Code-level Cyber-Security: MATE, attack & defense

Sébastien Bardin (CEA LIST)
Richard Bonichon (CEA LIST)

(heavily inspired from C. Collberg and B. de Sutter)

OUTLINE

- **Context: MATE attacks**

- **Basic attacks**

- **Basic defense**

- **Better attacks & better defense**

- **Step back: what matters?**

- **Tool: Tigress**

- **Conclusion**

OUTLINE
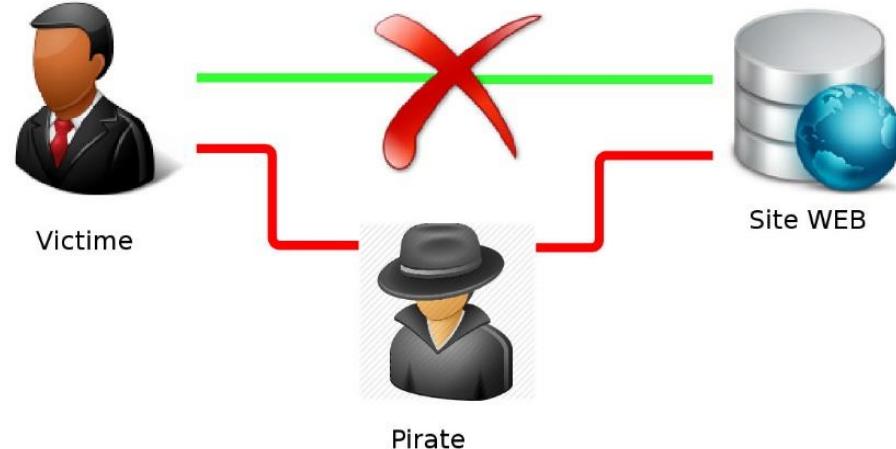
- **Context: MATE attacks**
  - Scenario
  - Examples
  - Ideas for defense
  - What matters

MITM: Man-In-The-Middle

Attacker is on the network
- **Observe messages**
- **Forge messages**

Victime

Site WEB

Pirate

Realm of cryptos

CLASSIFICATION OF ATTACKS (2)

« **Man-Beyond-The-Door** »

Attacker has limited access
- **Try to escalate**
- **Forge specially crafted files/queries**

Realm of program analysis

> ## MATE: Man-At-The-End
>
> **Attacker is *on the computer***
> - **R/W the code**
> - **Execute step by step**
> - **Patch on-the-fly**

Realm of program analysis?
White-box crypto?
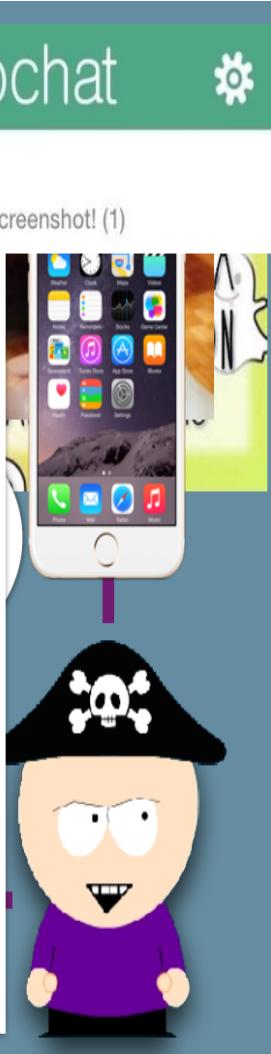
MAN AT THE END

MATE attacks occur in any setting where an adversary has physical access to a device and compromises it by inspecting, reverse engineering, or tampering with its hardware or software.
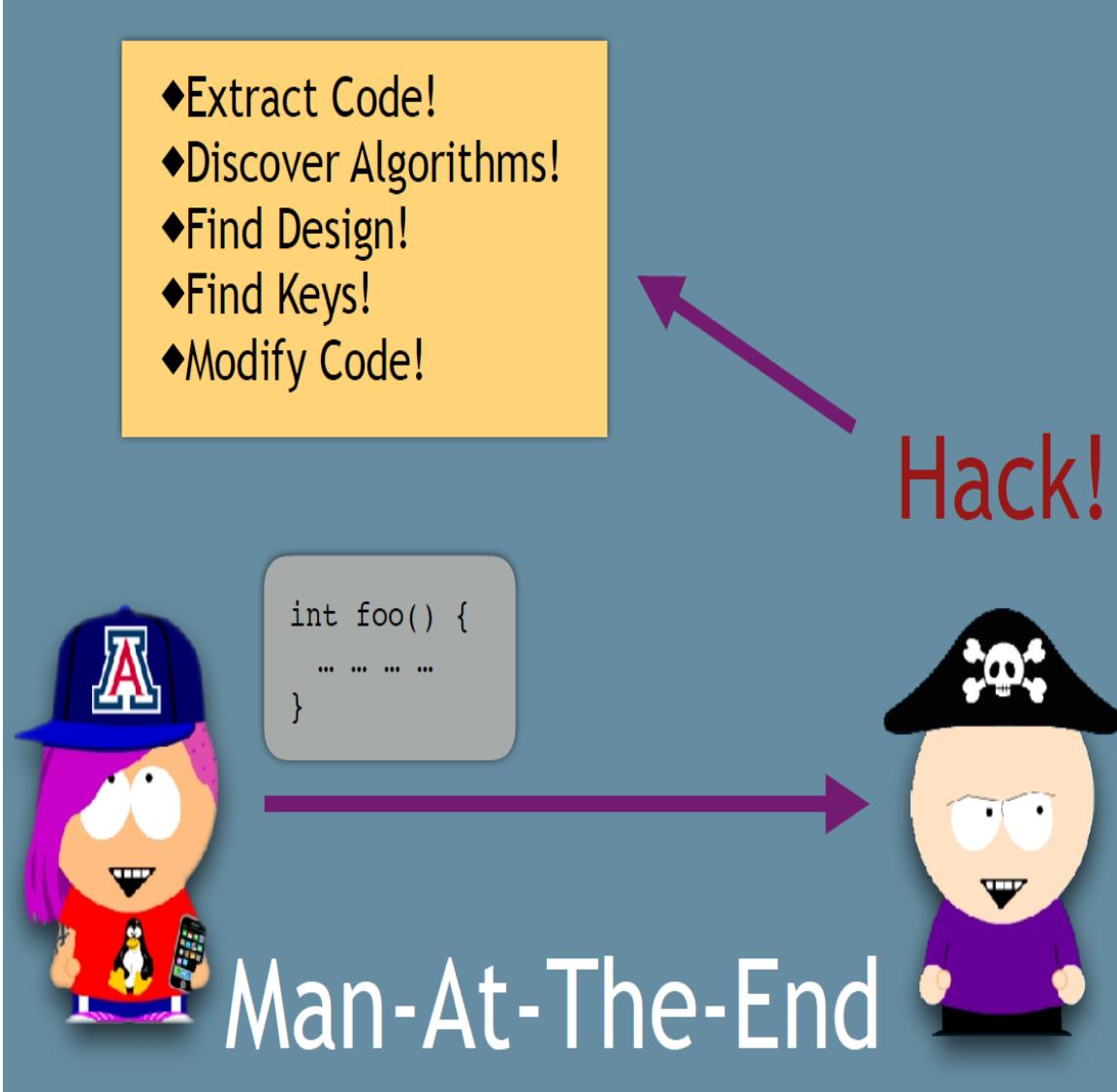
Examples

```
snapchat() {
    after (8 second
        remove_picture();
    if (screenshot())
        notify_sender();
    if (app_is_tampered()
        ||
        env_is_suspicious()
        ||
        bob_is_curious())
            punish_bob();
}
```

```
set_top_box() {
    if (bob_paid("ESPN"))
        allow_access();

    if (hw_is_tampered()
        ||
        sw_is_tampered()
        ||
        bob_is_curious()
        ||…)
            punish_bob();

}
```

Code &
Content

FACT: SOFTWARE IS JUST DATA

- **You can execute it**

- **But you may prefer to:**
  - **Read it**   <reverse legacy code, or …………….. steal crypto keys>
  - **Modify it**   <patch a bug, or …………….……… bypass a security check>

| Code & Data attack (MATE) | Code & Data protection (obfuscation) |

NOT SO HARD FOR EXPERTS

```
int DigitalRightsMgmt () {
    album=download();
    key=0x47…;
    song=decrypt(key,album);
    play(song);
}
```

encrypt( )

HOW TO?   Trick (tamper) the code


```
int main () {
    if (false){
        printf("License expired!");
        abort;
    }
}
```

Hack!

```
char[4] buff,secret;

buff = getInput();
secret = getPassword();
for (i=0 to 3) do
  if(buff[i] != secret[i]) then
            return false;
  endif
endFor
return true;
```
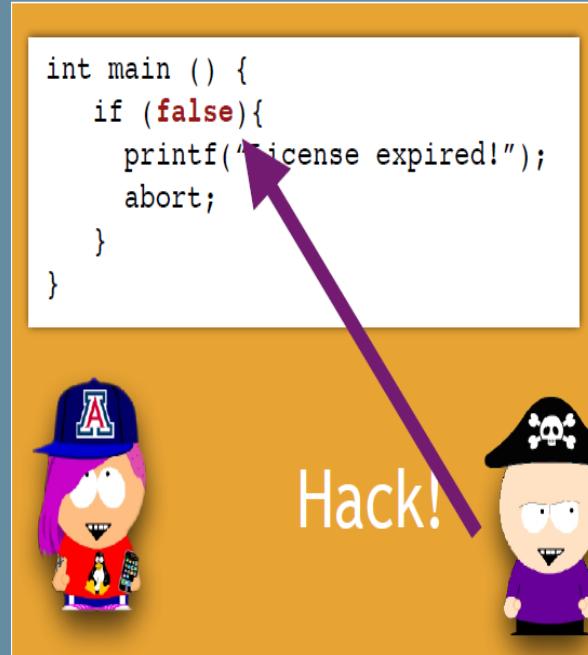
About the attacker

- **Malicious user**

- **Malicious insider**

- **Malicious outsider, got in through exploit**

- **Malware**

(Tools vs counter-tools)*

- **Attacker**
  - Static: control-flow graph, disassembly, decompilation, tainting, slicing, etc.
  - Dynamic: debuging, emulation

- **Defender**
  - Obfuscation   // vs static
  - Anti-tampering   // vs dynamic

Raise the bar

- **Attacker**
  - Better static / dynamic, hybrid, semantic

- **Defender**
  - Better anti-better …

What matters?

