Engineering Code Obfuscation

ISSISP 2017 - Obfuscation I

Christian Collberg

Department of Computer Science
University of Arizona

http://collberg.cs.arizona.edu

collberg@gmail.com

Supported by NSF grants 1525820 and 1318955 and
by the private foundation that shall not be named

Man-At-The-End Applications

Tools and Counter Tools

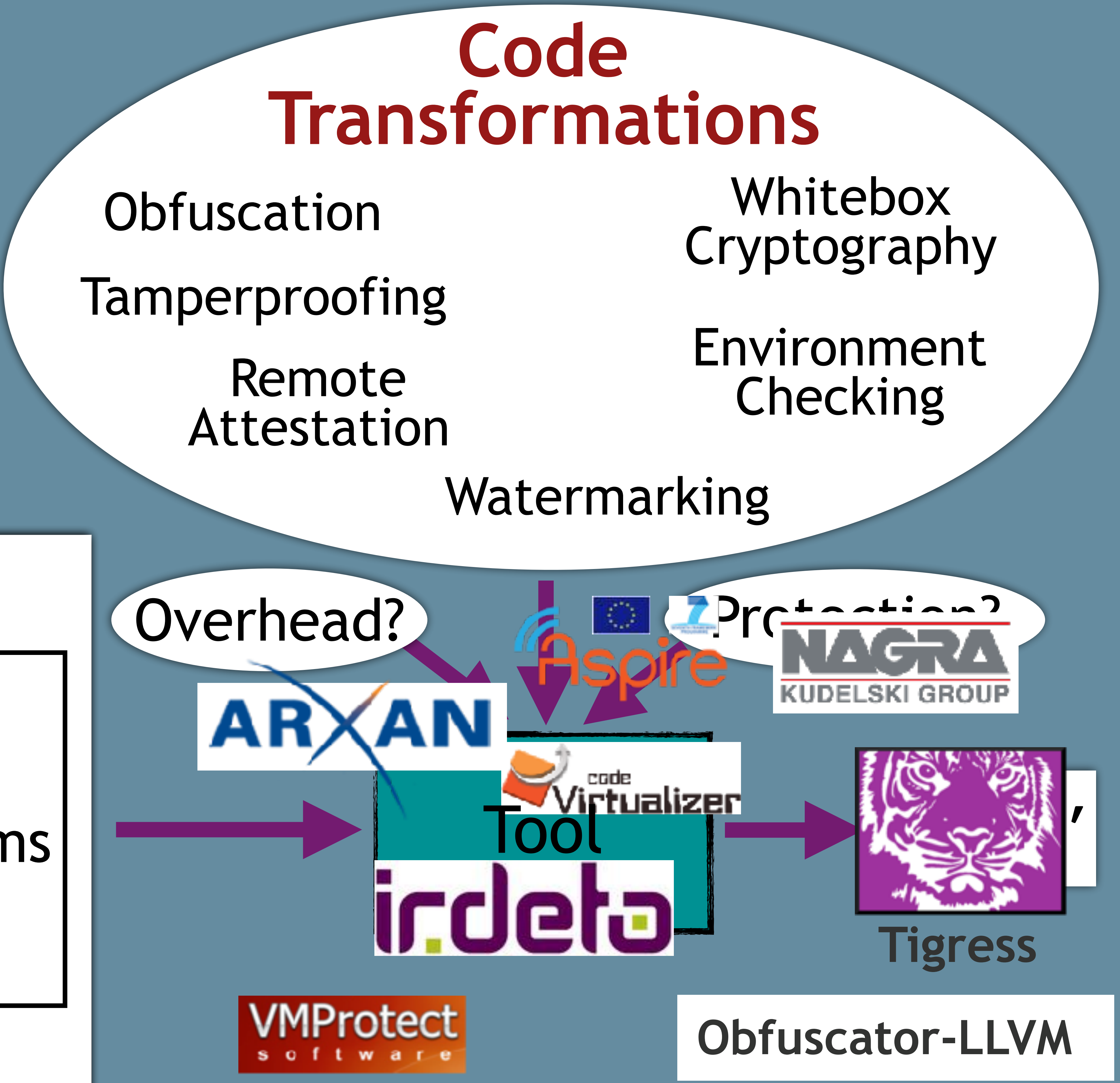
Obfuscation vs. Deobfuscation

Deploying Obfuscation

Evaluation

Discussion

Tools
vs.
Counter Tools



Code Transformations

Obfuscation

Tamperproofing

Remote Attestation

Whitebox Cryptography

Environment Checking

Watermarking

Overhead?

Protection?

Tool

Tigress

Obfuscator-LLVM

```
Prog() {
```

Assets

- Source
- Algorithms
- Keys
- Media

```
}
```

Code Analyses

Static analysis

Dynamic analysis

Concolic analysis

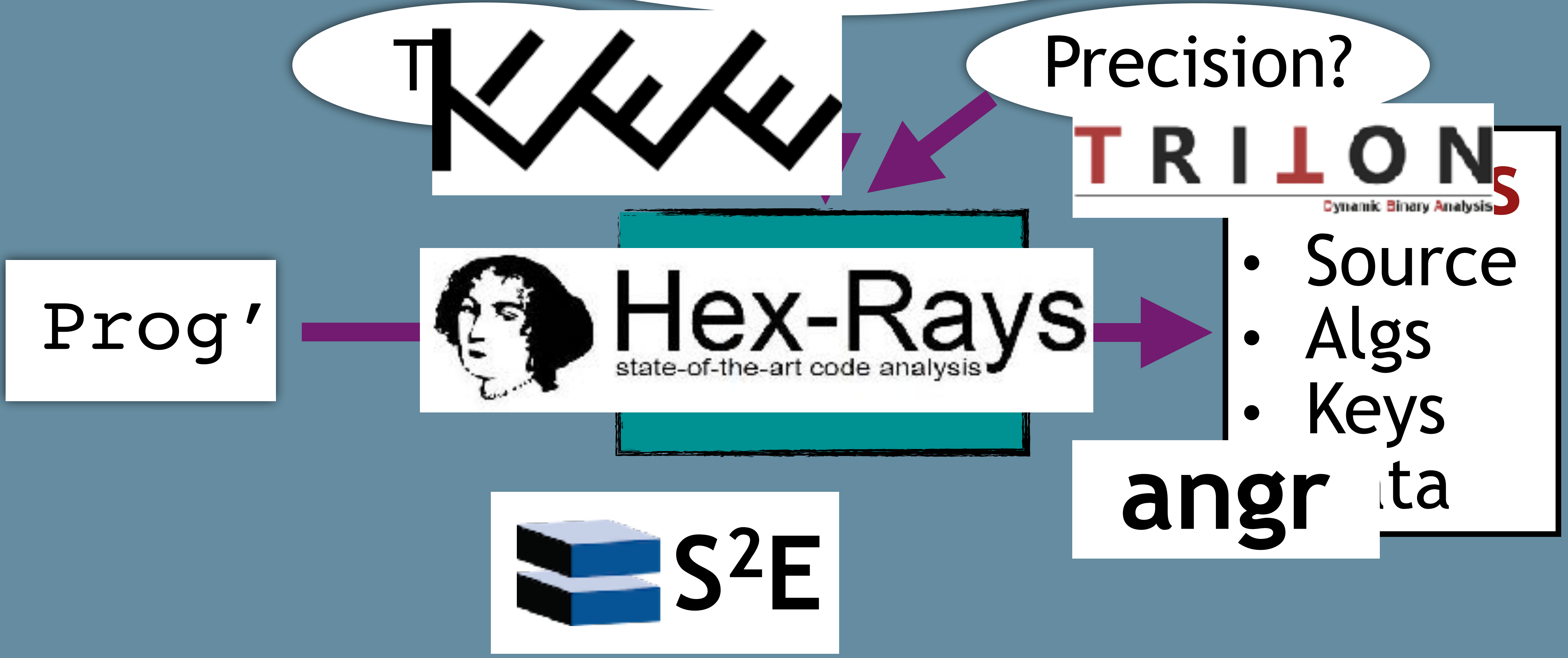
Disassembly

Decompilation

Slicing

Debugging

Emulation



T

KLEE

Precision?

TRITON

Dynamic Binary Analysis

s

Prog’

Hex-Rays

state-of-the-art code analysis

- Source
- Algs
- Keys
- ta

angr

S2E

What Matters?

Performance

Time-to-Crack

S^{2}E

angr

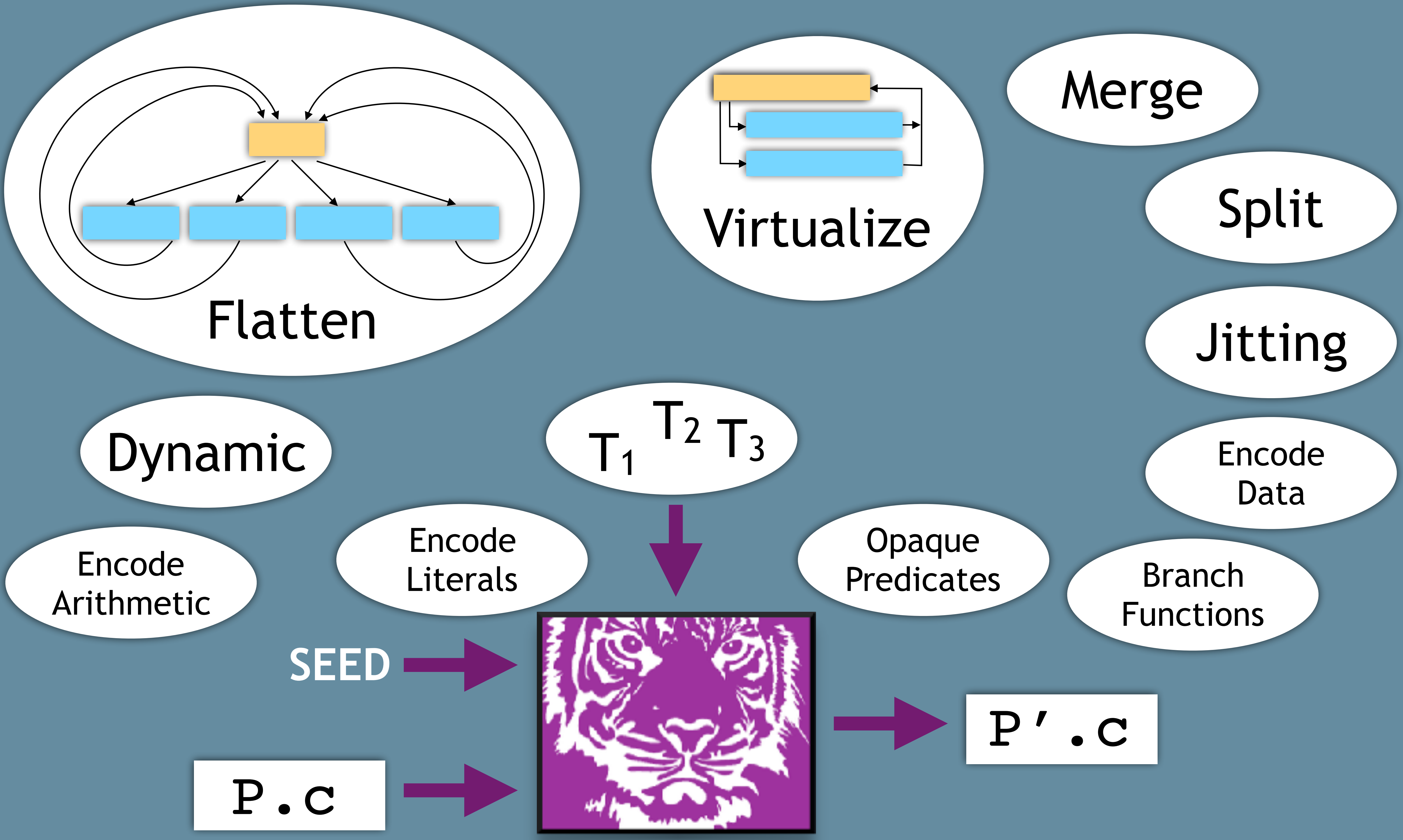
TRITON Dynamic Binary Analysis

KLEE

Hex-Rays state-of-the-art code analysis

Stealth

The Tigress Obfuscator



Flatten

Virtualize

Merge

Split

Jitting

Dynamic

T_1 T_2 T_3

Encode Data

Encode Arithmetic

Encode Literals

Opaque Predicates

Branch Functions

SEED

P.c

P'.c

tigress.cs.arizona.edu

```
#include<stdio.h>
#include<stdlib.h>
int fib(int n) {
   int a = 1; int b = 1; int i;
   for (i = 3; i <= n; i++) {
      int c = a + b; a = b; b = c;
   };
   return b;
}
int main(int argc, char** argv) {
   if (argc != 2) {
      printf("Give one argument!\n"); abort(); };
      long n = strtol(argv[1],NULL,10);
      int f = fib(n);
      printf("fib(%li)=%i\n",n,f);
}
```

- Install Tigress:

  http://tigress.cs.arizona.edu/#download

- Get the test program:

  http://tigress.cs.arizona.edu/fib.c

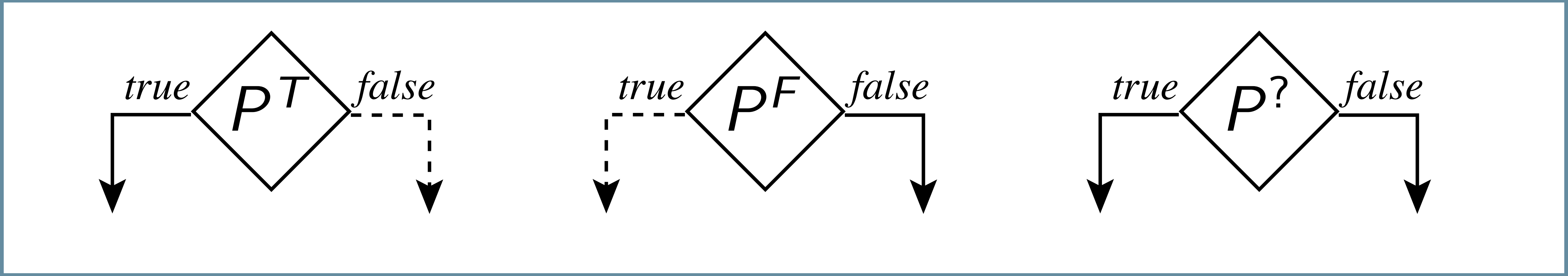
Opaque Expressions

Opaque Expressions

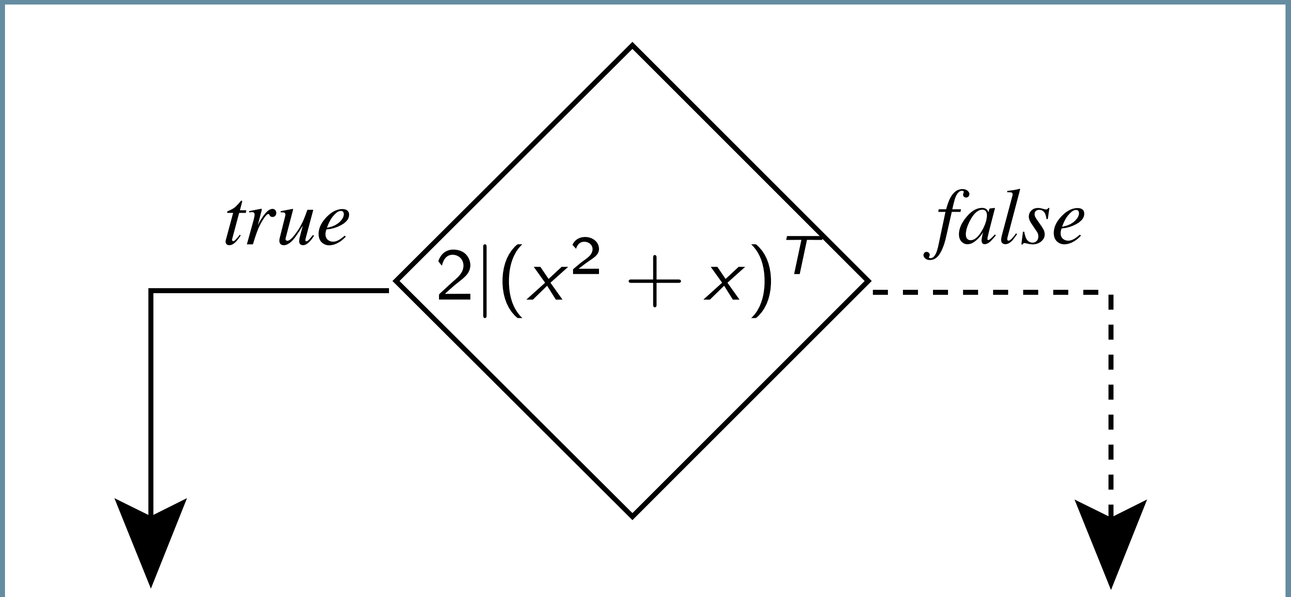
An expression whose value is known to you as the defender (at obfuscation time) but which is difficult for an attacker to figure out

