Code Obfuscation

Code Obfuscation and Diversity

- ✓Code Obfuscation was first introduced by Fred Cohen for the purpose of program diversity in order to defend against automatic attacks on operating systems
- ✓Without physical protection it is unlikely to fully protect a system

"Any protection scheme other than a physical one depends on the operation of a finite state machine, and ultimately, any finite state machine can be examined and modified at will, given enough time and effort. The best we can ever do is delay the attack by increasing the complexity of making desired alterations (security through obscurity)"

Operating system protection through program evolution. Cohen J. Comp. and Sec.1993.

Code Obfuscation and Diversity

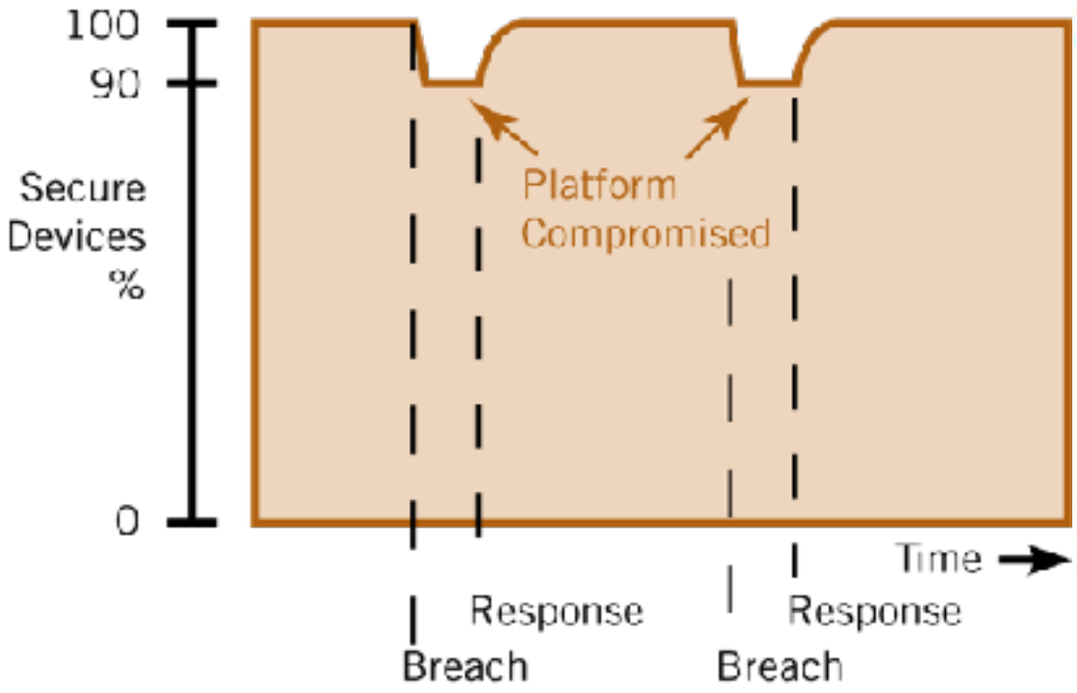
"The goal of security through obscurity is to make the difficulty of attack so great that in practice, it is not worth performing, even though it could eventually be successful. Successful attacks against obscurity defenses depend on the ability to guess some key piece of information.

The most obvious example is attacking and defending passwords: 1) make the size of the password space large, so that the potential number of guesses is enormous, 2) spread the probability density out so that there is relatively little advantage to searching the space selectively, 3) obscure the stored password information so that the attacker cannot simply read it in stored form."

Operating system protection through program evolution. Cohen J. Comp. and Sec.1993.

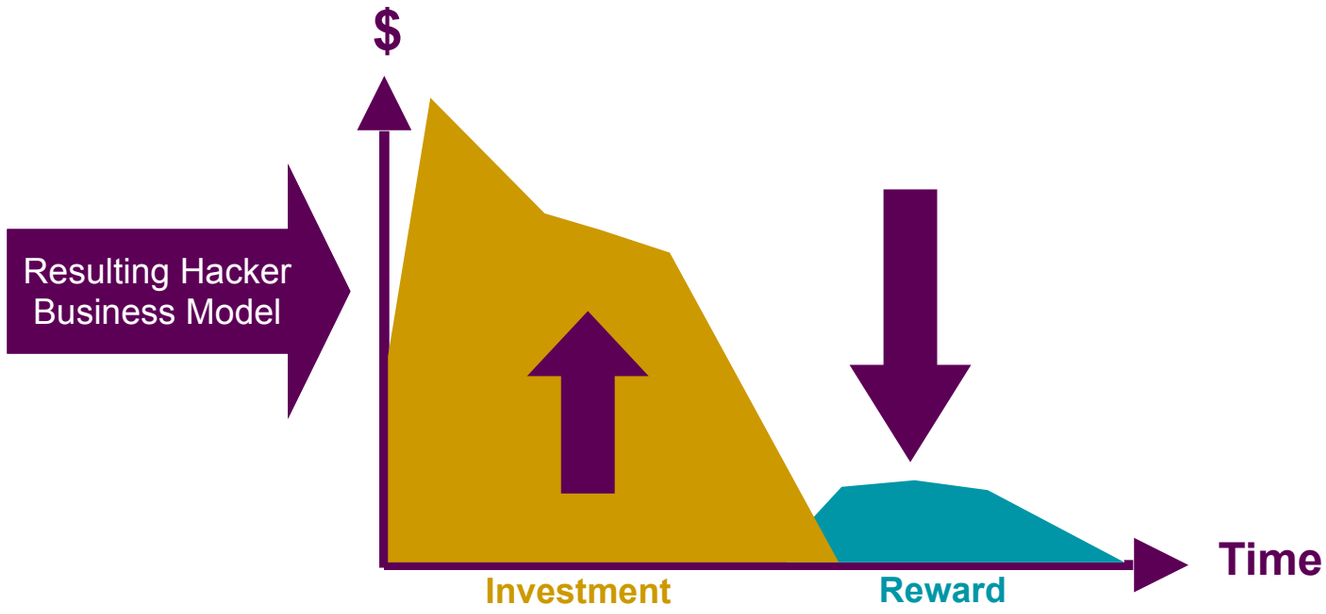
Mitigation

- **Strong attack response**
- **Reduces duration of attack**



Tamper resistance
Raises cost of attack

Diverse production
Reduces scope of attack



Software Diversity Benefits

Minimise scope of attack -- Prevent automated attacks

Provide rapid recovery in the event of an attack

Make the business unattractive to the hacker

Diversifying Transformations

✓ Instruction equivalence: replace equivalent op-code in assembly

✓ Equivalent instruction sequences, possible infinite number of evolutions. For example there are infinite ways to turn an arithmetic expression into a sequence of elementary instructions.

✓ Instruction reordering without altering program execution

✓ variable substitutions in high level languages, alter memory locations in compiled code

✓ Adding and removing jumps

✓ Garbage insertion

✓ …

Operating system protection through program evolution. Cohen J. Comp. and Sec.1993.

Obfuscating Transformations

DEFINITION 1 (OBFUSCATING TRANSFORMATION) Let $P \xrightarrow{\tau} P'$ be a transformation of a source program P into a target program P'.

$P \xrightarrow{\tau} P'$ is an *obfuscating transformation*, if P and P' have the same *observable behavior*. More precisely, in order for $P \xrightarrow{\tau} P'$ to be a legal obfuscating transformation the following conditions must hold:

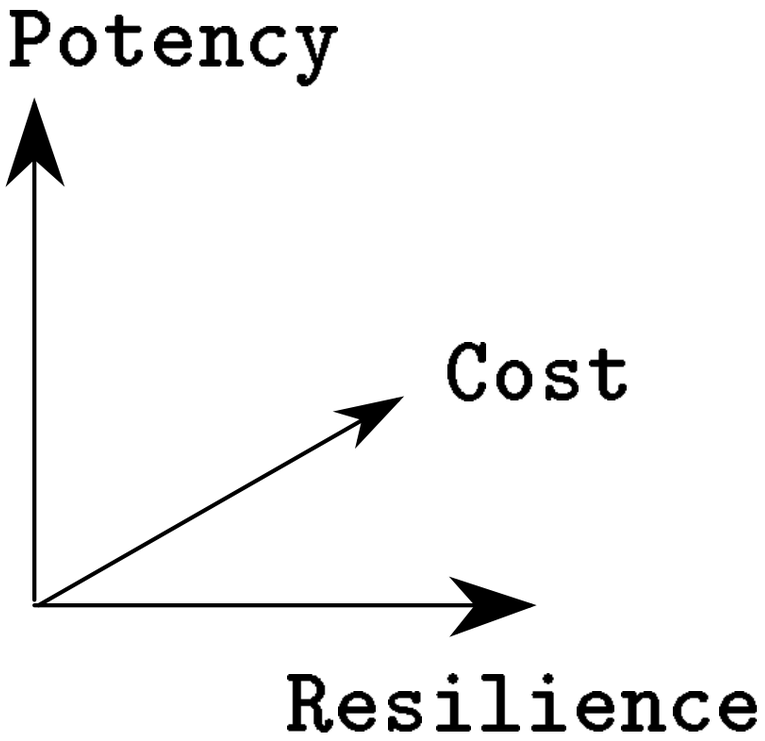
- If P fails to terminate or terminates with an error condition, then P' may or may not terminate.
- Otherwise, P' must terminate and produce the same output as P.

□

Behavior experimented by the user

Manufacturing cheap, resilient and stealthy opaque constructs. Collberg et al. POPL 1998

Evaluating Obfuscating Transformations



✓Potency: measures how much more difficult the obfuscated code is to understand than the original one

✓Resilience: measures how well a transformation holds up under attack from an automatic deobfuscator

✓Cost: measures the execution time/space penalty which a transformation incurs on an obfuscated application

Manufacturing cheap, resilient and stealthy opaque constructs. Collberg et al. POPL 1998

Potency

DEFINITION 2 (TRANSFORMATION POTENCY) Let $\mathcal{T}$ be a behavior-conserving transformation, such that $P \xrightarrow{\mathcal{T}} P'$ transforms a source program P into a target program P'. Let $E(P)$ be the complexity of P, as defined by one of the metrics[2] in Table 1.

$\mathcal{T}_{\text{pot}}(P)$, the *potency* of $\mathcal{T}$ with respect to a program P, is a measure of the extent to which $\mathcal{T}$ changes the complexity of P. It is defined as

$$\mathcal{T}_{\text{pot}}(P) \stackrel{\text{def}}{=} E(P')/E(P) - 1.$$

$\mathcal{T}$ is a *potent obfuscating transformation* if $\mathcal{T}_{\text{pot}}(P) > 0$.

□

Manufacturing cheap, resilient and stealthy opaque constructs. Collberg et al. POPL 1998

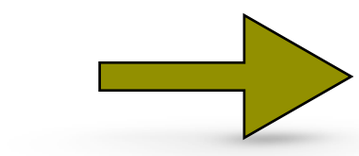
Software Complexity Metrics

Metric	Metric Name	Citation
μ_1	Program Length	Halstead [8]
	$E(P)$ increases with the number of operators and operands in P.	
μ_2	Cyclomatic Complexity	McCabe [20]
	$E(F)$ increases with the number of predicates in F.	
μ_3	Nesting Complexity	Harrison [9]
	$E(F)$ increases with the nesting level of conditionals in F.	
μ_4	Data Flow Complexity	Oviedo [23]
	$E(F)$ increases with the number of inter-basic block variable references in F.	
μ_5	Fan-in/out Complexity	Henry [10]
	$E(F)$ increases with the number of formal parameters to F, and with the number of global data structures read or updated by F.	
μ_6	Data Structure Complexity	Munson [21]
	$E(P)$ increases with the complexity of the static data structures declared in P. The complexity of a scalar variable is constant. The complexity of an array increases with the number of dimensions and with the complexity of the element type. The complexity of a record increases with the number and complexity of its fields.	
μ_7	OO Metric	Chidamber [3]
	$E(C)$ increases with (μ_7^{a}) the number of methods in C, (μ_7^{b}) the depth (distance from the root) of C in the inheritance tree, (μ_7^{c}) the number of direct subclasses of C, (μ_7^{d}) the number of other classes to which C is coupled[a], (μ_7^{e}) the number of methods that can be executed in response to a message sent to an object of C, (μ_7^{f}) the degree to which C's methods do not reference the same set of instance variables. Note: μ_7^{f} measures *cohesion*; i.e. how strongly related the elements of a module are.	