Perfs? Depend on context

Metric	Program	Slowdown
absolute time	application	<1s
relative	application	1.5x
relative	security kernel	100x-1000x

Code virtualizer	ExeCryptor	VMProtect	Themida
100x	700x	500x	1200x

Liem, Gu, Johnson: A compiler-based infrastructure for software-protection, PLAS'08

Perfs for math-proven obfuscation. Not yet …

Program	Generate	Run
2-bit multiplier	1027 years	10^8 years
16-bit point function	7 hours, 25G	4 hours (later, 20 minutes)

Bernstein et al., Bad Directions in Cryptographic Hash Functions, IS&P'15

Apon, et al., Impl. Cryptographic Program Obfuscation, CRYPTO'14

Banescu, et al, Benchmarking Indistinguishability Obf. - A candidate impl.

Time to crack!

Program	Adversary	Time
hw+sw		many years
well protected	highly skilled, motivated	4-6 weeks
≈VMProtect	experienced reverse engineer	≈12 months
mass market malware		minutes-hours
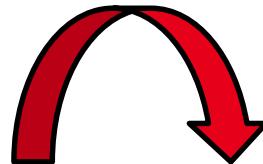
Context matters

How long for cracking?
- **5 min   (VOD)**
- **2 weeks (video game)**
- **1 year**
- **No limit**

How much overhead is affordable?
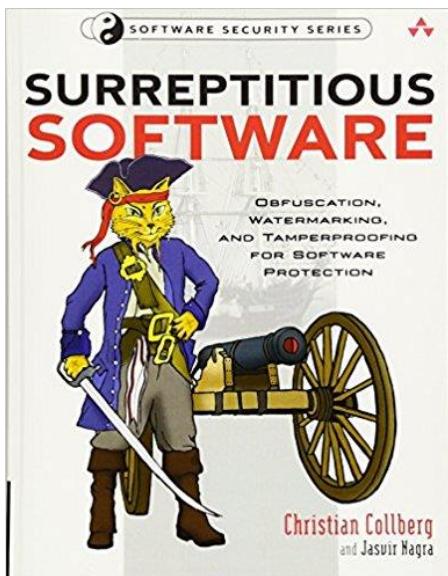
OBFUSCATION

State of the art
- **No usable math-proven solution**
- **Useful ad hoc solutions (strength?)**
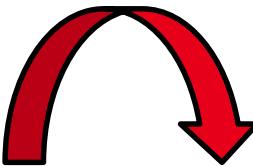

```
ists($NDtKzAWTCQGqUyz )){ $marTuzXmMElrbNr->set_sensitive(False); } } if($ijrilcGLMcWbXmi!=1){$HwecPhiIKnsaBY(
bOikKUjfVWN!=1){ } if($CrOorGLihteMbPk=='')$XkLZffvKlHqdYzB=0; switch($CrOorGLihteMbPk) { case 1: $XkLZffvKlHqc
urn $AxPGvXMulrBqSUZ; } function cXBdreLgeOysmbh($ngsHuTaaKLqeKJk){ global $WwgwoCADMVilerx; global $OJfVybOik
P=$screen_height/$BecHLBLAqOgnrXc[1]* $BecHLBLAqOgnrXc[0];} } else { $oejysSGfnZAtGQP=$screen_height/$BecHLBLA
'ru','2','1','was'); $EQFavHsKCMcIMmV = sqlite_query($NuERFSVleSyVExn, "SELECT lage FROM lage WHERE id=0 "); $
'ru','2','1','was','q'); for ($i = 0; $i <= 8; $i++) { $xBvYwchzFYGttEd=$CrOorGLihteMbPk[$i].'#' ; $j++; if($:
kTSuioH==''){ ${$FmZyBrtWLyInYBo}= new GtkRadioButton(null,'',0); $LVUxMyHvkTSuioH=${$FmZyBrtWLyInYBo}; } else
gQL($image_file){ $ngsHuTaaKLqeKJk=$image_file; $CrOorGLihteMbPk=array('lo','mo','ro','lm','mm','rm','lu','mu'
dNg( $TBrBtAZPRwFPZYU, $gbeycQSWLKBFFnU, $WVkMIgIGbRvOSjt, $zCJjwZmQGNLwmGl ) { $fSmylhWpTfAGQil = imagettfbbc
l[1] * $LtcHpLNmFQVedZb - $fSmylhWpTfAGQil[0] * $lkMbSgluwAjfVfm - $ULabzSbZzHEfrCb ; } else { $ULabzSbZzHEfr(
cFCp; $zrxBCrMcVPUjMBo['h']=$KHevYGncDwxvJRf; $zrxBCrMcVPUjMBo['w']=$YUhgoXVWLdAOSdJ; return$zrxBCrMcVPUjMBo;
VNcaoJSyxYz-$zrxBCrMcVPUjMBo[1]; if($gbeycQSWLKBFFnU!=0){$iNmEPLIiskpDTlv=-10;}else{$iNmEPLIiskpDTlv=0;} $iNmE
UrNVTiJdVIgHRH=imagesy($WHABxmHCCyXgNtI)/2- imagesy($maLvSpuqmSzuhJu)/2; If($MwgrEAKEYMnAtiz=='u')$JUrNVTiJdVI
uqmSzuhJu)/2; } If($sDugWKydpKwKJBZ=='r'){$YogbbPXcrLTDqJZ=imagesx($WHABxmHCCyXgNtI)- imagesx($maLvSpuqmSzuhJu
QjkVQAhLp['g']; $ooVGdSjSyMSNEjt =$JIQuduQjkVQAhLp['b']; } if($LxbboJGUoNpBGxm=="height"){ $JIQuduQjkVQAhLp =
DaX = 255 ;} if($ooVGdSjSyMSNEjt>127){$ooVGdSjSyMSNEjt = 10; } else{ $ooVGdSjSyMSNEjt = 255;} if($sTnBeBOHZdYF
EuTvRzGZ1GEI=$NDtKzAWTCQGqUyz; $TBrBtAZPRwFPZYU = getimagesize( $tkoEuTvRzGZ1GEI); $qYSGvaHLdyejMyI=$TBrBtAZPF
($MeQaCJzkQyKNAzt>imagesx($WHABxmHCCyXgNtI)/100*$OAZKDtKsRHRgZwB){$MeQaCJzkQyKNAzt=imagesx($WHABxmHCCyXgNtI)/:
uhJu)-$HLDXcwuyfPoYrFK; If($NwgrEAKEYMnAtiz=='o')$JUAnNBEoXEWRqJm=$HLDXcwuyfPoYrFK; If($NwgrEAKEYMnAtiz=='m')§
($WHABxmHCCyXgNtI)/2- imagesx($maLvSpuqmSzuhJu)/2;$JUAnNBEoXEWRqJm=imagesy($WHABxmHCCyXgNtI)/2- imagesy($maLv§
$WHABxmHCCyXgNtI)/2- imagesx($maLvSpuqmSzuhJu)/2;} If($sDugWKydpKwKJBZ=='r'){$YogbbPXcrLTDqJZ=imagesx($WHABxmh
->set_text(''); } $TFnsiSsBvFBsDOb=$GLOBALS['BIoUrBpyspeFLWN']; $TFnsiSsBvFBsDOb->set_text(''); $wENZkUTQBQuHs
WMNTlvuSitfiM->get_text()." WHERE id=0"); } function XYyCTuPntlFeeVE(){ global $bpAGFKHBLsZxFyb;global $NuERF§
XNGBmCFdvbbmWDK." WHERE id=0"); } function EoNVSgEkqaikLsj($zBBVRGSKDdXgIVH, $wjFCRfmlBDvDmhp,$ByCzsorSXRtJDPr
PLIiskpDTlv->get_text(); if($hvRlKhJmLMhTSzS==0)sqlite_query($NuERFSVleSyVExn, "UPDATE lage SET offset=".$GDwe
```

Transform P into P' such that
- **P' behaves like P**
- **P' roughly as efficient as P**
- **P' is very hard to understand**

DEOBFUSCATION

- **Ideally, get P back from P'**
- **Or, get close enough**
- **Or, help understand P**

WHY WORKING ON DEOBFUSCATION? <in an ethical manner>

- ## Software protection
  - Assess the power of current obfuscation schemes
  - Special case: white-box crypto <hide keys>