Notation

- $P^{=T}$ for an opaquely true predicate
- $P^{=F}$ for an opaquely false predicate
- $P^{=?}$ for an opaquely indeterminate predicate
- $E^{=v}$ for an opaque expression of value v



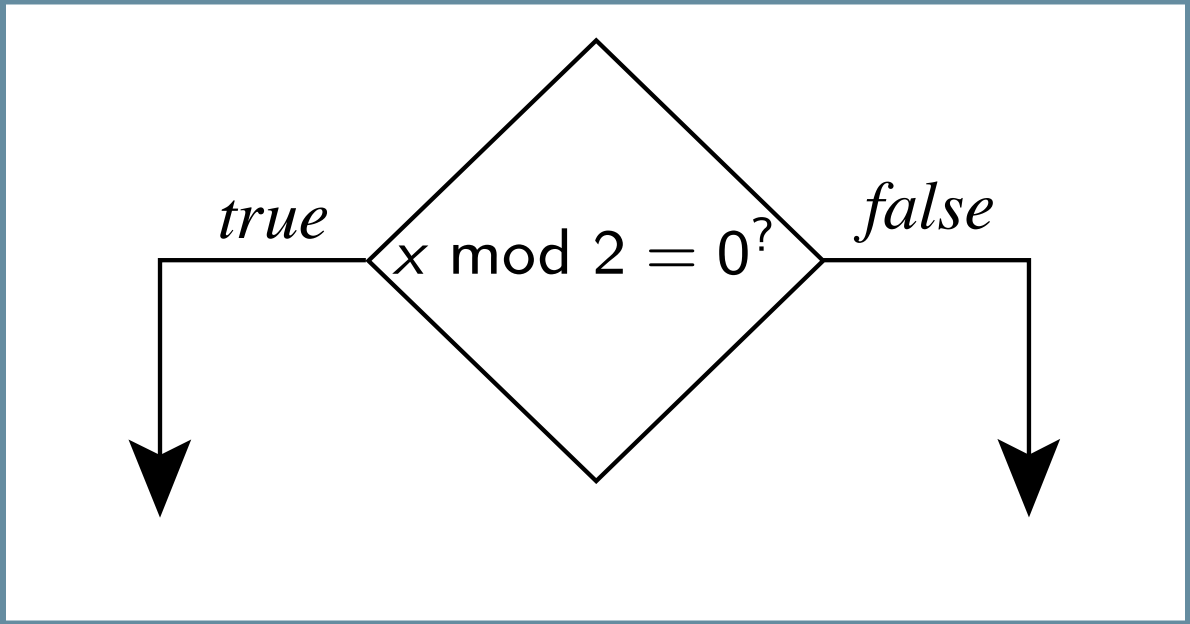
Examples



true

$2|(x^2 + x)^T$

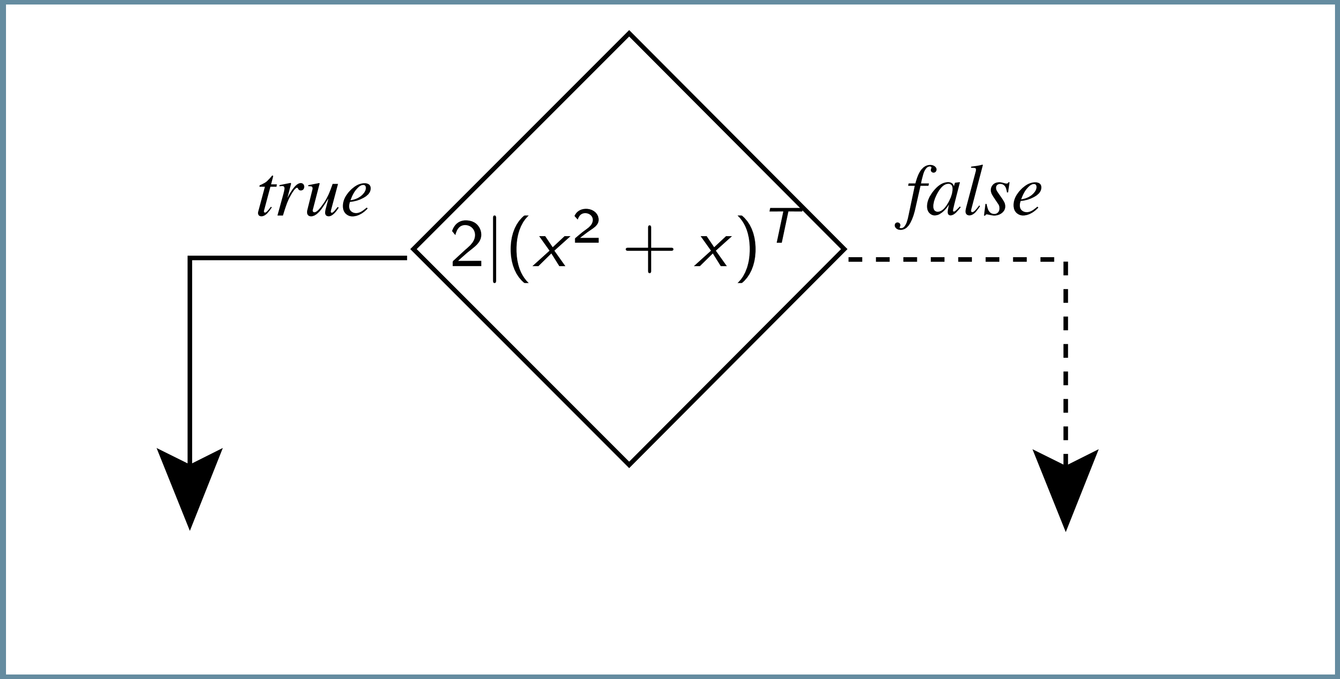
false



true

$x \bmod 2 = 0^?$

false



true

$2|(x^2 + x)^T$

false

Inserting Bogus Control Flow

Examples

```
if (x[k] == 1)
   R = (s*y) % n
else
   R = s;
s = R*R % n;
L = R;
```

```
if (x[k] == E=1)
   R = (s*y) % n
else
   R = s;
s = R*R % n;
L = R;
```

Examples

```
if (x[k] == 1)
    R = (s*y) % n
else
    R = s;
s = R*R % n;
L = R;
```

```
if (x[k] == 1)
    R = (s*y) % n
else
    R = s;
if (expr^{=T})
    s = R*R % n;
else
    s = R*R * n;
L = R;
```

Examples

```
if (x[k] == 1)
   R = (s*y) % n
else
   R = s;
s = R*R % n;
L = R;
```

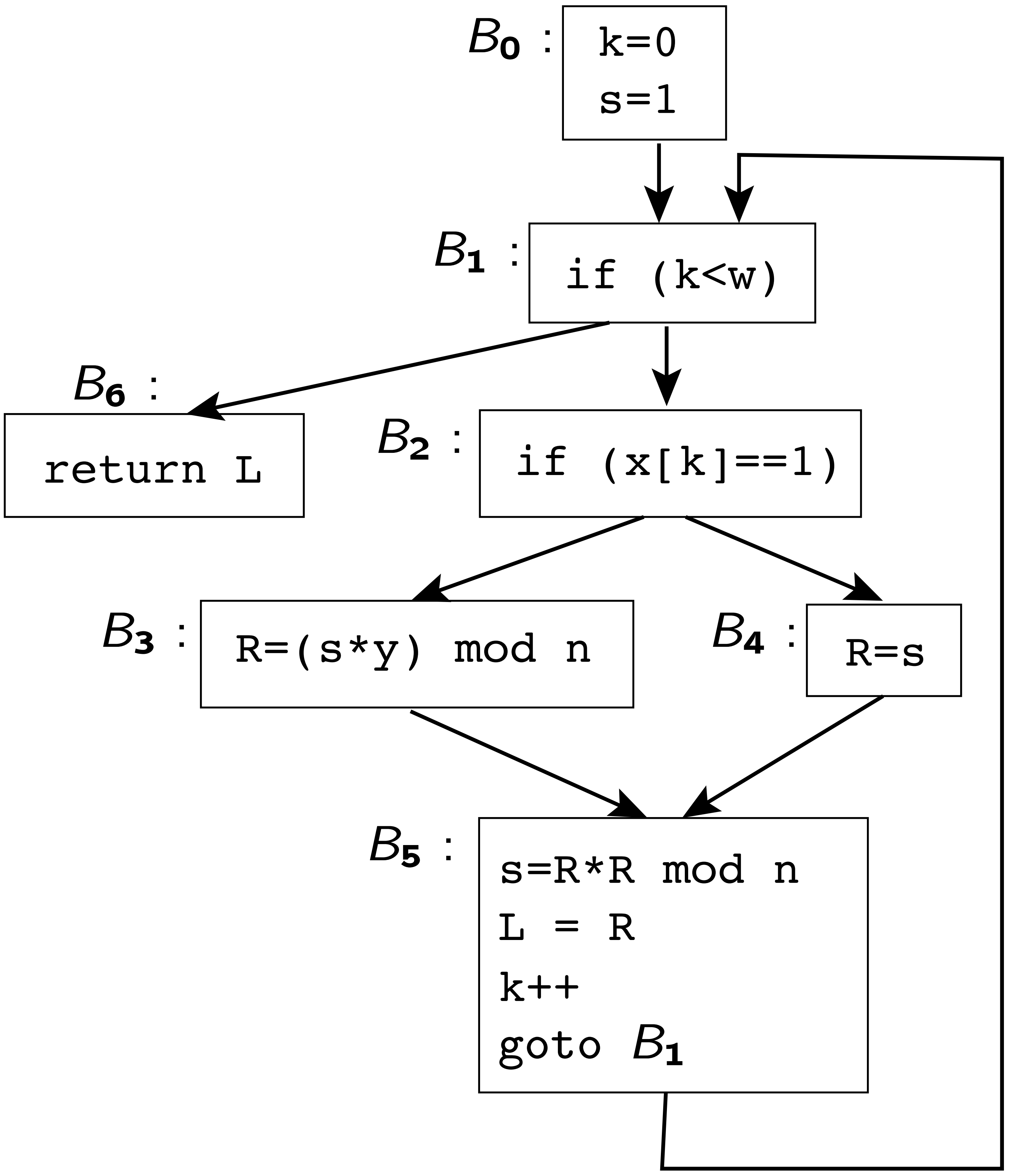
```
if (x[k] == 1)
   R = (s*y) % n
else
   R = s;
if (expr=?)
   s = R*R % n;
else
   s = (R%n)*(R%n)%n;
L = R;
```

Exercise!

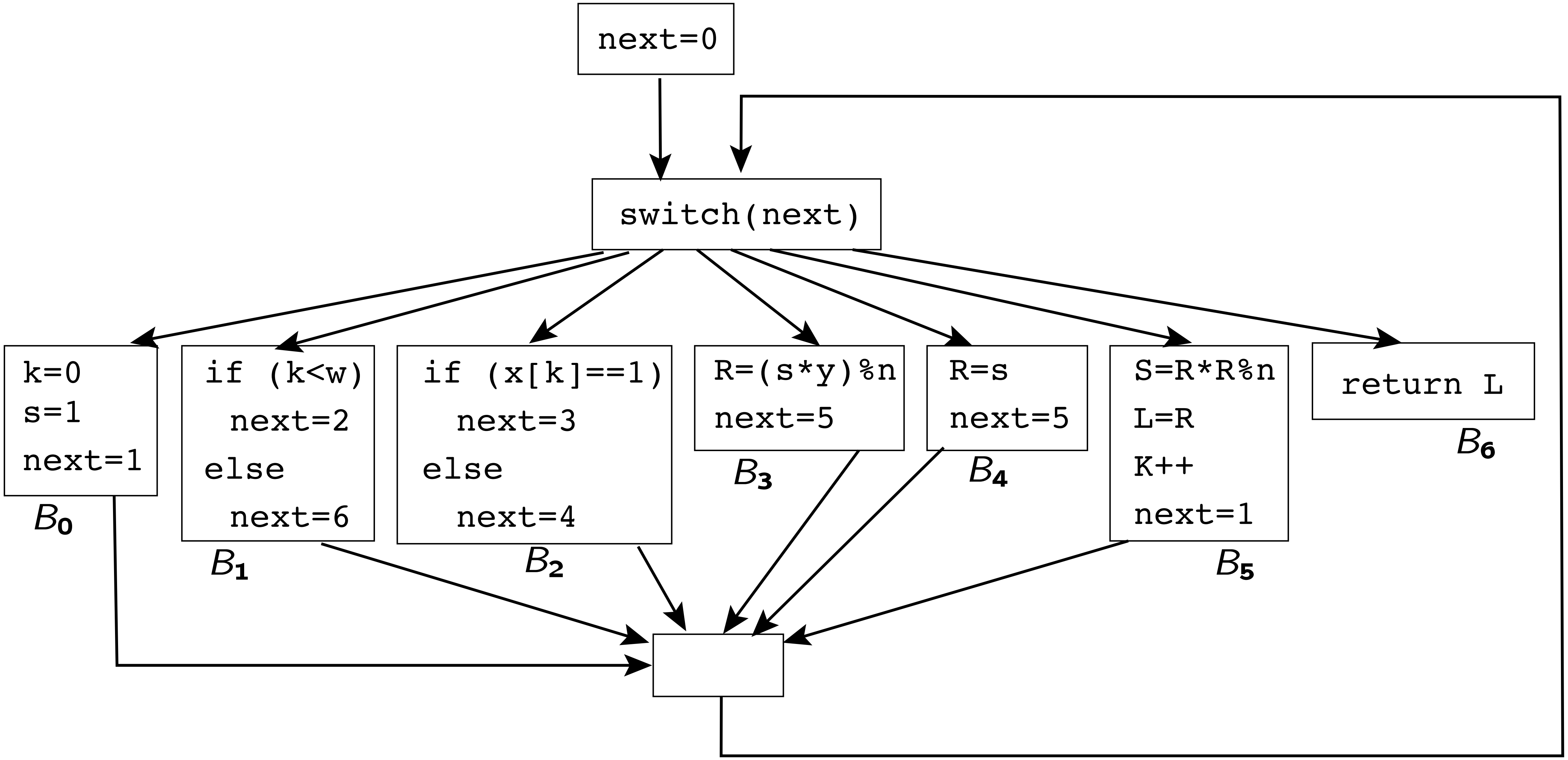
```
tigress --Seed=0 \
   --Transform=InitEntropy \
   --Transform=InitOpaque \
      --Functions=main\
      --InitOpaqueCount=2\
      --InitOpaqueStructs=list,array \
   --Transform=AddOpaque\
      --Functions=fib\
      --AddOpaqueKinds=question \
      --AddOpaqueCount=10 \
    fib.c –out=fib_out.c
```

Control Flow Flattening

```
int modexp(int y,int x[],int w,int n){
   int R, L;
   int k=0; int s=0;
   while (k < w) {
      if (x[k] == 1)
         R = (s*y) % n
      else
         R = s;
      s = R*R % n;
      L = R;
      k++;
   }
   return L;
}
```



```
int modexp(int y, int x[], int w, int n) {
int R, L, k, s;
   int next=0;
   for(;;)
      switch(next) {
         case 0 :
             k=0; s=1; next=1; break;
         case 1 :
             if (k<w) next=2; else next=6; break;
         case 2 :
             if (x[k]==1) next=3; else next=4; break;
         case 3 :
             R=(s*y)%n; next=5; break;
         case 4 :
             R=s; next=5; break;
         case 5 :
             s=R*R%n; L=R; k++; next=1; break;
         case 6 : return L;
   }
}
```



```
next=0

switch(next)

k=0
s=1
next=1

if (k<w)
  next=2
else
  next=6

if (x[k]==1)
  next=3
else
  next=4

R=(s*y)%n
next=5

R=s
next=5

S=R*R%n
L=R
K++
next=1

return L
```

Exercise!