Manufacturing cheap, resilient and stealthy opaque constructs. Collberg et al. POPL 1998

Resilience

```
main() {
     S1;
     S2;
    }
```



```
main() {
     S1;
if (5==2) S1;
     S2;
if (1>2) S2;
     }
```

According to the software complexity measures that we have seen it seems easy to increase potency, for example by augmenting the number of if statements

These transformations are useless, since they can easily be undone

Manufacturing cheap, resilient and stealthy opaque constructs. Collberg et al. POPL 1998

Resilience

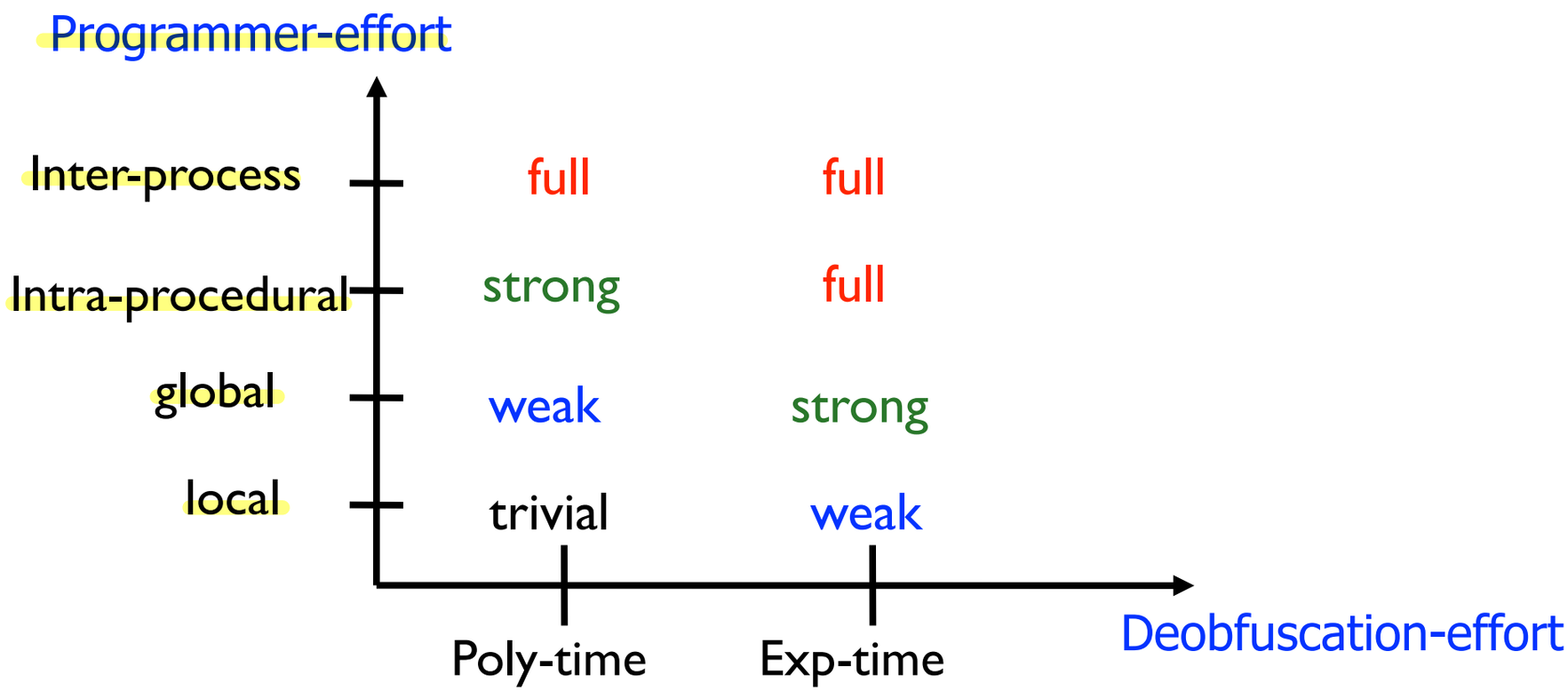
The resilience of a transformation τ is given by the combination of:

Programmer effort: the amount of time required to construct an automatic deobfuscator able to reduce the potency of τ

Deobfuscator effort: the execution time and space required by such an automatic deobfuscator to effectively reduce the potency of τ

Manufacturing cheap, resilient and stealthy opaque constructs. Collberg et al. POPL 1998

Resilience



trivial < weak < strong < full < one-way

Manufacturing cheap, resilient and stealthy opaque constructs. Collberg et al. POPL 1998

Resilience

DEFINITION 3 (TRANSFORMATION RESILIENCE) Let $\mathcal{T}$ be a behavior-conserving transformation, such that $P \xrightarrow{\mathcal{T}} P'$ transforms a source program P into a target program P'. $\mathcal{T}_{\text{res}}(P)$ is the *resilience* of $\mathcal{T}$ with respect to a program P.

$\mathcal{T}_{\text{res}}(P)$=*one-way* if information is removed from P such that P cannot be reconstructed from P'. Otherwise,

$$\mathcal{T}_{\text{Res}} \stackrel{\text{def}}{=} \texttt{Resilience}(\mathcal{T}_{\substack{\texttt{Deobfuscator} \\ \texttt{effort}}}, \mathcal{T}_{\substack{\texttt{Programmer} \\ \texttt{effort}}}),$$

Manufacturing cheap, resilient and stealthy opaque constructs. Collberg et al. POPL 1998

Cost

The cost of an obfuscating transformation is the execution time/ space penalty of the obfuscated program wrt the original one. Cost can be classified on a four-point scale as follows:

$$\mathcal{T}_{\text{cost}}(P) \stackrel{\text{def}}{=} \begin{cases} \textit{dear} & \text{if executing } P' \text{ requires } \textit{exponentially} \text{ more resources than } P. \\ \textit{costly} & \text{if executing } P' \text{ requires } \mathcal{O}(n^p),\ p > 1, \text{ more resources than } P. \\ \textit{cheap} & \text{if executing } P' \text{ requires } \mathcal{O}(n) \text{ more resources than } P. \\ \textit{free} & \text{if executing } P' \text{ requires } \mathcal{O}(1) \text{ more resources than } P. \end{cases}$$

Manufacturing cheap, resilient and stealthy opaque constructs. Collberg et al. POPL 1998

Stealth

- ✓ While a resilient transformation may not be susceptible to attacks by automatic deobfuscators, it may still be susceptible to *attacks by humans*

- ✓ In particular, if a transformation introduces new code that *differs widely* from what is in the original program it will be easy to spot for a reverse engineer. Such transformations are *unstealthy*.

if isPrime(8375374…7765456543) then

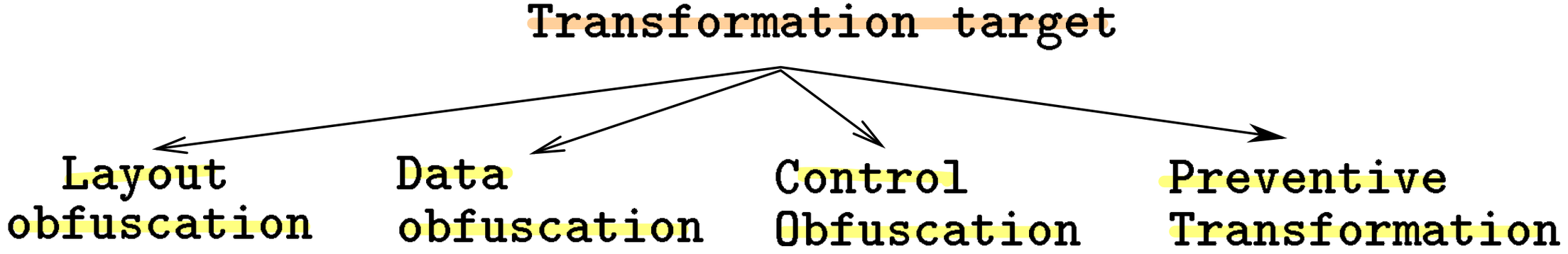
512 bit integer

- ✓ Stealth is a *context-sensitive notion.* Code may be stealthy in one program but extremely unstealthy in another one.

Manufacturing cheap, resilient and stealthy opaque constructs. Collberg et al. POPL 1998

A Taxonomy of Obfuscations

Obfuscating transformations are usually classified according to the kind of information they target



Manufacturing cheap, resilient and stealthy opaque constructs. Collberg et al. POPL 1998

Code Obfuscation

When developing and evaluating an obfuscating transformation we have to take into account

- ✓What is the target of protection
- ✓Who is the attacker
- ✓Define the obfuscation technique
- ✓Prove the quality of the proposed obfuscation

Code Obfuscation Techniques

Layout Obfuscation

Layout Obfuscation

- ✓ Changes or removes useful information from the code without affecting real instructions:
  - Comment stripping
  - Identifier renaming: new names can utilize different schemes like "a", "b", "c", or numbers, unprintable characters or invisible characters. And names can be overloaded as long they have different scopes.
- ✓ Used in many commercial obfuscators for.NET, iOS, Java and Android

Layout Obfuscation

- ✓ These are *one-way* transformations since the original formatting cannot be recovered
- ✓ It has *low potency* since there is little semantic content in formatting, more potent against human attack
- ✓ As regarding cost it is *free*
- ✓ It is very useful in obfuscating *authorship*…

Data Obfuscation

Data Obfuscation

- ✓ *Data obfuscation modifies the form in which data is stored in a program in order to*
- ✓ Prevent data form direct analysis
- ✓ Hide the content of data

Data Encoding

- ✓ To obfuscate a variable v in a program, you convert it from the representation the programmer initially chose for it to a *representation that is harder for the attacker to analyze*
- ✓ Any *value* that a variable v can assume must be represented in the new obfuscated representation. We need two functions:

encode E: T $\longrightarrow$ T ' *decode* D: T ' $\longrightarrow$ T

- ✓ The *operations* on variables need to be interpreted in the new representation

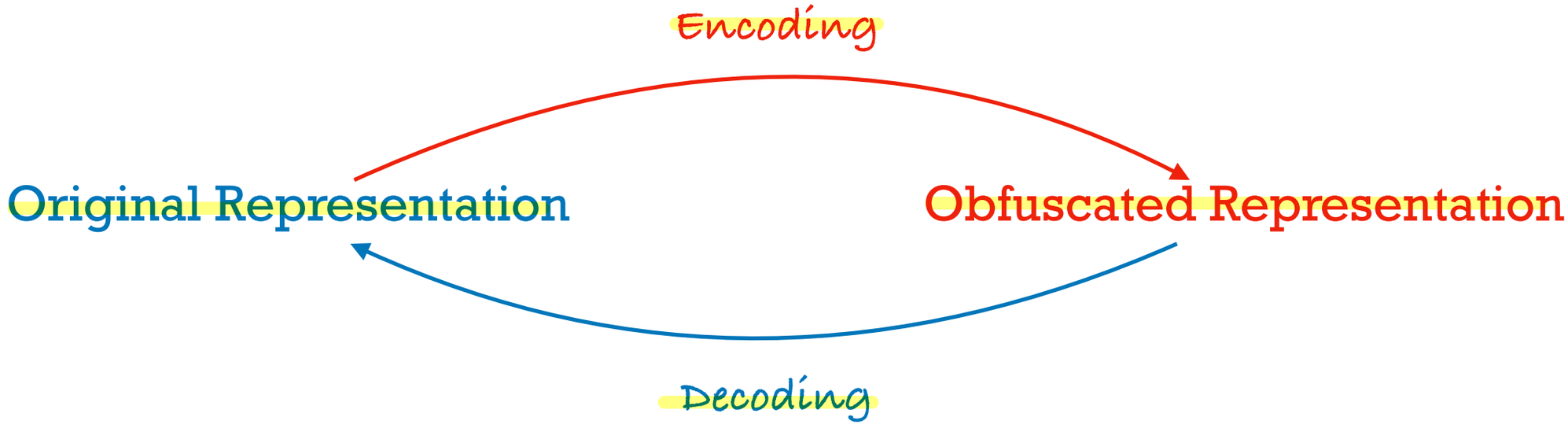
op: T x T $\longrightarrow$ T op': T ' x T ' $\longrightarrow$ T '

Encoding

Harder to analyse

Original Representation

Obfuscated Representation



XOR Masking

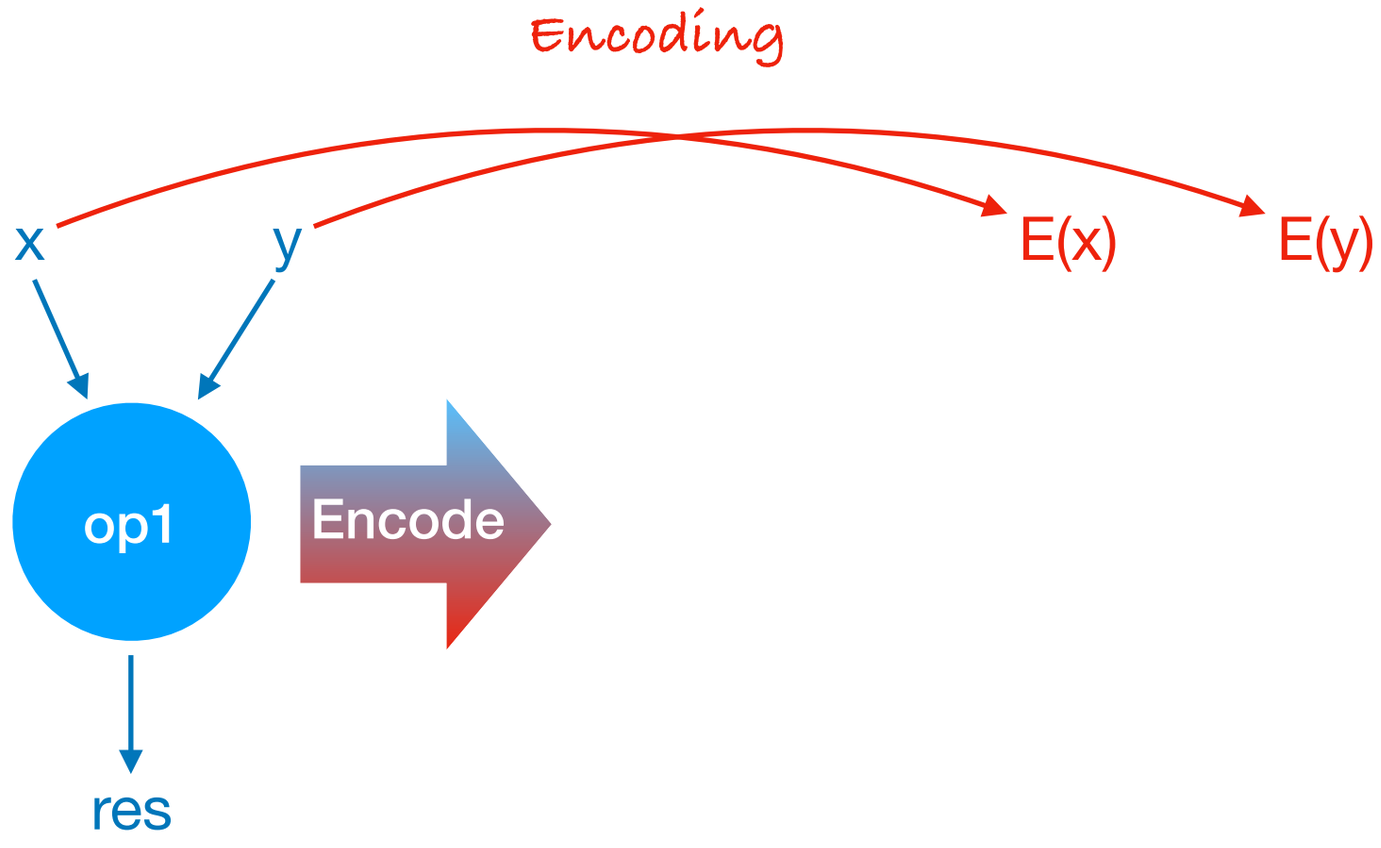
- ✓ Encoding and decoding function may depend on a *parameter* **p**
- ✓ Commonly used parametric encoding function is bitwise XOR
- ✓ **E(x) = x XOR p**
- ✓ For the property of XOR we have that **D = E**

