16 = 0;
while (size_iter16 < 4) {
TempVar = 0; signal(31, handler);
BitVar = 0;
while (BitVar < 8) {
if ((*argc_origPtr13 >> BitVar) & 1) {
raise(31);
}
BitVar ++; }
signal(31, (void (*)(void *sig ))1);
*argc_copyPtr15 = TempVar;
argc_origPtr13 ++;
argc_copyPtr15 ++;
size_iter16 ++; }
```

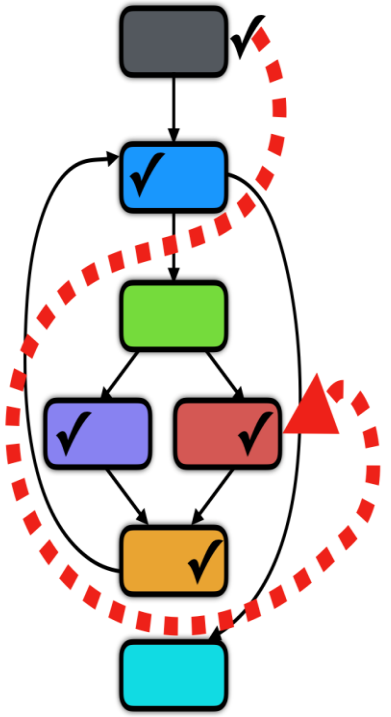
- Variable copied by throwing exceptions caught in a signal handler.

```
unsigned char TempVar;
int BitVar;
void handler(int sig ) {
TempVar |= 1 << BitVar;
}
```

Implicit Flow

Ca' Foscari University of Venice

- Several transformations, in particular AntiTaintAnalysis, use implicit flow as a basic building block.
- Before you can use these you need to call the --*Transform=InitImplicitFlow* and list which of the implicit flow variants you are going to use later.
- An Example of Tigress script for Implicit Flow is in the next slide



Implicit Flow: Example Tigress Command

Ca' Foscari University of Venice

```
tigress --Seed=0 --Verbosity=1 --Environment=x86_64:Linux:Gcc:4.6 \
   --Transform=InitEntropy --Functions=main \
   --Transform=InitImplicitFlow --Functions=main \
      --InitImplicitFlowKinds=trivial_thread_1,trivial_counter,\
                              mem_cache_time,mem_cache_thread_1,\
                              file_cache_time,file_cache_thread_1,\
                              jit_time \
      --InitImplicitFlowHandlerCount=1 \
      --InitImplicitFlowJitCount=1 \
      --InitImplicitFlowJitFunctionBody="(for (if (bb 50) (bb 50)))" \
      --InitImplicitFlowTrace=false \
      --InitImplicitFlowTrain=false \
      --InitImplicitFlowTime=false \
      --InitImplicitFlowTrainingTimesClock=500 \
      --InitImplicitFlowTrainingTimesThread=500 \
      --InitImplicitFlowTrainingMinGap=90 \
      --InitImplicitFlowTrainingConfidenceLevel=0.99 \
      --InitImplicitFlowTrainingTargetErrorRate=0.00001 \
      --InitImplicitFlowTrainingKind=statistics \
   --Transform=AntiTaintAnalysis --Functions=main \
      --AntiTaintAnalysisKinds=vars \
      --LocalVariables=main:b \
      --AntiTaintAnalysisImplicitFlow="(repeat mem_cache_time 3)" \
   input.c --out=output.c
```

Parallelized (Multi-Threaded) Code

Ca' Foscari University of Venice

- Threads in the code makes it more difficult to reverse engineer as it hides the actual flow of control
  1. Create dummy processes that perform no useful task and run them in parallel with the actual instructions.
  2. Split a sequential section of the application into multiple sections executing in parallel.
  3. A code block that does not have any data dependencies can easily be parallelized.
  4. A code block that has data dependencies can be parallelized by using synchronization functions.

Parallelized (Multi-Threaded) Code

- Parallelizing can have a considerable time overhead associated for making sure the parallel threads are properly synchronized so that the program functionality stays unchanged.
- However, the resilience and potency of this method is very high
- The introduction of parallel paths makes analysis difficult both for a deobfuscator program as well as a reverse engineer.