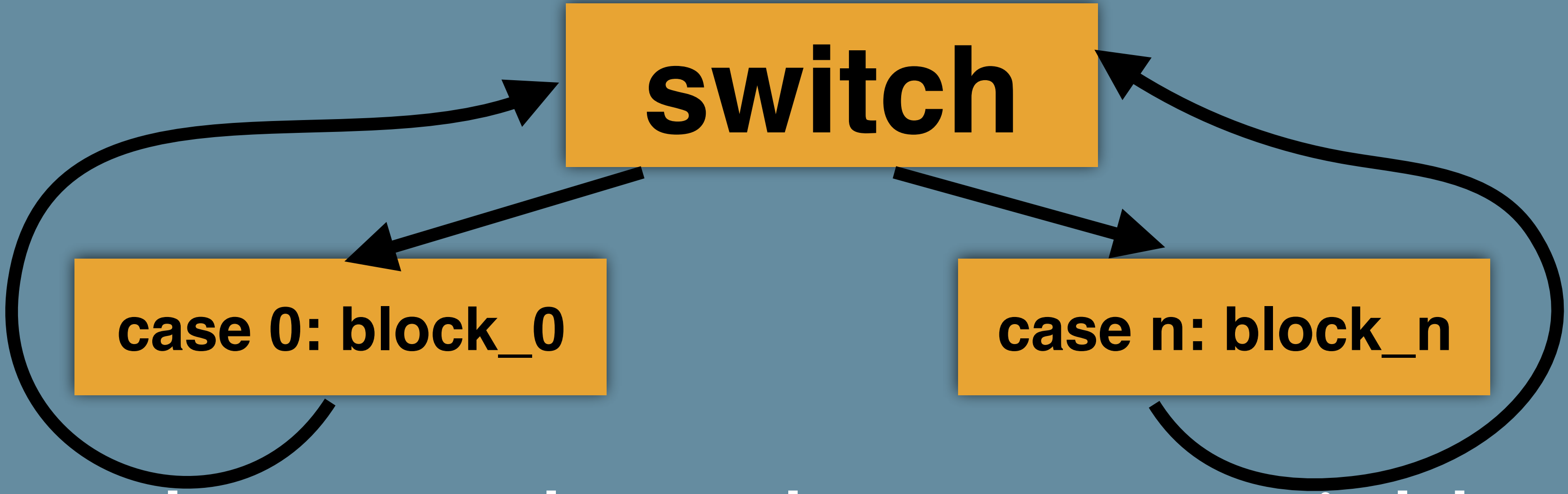
```
tigress \
      --Seed=42 \
      --Transform=InitOpaque \
        --Functions=main \
      --Transform=Flatten \
        --FlattenDispatch=switch \
        --FlattenOpaqueStructs=array \
        --FlattenObfuscateNext=false \
        --FlattenSplitBasicBlocks=false \
        --Functions=fib \
    fib.c --out=fib1.c
```

Exercise...

- Try different kinds of dispatch
  switch, goto, indirect
- Turn opaque predicates on and off.
- Split basic blocks or not.

Algorithm

1. Construct the CFG
2. Add a new variable **int next=0;**
3. Create a switch inside an infinite loop, where every basic block is a case:



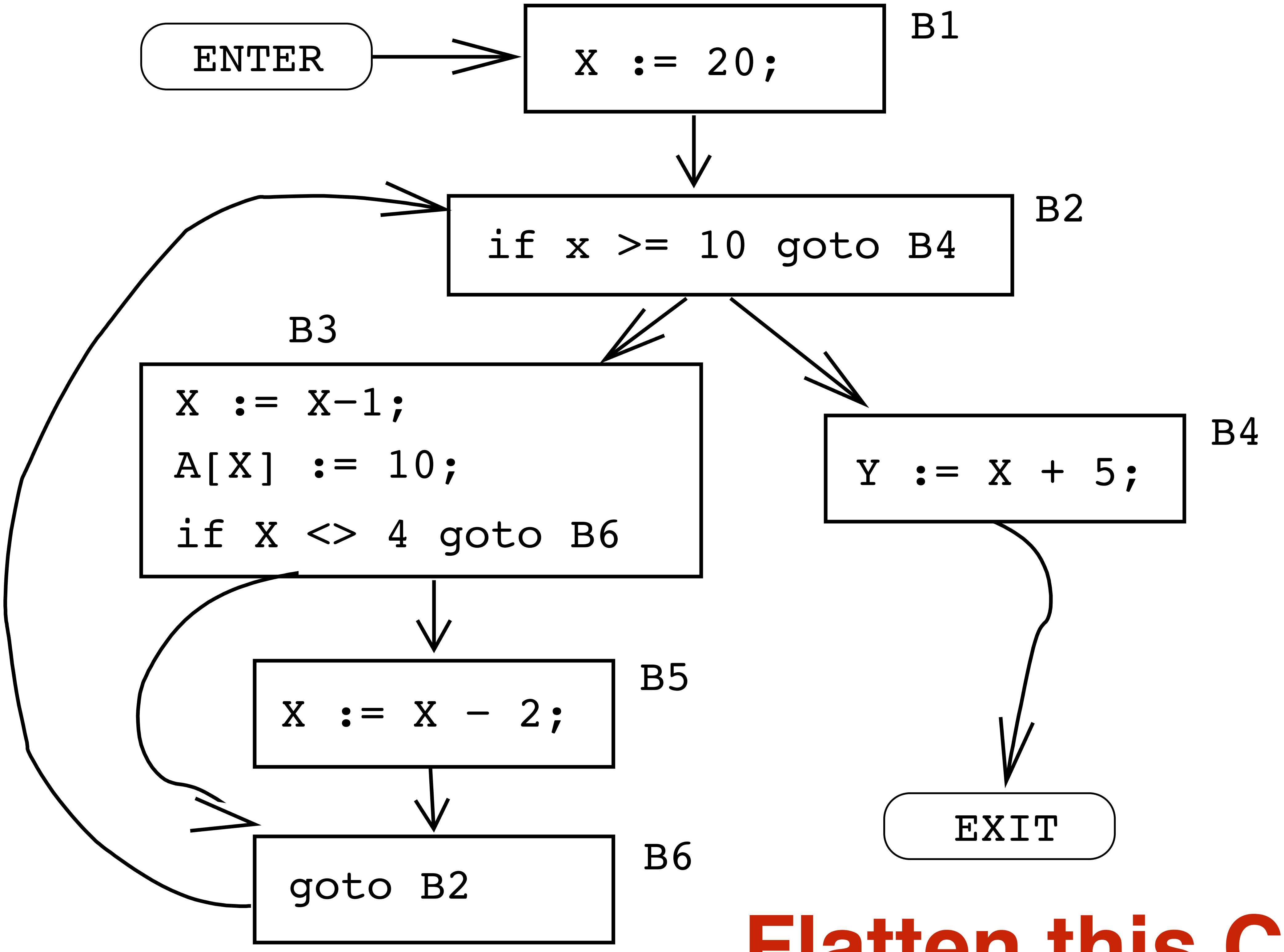
switch

case 0: block_0

case n: block_n

4. Add code to update the **next** variable:

```
case n: {
  if (expression)
    next = ...
  else
    next = ...
}
```



Flatten this CFG!

Work with your friends!

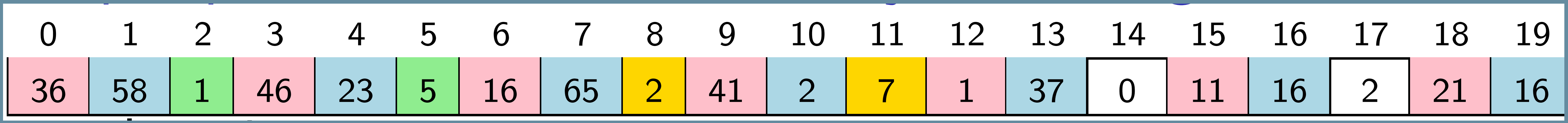
Attacks against Flattening

- Attack:
  - Work out what the next block of every block is.
  - Rebuild the original CFG!
- How does an attacker do this?
  - use-def data-flow analysis
  - constant-propagation data-flow analysis

```
int modexp(int y, int x[], int w, int n) {
   int R, L, k, s;
   int next=^{E=0};
   for(;;)
    switch(next) {
    case 0: k=0; s=1; next=^{E=1}; break;
    case 1: if (k<w) next=^{E=2};
            else next=^{E=6}; break;
    case 2: if (x[k]==1) next=^{E=3};
            else next=^{E=4}; break;
    case 3: R=(s*y)%n; next=^{E=5}; break;
    case 4: R=s; next=^{E=5}; break;
    case 5: s=R*R%n; L=R; k++;
            next=^{E=1}; break;
    case 6: return L;
} }
```

next=$^{E=1}$

Opaque Predicates



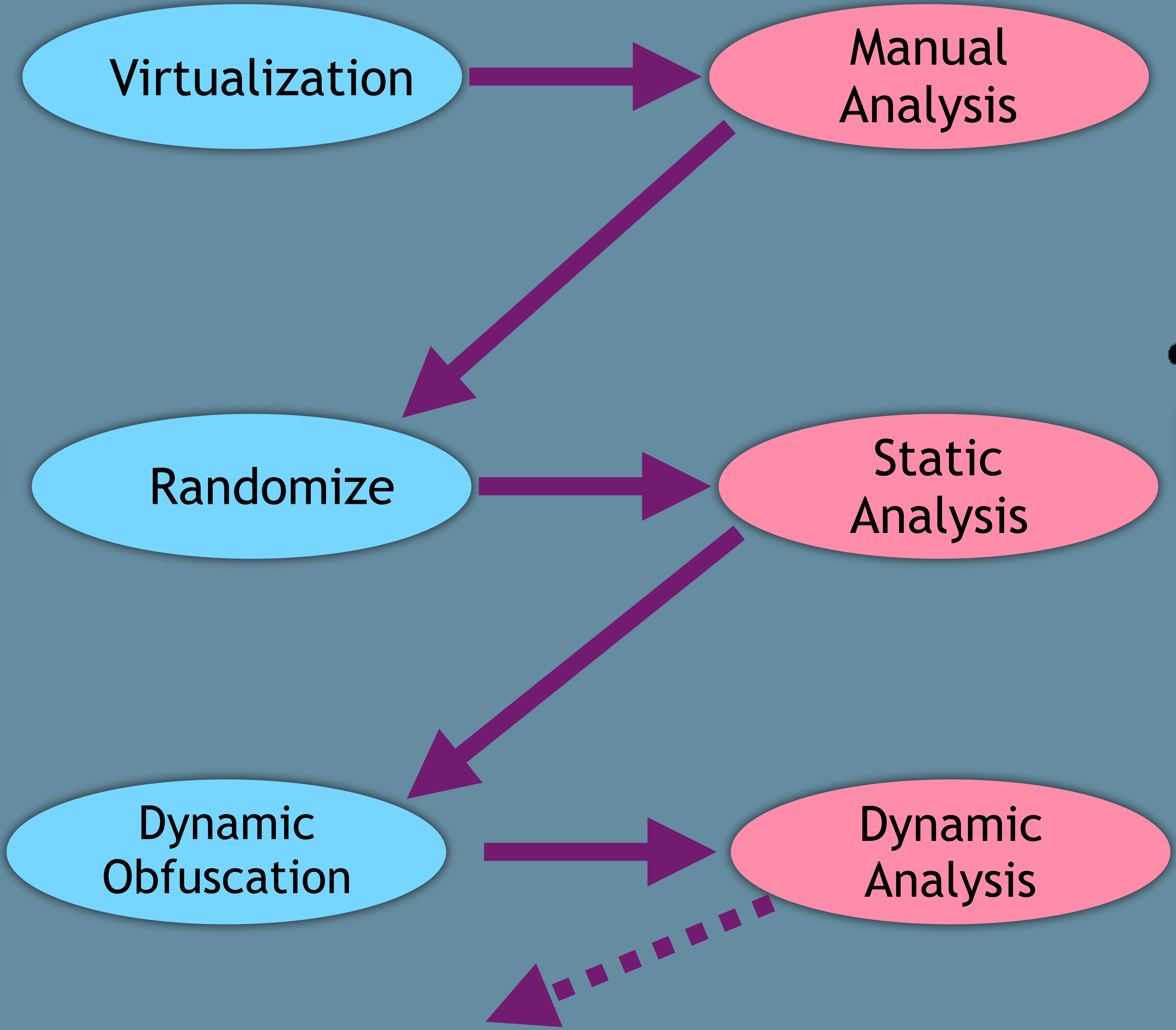
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
36	58	1	46	23	5	16	65	2	41	2	7	1	37	0	11	16	2	21	16

Invariants:

- every third cell (in pink), starting will cell 0, is ≡ 1 mod 5;
- cells 2 and 5 (green) hold the values 1 and 5, respectively;
- every third cell (in blue), starting will cell 1, is ≡ 2 mod 7;
- cells 8 and 11 (yellow) hold the values 2 and 7, respectively.

```
int modexp(int y, int x[], int w, int n) {
  int R, L, k, s; int next=0;
  int g[] = {10,9,2,5,3};
  for(;;)
    switch(next) {
    case 0 : k=0; s=1; next=g[0]%g[1]=1; break;
    case 1 : if (k<w) next=g[g[2]]=2;
                else next=g[0]-2*g[2]=6; break;
    case 2 : if (x[k]==1) next=g[3]-g[2]=3;
                else next=2*g[2]=4; break;
    case 3 : R=(s*y)%n; next=g[4]+g[2]=5; break;
    case 4 : R=s; next=g[0]-g[3]=5; break;
    case 5 : s=R*R%n; L=R; k++;
               next=g[g[4]]%g[2]=1; break;
    case 6 : return L;
} }
```