✓ Example of XOR masking with **p = 12**

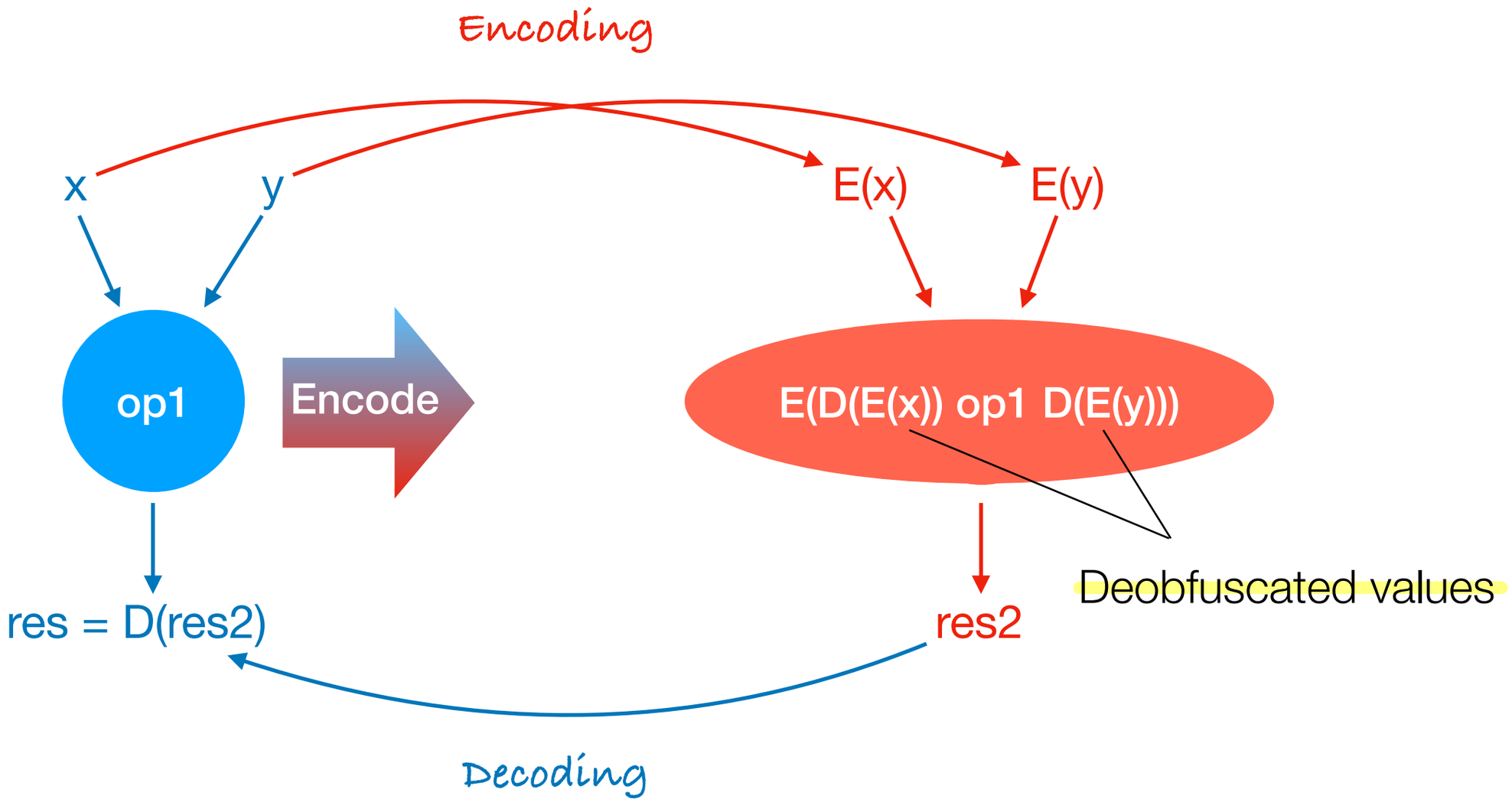
```
int a = 5;                          int a = 9; // 9 = 5^12
int b = 8;                          int b = 4; // 4 = 8^12
int x = a+b;              ⇒         int x = ((a^12)+(b^12))^12;
...                                 ...
printf("%d\n",x);                   printf("%d\n",x^12);
```

The parameter needs to be protected!!

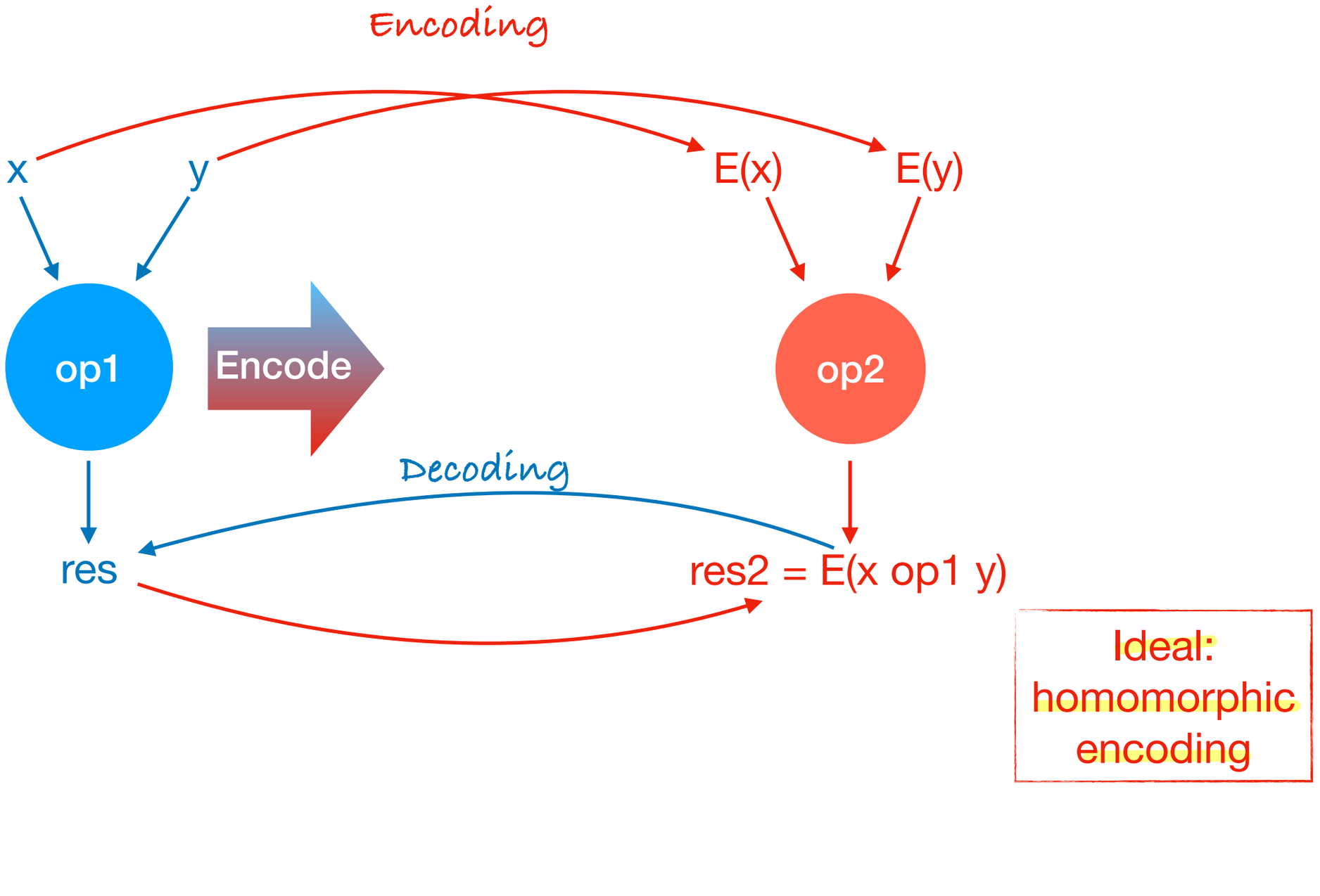
Encoding & Operations



Encoding & Operations



Encoding & Operations



Homomorphic Encoding

$E(x) = N*p + x$

$D(y) = y \bmod N$

- ✓ Homomorphic for addition and product:
- ✓ $E(x) +' E(y) = N*p_1 + x +' N*p_2 + y = N(p1 + p2) + (x + y) = E(x+y)$
- ✓ $E(x) *' E(y) = (N*p_1 + x) *' (N*p_2 + y) = N*(N*p_1*p_2 + p_2*x + p_1*y) + x*y = E(x*y)$
- ✓ Not homomprhic for comparison tests <,> =

- ✓ *Analysis of the use of the data to protect*

Residue Number Coding RNC

N1 = 31
N2 = 37

Original values
split in 2 values

```
1: int x = 12;
2: int y = 7;

3: int z = x+y;
4: int w = x*y;

5: printf(
   "z=%d, w=%d\n",
    z, w);
```

→

```
// 1965 % 31 = 12
int x1 = 1965;
// 1973 % 37 = 12
int x2 = 1973;

// 1433 % 31 = 7
int y1 = 1433;
// 2634 % 37 = 7
int y2 = 2634;

int z1 = x1+y1;
int z2 = x2+y2;

int w1 = x1*y1;
int w2 = x2*y2;

printf("z=%d, w=%d\n",
       d(z1,z2),d(w1,w2));
```

Choose randomly

Sum and multiplication performed on the obfuscated domain

Decode z and w before being printed,
the clear value of x and y never exposed

Some Issues

✓ Data encoding *modifies the range of variables*:

- *Value range propagation analysis or programmer annotations are needed to preserve the semantics*

✓ *Integration of different representations* in the same program:

```
int a = … ;
int x = … ;
int y = … ;
x = … a … ;
y = … x … ;
```

Encode

```
T1 a = … ;
T1 x = … ;
T2 y = … ;
x = … a … ;
y = … E2(D1(x)) … ;
```

- *Program slicing analysis in order use the same representation for a program slice*

Encoding Integers

Example: every value v is encoded in v+1

Check overflow

```
typedef int T1
T1 E1(int e) {return e+1}
int D1(T1 e ) {return e-1}
T1 ADD1(T1 a, T1 b){return E1(D1(a)+D1(b))}
T1 MUL1(T1 a, T1 b){return E1(D1(a)*D1(b))}
T1 LT1(T1 a, T1 b){return D1(a)<D1(b)}
```

deobfuscate before applying operations

```
typedef int T2
T2 E2(int e) {return e+1}
int D2(T2 e ) {return e-1}
T2 ADD2(T2 a, T2 b){return a+b-1}
T2 MUL2(T2 a, T2 b){return a*b-a-b+2}
T2 LT1(T2 a, T2 b){return a<b}
```

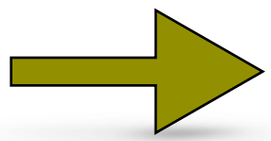
arithmetic operations performed on the obfuscated values

Homomorphic data transformations:

E(x) op E(y) = E(x op y)

Encoding Integers

```
int v = 7;
v = v * 5;
v = v + 7;
While (v < 50) v++;
```



```
T1 v = E1(7);
v = MUL1(v,E1(5));
v = ADD1(v,E1(7));
While (LT1(v,E1(50)))
   v = ADD1(v,E1(1));
```

Example of how the code changes by adding definition T1

Encoding Integers

- ✓ Many possible alternative ways for encoding integers:
- ✓ Number theoretic tricks. For example an integer y can be represented as (p * N + y) where N is the product of two close prime numbers, p is a random number. Deobufscation then is removing (p * N) by reducing modulo N.

```
typedef int T3
#define N (53 * 59)
T3 E3(int e, int p) {return p * N + e}
int D3(T3 e ) {return e % N}
T3 ADD3(T3 a, T3 b){return a + b}
T3 MUL3(T3 a, T3 b){return a * b}
T3 LT3(T3 a, T3 b){return D3(a) < D3(b)}
```

N has to be larger than any integer that we need to represent. Addition and multiplication are performed in the obfuscated space

Encoding Integers

- ✓ Encrypt integers by using one of the standard cryptographic algorithms
  - ✓ **DES**: you cannot perform operations on encrypted data and you have to decrypt operands before any arithmetic operations. This causes a considerable overhead. Moreover, the DES key has to be stored somewhere in the program.
  - ✓ **RSA** is homomorphic in multiplication
  - ✓ Homomorphic encryption

Encoding Booleans

- ✓ Since there are so few Boolean values, we use *multiple values* to encode ***true*** and multiple values to encode ***false***
- ✓ For example:
  - ✓ ***true***: any integer divisible by 2
  - ✓ *false*: any integer divisible by 3
  - ✓ E: Bool —> Integer
  - ✓ E(true) = 2*(3(rand()%10000) + rand()%2 + 1)

    not divisible by 3 and divisible by 2
  - ✓ E(false) = 3*(2(rand()%10000) + 1)

    not divisible by 2 and divisible by 3

Encoding Booleans

- ✓ Easy to obfuscate since their *value range is known*
- ✓ We follow the same approach used for encoding integers
- ✓ Since there are so few boolean values, we use multiple values to encode true and multiple values to encode false

v	p	q
T	F	F
F	F	T
F	T	F
T	T	T

- ✓ Boolean values are used in control flow statements (OR, AND) and in assignment statements (ORV, ANDV)

Encoding Booleans

```
typedef struct {int p; int q;} TD

TD trues[] = {(TD){0,0} , (TD){1,1}};
TD falses[] = {(TD){0,1} , (TD){1,0}};

TD  ED(BOOL  e)  {return  (e==TRUE?trues[rand()%2]:
falses[(rand()%2)]);}

BOOL DD(TD e) {return (e.p==e.q)?TRUE:FALSE}
```

- ✓ There are many possible ways for DD to convert back from the obfuscated to the original representation. We have choose to check p=q.