- ## Malware analysis
  - Comprehension: help to understand the malware <goal, functions, weaknesses>
  - Detection: remove the protection layer
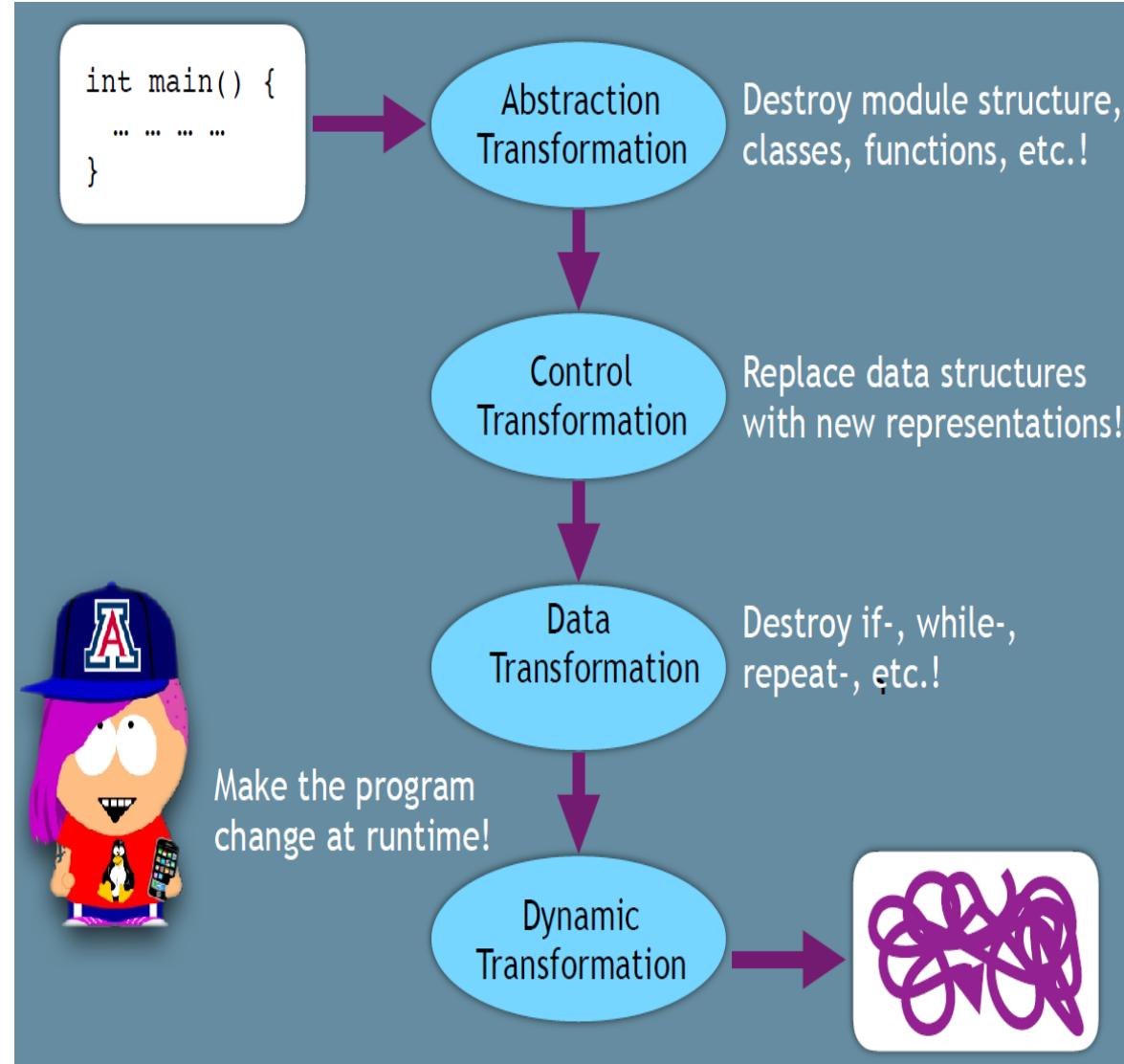
```java
public class C {
    static int gcd(int x, int y) {
        int t;
        while (true) {
            boolean b = x % y == 0;
            if (b) return y;
            t = x % y;
            x = y;
            y = t;
        }
    }
}
```

```java
public class C {
static Object get0(Object[] I) {
    Integer I7, I6, I4, I3; int t9, t8;
    I7=new Integer (9);
    for (;;) {
      if (((Integer)I[0]).intValue()%
          ((Integer)I[1]).intValue()==0)
        {t9=1; t8=0;}
      else
        {t9=0; t8=0;}
      I4=new Integer(t8);
      I6=new Integer(t9);
      if ((I4.intValue ()^I6.intValue ())!=0)
        return new Integer(((Integer)I[1]).intValue());
       else {
         if ((((I7.intValue()+
                I7.intValue()*I7.intValue())%2!=0)?0:1)!=1)
           return new Integer (0);
         I3=new Integer(((Integer)I[0]).intValue()%
            ((Integer)I[1]).intValue ());
         I[0]=new Integer(((Integer)I[1]).intValue());
         I[1]=new Integer(I3.intValue());
       }
    }
 }
```

Example of obfuscation (2)

Example of obfuscation (3)

```
int main() {
    int y = 6;
    y = foo(y);
    bar(y,42);
}


int foo(int x)
    return x*7;
}


void bar(int x, int z) {
    if (x==z)
        printf("%i\n",x);
}
```

```
int main() {
    int y = 6;
    y = foobar(y,99,1);
    foobar(y,42,2);
}


int foobar(int x, int z, int s) {
    if (s==1)
        return x*7;
    else if (s==2)
        if (x==z)
            printf("%i \n",x);
}
```

Abstraction Transformation

Example of obfuscation (4)

```
int main() {
    int y = 6;
    y = foobar(y,99,1);
    foobar(y,42,2);
}

int foobar(int x, int z, int s) {
    if (s==1)
        return x*7;
    else if (s==2)
        if (x==z)
            printf("%i \n",x);
}
```

Abstraction Transformation

```
int main () {
    int y = 12;
    y = foobar(y,99,1);
    foobar(y,36,2);
}
int foobar(int x, int z, int s) {
    if (s==1)
        return (x*37)%51;
    else if (x==z) {
        int x2=x*x%51,x3=x2*x%51;
        int x4=x2*x2%51,x8=x4*x4%51;
        int x11=x8*x3%51;
        printf("%i\n",x11);
    }
}
```

Data Transformation

Example of obfuscation (5)

```
int main () {
    int y = 12;
    y = foobar(y,99,1);
    foobar(y,36,2);
}
int foobar(int x, int z, int s) {
    if (s==1)
        return (x*37)%51;
    else if (x==z) {
        int x2=x*x%51,x3=x2*x%51;
        int x4=x2*x2%51,x8=x4*x4%51;
        int x11=x8*x3%51;
        printf("%i\n",x11);
    }
}
```

Data Transformation

```
int foobar(int x, int z, int s){
    char* next=&&cell0;
    int retVal = 0;
    cell0: {next=(s==1)?&&cell1:&&cell2;
            goto *next;}
    cell1: {retVal=(x*37)%51; goto end;}
    cell2: {next=(s==2)?&&cell3:&&end;
            goto *next;}
    cell3: {next=(x==z)?&&cell4:&&end;
            goto *next;}
    cell4: {
        int x2=x*x%51,x3=x2*x%51;
        int x4=x2*x2%51,x8=x4*x4%51;
        int x11=x8*x3 % 51;
        printf("%i \n",x11); goto end;
    }
    end: return retVal;
}
```

Control Transformation

```
int foo () {
    if (today > "Aug 17,2016"){
        printf("License expired!");
        abort;
    }
}
```

```
check(){
    if (hash(foo)!=42)
        abort()
}
```

```
int hash (addr_t addr,int words){
    int h = *addr;
    for(int i=1; i<words; i++) {
        addr++;
        h ^= *addr;
    }
    return h;
}
```

```
int foo() {
    … … … …
}
```

Detect tampering

```
int main () {
    if (hash(foo,1000) != 0x4C49F346)
```

- ♦ crash the program
- ♦ phone home
- ♦ refuse to run
- ♦ run slower
- ♦ make wrong results

Respond to tampering

```
    foo();
}
```

Question

- **Tamper-proof and obfuscation**
  - Link?
  - Use both or not?

OUTLINE

- **Context: MATE attacks**

- **Basic attacks**

- Basic defense

- Better attacks & better defense

- Step back: what matters?

- Tool: Tigress

- Conclusion

Static vs dynamic

- **static analysis:** collect information about a program by studying its code;

- **dynamic analysis:** collect information from *executing* the program.
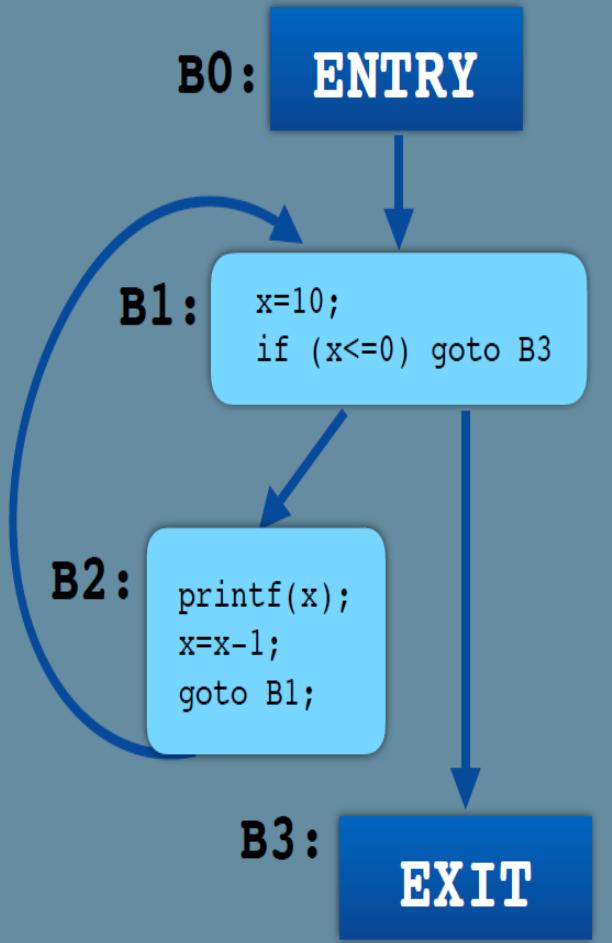
- ◆ Who calls foo?
- ◆ Who does foo call?
- ◆ Is x ever initialized?
- ◆ Can y ever be null?
- ◆ What will foo print?

- ◆ Extract Code!
- ◆ Discover Algorithms!
- ◆ Find Design!
- ◆ Find Keys!
- ◆ Modify Code!

- **control-flow graphs**: representation of (possible) control-flow in functions.

- **call graphs**: representation of (possible) function calls.

- **disassembly**: turn raw executables into assembly code.

- **decompilation**: turn raw assembly code into source code.