Virtualization



Virtualization

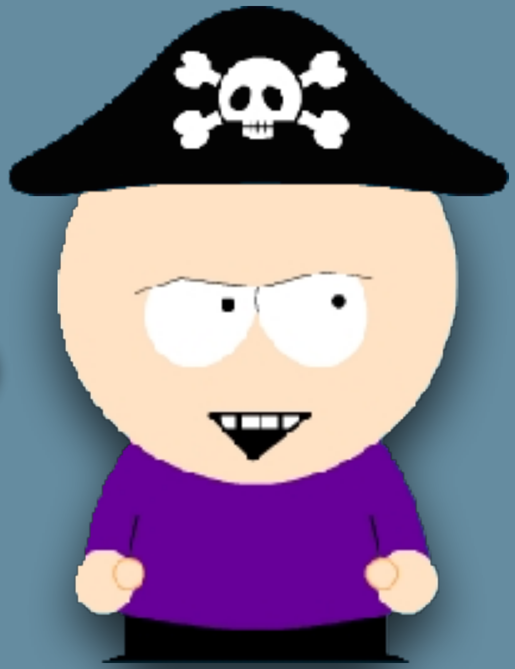
Manual Analysis

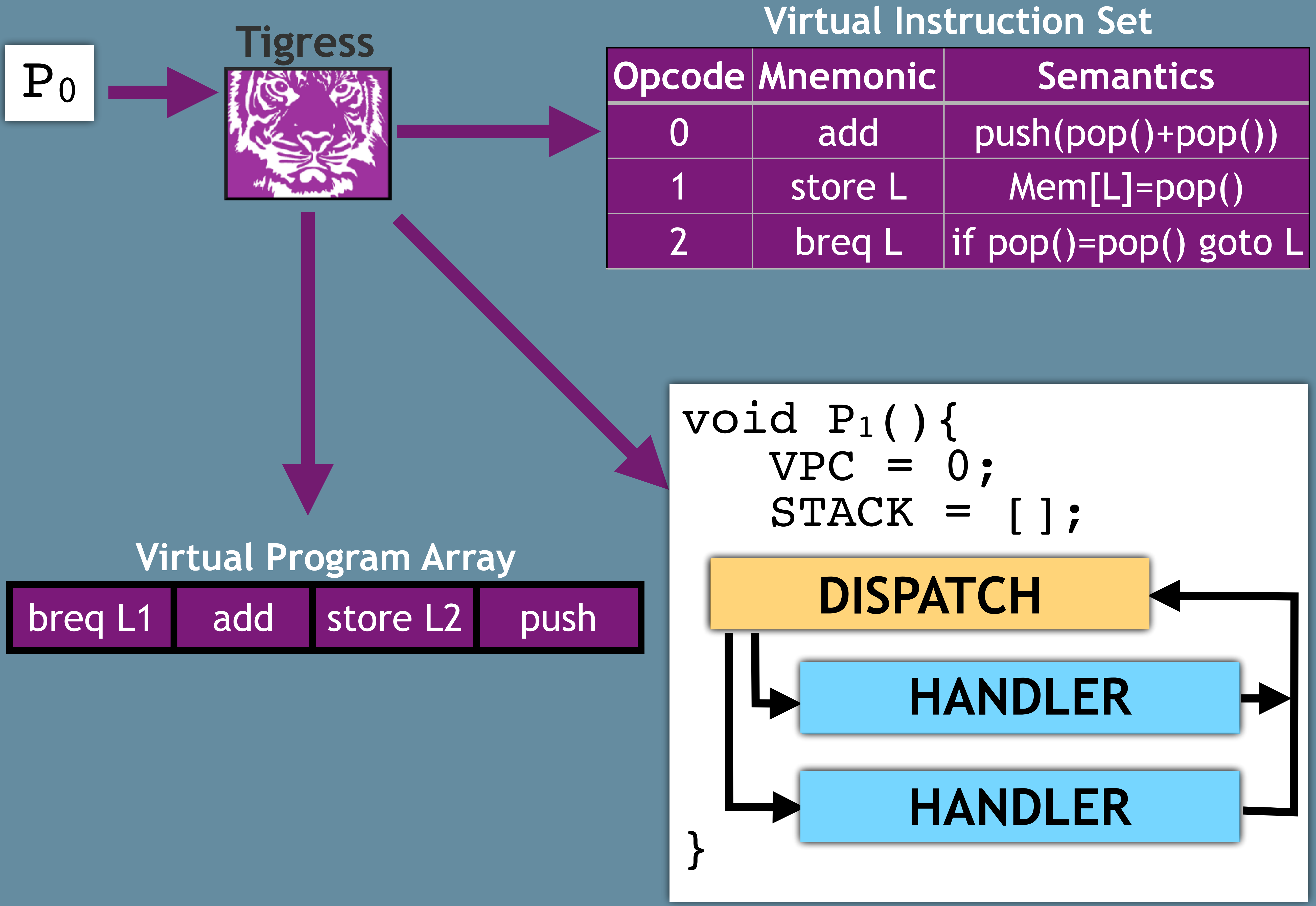
Randomize

Static Analysis

Dynamic Obfuscation

Dynamic Analysis





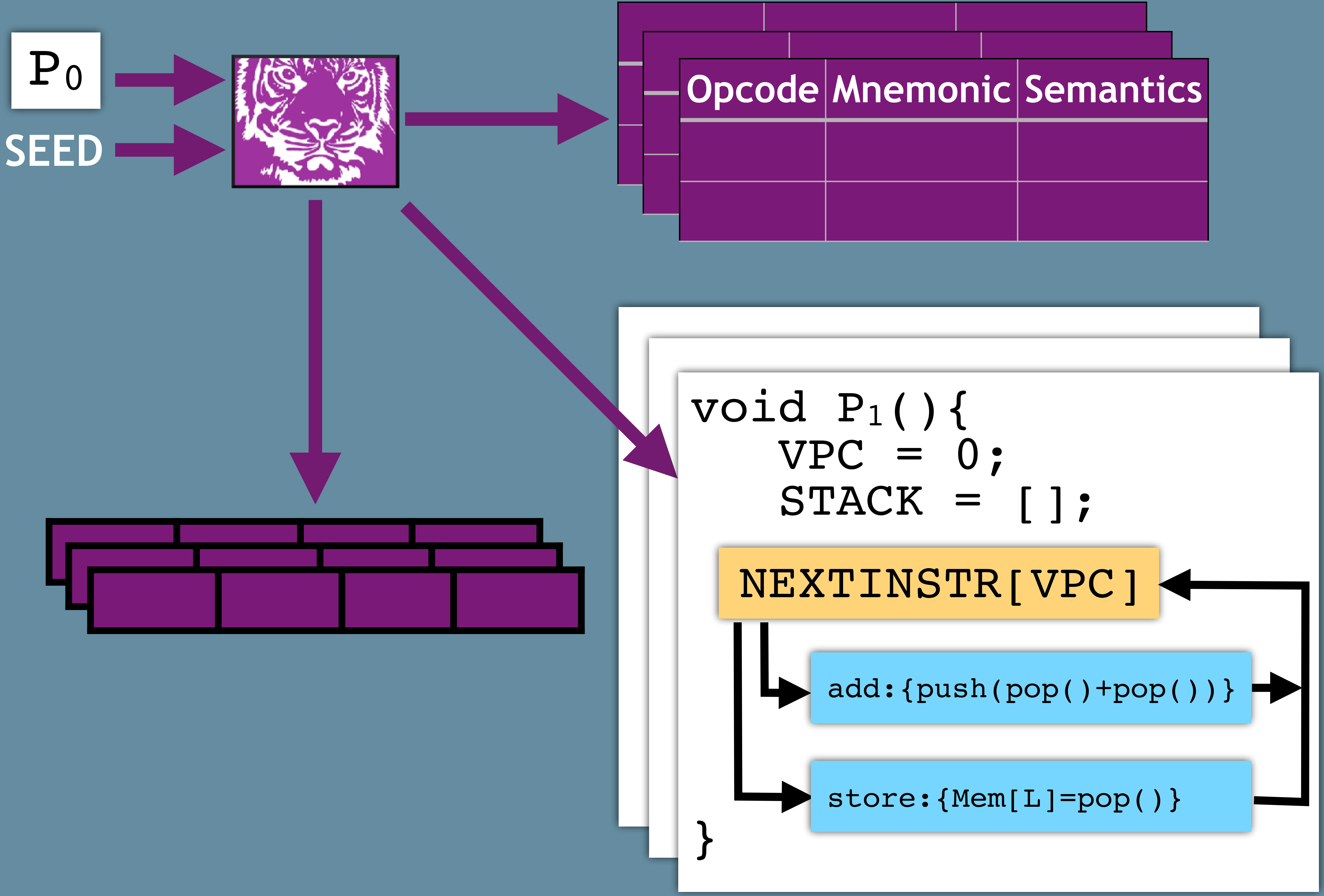
Virtual Instruction Set

Opcode	Mnemonic	Semantics
0	add	push(pop()+pop())
1	store L	Mem[L]=pop()
2	breq L	if pop()=pop() goto L

Virtual Program Array

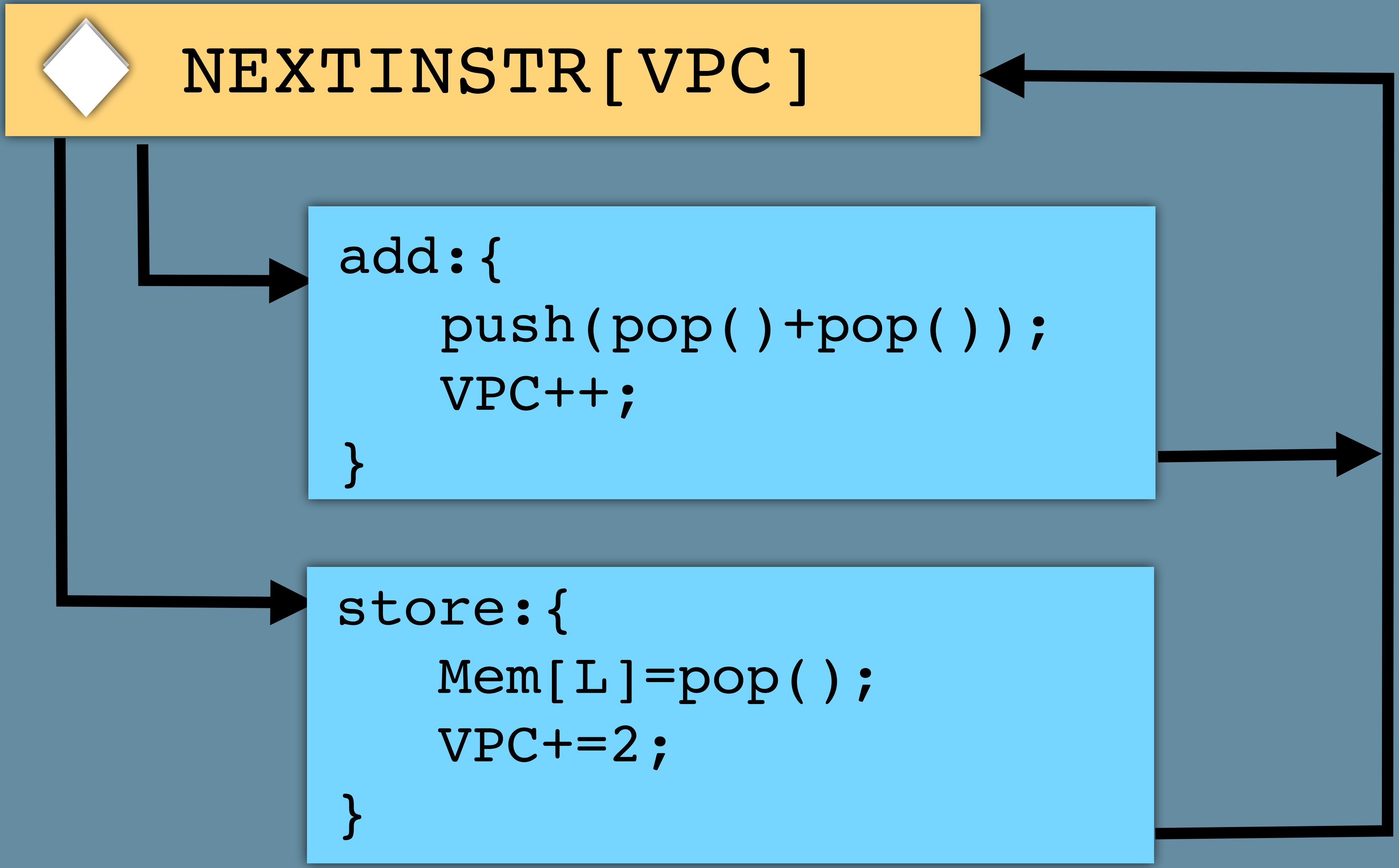
breq L1	add	store L2	push

```
void P1(){
    VPC = 0;
    STACK = [];
      DISPATCH
        HANDLER
        HANDLER
}
```



Opcode	Mnemonic	Semantics

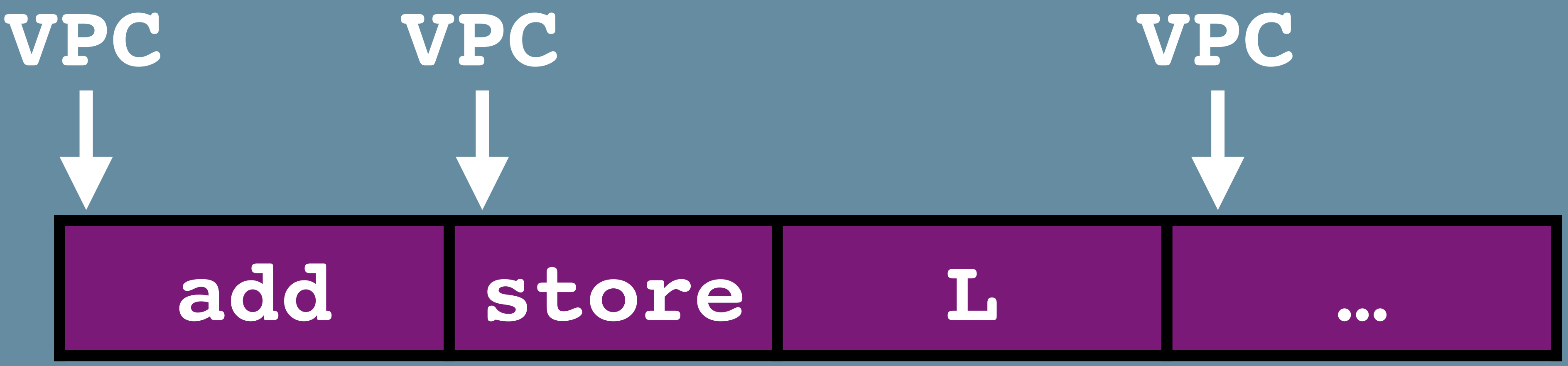
```
void P1(){
    VPC = 0;
    STACK = [];
  NEXTINSTR[VPC]
      add:{push(pop()+pop())}
      store:{Mem[L]=pop()}
}
```



```
NEXTINSTR[VPC]
```

```
add:{
    push(pop()+pop());
    VPC++;
}
```

```
store:{
    Mem[L]=pop();
    VPC+=2;
}
```

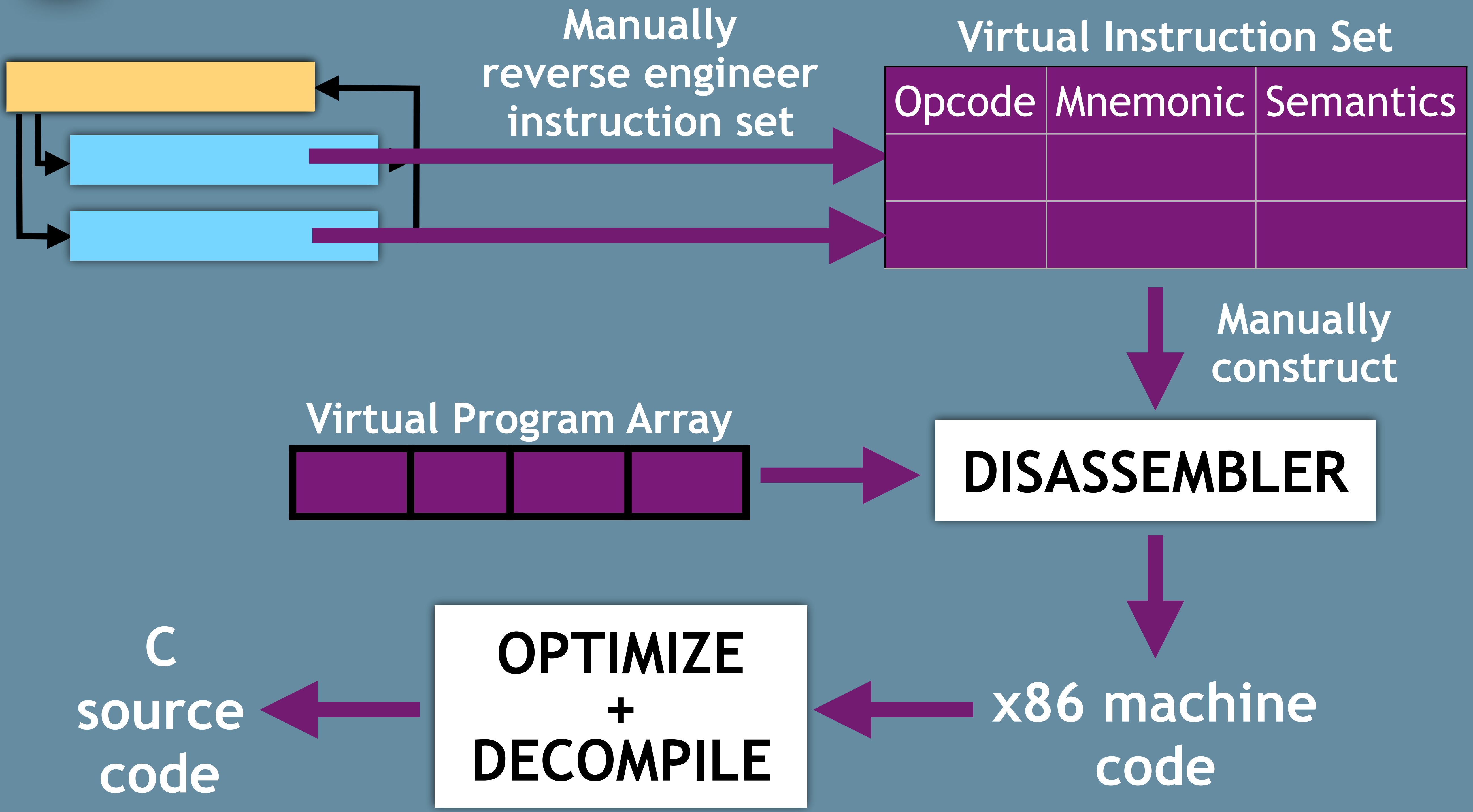


Exercise!

```
tigress\
   --Transform=Virtualize\
      --Functions=fib\
      --VirtualizeDispatch=switch\
   —out=v1.c fib.c
```

- Try a few different dispatchers: direct, indirect, call, ifnest, linear, binary, interpolation.
- Are some of them better obfuscators than others? Why?