Encoding Booleans

```
TD and[] = {(TD){0,0}, // {0,0} && {0,0} = T && T = T
            (TD){1,0}, // {0,0} && {0,1} = T && F = F
            (TD){0,1}, // {0,0} && {1,0} = T && F = F
            (TD){1,1}, // {0,0} && {1,1} = T && T = T
            (TD){1,0}, // {0,1} && {0,0} = F && T = F
            (TD){1,0}, // {0,1} && {0,1} = F && F = F
            (TD){0,1}, // {0,1} && {1,0} = F && F = F
            (TD){0,1}, // {0,1} && {1,1} = F && T = F
            (TD){0,1}, // {1,0} && {0,0} = F && T = F
            (TD){1,0}, // {1,0} && {0,1} = F && F = F
            (TD){1,0}, // {1,0} && {1,0} = F && F = F
            (TD){0,1}, // {1,0} && {1,1} = F && T = F
            (TD){0,0}, // {1,1} && {0,0} = T && T = T
            (TD){1,0}, // {1,1} && {0,1} = T && F = F
            (TD){0,1}, // {1,1} && {1,0} = T && F = F
            (TD){1,1}, // {1,1} && {1,1} = T && T = T
```

Lookup table generated at obfuscation time, if generated at runtime more resilient against static analysis

```
BOOL ORD(TD a, TD b) {return ((a.p==a.q)+(b.p==b.q))!=0}

TD ORVD(TD a, TD b) {return ED(ORD(a,b))}

TD ANDVD(TD a, TD b) {return and[8*a.p + 4*a.q + 2*b.p + b.q];}

BOOL ANDD(TD a, TD b) {return DD(ANDVD(a,b))}
```

Encoding Booleans

- ✓ The *potency*, *resilience* and *cost* of variable splitting *grow with the number of variables into which the original variable is split*
- ✓ *Resilience* can be further enhanced by *selecting the encoding at run-time*: the look-up tables are not constructed at compile time (which would make them susceptible to static analysis) but by algorithms included in the obfuscated application.

Encoding Literal Data

- ✓ Literal data may carry much *semantic information*:
  - "please enter your pwd"
  - highly random 1024-bit string (probably a cryptographic key)
- ✓ Possible ways to hide literal data:
  - *Break up the data* in small peaces and spread them in the program
  - **xor** a string with a known constant value
  - convert the literal into *code* that generates the decoded value at runtime (mealy machine)

Convert Static to Procedural Data

```
main() {
   String S1,S2,S3,S4;
   S1 = "AAA";
   S2 = "BAAA";
   S3 = "CCB";
   S4 = "CCB";
}
```

$\Downarrow \mathcal{T}$

```
main() {
   String S1,S2,S3,S4,S5;
   S1 = G(1);
   S2 = G(2);
   S3 = G(3);
   S4 = G(5);
   if (P^F) S5 = G(9);
}
```

$\mathcal{T} \Rightarrow$

```
static String G (int n) {
   int i=0;
   int k;
   char[] S = new char[20];
   while (true) {
      L1:  if (n==1) {S[i++]='A'; k=0; goto L6};
      L2:  if (n==2) {S[i++]='B'; k=-2; goto L6};
      L3:  if (n==3) {S[i++]='C'; goto L9};
      L4:  if (n==4) {S[i++]='X'; goto L9};
      L5:  if (n==5) {S[i++]='C'; goto L11};
           if (n>12) goto L1;
      L6:  if (k++<=2) {S[i++]='A'; goto L6}
           else goto L8;
      L8:  return String.valueOf(S);
      L9:  S[i++]='C'; goto L10;
      L10: S[i++]='B'; goto L8;
      L11: S[i++]='C'; goto L12;
      L12: goto L10;
   }
}
```

- ✓ A simple way to obfuscate strings is to convert them into a program that produces the strings as output (and possibly other strings)

Convert Static to Procedural Data

```
main() {
   String S1,S2,S3,S4;
   S1 = "AAA";
   S2 = "BAAA";
   S3 = "CCB";
   S4 = "CCB";
}
```

$\Downarrow \mathcal{T}$

```
main() {
   String S1,S2,S3,S4,S5;
   S1 = G(1);
   S2 = G(2);
   S3 = G(3);
   S4 = G(5);
   if (P^F) S5 = G(9);
}
```

$\Rightarrow \mathcal{T}$

```
static String G (int n) {
   int i=0;
   int k;
   char[] S = new char[20];
   while (true) {
      L1:  if (n==1) {S[i++]='A'; k=0; goto L6};
      L2:  if (n==2) {S[i++]='B'; k=-2; goto L6};
      L3:  if (n==3) {S[i++]='C'; goto L9};
      L4:  if (n==4) {S[i++]='X'; goto L9};
      L5:  if (n==5) {S[i++]='C'; goto L11};
           if (n>12) goto L1;
      L6:  if (k++<=2) {S[i++]='A'; goto L6}
           else goto L8;
      L8:  return String.valueOf(S);
      L9:  S[i++]='C'; goto L10;
      L10: S[i++]='B'; goto L8;
      L11: S[i++]='C'; goto L12;
      L12: goto L10;
   }
}
```

- ✓ Aggregating the computation of all static string data into just one function would be unstealthy in most codes
- ✓ Much higher potency and resilience may be achieved by breaking the function into smaller components embedded into the “normal” control flow of the source program

Encoding Arrays

- ✓ There are two categories of array obfuscations:
  - Reorder the elements of the array. This breaks the linear order than an attacker would expect
  - Restructure the array by breaking it into peaces, merging with other unrelated arrays, or changing its dimensionality, to hide from the adversary the structure that the programmer originally intended

Evaluation

- ✓ Preliminary analysis: range analysis, slicing, …
- ✓ Parametric encoding: hide parameters (opaque constants)
- ✓ Homomorphic obfuscation: Support basic operations directly on the obfuscated version
- ✓ *Static analysis of data is more complex*
- ✓ *Dynamic analysis has to be prevented with other means*

Aggregation

Aggregation

- ✓ Merge scalar variables
- ✓ Array:
  - ✓ Split
  - ✓ Merge
  - ✓ Fold
  - ✓ Flatten

Merge Scalar Variables

- ✓ X,Y 32-bit integer variables merged in Z 64-bit integer variable
- ✓ Z(X,Y) = 2^32 *Y + X
- ✓ *Resilience*: low, can be increased by adding opaque operations on the whole variable

$$
\begin{array}{lcll}
\mathtt{Z}(\mathtt{X}+r,\mathtt{Y}) & = & 2^{32}\cdot\mathtt{Y}+(r+\mathtt{X}) & = & \mathtt{Z}(\mathtt{X},\mathtt{Y})+r \\
\mathtt{Z}(\mathtt{X},\mathtt{Y}+r) & = & 2^{32}\cdot(\mathtt{Y}+r)+\mathtt{X} & = & \mathtt{Z}(\mathtt{X},\mathtt{Y})+r\cdot 2^{32} \\
\mathtt{Z}(\mathtt{X}\cdot r,\mathtt{Y}) & = & 2^{32}\cdot\mathtt{Y}+\mathtt{X}\cdot r & = & \mathtt{Z}(\mathtt{X},\mathtt{Y})+(r-1)\cdot\mathtt{X} \\
\mathtt{Z}(\mathtt{X},\mathtt{Y}\cdot r) & = & 2^{32}\cdot\mathtt{Y}\cdot r+\mathtt{X} & = & \mathtt{Z}(\mathtt{X},\mathtt{Y})+(r-1)\cdot 2^{32}\cdot\mathtt{Y}
\end{array}
$$

```
(1) int X=45,Y=95;                (1') long Z=167759086119551045;
(2) X += 5;                       (2') Z += 5;
(3) Y += 11;          ⇒ T         (3') Z += 47244640256;
(4) X *= c;                       (4') Z += (c-1)*(Z & 4294967295);
(5) Y *= d;                       (5') Z += (d-1)*(Z & 18446744069414584320);
```

Array Splitting

```
#include <stdio.h>
int x [] = {1,2,3,4,5,6,7,8,9,10 };
int main(int argc, char * argv[]) {
    int s = 0;
    int i;
    for (i=0; i<10; i++) {
        s += x[i];
    }
    printf("sum=%d\n",s);
}
```

⇒

```
#include <stdio.h>
int x1[] = {1,3,5,7,9 };
int x2[] = {2,4,6,8,10};
int main(int argc, char * argv[]) {
    int s = 0;
    int i;
    for (i=0; i<10; i++) {
        s += (i % 2 == 0?x1[i/2]:x2[i/2]);
    }
    printf("sum=%d\n",s);
}
```

Array Merging

```
#include <stdio.h>
int x1[] = {1,3,5,7,9 };
int x2[] = {2,4,6,8,10};
int main(int argc, char * argv[]) {
    int s = 0;
    int i;
    for (i=0; i<5; i++) {
        s += x1[i]*x2[i];
    }
    printf("cprod=%d\n",s);
}
```

⇒

```
#include <stdio.h>
int x[] = {1,2,3,4,5,6,7,8,9,10};
int main(int argc, char * argv[]) {
    int s = 0;
    int i;
    for (i=0; i<5; i++) {
        s += x[2*i]*x[2*i+1];
    }
    printf("cprod=%d\n",s);
}
```

Array Folding

```
#include <stdio.h>
int x [] = { 1,2,3,4,5,6,7,8,9,10 };
int main(int argc, char * argv[]) {
    int s = 0;
    int i;
    for (i=0; i<10; i++) {
        s += x[i];
    }
    printf("sum=%d\n",s);
}
```

⇒

```
#include <stdio.h>
int x[][5] = { { 1,2,3,4,5 },
               {6,7,8,9,10}};
int main(int argc, char * argv[]) {
    int s = 0;
    int i;
    for (i=0; i<10; i++) {
        s += x[i/5][i%5];
    }
    printf("sum=%d\n",s);
}
```

Array Flattening

```
#include <stdio.h>
int x[][5] = { {1,2,3,4,5 },
               {6,7,8,9,10},
               {11,12,13,14,15}};
int main(int argc, char * argv[]) {
    int s = 0;
    int i,j;
    for (i=0; i<3; i++) {
        for (j=0; j<5; j++) {
            s += x[i][j];
        }
    }
    printf("sum=%d\n",s);
}
```

⇒

```
#include <stdio.h>
int x[] = { 1,2,3,4,5,6,7,8,9,10,
            11,12,13,14,15};
int main(int argc, char * argv[]) {
    int s = 0;
    int i,j;
    for (i=0; i<3; i++) {
        for (j=0; j<5; j++) {
            s += x[i*5+j];
        }
    }
    printf("sum=%d\n",s);
}
```

Aggregation

- ✓ *Potency* given by the fact that they introduce structure where there was originally none or they remove structure from the original program. This can incense the obscurity of a program for *human inspection and reverse engineering in general*

Array Reordering

✓ Reorder the elements of the array according to the permutation array P or by defining a mapping function

```
#include <stdio.h>
char * m = "secret message";
int main(int argc, char * argv[]) {
    printf("%s\n",m);
}
```

⇒

```
#include <stdio.h>
char m [] = {
        't', 'r', 'g', 'e', 'e', ' ', 'm',
        'c', 's', 'e', 's', 's', 'e', 'a' };
char p [] = {
        11, 3, 7, 1, 4, 0, 5, 6, 9, 10, 8,
        13, 2, 12 };
int main(int argc, char * argv[]) {
    int i;
    for (i=0; i<14; i++) {
        putchar(m[p[i]]);
    }
    putchar('\n');
}
```

Array Reordering

- ✓ *Potency* given by the fact that an unnatural ordering of the elements is used. This breaks locality and it can *incense the obscurity of a program for human inspection and reverse engineering in general*

Hide Parameters

- ✓ Parametric encoding
- ✓ Crypto keys
- ✓ Permutation arrays
- ✓ …
- ✓ *Need to be protected!!!*

Code Obfuscation Evaluation

- ✓ Model the attacker
- ✓ Model the attacker goal
- ✓ Analyze the specific code transformations

Protecting software through obfuscation: can it keep pace with progress in code analysis?
ACM Computing Survey 2016

Code Analysis Categories

- ✓ **Pattern matching:** simplest and fastest analysis. It's a syntactic analysis for the identification of static sequences of instructions, regular expressions or machine learning-based classifier.