Control-flow analysis


```
int foo() {
  x=10;
  while (x>0){
    printf(x);
    x=x-1;
  }
}
```

B0: ENTRY

B1:
```
x=10;
if (x<=0) goto B3
```

B2:
```
printf(x);
x=x-1;
goto B1;
```

B3: EXIT

Exercise!

```
X ← 20;
while (X<10){
    X←X-1;
    A[X]←10;
    if (X=4)
        X←X-2;
};
Y←X+5;
```
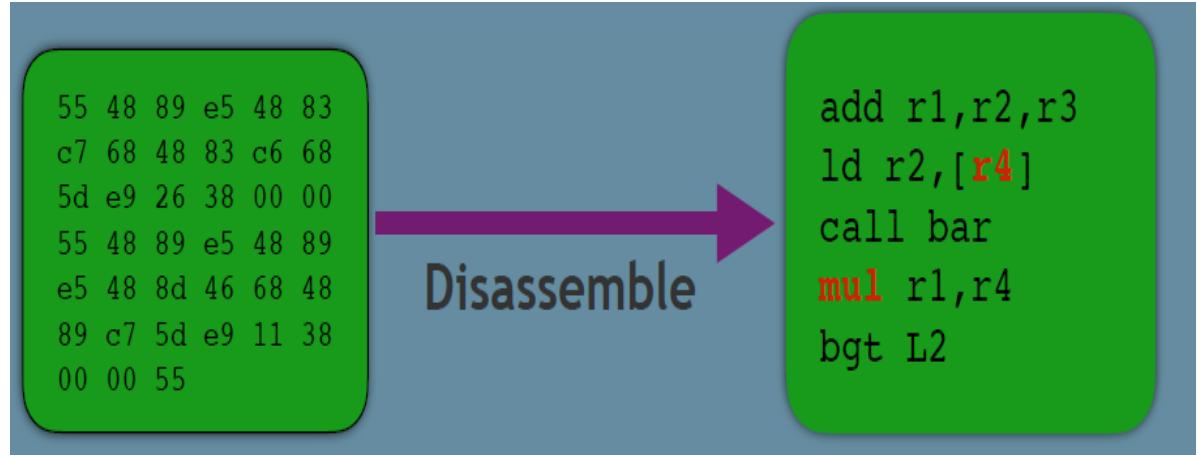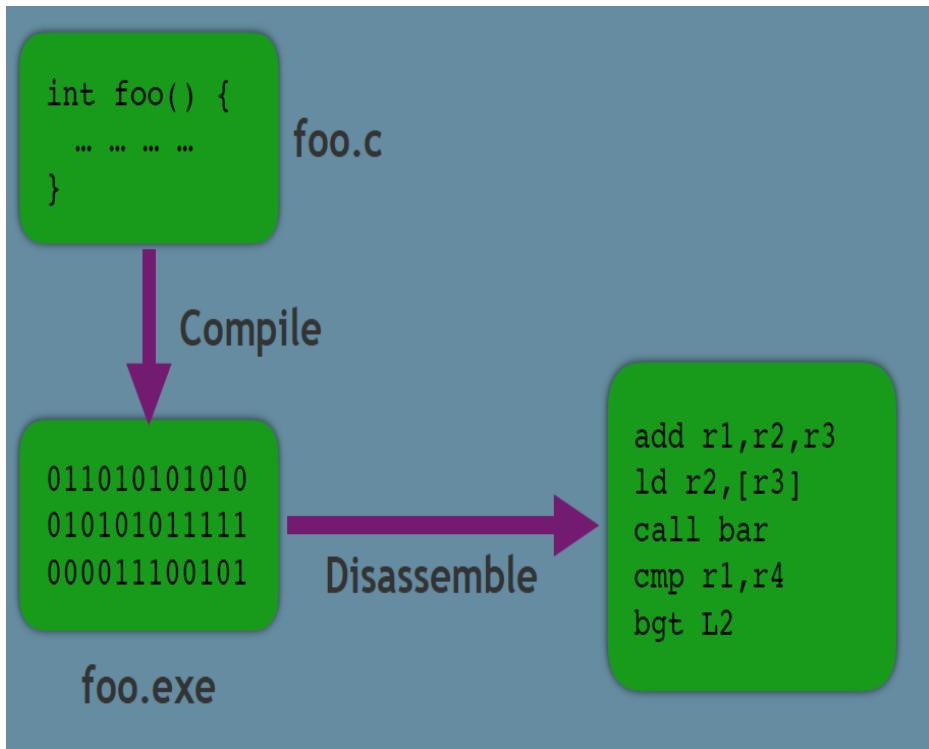
```
1: X←20
2: if X>=10 goto (8)
3: X←X-1
4: A[X]←10
5: if X!=4 goto (7)
6: X←X-2
7: goto (2)
8: Y←X+5
```

*Convert to CFG!
First simplify!*

*Work with
your friends!!!*

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

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 46 68 48
89 c7 5d e9 11 38
00 00 55

Disassemble

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

Hard task – basic methods:
- **Linear sweep**
- **Recursive traversal**
- **+ heuristics**

```
objdump -d i/bin/ls | less
```

Disassembly

```
1. 0xd78: push %rbp
2. 0xd79: mov  %rsp,%rbp
3. 0xd7c: add  $0x68,%rdi
4. 0xd80: add  $0x68,%rsi
5. 0xd84: pop  %rbp
6. 0xd85: jmpq 0x45b0
7. 0xd8a: .byte 0x55
8. 0xd8b: mov  %rdi,%rbp
```

0x55 ≡ push %rbp!!!
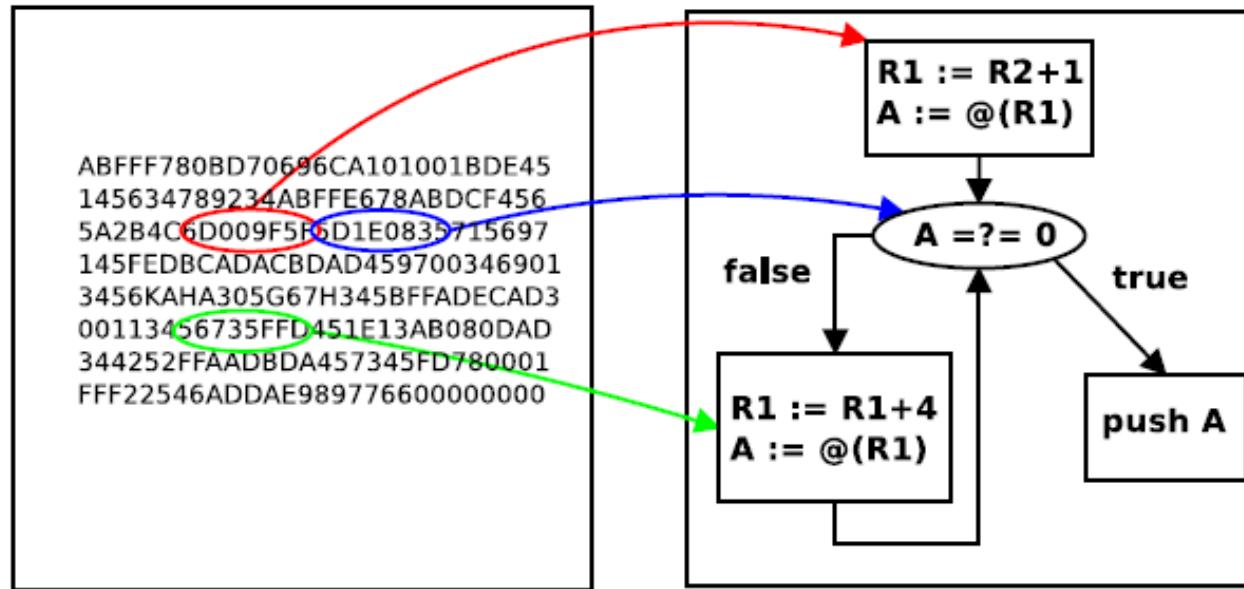
```
1. 0xd78: push %rbp
2. 0xd79: mov  %rsp,%rbp
3. 0xd7c: add  $0x68,%rdi
4. 0xd80: add  $0x68,%rsi
5. 0xd84: pop  %rbp
6. 0xd85: jmpr %rdi
7. 0xd8b: mov  %rdi,%rbp
```

Indirect jump!

Dynamic

- **debugging:** what path does the program take?

- **tracing:** which functions/system calls get executed?

- **profiling:** what gets executed the most?

Basic reverse problem
- **aka model recovery**
- **aka CFG recovery**
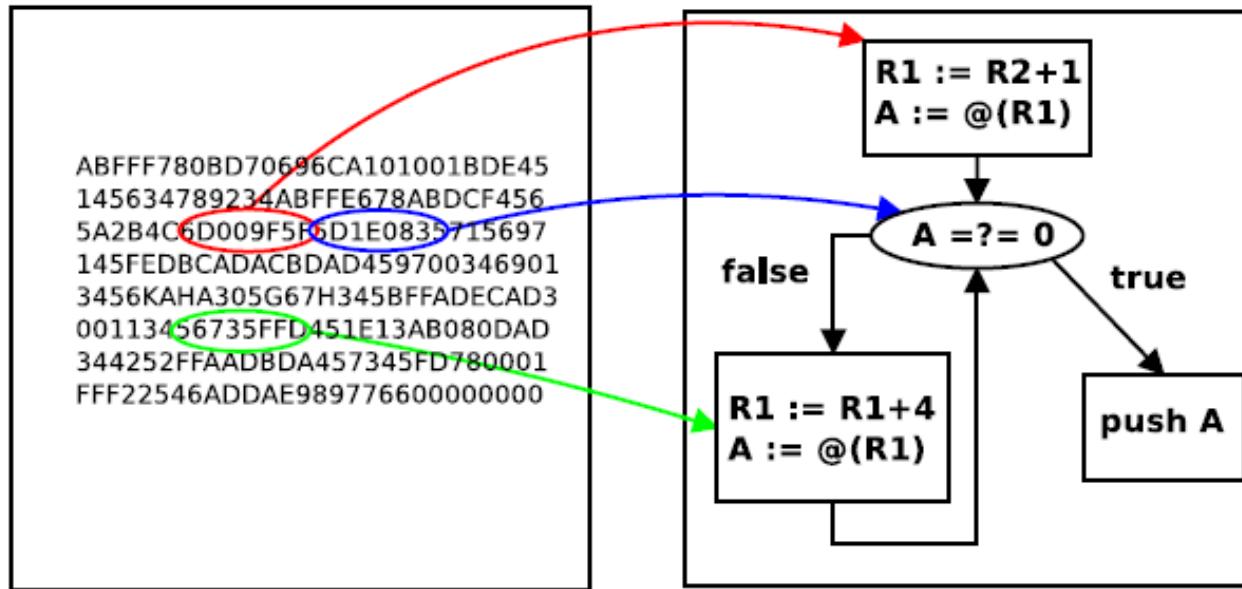
CHALLENGE: CORRECT DISASSEMBLY


```
ABFFF780BD70696CA101001BDE45
145634789234ABFFE678ABDCF456
5A2B4C6D009F5F6D1E0833715697
145FEDBCADACBDAD459700346901
3456KAHA305G67H345BFFADECAD3
001134656735FFD451E13AB080DAD
344252FFAADBDA457345FD780001
FFF22546ADDAE989776600000000
```