Manual Analysis



Virtual Instruction Set

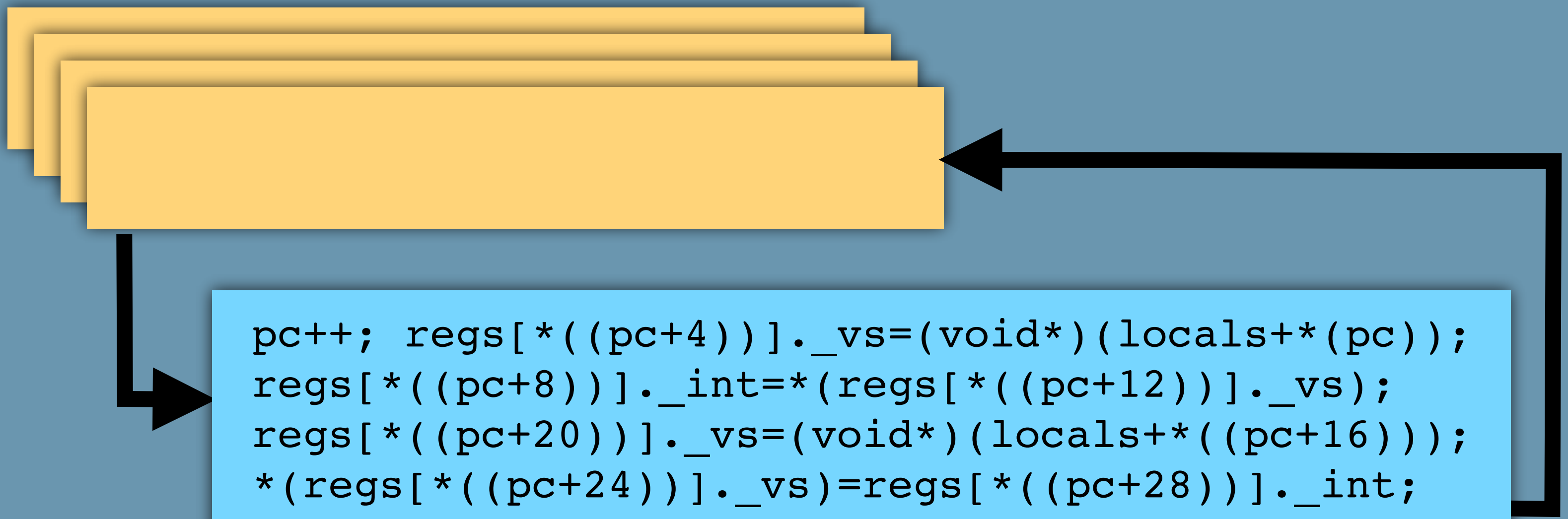
Opcode	Mnemonic	Semantics

Rolles, Unpacking virtualization obfuscators, WOOT'09

Randomize

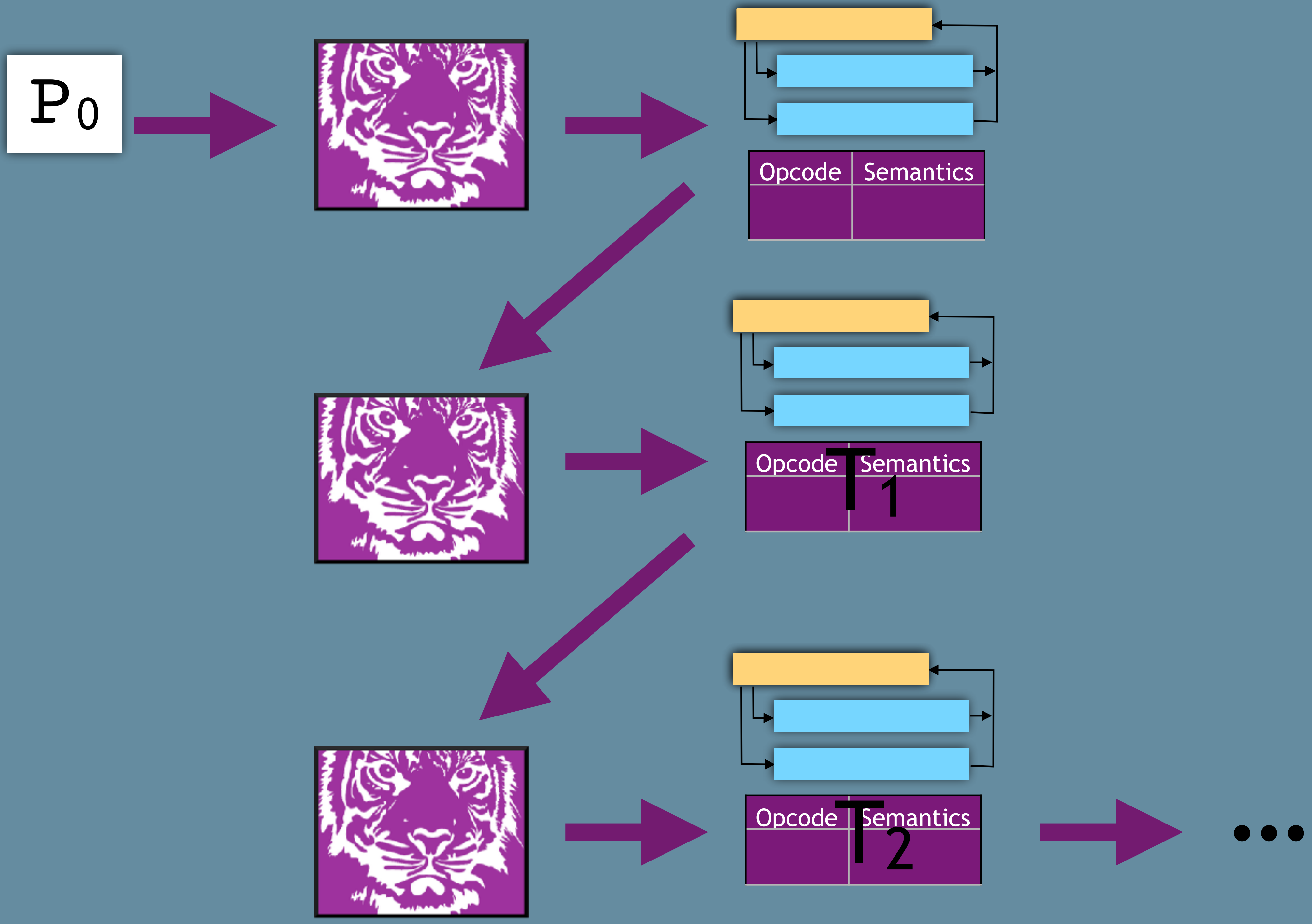
- Superoperators
- Randomize operands
- Randomize opcodes
- Random dispatch

Opcode	Semantics
93	R[b]=L[a];R[c]=M[R[d]];R[f]=L[e]; M[R[g]]=R[h];R[i]=L[j];R[l]=L[k]; S[++sp]=R[m];pc+=53;



```
pc++; regs[*((pc+4))]._vs=(void*)(locals+*(pc));
regs[*((pc+8))]._int=*(regs[*((pc+12))]._vs);
regs[*((pc+20))]._vs=(void*)(locals+*((pc+16)));
*(regs[*((pc+24))]._vs)=regs[*((pc+28))]._int;
regs[*((pc+32))]._vs=(void*)(locals+*((pc+36)));
regs[*((pc+44))]._vs=(void*)(locals+*((pc+40)));
stack[sp+1]._int=*(regs[*((pc+48))]._vs);
sp++;pc+=52;break;
```

Composition



Exercise!

```
tigress\
   --Transform=Virtualize
      --Functions=fib \
      --VirtualizeDispatch=switch\
   --Transform=Virtualize\
      --Functions=fib \
      --VirtualizeDispatch=indirect \
   --out=v2.c fib.c
```

- Try combining different dispatchers. Does it make a difference?
- Try three levels of interpretation! Do you notice a slowdown? What about the size of the program?

Obfuscating Arithmetic

Encoding Integer Arithmetic

$$x+y = x - \neg y - 1$$

$$x+y = (x \oplus y) + 2 \cdot (x \wedge y)$$

$$x+y = (x \vee y) + (x \wedge y)$$

$$x+y = 2 \cdot (x \vee y) - (x \oplus y)$$

Example

One possible encoding of

```
   z=x+y+w
```

is

```
   z = (((x ^ y) + ((x & y) << 1)) | w) +
         (((x ^ y) + ((x & y) << 1)) & w);
```

Many others are possible, which is good for diversity.

Exercise!

- The virtualizer's **add** instruction handler could still be identified by the fact that it uses a + operator!
- Try adding am arithmetic transformer:

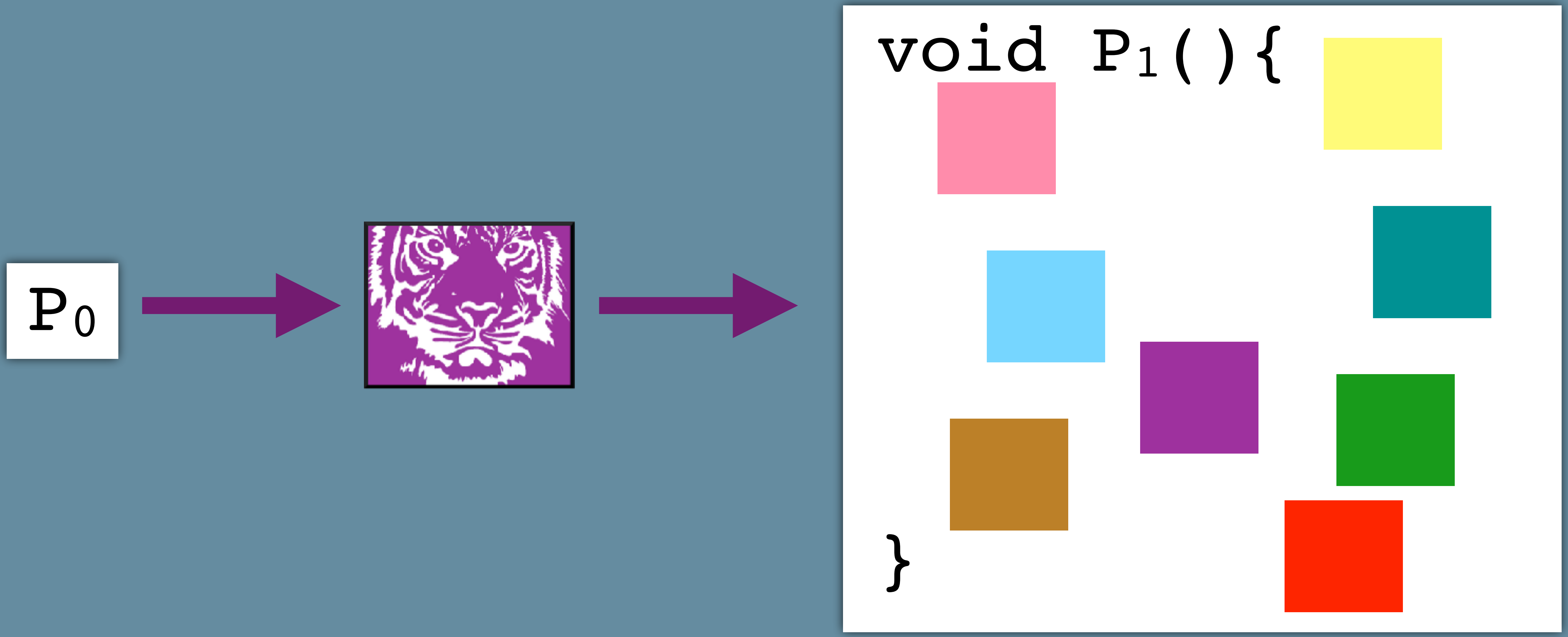
```
 --Transform=EncodeArithmetic \
    --Functions=fib,main ...
```

- What differences do you notice?
- Should this transformation go before or after the virtualization transformation?

Dynamic Obfuscation

Dynamic Obfuscation

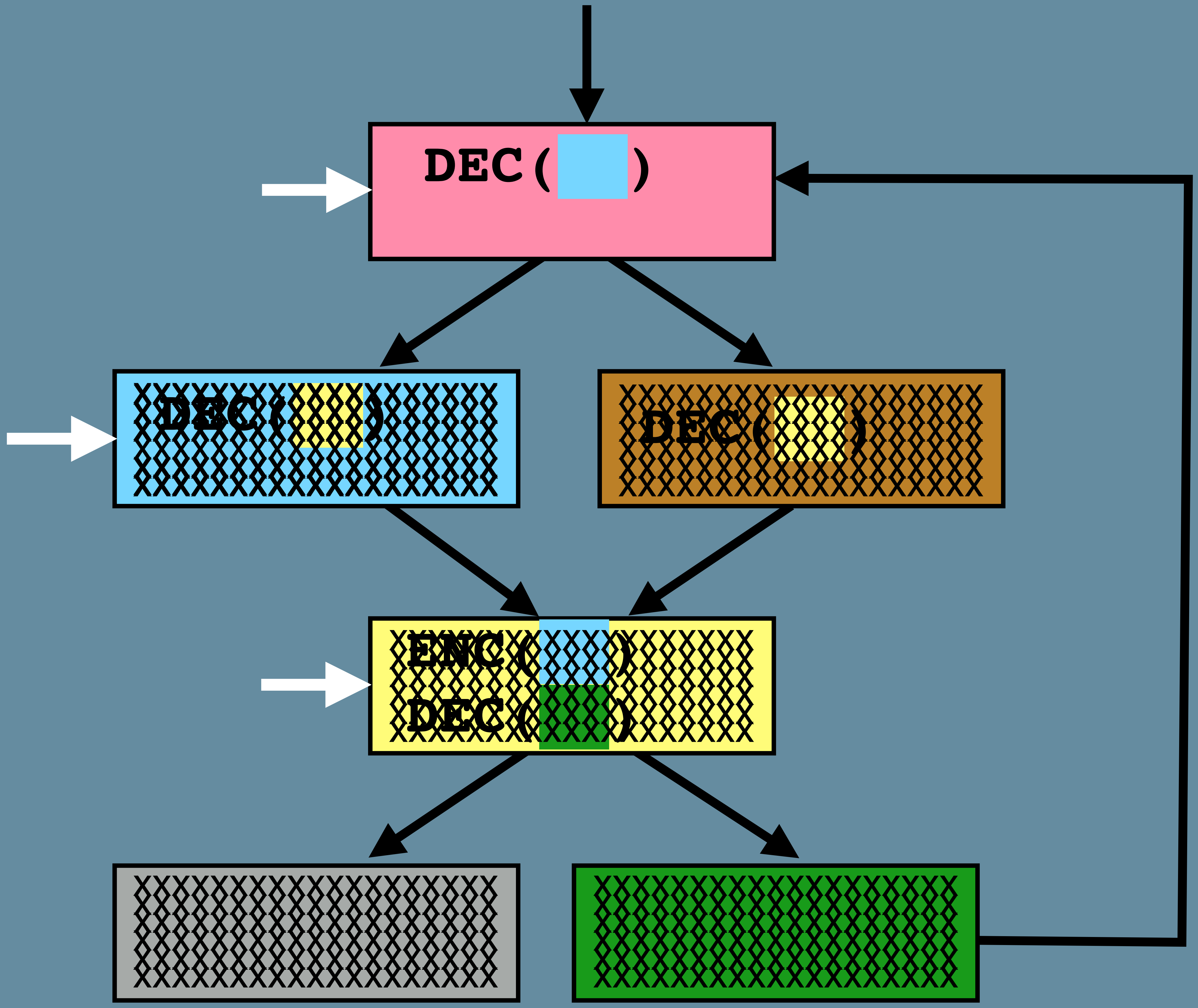
- Keep the code in constant flux at runtime
- At no point should the entire code exist in cleartext