- ✓ **Automated static analysis**: statically reasons about program semantics. Simple forms include disassemblers that interpret branch targets. Often used to reconstruct high-level information about programs.

- ✓ **Automated dynamic analysis**: it is able to very precisely reason about program behaviour along the observed traces. Code coverage problem.

- ✓ **Human-assisted analysis**: is the most capable "analysis". In this reverse engineering process the analyst aim at understanding the program structure and behaviour with the help of a variety of tools.

Protecting software through obfuscation: can it keep pace with progress in code analysis? ACM Computing Survey 2016

Analyst's Aims

- ✓ *Finding the location of data:* the analyst wants to retrieve some data embedded in the program (cryptographic key, licensing keys, certificates, credentials..)

- ✓ *Finding the location of program functionality*: the analyst wants to identify the entry point of a particular function within the obfuscated program (entry point of a cryptographic algorithm, identify copy protection mechanisms,…)

- ✓ *Extraction of code fragments*: the analyst wants to extract a piece of code including all possible dependences that implement a particular functionality from an obfuscated program (competitors code, decryption routine for breaking DRM, alter the functionality at runtime - in situ reuse, similar to return oriented programming).

- ✓ *Understanding the program*: the analysis wants to fully understand the obfuscated program (de-obfuscation, find vulnerabilities, intellectual property theft)

Protecting software through obfuscation: can it keep pace with progress in code analysis? ACM Computing Survey 2016

Table II. Analysis of the strength of code obfuscation classes in different analysis scenarios (PM = Pattern Matching LD = Locating Data, LC = Locating Code, EC = Extracting Code, UC = Understanding Code).

	PM		**Autom. Static**				**Autom. Dynamic**				**Human Assisted**			
Name	**LD**	**LC**	**LD**	**LC**	**EC**	**UC**	**LD**	**LC**	**EC**	**UC**	**LD**	**LC**	**EC**	**UC**
Data obfuscation														
Reordering data			✓				✓							
Changing encodings	✓						✓							
Converting static data to procedures	✓						✓				✓			

Legend		obfuscation breaks analysis fundamentally
		obfuscation is not unbreakable, but makes analysis more expensive
		obfuscation only results in minor increases of costs for analysis
	✓	A checkmark indicates that the rating is supported by results in the literature
		Scenarios without a checkmark were classified based on theoretical evaluation

Protecting software through obfuscation: can it keep pace with progress in code analysis?
ACM Computing Survey 2016

Code Obfuscation

Code Obfuscation

- ✓ *Static code rewriting:* A static rewriter is similar to an optimizing compiler, as it modifies program code during obfuscation, but allows its output to be executed **without further run-time modifications.** All data obfuscation techniques described above would also fall into the category of static code rewriting.
  - ✓ Replacing instructions
  - ✓ Opaque predicates
  - ✓ Insert dead or irrelevant code
  - ✓ Reordering
  - ✓ Loop transformations
  - ✓ Function splitting/recombination
  - ✓ Name scambling
  - ✓ Control flow obfuscation
  - ✓ …..

Protecting software through obfuscation: can it keep pace with progress in code analysis? ACM Computing Survey 2016

Code Obfuscation

- ✓ Dynamic code rewriting: The main characteristic of code obfuscation schemes in this category is that **the executed code differs from the code that is statically visible in the executable**.
  - ✓ Packing/Encryption
  - ✓ Dynamic code modifications
  - ✓ Environment requirements (keys constructed from the environment, outside the activation environment the program does not reveal its secrets)
  - ✓ HW-assisted code obfuscation (HW-SW binding by making the execution of the software dependant on some HW token. Without the token the analysis of the software will fail)
  - ✓ Virtualization

Protecting software through obfuscation: can it keep pace with progress in code analysis? ACM Computing Survey 2016

Control Obfuscation

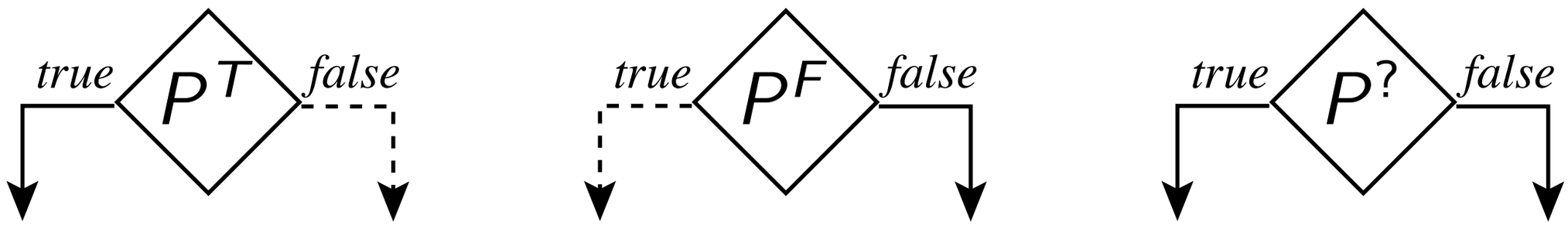
Control Flow Obfuscation

- ✓ Transform a well-structured control flow of an executable into a difficult analyzable program structure by control flow transformations
- ✓ Such control flow obfuscation can *obstruct and stop static attacks to the control flow of the executable code under white-box attacks*
- ✓ The attacker must take dynamic approaches to get the control flow so that it becomes much harder to do and increase attacking effort significantly

Opaque Predicate

- ✓ Opaque predicates are the major building block in transformations that obfuscate the control flow
- ✓ *idea*: construct at obfuscation time an expression whose value is known to the defender, but it is difficult for an attacker to deduce
- ✓ Opaque predicates are boolean value expressions for which the defender knows whether they will return *true*, *false* or sometimes *true* and sometimes *false*.

.



Opaque Predicates from Number Theory

$$\forall x, y \in \mathbb{Z}: \quad 7y^2 - 1 \neq x^2$$

$$\forall x \in \mathbb{Z}: \quad 2 \mid (x + x^2)$$

$$\forall x \in \mathbb{Z}: \quad 3 \mid (x^3 - x)$$

$$\forall n \in \mathbb{Z}^+, x, y \in \mathbb{Z}: \quad (x - y) \mid (x^n - y^n)$$

$$\forall n \in \mathbb{Z}^+, x, y \in \mathbb{Z}: \quad 2 \mid n \ \vee \ (x + y) \mid (x^n + y^n)$$

$$\forall n \in \mathbb{Z}^+, x, y \in \mathbb{Z}: \quad 2 \nmid n \ \vee \ (x + y) \mid (x^n - y^n)$$

$$\forall x \in \mathbb{Z}^+: \quad 9 \mid (10^x + 3 \cdot 4^{(x+2)} + 5)$$

$$\forall x \in \mathbb{Z}: \quad 3 \mid (7x - 5) \ \Rightarrow \ 9 \mid (28x^2 - 13x - 5)$$

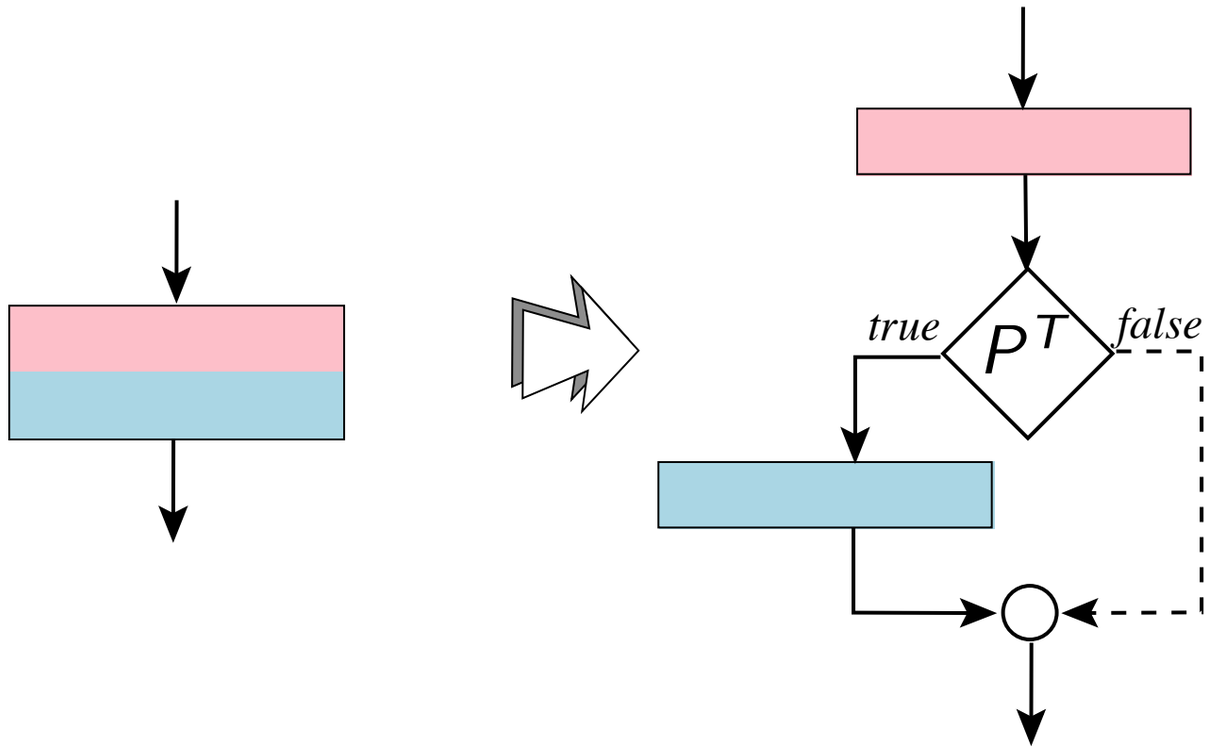
$$\forall x \in \mathbb{Z}: \quad 5 \mid (2x - 1) \ \Rightarrow \ 25 \mid (14x^2 - 19x - 19)$$

$$\forall x, y, z \in \mathbb{Z}: \quad (2 \nmid x \ \wedge \ 2 \nmid y) \ \Rightarrow \ x^2 + y^2 \neq z^2$$

$$\forall x \in \mathbb{Z}^+: \quad 14 \mid (3 \cdot 7^{4x+2} + 5 \cdot 4^{2x-1} - 5)$$