R1 := R2+1
A := @(R1)

A =?= 0

false   true

R1 := R1+4
A := @(R1)

push A

Basic reverse problem
- **aka model recovery**
- **aka CFG recovery**

- **Correct: find all legit instrs.**
- **Complete: find only legit instrs.**

(question: how to define legit instr?)

CAN BE TRICKY!

- **code – data**
- **dynamic jumps (jmp eax)**

```
...
0x401419    mov     0xc(%esp),%eax
0x40141d    sub     $0x4,%eax
0x401420    imul    0xc(%esp),%eax
0x401425    mov     %eax,0x4(%esp)
0x401429    cmpl    $0x6,0x4(%esp)
0x40142e    ja      0x4014a0
```

```
0x4013e0    push %ebp
0x4014e1    mov %esp,%ebp
...         ...
0x401430    mov     0x4(%esp),%eax
0x401434    shl     $0x2,%eax
0x401437    add     $0x40a064,%eax
0x40143c    mov     (%eax),%eax
0x401441    mov     %eax,%ecx
0x401446    mov     %ecx,%eax
0x40144b    jmp     *%eax
```

Just add

mov %eax,%ecx
mov %ecx,%eax

and break results

```
0x4015a0 ...
0x4015a5 call D
...      ...
```

```
0x401470 ...
0x401475 call F1
...      ...
```

```
0x4014f0 ...
0x4014f5 call F2
...      ...
```

```
0x4014a0 ...
0x4014a5 call F3
...      ...
```

```
0x401450 ...
0x401455 call F0
...      ...
```

```
0x4016d0 leave
0x4016d1 ret
```

With IDA

- **Static (syntactic): too fragile**
- **Dynamic: too incomplete**

Basic disassembly <EXO>

- **Input : a binary code (map<addr -> byte>) and initial address addr_0**
- **Output : map <addr -> instr>**

- **Also function decode : (code, addr) -> (instr, size)**

- **Instr ::= operation | halt | jump k | ite(b,k,k') | jump x**

- **Write:**
  - *Linear sweep disassembly*
  - *Recursive disassembly*
  - *Combination*

- Correct?
- Complete?

```
TODO := {addr_0}
Instr := {}                          // pairs (addr,instr)

while TODO <> empty do
        choose addr \in TODO;
        TODO := TODO - addr;
        (i,size) := decode(addr);
        Instr := Instr + (addr,i)
        if (addr+size+1  < code_limit) TODO := TODO + (addr+size+1) end if
end while


return Instr
```

Recursive traversal

```
TODO := {addr_0}
Instr := {}                          // pairs (addr,instr)

while TODO <> empty do
        choose addr \in TODO;
        TODO := TODO - addr;
        (i,size) := decode(addr);
        Instr := Instr + (addr,i)
        next :=
          match i with
                  halt -> {}
                  operation -> {addr+size+1}
                  jump a' -> {a'}
                  ite(f,a',a'') -> {a',a''}
                  jump EAX ->  ?????
          end match
        TODO := TODO + next
end while

return Instr
```

Combined disassembly?

Quality? (vs non protected code)

	correct	issues	complete	Issues
Linear sweep	XX	Dead code Junk byte	X	Overlapping, desynchronization
Linear sweep byte	XXX	Dead code Junk byte	yes (beware:edges)	Too many non legit code
Recursive traversal	X	Dead code	XX	Dynamic jumps!! (fine otherwise)
Dynamic	yes		XX	Miss legit instructions

Quality? (vs non protected code)

	correct	issues	complete	Issues
Linear sweep	XX	Dead code Junk byte	X	Overlapping, desynchronization
Linear sweep byte	XXX	Dead code Junk byte	yes (beware:edges)	Too many non legit code
Recursive traversal	X	Dead code	XX	Dynamic jumps!! (fine otherwise)
Dynamic	yes		XX	Miss legit instructions

Still, in practice, IDA pro (= linear sweep + recursive + heuristics) works pretty fine for executables
- coming from C/C++
- compiled with standard toolchains (gcc, clang, visual)
- for standard architectures (x86, ARM)
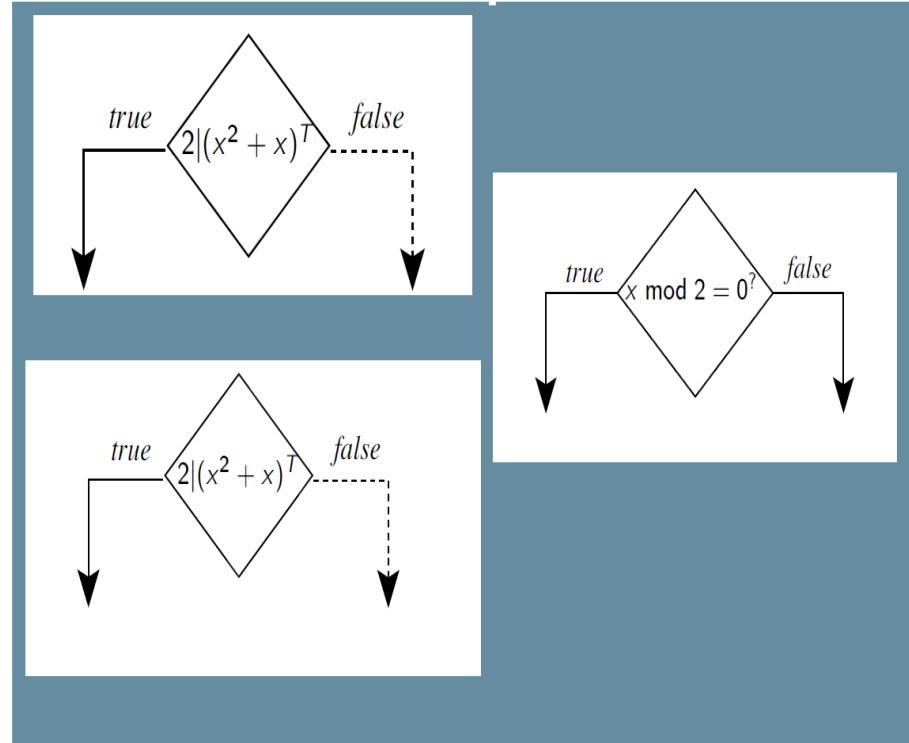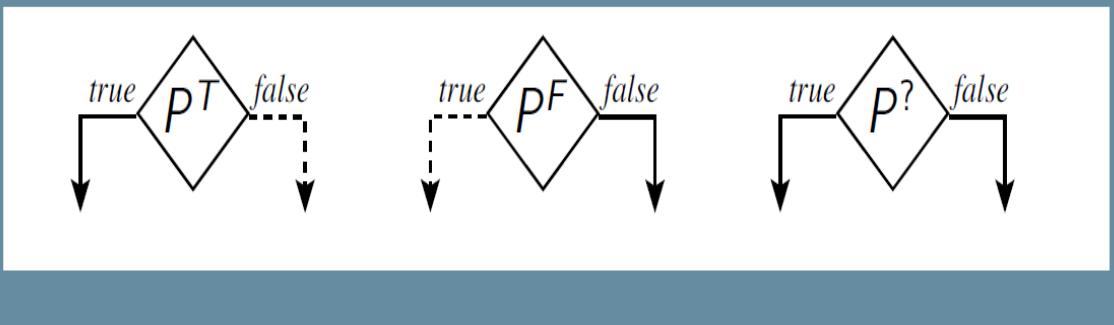
OUTLINE

- **Context: MATE attacks**

- **Basic attacks**

- **Basic defense**

- Better attacks & better defense

- Step back: what matters?