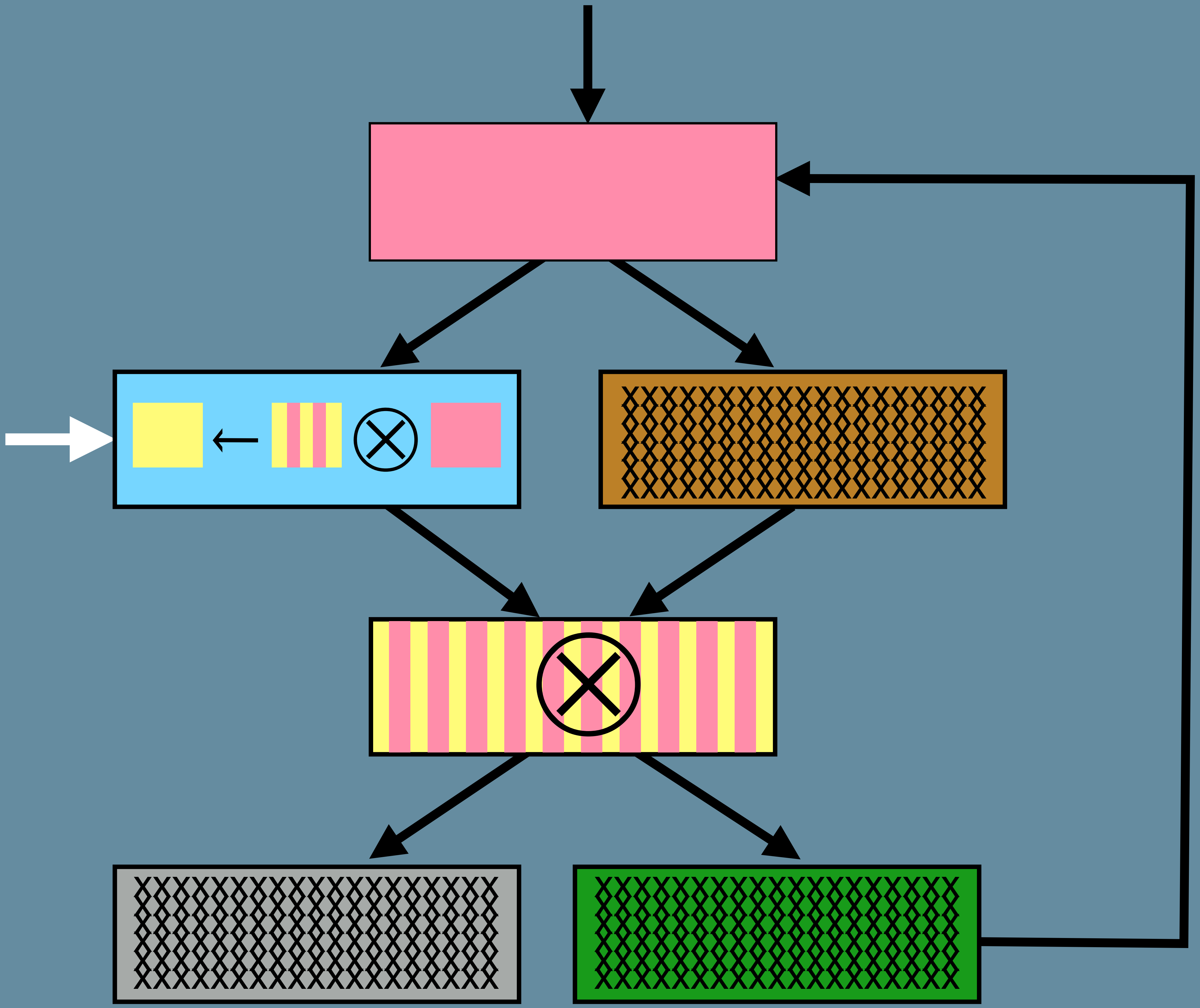


P_0

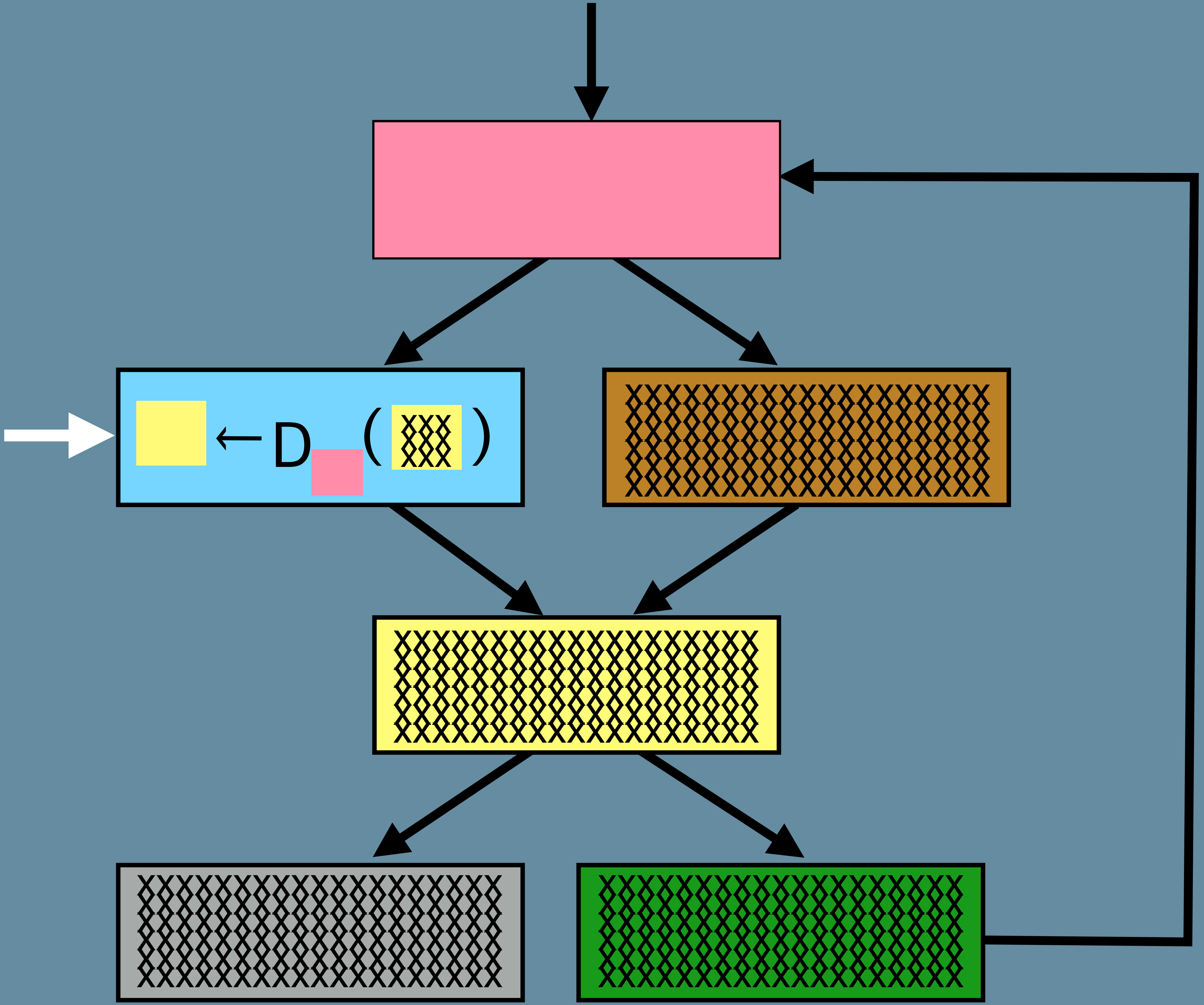
void P_1(){

}

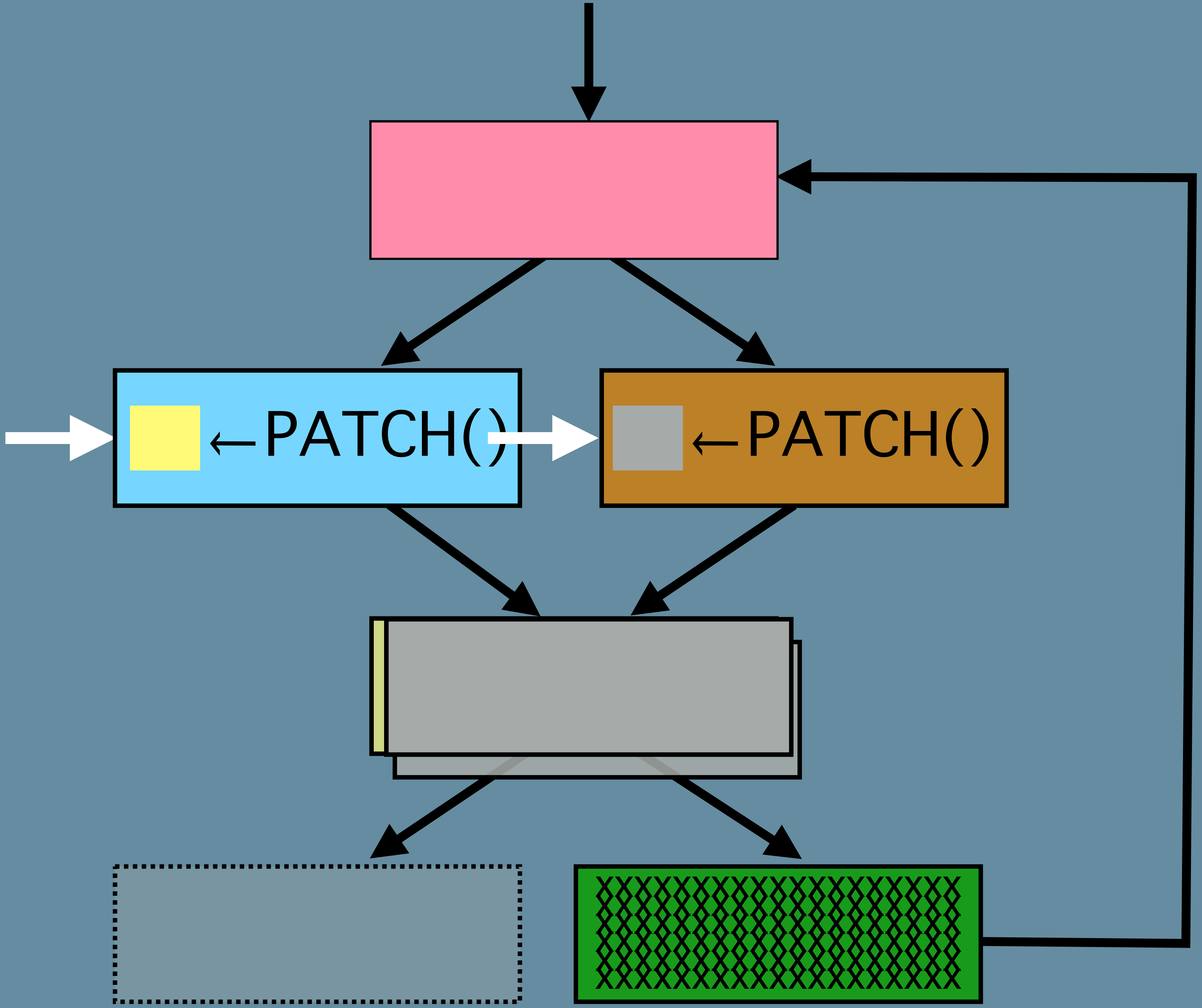

DEC( )
DEC( )
DEC( )
ENC( )
DEC( )



Aucsmith, Tamper Resistant Software: An Implementation, IH’96



Cappaert, Preneel, et al. Towards Tamper Resistant Code Encryption P&E, ISPEC'08



Madou, et al., Software protection through dynamic code mutation, WISA'05

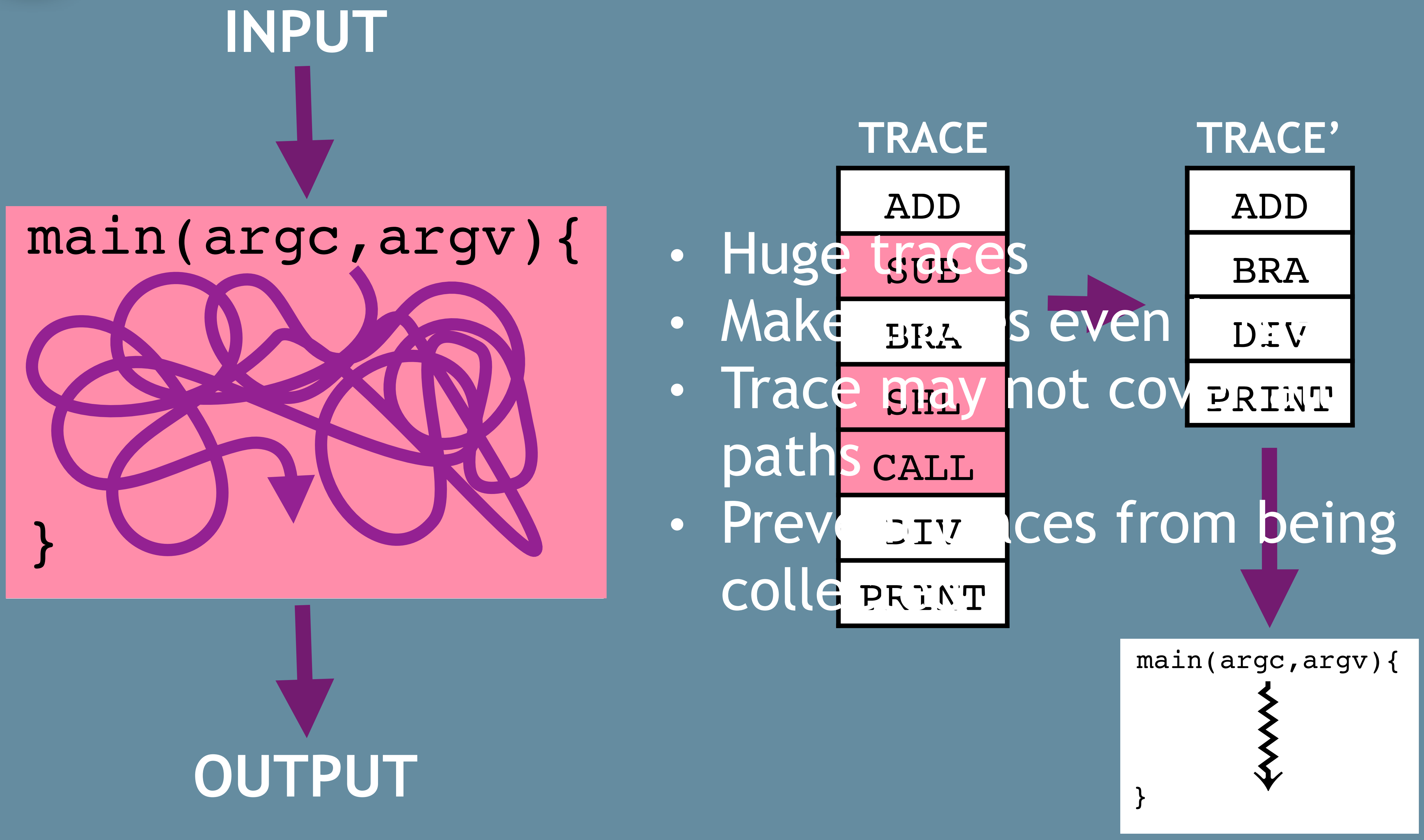
Exercise!

```
tigress \
   --Transform=Dynamic \
      --Functions=fib \
         --DynamicCodecs=xtea \
         —DynamicDumpCFG=false \
         --DynamicBlockFraction=%50 \
         --out=fib_out.c fib.c
```

- If you have “dot” (graphviz) installed, you can set DynamicDumpCFG=true and look at the generated .pdf files of the transformed CFGs.