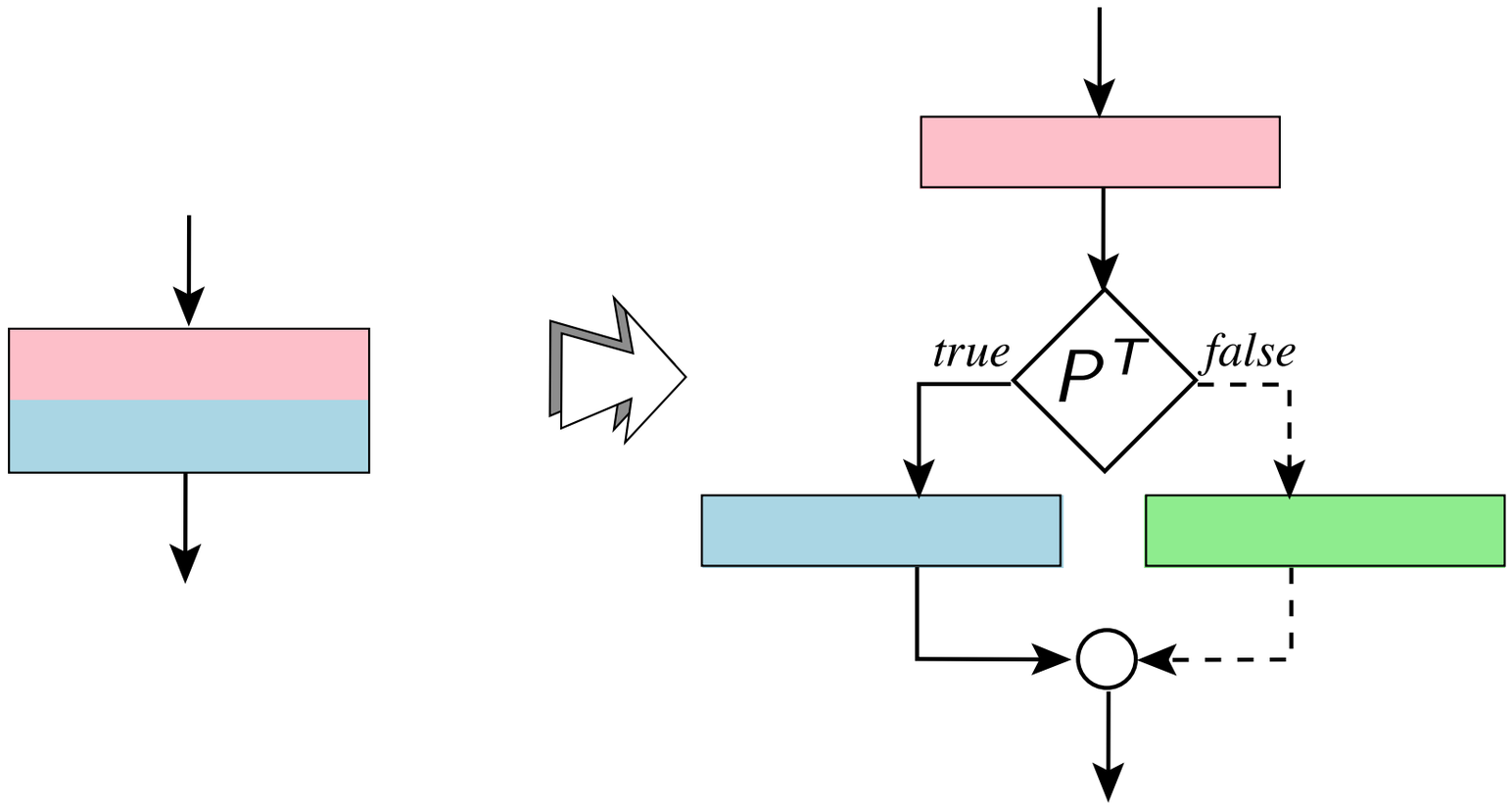
Opaque Predicate Insertion

✓ The simplest block splitting



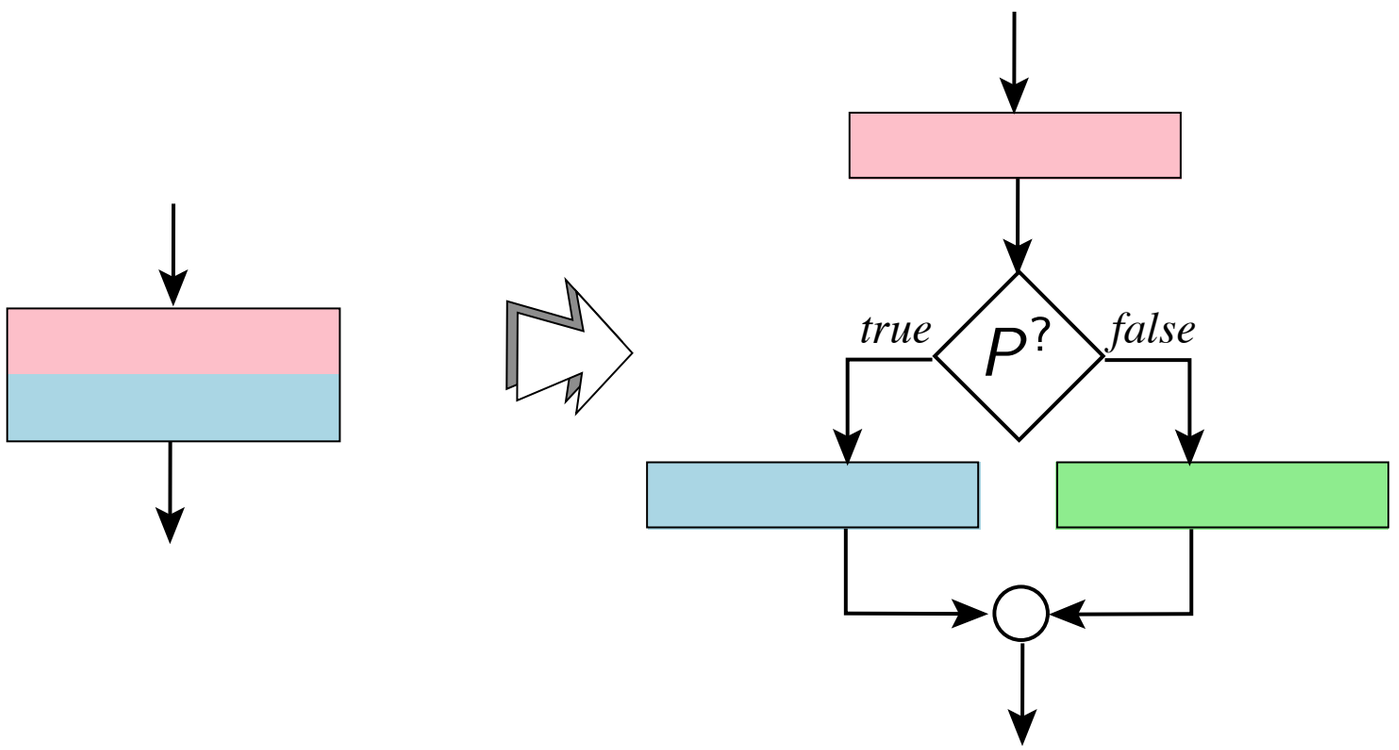
✓ The blue block seems to be executed only sometimes.

Opaque Predicate Insertion



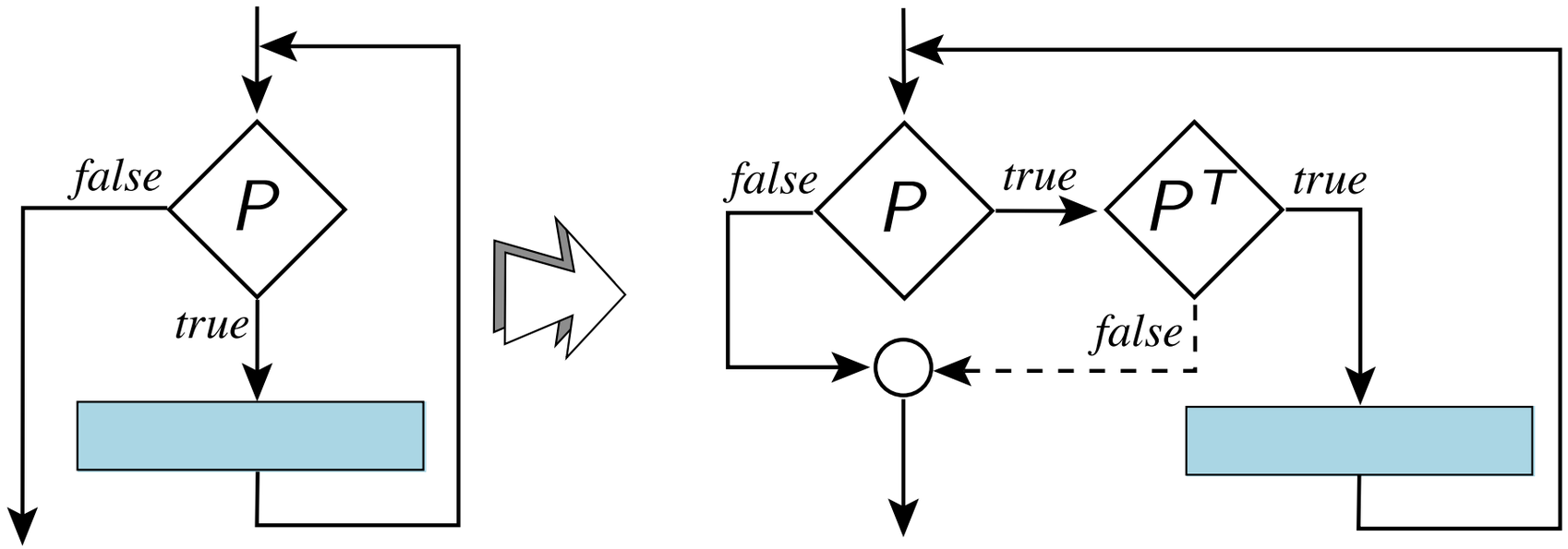
- ✓ A bogus block (green) appears as it might be executed while, in fact it never will

Opaque Predicate Insertion



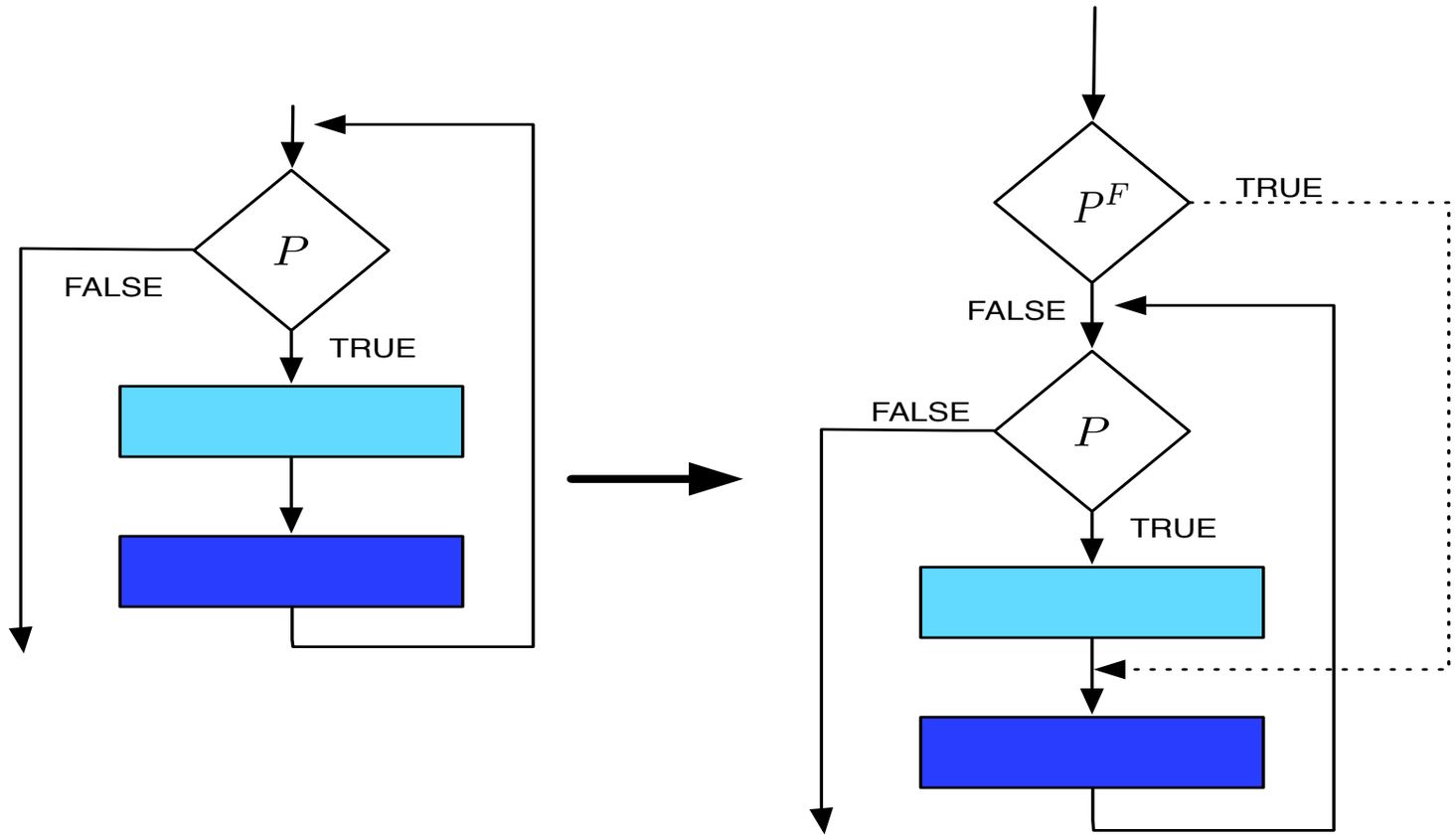
- ✓ Sometimes execute the blue block, sometimes the green block
- ✓ The green and blue block should be *semantically equivalent*

Opaque Predicate Insertion



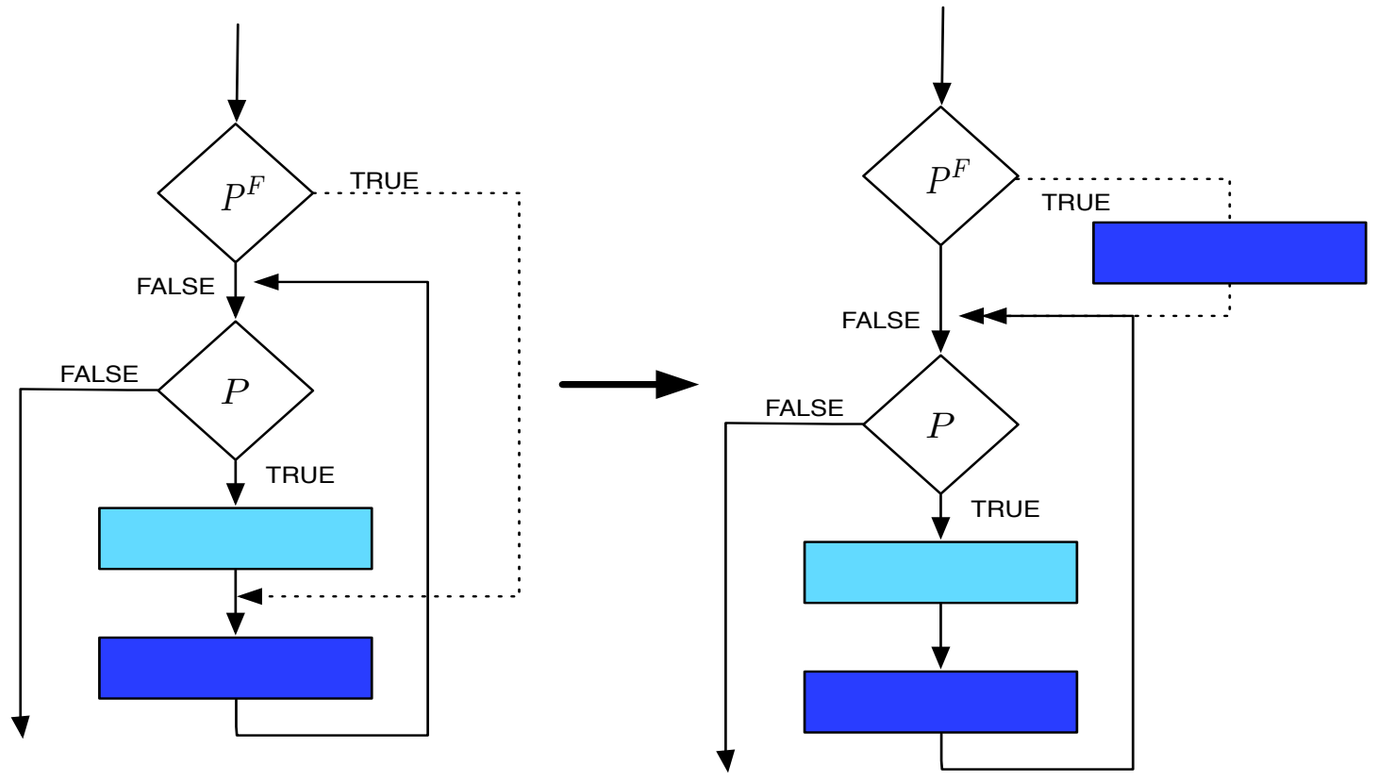
✓ Extend a loop condition P by conjoining it with an opaquely true predicate