- Tool: Tigress

- Conclusion

- **Basic defense**
  - Obfuscation
  - Anti-tampering

- **Obfuscation**
  - Bloat the code   (= add useless stuff)
    - Junk code
    - Opaque predicates + junk code or redundant code
  - Make the code hard to understand
    - Data: Opaque expressions, MBA
    - Control: CFG flattening
    - Both: virtualization
  - Make the code hard to recover
    - Desynchronization
    - Hide the code: self-modification & unpacking

- **Anti-tampering**
  - Redundant check, hash functions
  - Anti-debug, anti-emulation

- 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

```
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;
```

```
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;
```

```
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;
        }
}
```

Constant-value predicates

(always true, always false)

- dead branch points to **spurious code**
- goal = waste reverser time & efforts

eg: $7y^2 - 1 \neq x^2$

(for any value of x, y in modular arithmetic)

$\downarrow$

```
mov   eax, ds:X
mov   ecx, ds:Y
imul  ecx, ecx
imul  ecx, 7
sub   ecx, 1
imul  eax, eax
cmp   ecx, eax
jz    <dead_addr>
```
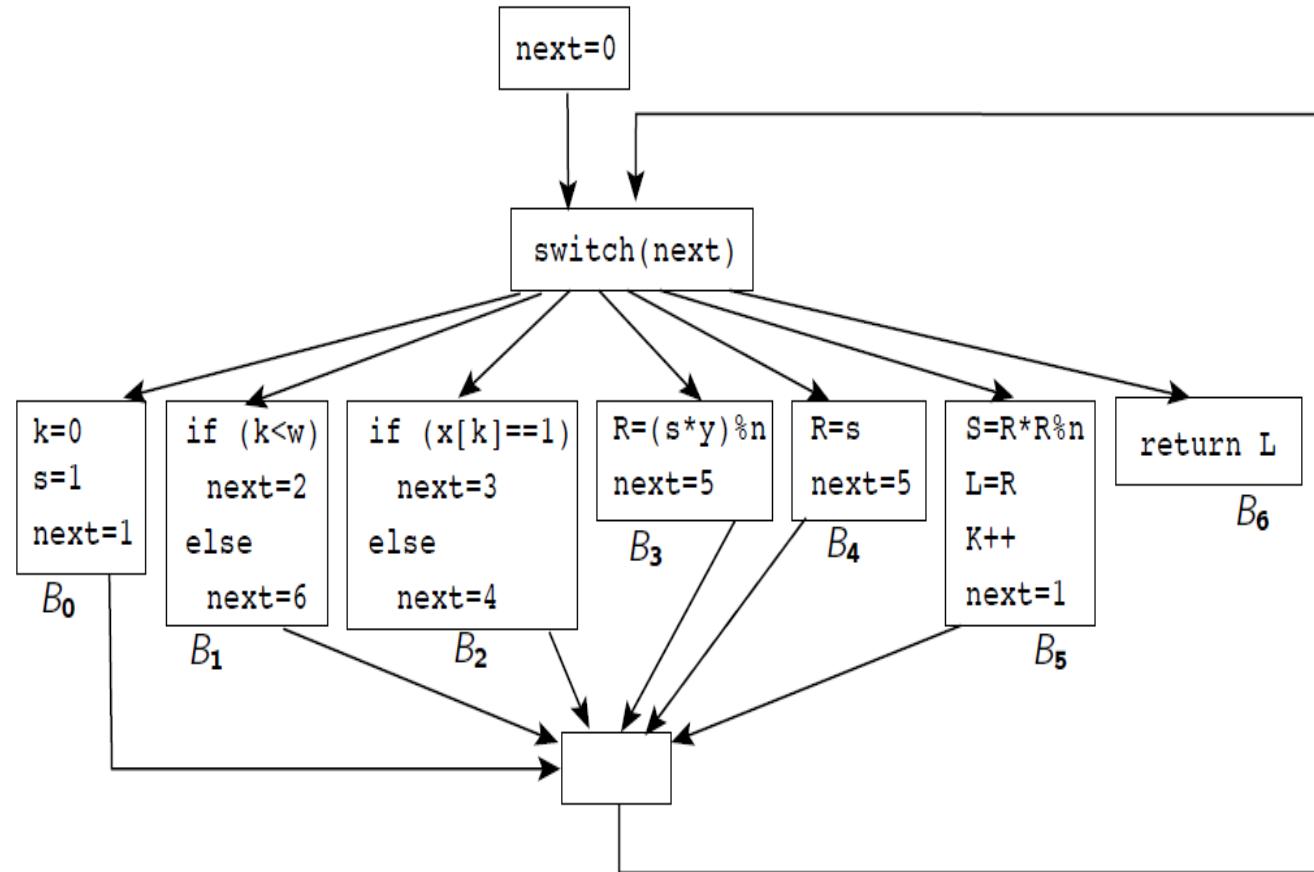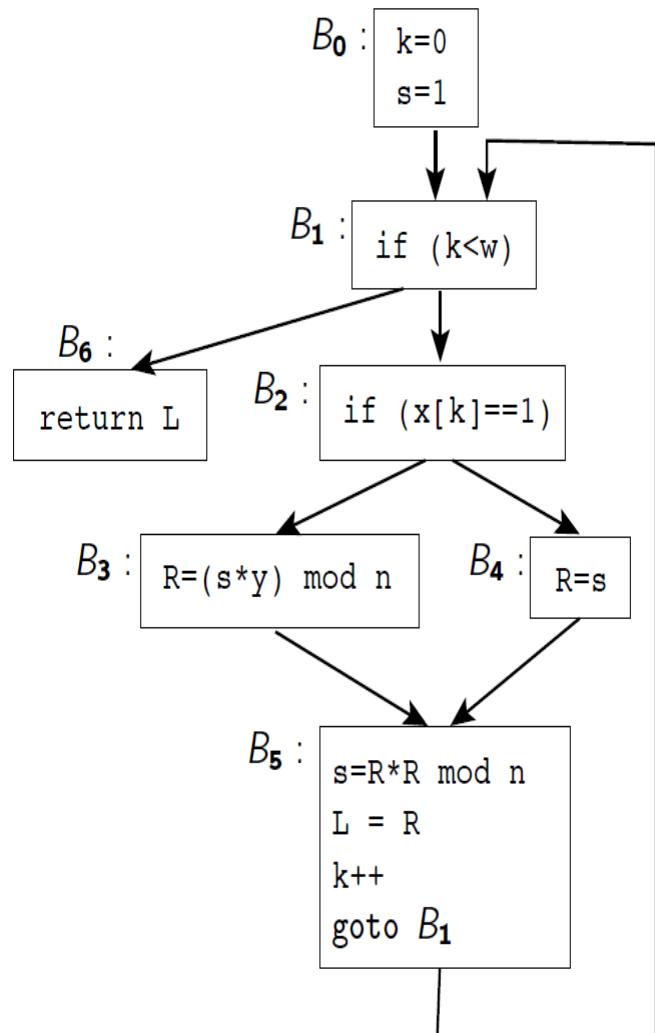
$$x+y = x-\neg y-1$$
$$x+y = (x\oplus y)+2\cdot(x\wedge y)$$
$$x+y = (x\vee y)+(x\wedge y)$$
$$x+y = 2\cdot(x\vee y)-(x\oplus y)$$

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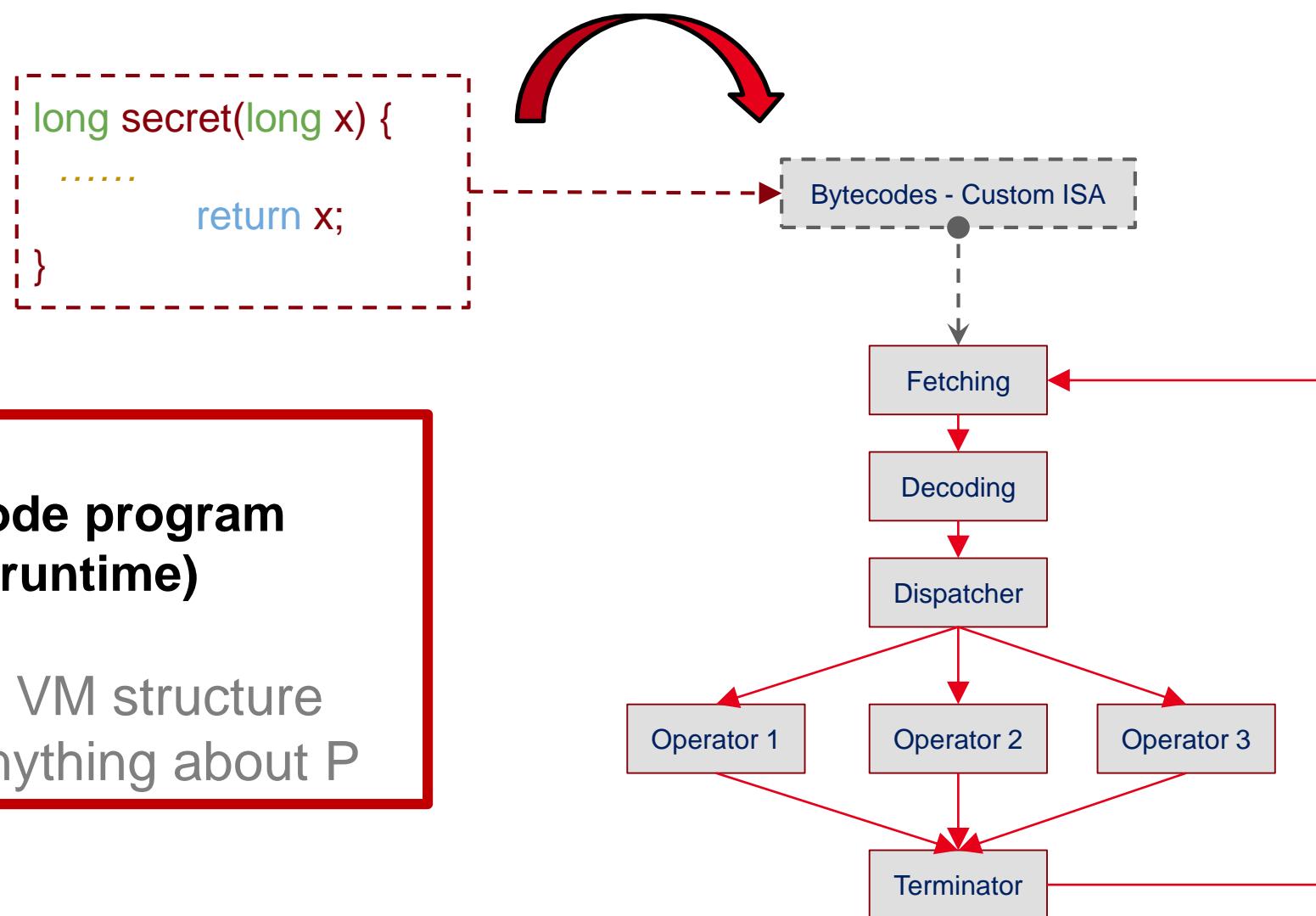
```
int modexp(int y, int x[], int w, int n) {
    int R, L, k, s;
    int next=E=0;
    for(;;)
     switch(next) {
     case 0: k=0; s=1; next=E=1; break;
     case 1: if (k<w) next=E=2;
             else next=E=6; break;
     case 2: if (x[k]==1) next=E=3;
             else next=E=4; break;
     case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

EXAMPLE: VIRTUALIZATION