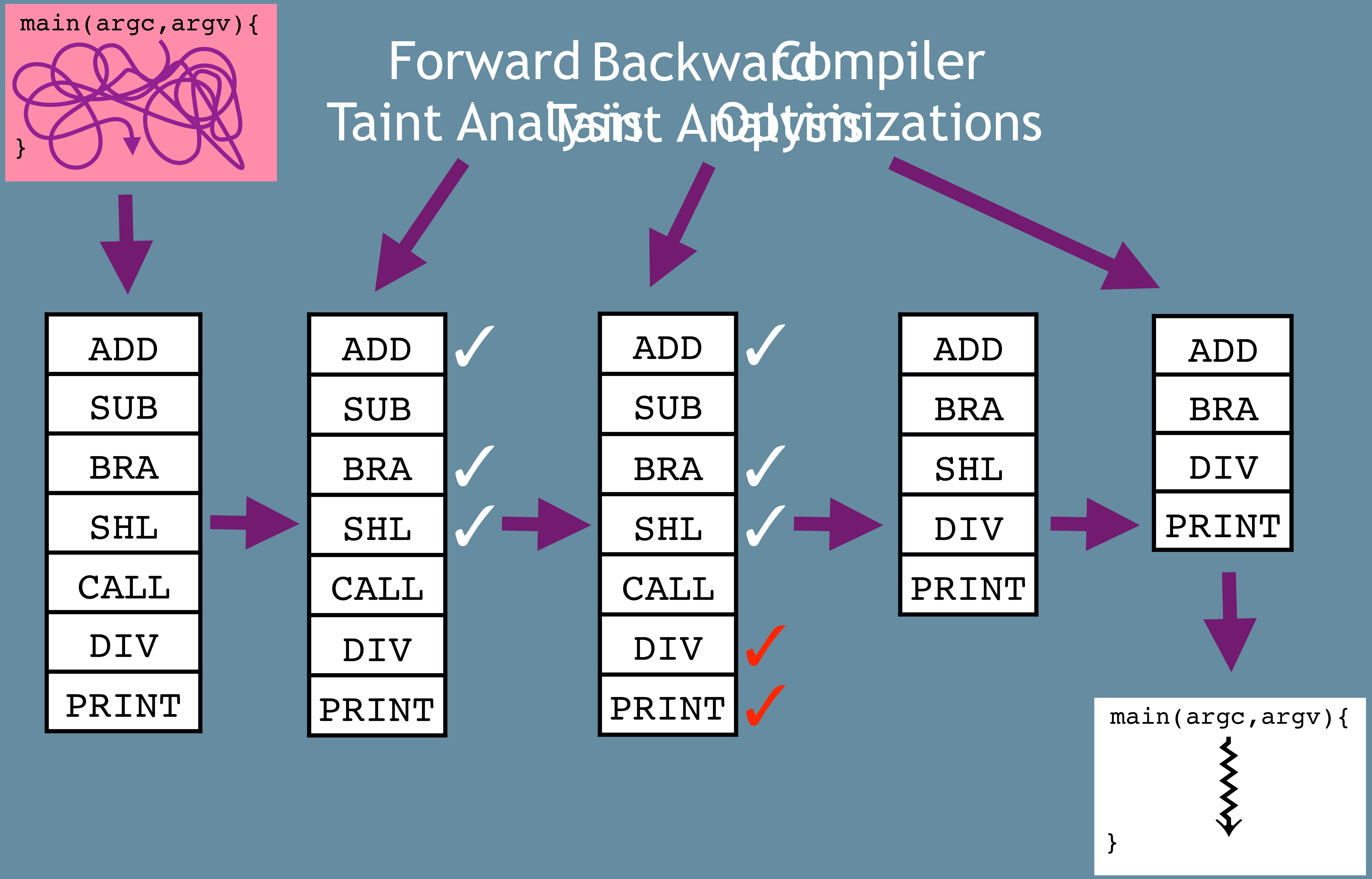
Dynamic Analysis

Dynamic Analysis



- Huge traces
- Make s even
- Trace may not cov paths
- Preve ces from being colle

Yadegari, et al., A Generic Approach to Deobfuscation. IEEE S&P’15



Forward Taint Analysis

Backward Taint Analysis

Compiler Optimizations

```
main(argc,argv){
}
```

ADD
SUB
BRA
SHL
CALL
DIV
PRINT

ADD ✓
SUB
BRA ✓
SHL ✓
CALL
DIV
PRINT

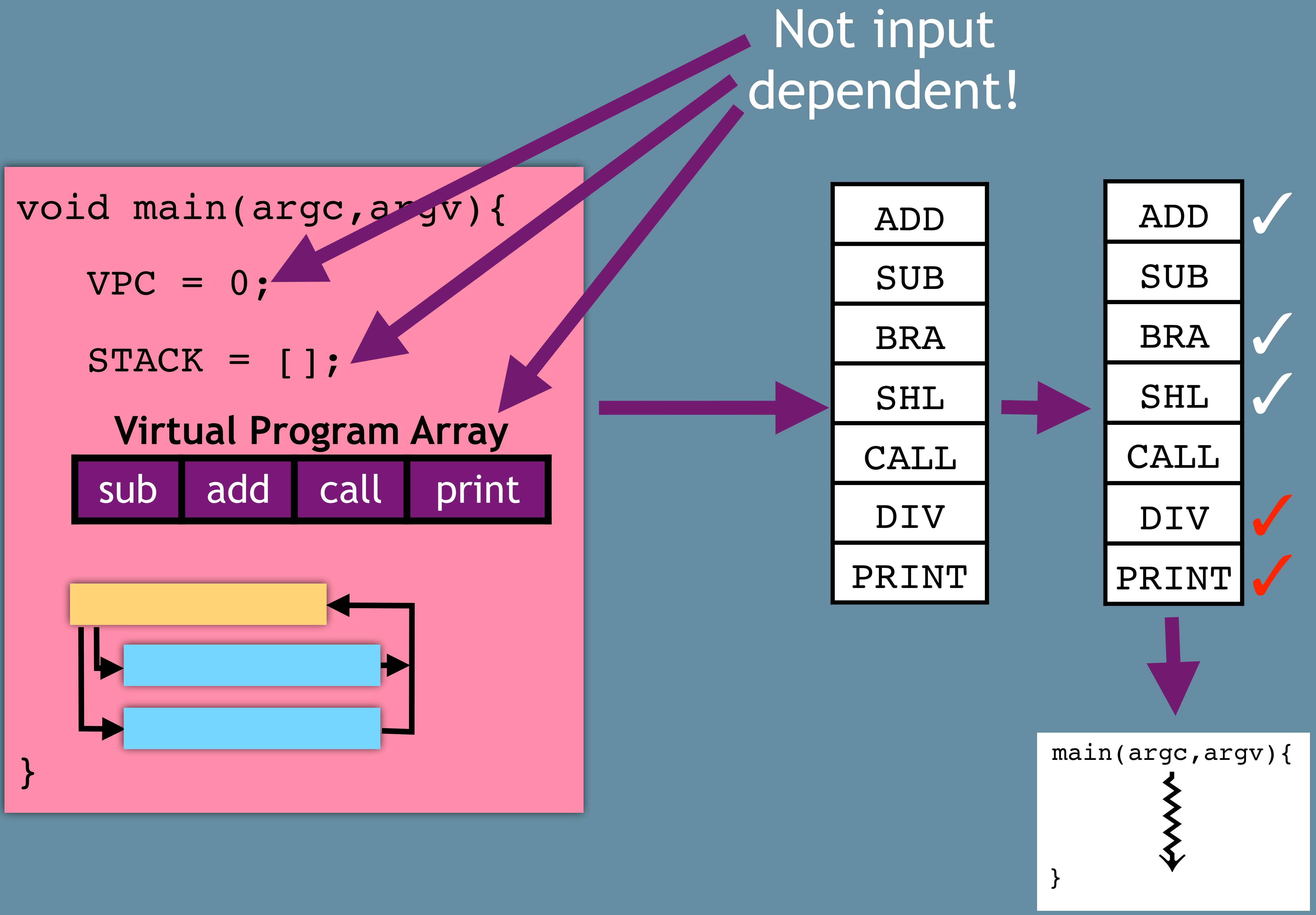
ADD ✓
SUB
BRA ✓
SHL ✓
CALL
DIV ✓
PRINT ✓

ADD
BRA
SHL
DIV
PRINT

ADD
BRA
DIV
PRINT

```
main(argc,argv){
}
```

Yadegari, et al., A Generic Approach to Deobfuscation. IEEE S&P'15



Not input dependent!

```
void main(argc,argv){
    VPC = 0;
    STACK = [];
```

Virtual Program Array

sub	add	call	print

```
}
```

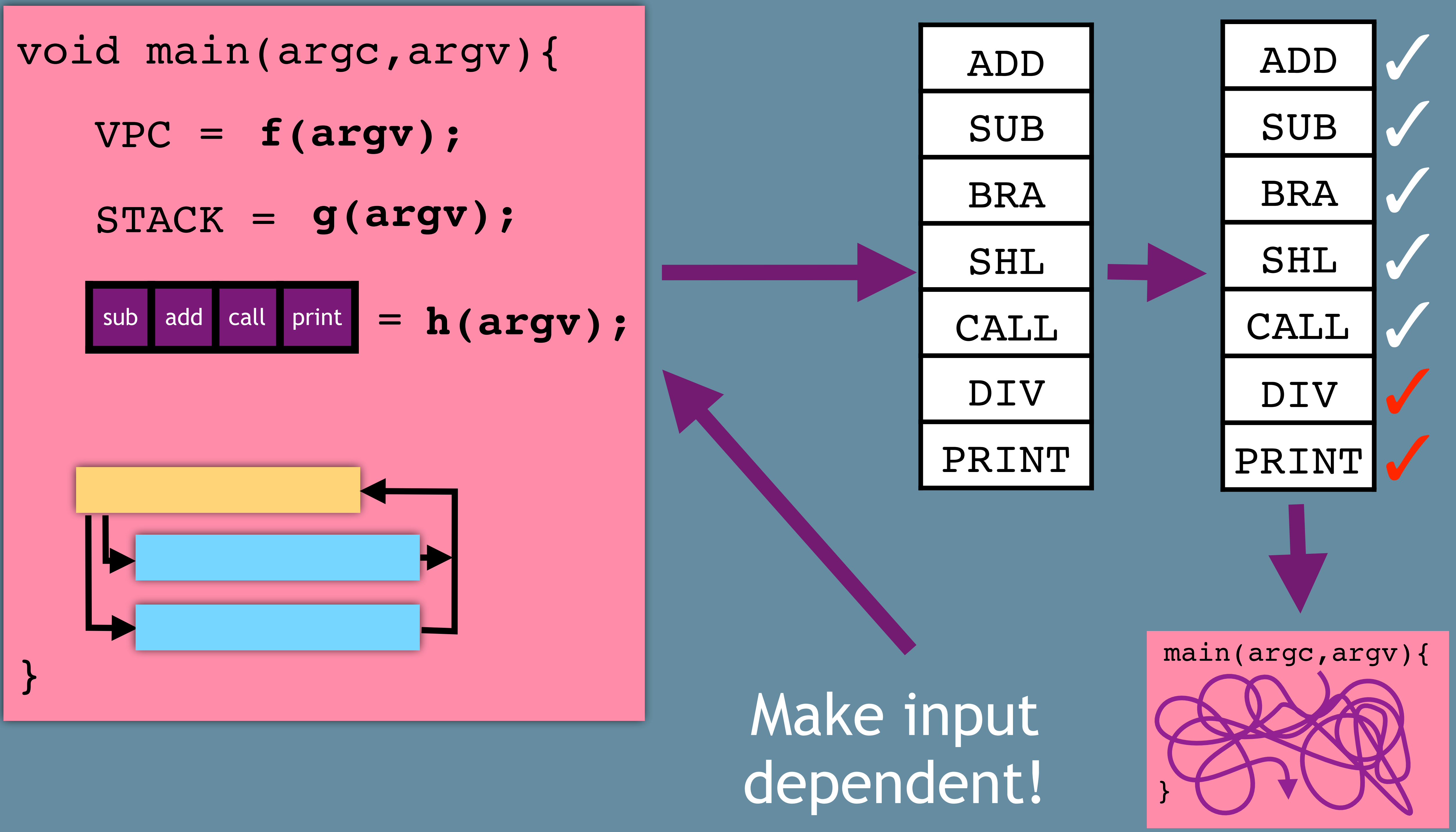
ADD
SUB
BRA
SHL
CALL
DIV
PRINT

ADD ✓
SUB
BRA ✓
SHL ✓
CALL
DIV ✓
PRINT ✓

```
main(argc,argv){
}
```

Yadegari, et al., A Generic Approach to Deobfuscation. IEEE S&P'15

Anti-Taint Analysis

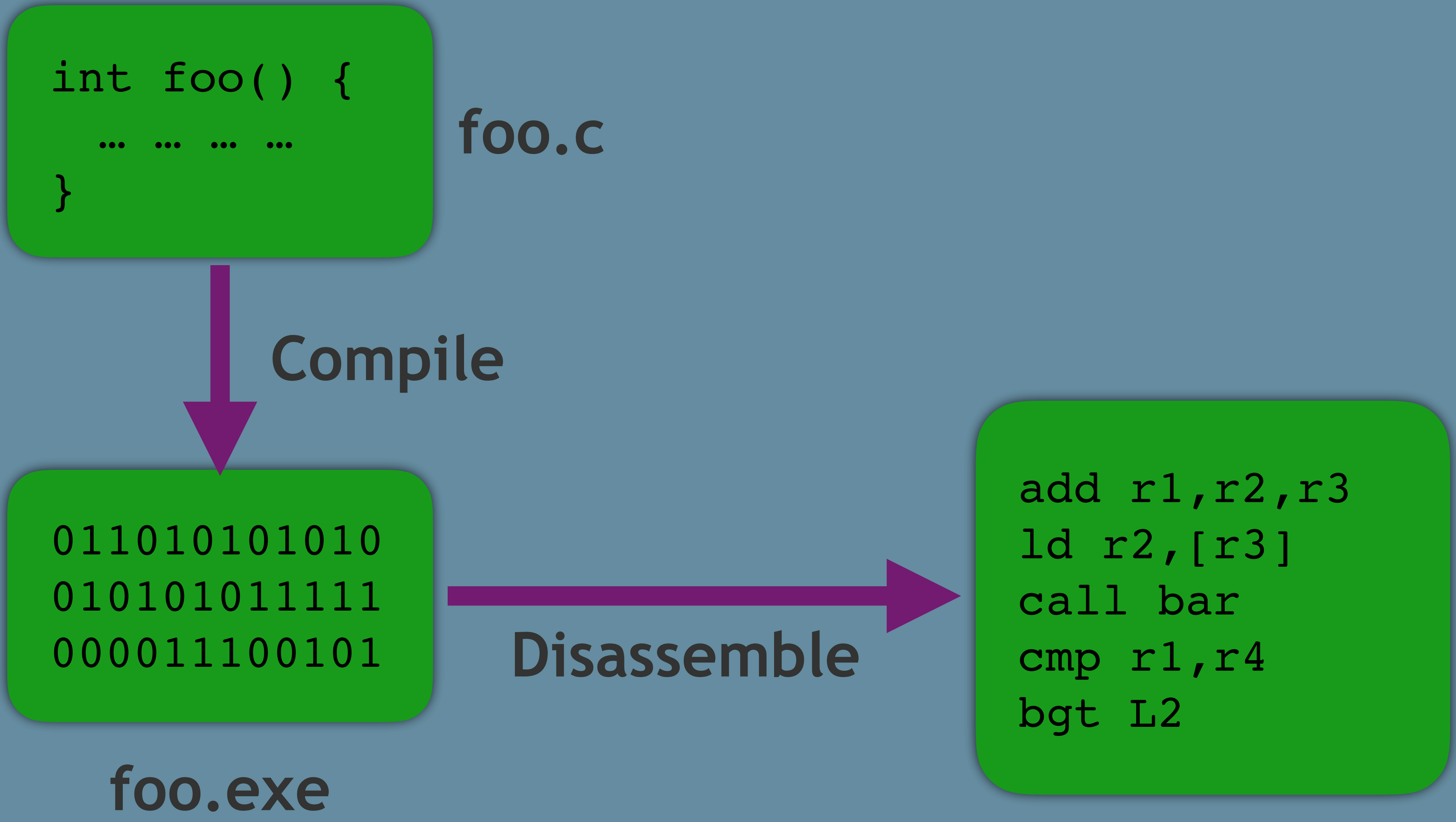


```
void main(argc,argv){
    VPC = f(argv);
    STACK = g(argv);
    sub add call print = h(argv);
}
```

Make input dependent!