Opaque Predicate Insertion



- ✓ Making loops *irreducible* by jumping into a loop
- ✓ Data flow analysis algorithms assume reducible CFG and on these CFG they are efficient

Opaque Predicate Insertion



- ✓ Making obfuscated loops *reducible* by *code replication*
- ✓ Hard for the attacker due to *exponential blow up*

Folklore Confirmed: Reducible Flow Graphs are Exponentially Larger, L. Carter et al. POPL 2003

Constructing Opaque Predicates

- ✓ Construct them based on:
  - Numeric theoretic results

    $$\forall x, y \in \mathbb{Z} : x^2 - 34y^2 \neq 1$$
    $$\forall x \in \mathbb{Z} : 2 | x^2 + x$$

  - The hardness of the alias analysis
  - The hardness of the concurrency analysis
- ✓ Protect them by:
  - Making them hard to find
  - Making them hard to break
- ✓ If your obfuscation keeps a table of predicates, your adversary will too

Opaque Predicates from Pointer Analysis

- ✓ Create an obfuscating transformation from a known computationally hard static analysis problem
- ✓ Assume that
  - The attacker will analyze the program *statically*
  - We can force the attacker to solve a particular static analysis problem to discover the secret he is after
  - We can generate an *actual hard instance* of this problem for the attacker to solve (open problem to create an algorithm that generates provably hard instances of a static analysis problem)
- ✓ Of course, these assumptions could be *false* !

Opaque Predicates from Pointer Analysis

- ✓ Alias analysis is known to be complex
- ✓ Two pointers are said to *alias* each other if they point to the same memory location
- ✓ *Alias analysis*: is the process of determining if, at a given program point, two variables may refer to the same location
  - ✓ 1-level aliasing is easy **P** [Banning'79]
  - ✓ >= 2-level aliasing is hard **NP** [Horowitz'97]
  - ✓ With dynamic memory allocation is undecidable!!
- ✓ *Understanding control flow = solve a >= 2-level aliasing problem*

Opaque Predicates from Pointer Analysis

- ✓ Creates opaque predicates from pointer analysis problems by going beyond the capabilities of known analysis algorithms
- ✓ Many algorithms have problems with *destructive updates* They are able to properly analyze programs that build a linked list, but they may fail on one that modifies the list

> Despite the great deal of work on both flow-sensitive and context-sensitive algorithms [...], none has been shown to scale to programs with millions of lines of code, and most have difficulty scaling to 100.000 lines of code. [Hardekopf and Lin]

- ✓ Alias analysis algorithms are designed to perform well on *normal code* written by humans!

Manufacturing cheap, resilient and stealthy opaque predicates, Collberg et al. POPL 1998

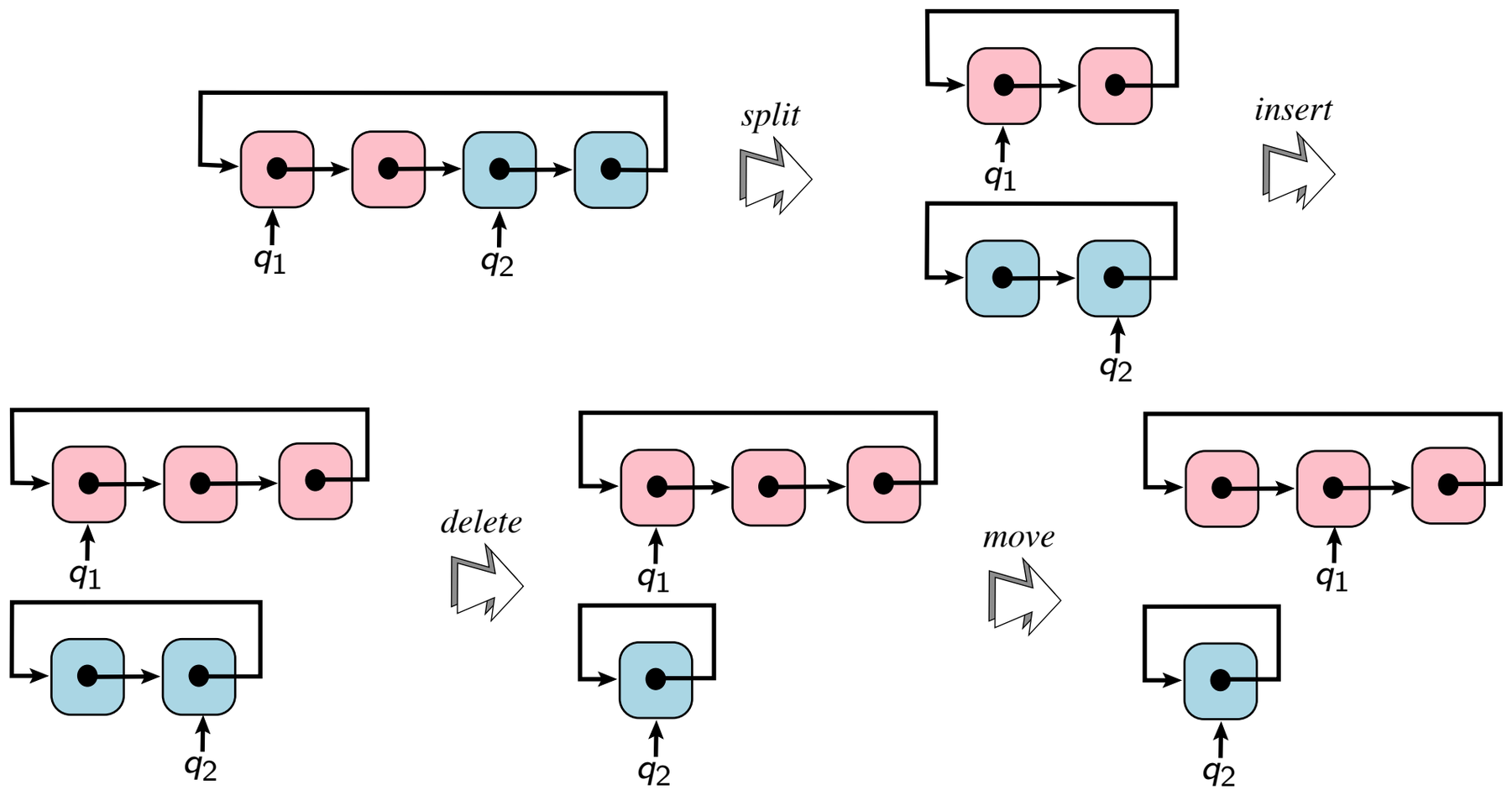
Opaque Predicates from Pointer Analysis

- ✓ Construct one or more heap-based graphs
- ✓ Keep pointers into those graphs
- ✓ Create opaque predicates by checking properties known to be true
- ✓ To confound analysis add operations to the graphs that move the pointers around and that perform destructive update operations on the graphs *while maintaining the set of invariants* your opaque predicate will test for

Manufacturing cheap, resilient and stealthy opaque predicates, Collberg et al. POPL 1998

Opaque Predicates from Pointer Analysis

- ✓ Invariants:
  - G_1 (pink) and G_2 (blue) are circular linked lists
  - q_1 points to a node in G_1 and q_2 points to a node in G_2
- ✓ Perform enough operations to confuse alias analysis
- ✓ Insert opaque predicate $(q_1 \neq q_2)^T$



Manufacturing cheap, resilient and stealthy opaque predicates, Collberg et al. POPL 1998

Opaque Predicates from Pointer Analysis

CreateOpaquePredicate(P)

1. Add to **P** code to build a set of dynamically allocated global point-structures **G = {G_1,G_2…}**
2. Add to **P** a set of pointers **Q = {q_1,q_2…}** that point to the structures in **G**
3. Construct a set of invariants **I = {I_1,I_2,…}** over **G** and **Q** such that
   - **(q_i =! q_j)^T** if **q_i** and **q_j** are known to point to different graphs **G_n** and **G_m**
   - **(q_i =! q_j)$^?$** if **q_i** and **q_j** are known to both point into a graph **G_k**
4. Add code to **P** that occasionally modifies the graphs in **G** and the points in **Q** while maintaining invariants **I**
5. Using the invariants **I**, construct opaque predicates over **Q** such that
   - **q_i** always points to nodes in **G_i**
   - **G_i** is always strongly connected

Manufacturing cheap, resilient and stealthy opaque predicates, Collberg et al. POPL 1998

Opaque Predicates from Array Analysis

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

Invariants:

1. every third cell (in pink), starting will cell 0, is $\equiv 1 \bmod 5$;
2. cells 2 and 5 (green) hold the values 1 and 5, respectively;
3. every third cell (in blue), starting will cell 1, is $\equiv 2 \bmod 7$;
4. cells 8 and 11 (yellow) hold the values 2 and 7, respectively.

You can update a pink element as often as you want, with any value you want, as long as you ensure that the value is always $\equiv 1 \bmod 5$!

Opaque Predicates from Array Analysis

```
int g[] = {36,58,1,46,23,5,16,65,2,41,
           2,7,1,37,0,11,16,2,21,16};

if ((g[3] % g[5])==g[2])
    printf("true!\n");

g[5] = (g[1]*g[4])%g[11] + g[6]%g[5];
g[14] = rand();
g[4] = rand()*g[11]+g[8];

int six = (g[4] + g[7] + g[10])%g[11];
int seven = six + g[3]%g[5];
int fortytwo = six * seven;
```

- ✓Pink: opaquely true predicate
- ✓Blue: ensures that g is constantly changing at runtime, while preserving the invariants
- ✓Green: construct an opaque value 42

Breaking opaque Predicates

```
...
x1 ← ...;
x2 ← ...;
...
b ← f(x1, x2, ...);
if b goto ...
```

- ✓ Find the instruction to make up *f(x₁,x2,…)*, backward slice
- ✓ Find the inputs to *f*, i.,e., *x₁,x₂..*
- ✓ Find the range of values *R₁* of *x₁*,..
- ✓ Compute the outcome of *f* for all possible values and kill the branch if *f* is always *TRUE*
- ✓ Use number-theory/brute force attack to determine opaqueness

Breaking opaque Predicates

- ✓ How to make attacker's task *more difficult*? Make it harder to:
  - Find $f(x_1,x2,\ldots)$
  - Find the inputs $x_1,x_2\ldots$ to f
  - Find the ranges $R_1,R_{2,\ldots}$ of $x_1,x_2\ldots$
  - Determine the outcome of f for alla argument values

Dynamic Opaque Predicates

- ✓ Opaque predicates are vulnerable to *dynamic analysis*: if an attacker can monitor heap, registers during execution then he/she may quickly understand that a predicate always evaluates to *TRUE*.

- ✓ *Dynamic opaque predicates*: a family of *correlated* opaque predicates which all evaluate to the same result in any given run, but in different runs they may evaluate to different results

Predicate	Run 1	Run 2	Run 3	Run 4	Run 5
Pred 1	T	T	F	T	F
Pred 2	T	T	F	T	F
Pred 3	T	T	F	T	F
Pred 4	T	T	F	T	F
Pred 5	T	T	F	T	F

- ✓ Difficult to determine which predicates are correlated (exponential sets of possible predicates)

Experience with software watermarking, Palsberg et al, ACSAC 2000

Distributed Obfuscation

- ✓ Analyzing distributed concurrent programs is difficult because of
  - *Interleaving semantics*: if you have *n* statements in a parallel region, then there are *n!* different orders in which they could execute
  - Badly designed concurrent programs are susceptible to *race conditions*, where several processes manipulate the same global data structure without the prerequisite locking statements
- ✓ These situations are difficult to analyze

Manufacturing opaque predicates in distributed systems for code protection. Majundar and Thomborson 2006