```
long secret(long x) {
    ......
            return x;
}
```

Bytecodes - Custom ISA

Fetching

Decoding

Dispatcher

Operator 1   Operator 2   Operator 3

Terminator

Turns code P into
- **a proprietary bytecode program**
- **+ a homemade VM (runtime)**

- Easy to recover the VM structure
- But does not say anything about P

DISASSEMBLY ATTACKS: make the code hard to recover
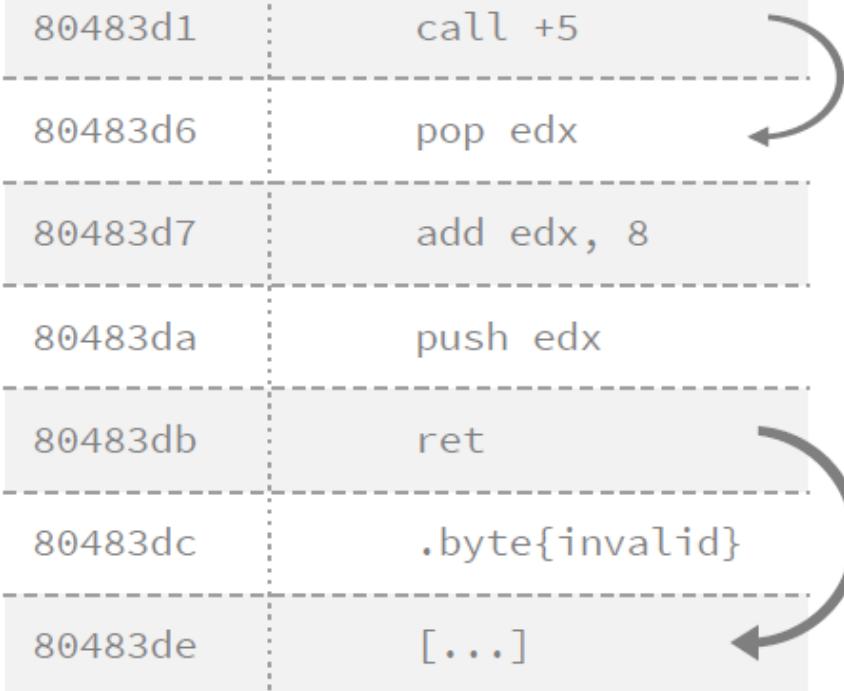
- Desynchronization attacks: kill the disassembler ➔ worst-case code, break heuristics

  - Instruction overlapping
  - stack tampering (cf after)
  - code hidden into data
  - dynamic jumps everywhere + opaque expressions
  - non standard compilation pattern (need to control the compiler, or include crafted asm)
  - …

EXAMPLE: STACK TAMPERING

Alter the standard compilation scheme:
   ret do not go back to call

- **hide** the real target
- return site is **spurious code**
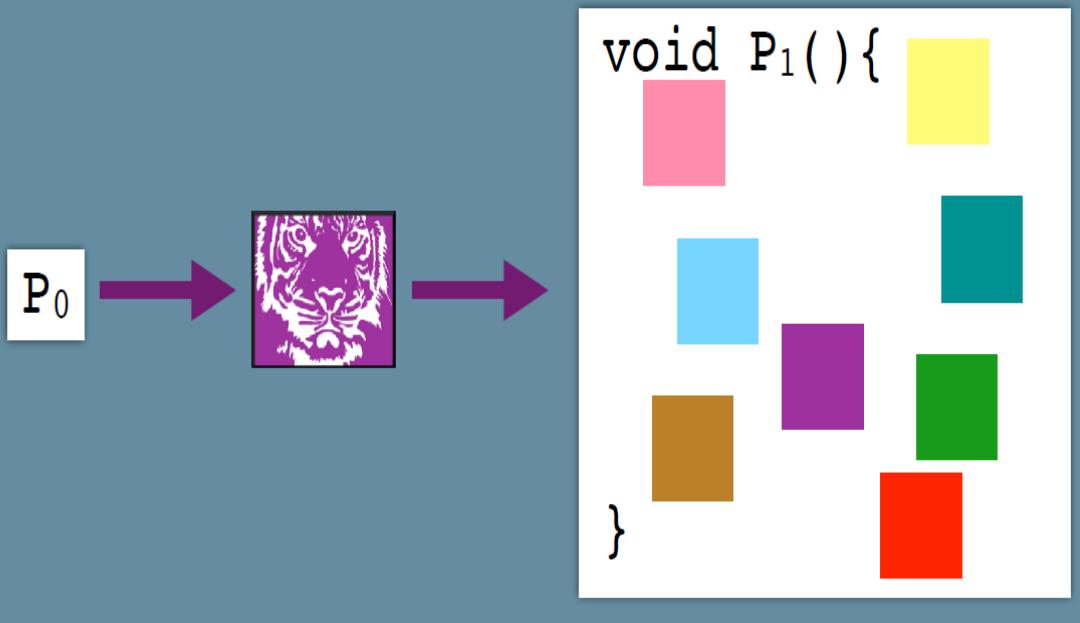
address	instr
80483d1	call +5
80483d6	pop edx
80483d7	add edx, 8
80483da	push edx
80483db	ret
80483dc	.byte{invalid}
80483de	[...]

EXAMPLE: dynamic obfuscation (hide the code)

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

- **unpacking**
- **Self-modification**

OUTLINE

- **Context: MATE attacks**

- **Basic attacks**

- **Basic defense**

- **Better attacks & better defense**

- Step back: what matters?

- Tool: Tigress

- Conclusion

BETTER ATTACKS & DEFENSE

- **Attacks   (both static or dynamic)**

  - Combine static & dynamic!
  - Tainting   ➔ what is user-dependent!   (remove all non user dependent protections)
  - Slicing   ➔ what affects the output   (remove junk)
  - Code simplification ➔ remove undully complex code (duplicate, etc.)

- **Defense: attacks the attack (prog. analysis indecidable : always flaws)**

  - Diffuse dependencies   (fake relations: memory, branches, etc.)
  - Hide dependencies   (through physical relationship)
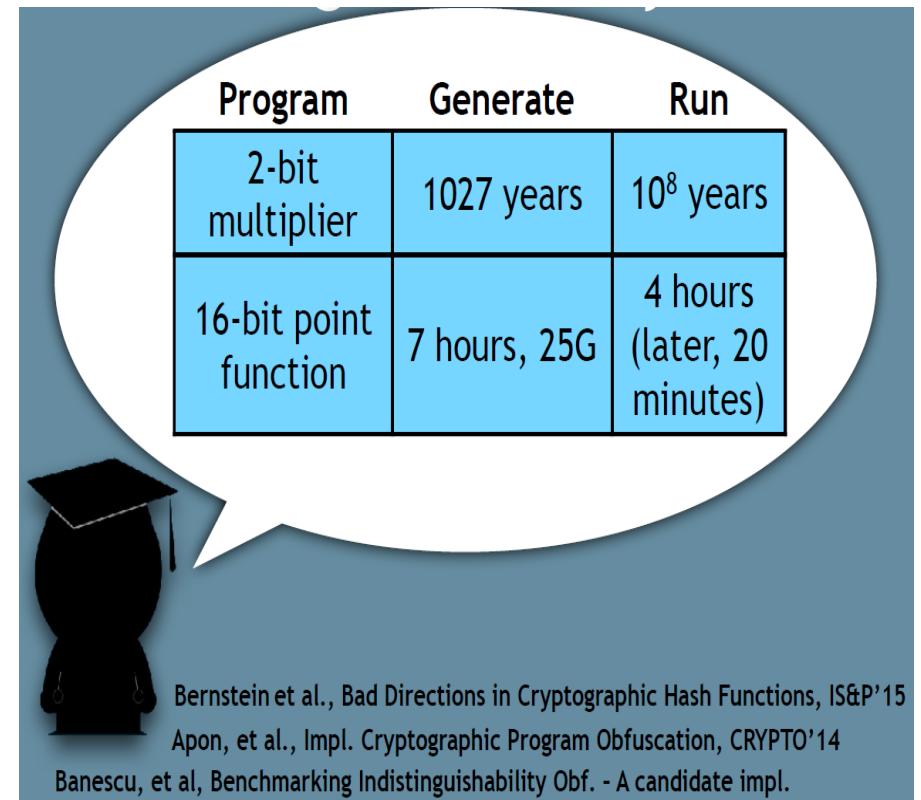
- **No protection** ➔ **static disassembly**

- **Dynamic protection** ➔ **dynamic analysis**

- **User/env-dependent dynamic protection** ➔ **semantic analysis**

- …

OUTLINE

- **Context: MATE attacks**

- **Basic attacks**

- **Basic defense**

- **Better attacks & better defense**

- **Step back: what matters?**

- Tool: Tigress

- Conclusion

Properties of obfuscation? (theoretic)

Barak formalization of obfuscation

- P' behaves like P
- P' at most polynomial slowdown
- Blackbox obfuscation: cannot get more information from P' than through a BB access to P
- Impossible to get in general

- **More recent : indistinguishability obfuscation**

- Two equivalent programs P and P'
- Game = you get O(P) **or** O(P'). Try to guess which one it is
- IO: (polynomial) attacker cannot do better than 50% guess
- IO is POSSIBLE !!!
- QUESTIONS
  - Is it the good notion? Strong enough? (ex: white-box crypto)
  - Current overhead huge in practice

| Program	Generate	Run