Anti-Disassembly

- ***Attackers**: prefer looking at assembly code than machine code*



```
int foo() {
  … … … …
}
```

foo.c

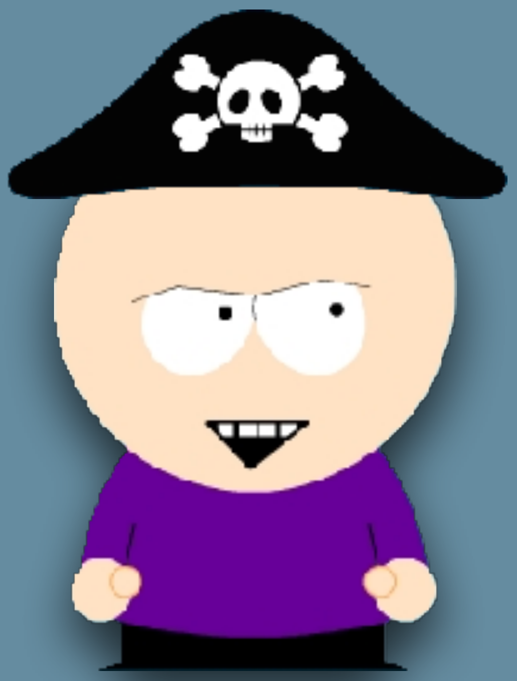
Compile

```
011010101010
010101011111
000011100101
```

foo.exe

Disassemble

```
add r1,r2,r3
ld r2,[r3]
call bar
cmp r1,r4
bgt L2
```



Address

Code bytes

Assembly

```
55 48 89 e5 48 83 c7
68 48 83 c6 68 5d e9
26 38 00 00 55 48 89
e5 48 89 e5 48 8d 4
68 48 89 c7 5d e9 1
38 00 00 55
```

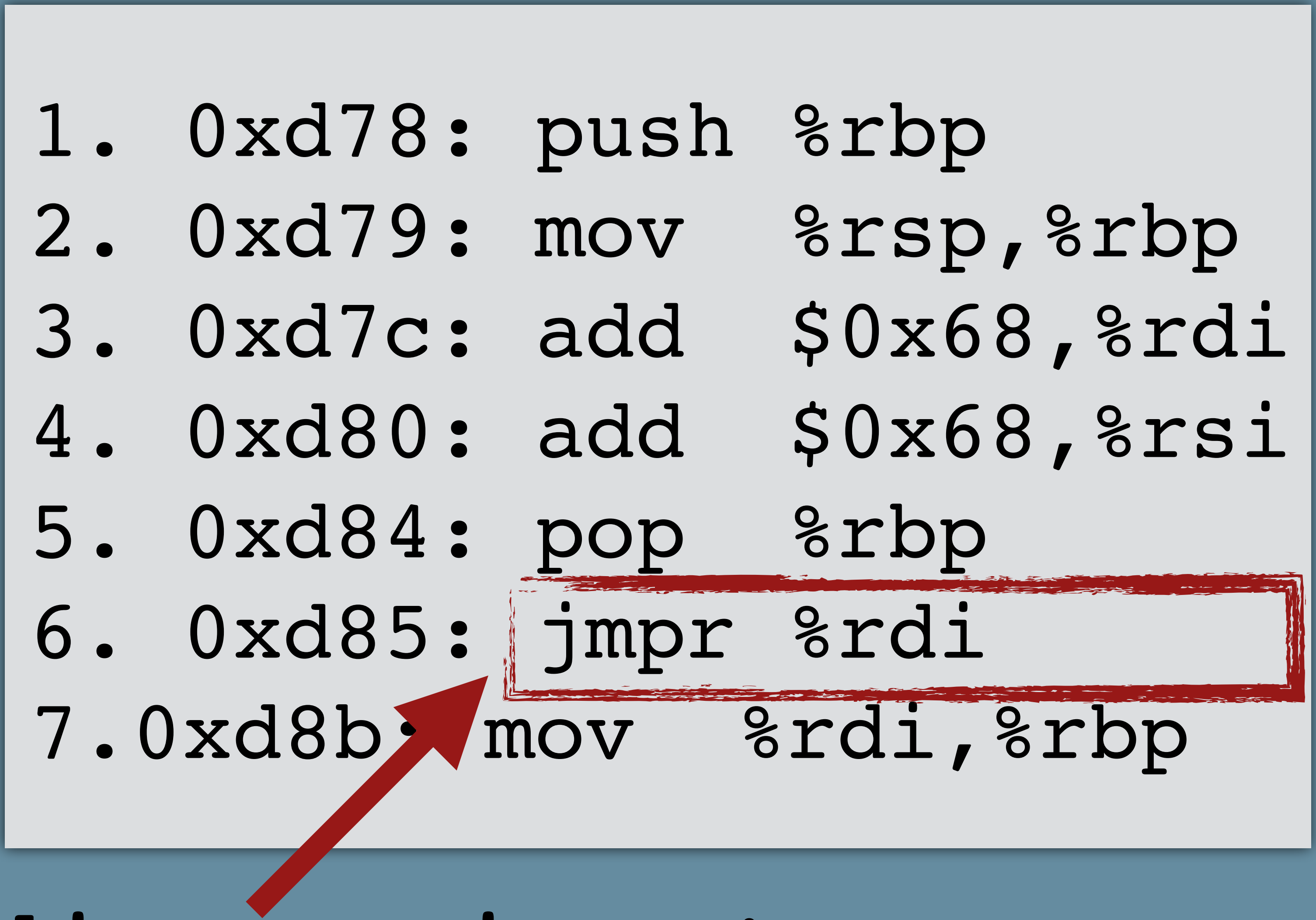
```
rbp
rsp,%rbp
0x68,%rdi
0x68,%rsi
rbp
0045b0

%rbp
%rsi),%rax
(%rdi),%rsi
x,%rdi

0045b0
%rbp
```

Linear Sweep Disassembly

```
0xd78:  push %rbp
0xd79:  mov  %rsp,%rbp
0xd7c:  add  $0x68,%rdi
0xd80:  add  $0x68,%rsi
0xd84:  pop  %rbp
0xd85:  jmpq 0x45b0
0xd8a:  .byte 0x55
0xd8b:  mov  %rdi,%rbp
```

- **Linear sweep** disassembly has problems with data mixed in with the instructions!

Exercise!



```
0xd78: push %rbp
0xd79: mov  %rsp,%rbp
0xd7c: add  $0x68,%rdi
0xd80: add  $0x68,%rsi
0xd84: pop  %rbp
0xd85: jmpr %rdi
0xd8b: mov  %rdi,%rbp
```

Indirect jump!

- How would a **recursive traversal** disassembly handle this code?

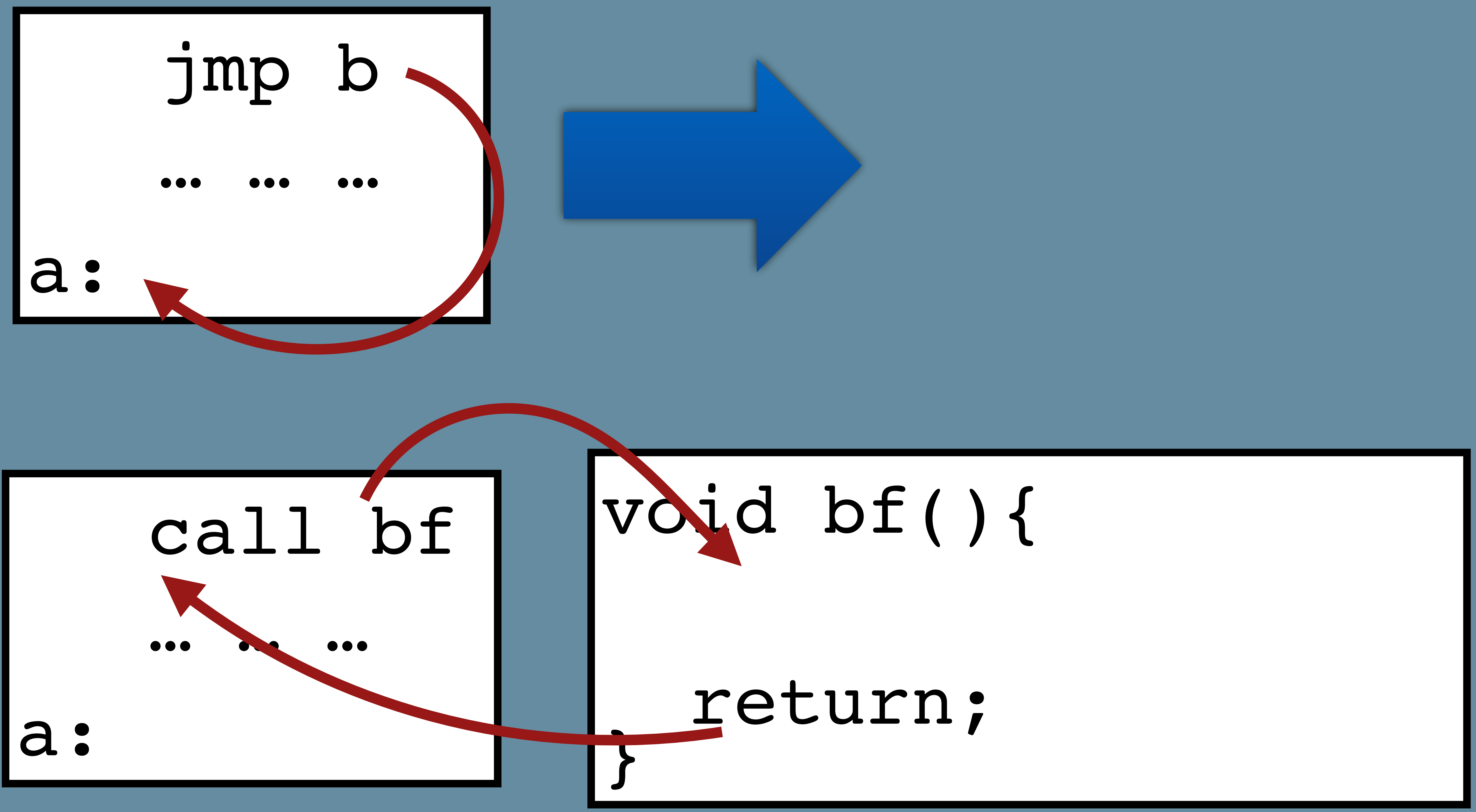
Insert Bogus Dead Code

- Insert unreachable bogus instructions:

```
if (opaquely false)
  asm(“.byte 0x55 0x23 0xff…”);
```

- This kind of lightweight obfuscation is common in malware.

Branch Functions

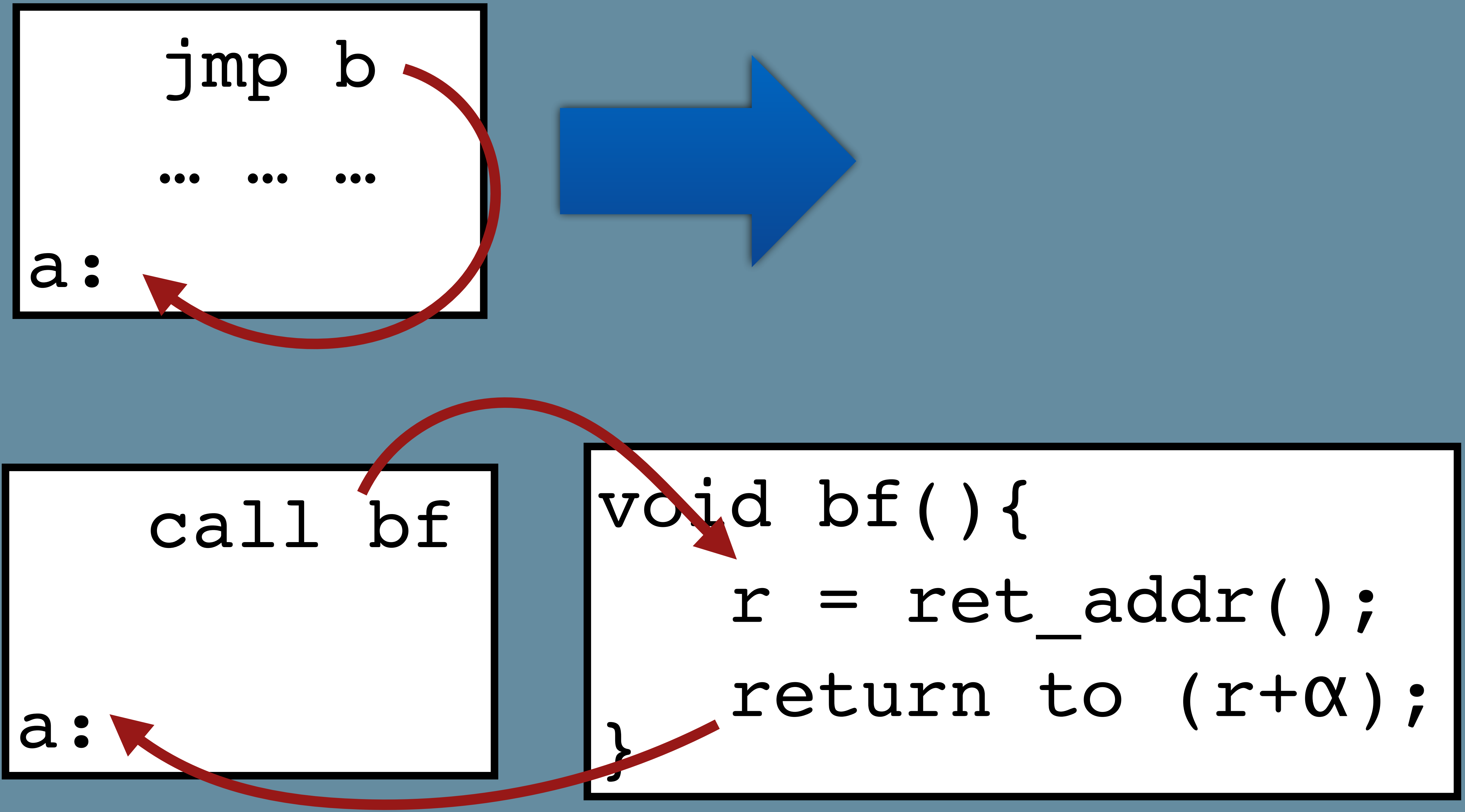


```
    jmp b
    … … …
a:
```

```
    call bf
    … … …
a:
```

```
void bf(){

   return;
}
```

Branch Functions

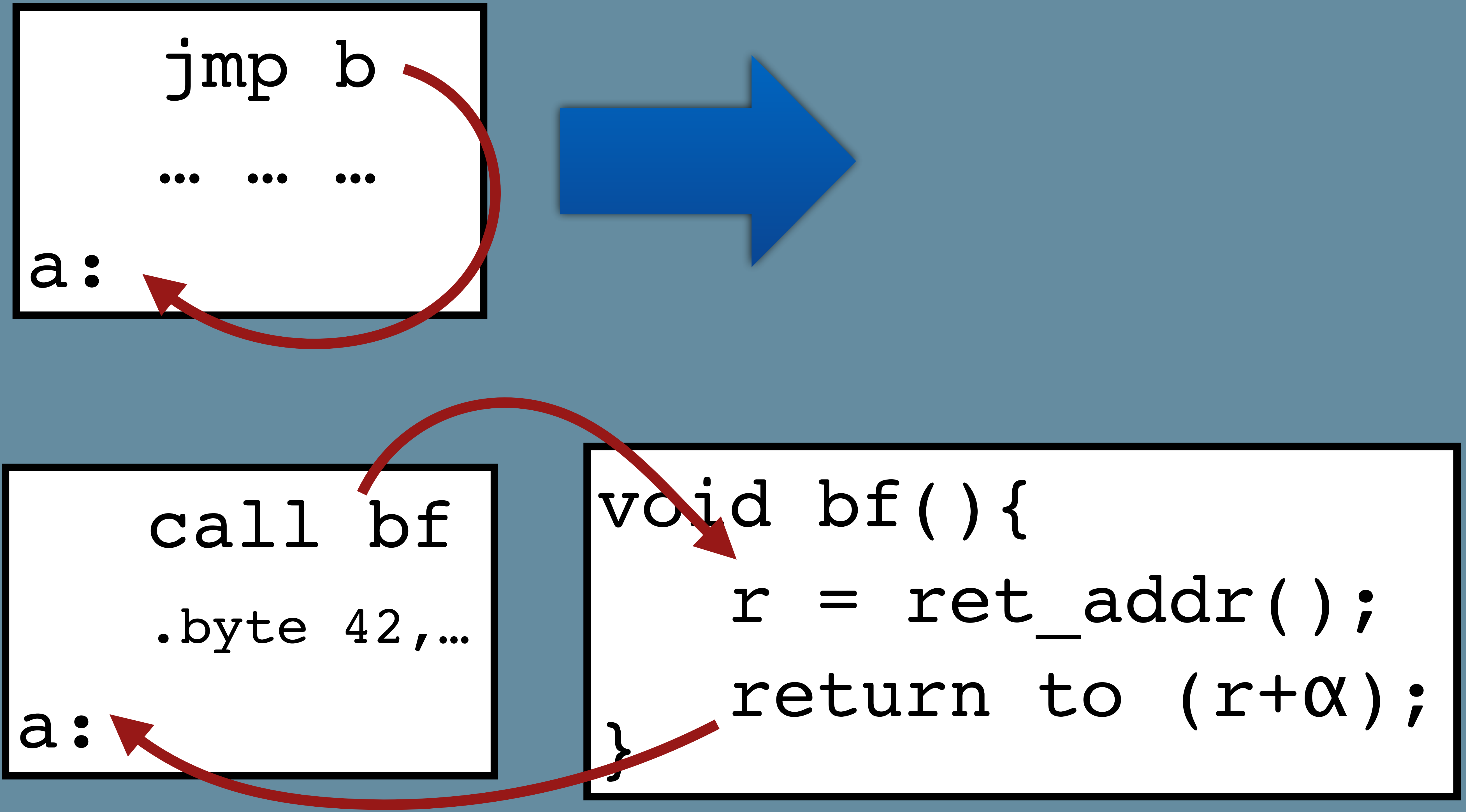


```
    jmp b
    ... ... ...
a:
```

```
    call bf


a:
```

```
void bf(){
    r = ret_addr();
    return to (r+α);
}
```

Branch Functions



```
    jmp b
    … … …
a:
```

```
    call bf
    .byte 42,…
a:
```

```
void bf(){
    r = ret_addr();
    return to (r+α);
}
```

Questions?

Exercise!

```
tigress \
   --Transform=InitBranchFuns \
      --InitBranchFunsCount=1 \
   --Transform=AntiBranchAnalysis \
      --AntiBranchAnalysisKinds=branchFuns \
      --Functions=fib \
   --out=fib_out.c fib.c
```