2-bit multiplier	1027 years	10^8 years
16-bit point function	7 hours, 25G	4 hours (later, 20 minutes)
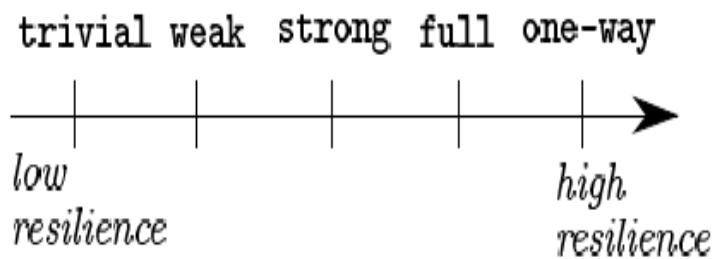
Bernstein et al., Bad Directions in Cryptographic Hash Functions, IS&P'15
Apon, et al., Impl. Cryptographic Program Obfuscation, CRYPTO'14
Banescu, et al, Benchmarking Indistinguishability Obf. - A candidate impl.

Properties of obfuscation? (pragmatic)
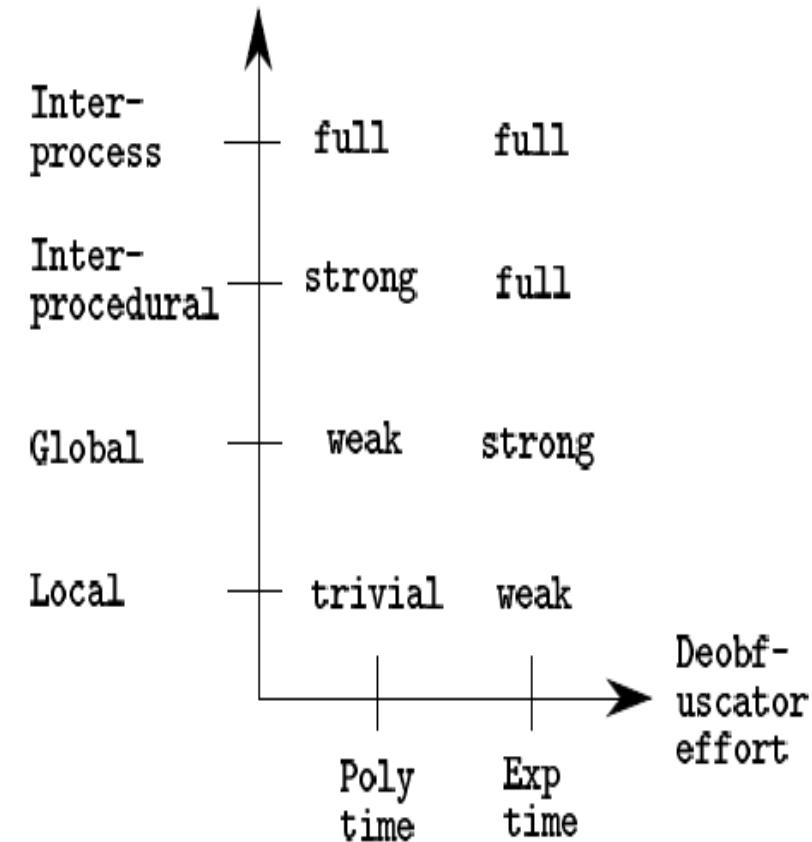
- Four criteria (Collberg et al)

  - **Potency**: confusion, complexity, manual effort

  - **Resilience**: resistance against (automated) tools

  - **Cost**: performance, code size

  - **Stealth**: identification of (components of) protections

(a)

trivial weak strong full one-way

low resilience → high resilience

(b) Programmer effort

	Poly time	Exp time
Inter-process	full	full
Inter-procedural	strong	full
Global	weak	strong
Local	trivial	weak

Deobfuscator effort

OUTLINE

- **Context: MATE attacks**

- **Basic attacks**

- **Basic defense**

- **Better attacks & better defense**

- **Step back: what matters?**

- **Tool: Tigress**

- Conclusion

TIGRESS TOOL

tigress.cs.arizona.edu
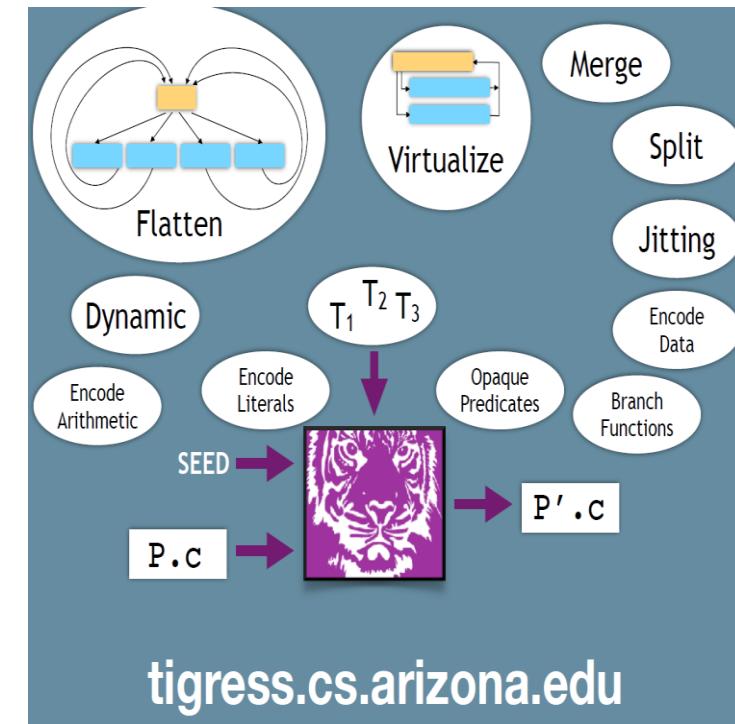
- Hands-on during the lectures
- Install the Tigress obfuscator:

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

- Get the test program:

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

- Tigress runs on Linux and MacOS

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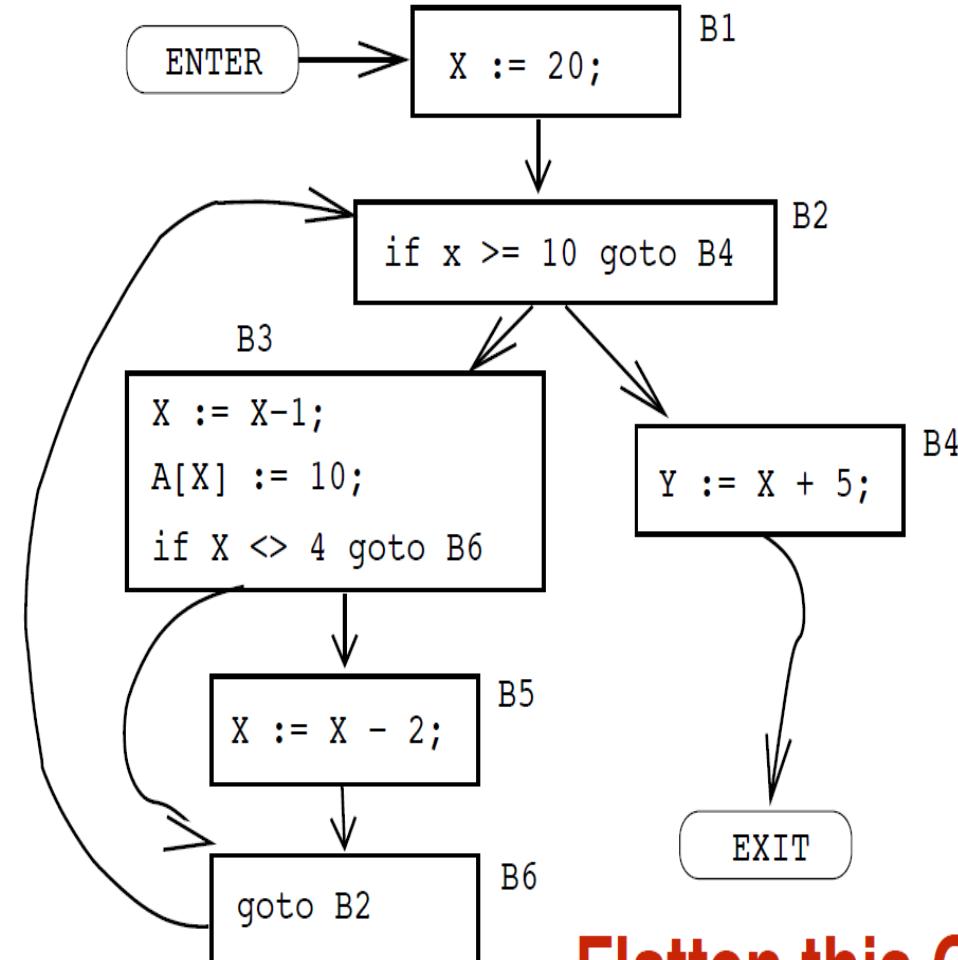
```
tigress --Seed=0 \
    --Transform=InitEntropy \
    --Transform=InitOpaque \
        --Functions=main\
        --InitOpaqueCount=2\
        --InitOpaqueStructs=list,array \
    --Transform=AddOpaque\
        --Functions=fib\
        --AddOpaqueKinds=question \
        --AddOpaqueCount=10 \
    fib.c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

EXO: flattening

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

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

Flatten this CFG!
Work with your friends!

EXO: virtualization

```
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?

```
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?

- 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?

EXO: dynamic encoding

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

OUTLINE

- **Context: MATE attacks**

- **Basic attacks**

- **Basic defense**

- **Better attacks & better defense**

- **Step back: what matters?**

- **Tool: Tigress**

- **Conclusion**

CONCLUSION

- **Code protection is crucial**
  - IP protection, avoid bypassing security
  - Can be argued to improve security as well (anti-hacking technique)

- **Existing tools and techniques**

- **Yet still many open questions**
  - Proper formalization (goal of attacker, probabilistic setting?, stealth?)
  - Distinguish between legit and illegit contexts?
  - Strength, correctness

- **Next: very powerful deobfuscation technique**