Distributed Obfuscation

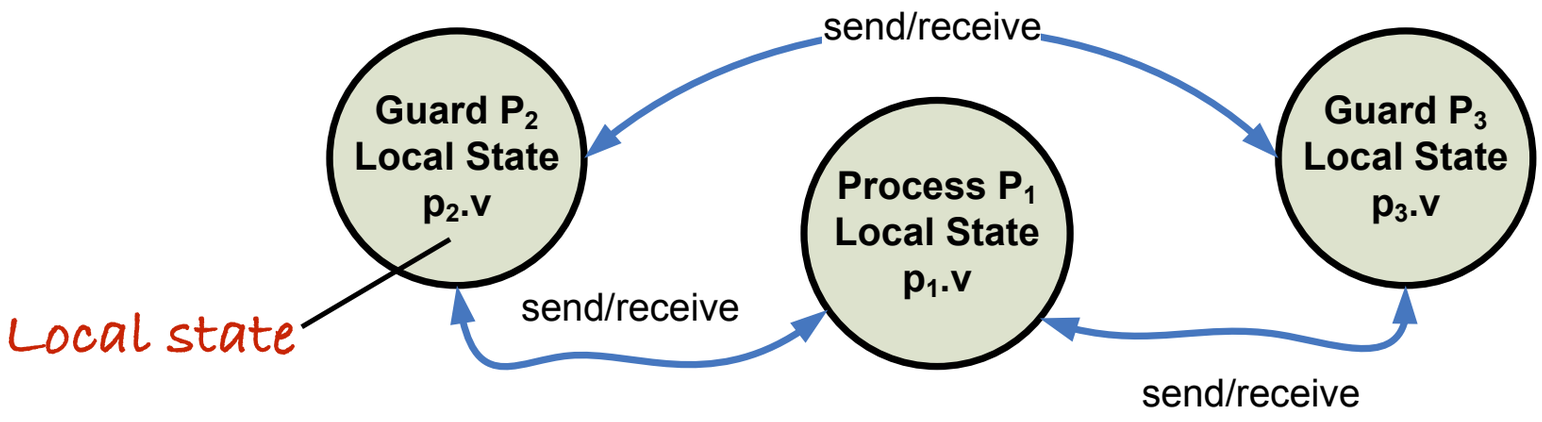
- ✓ Distributed opaque predicate: is an opaque predicate whose value depends on local states of multiple processes spread across the distributed system for its evaluation
- ✓ Idea:

```
1. add to P code to create a global data structure G
with a set of invariants I = {I1,I2,…}

2. add to P code to create threads T1,T2,… which make
concurrent updates to G while maintaining I

3. use the invariants I to create opaque predicates
```

Manufacturing opaque predicates in distributed systems for code protection. Majundar and Thomborson 2006

Distributed Obfuscation

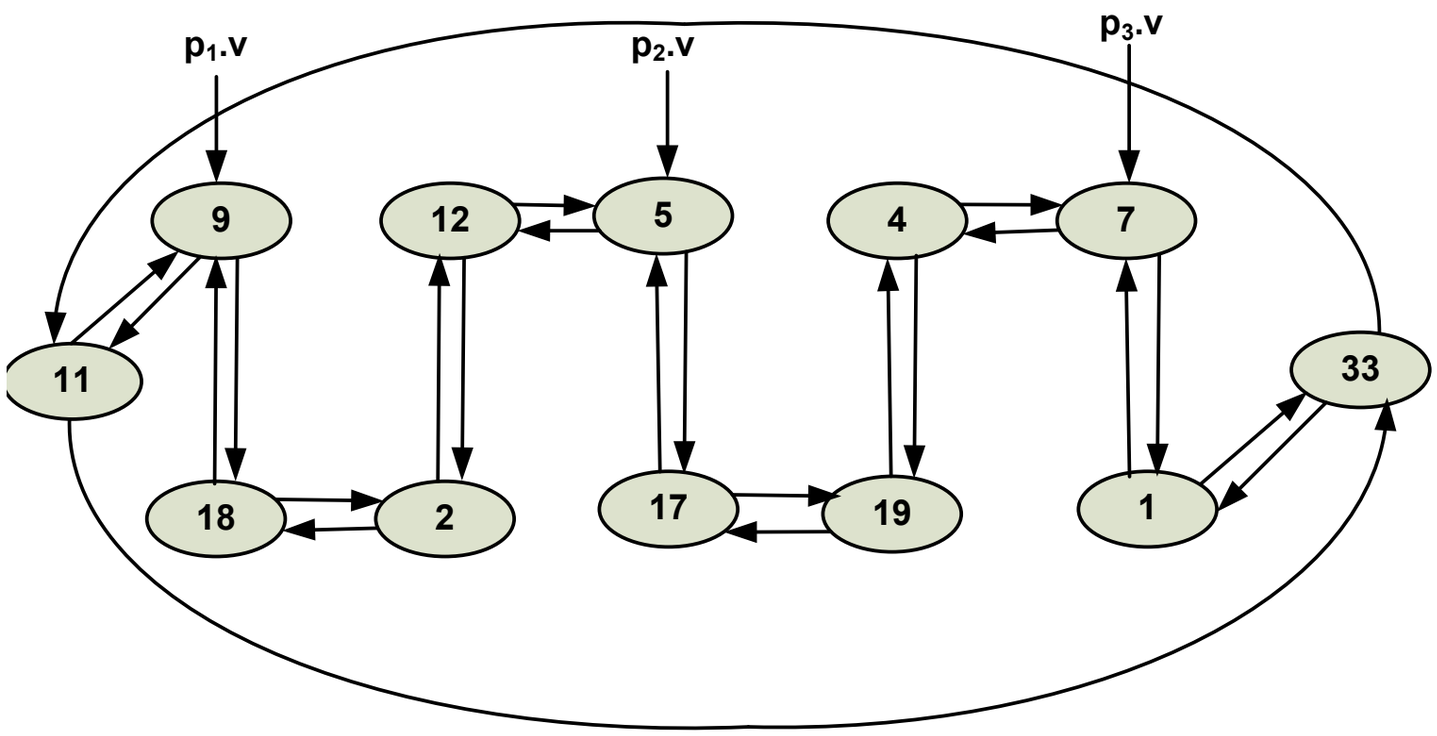
- ✓ Consider a distributed computing system consisting of a set of inter-communicating processes P_1,P_2,P_3…
- ✓ To obfuscate the control flow of P_1 we select a certain number of *guard* processes
- ✓ *Idea*: distribute the local states formed in the construction of distributed opaque predicate in P1 amongst the guards, and embed a communication pattern in the form of send/receive calls that update respective local states processes to previously known values



Manufacturing opaque predicates in distributed systems for code protection. Majundar and Thomborson 2006

Distributed Obfuscation

- ✓ 0/1 Knapsack problem is NP-hard: given a set of non-negative integers $S=\{a_1,a_2,\ldots,a_n\}$ and a sum $T=\sum x_i * a_i$ where x_i ranges over {0,1}
- ✓ Consider for example the set S={11,9,18,2,12,5,17,19,4,7,1,33}
- ✓ P_1,P_2 and P_3 are initialized by passing a dynamic data structure, such as a doubly circular linked-list, initialized with the element of the set S.
- ✓ Every list has also a pointer for each process representing the process local state used in the evaluation of the opaque predicate.



T = 27

x = {0,0,1,0,0,1,0,0,1,0,0,0}

p1.v = 18
p2.v = 5
p3.v = 4

Manufacturing opaque predicates in distributed systems for code protection. Majundar and Thomborson 2006

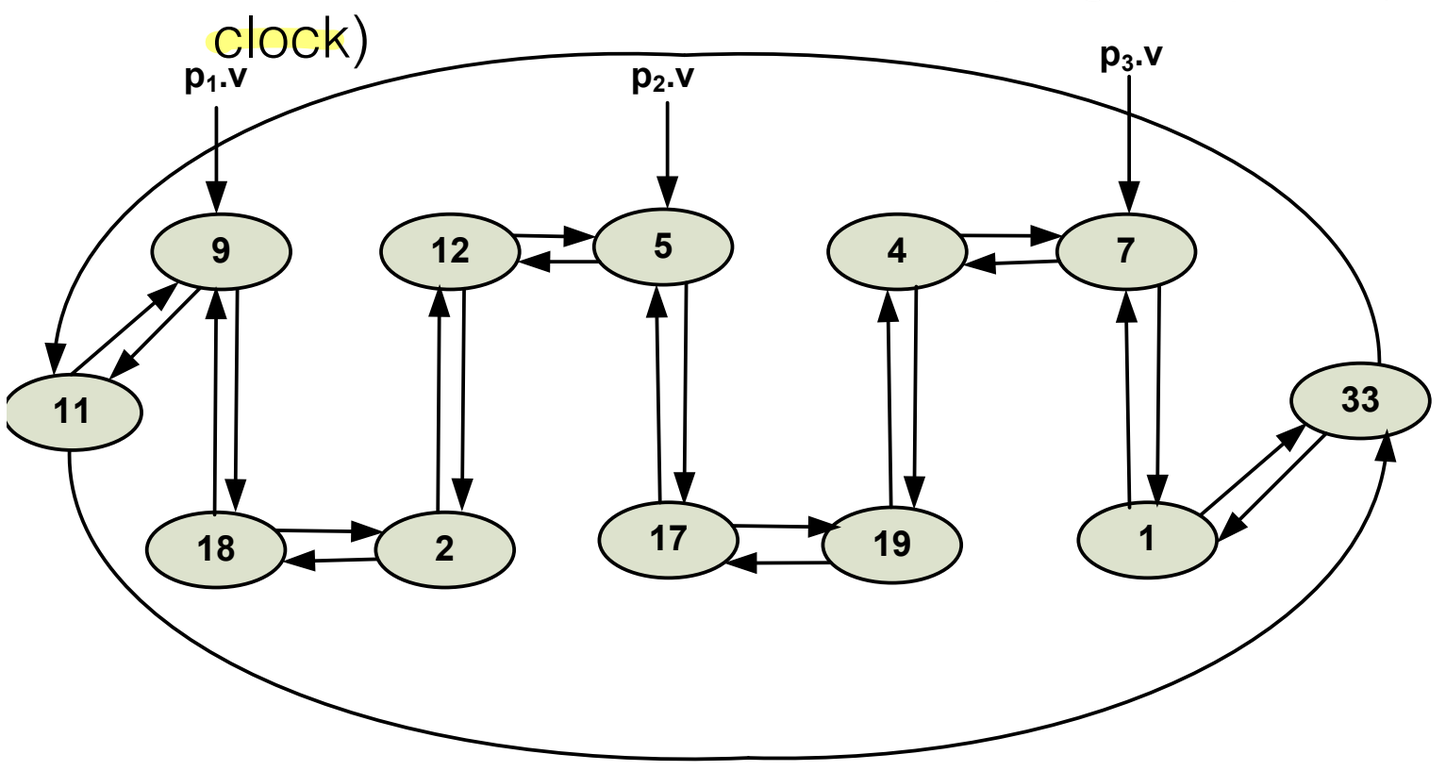
Distributed Obfuscation

- ✓ When processes communicate they update their local state by moving the pointer in the circular linked-list

- ✓ Local state update rules:

  - receive(m): pointer shifted right of the current node
  - send(m): pointer shifted left of the current node

- ✓ The value of the opaque predicates is “unstable” during execution but we can be sure of its value on certain program point (ensure casual delivery - vector clock)



T = 27

x = {0,0,1,0,0,1,0,0,1,0,0,0}

p1.v = 18
p2.v = 5
p3.v = 4

Manufacturing opaque predicates in distributed systems for code protection. Majundar and Thomborson 2006

Control Flow Flattening

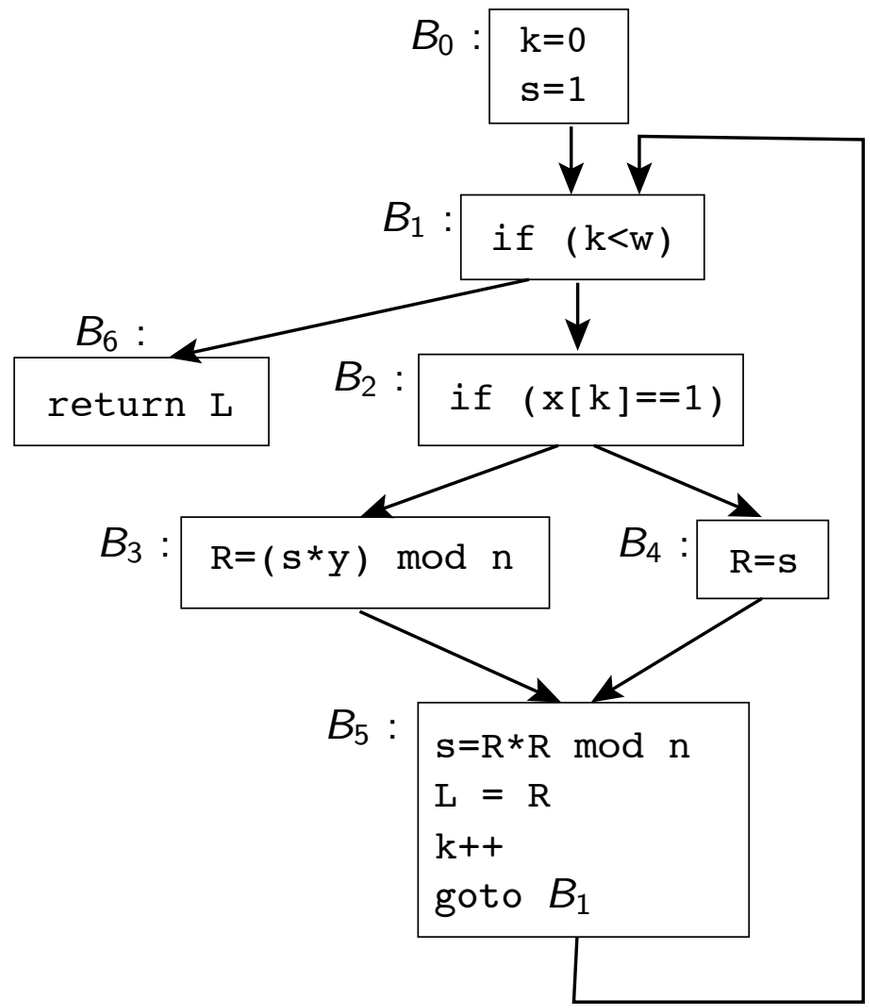
Control Flow Flattening

- ✓ Remove the control-flow structure of functions by *flattening* the corresponding CFG
- ✓ Known as *chenxify* from the inventor
- ✓ *Idea*: put each base block in a case inside a switch statement, and wrap the switch inside an infinite loop
- ✓ The algorithm maintains the correct control flow by making each basic block update a variable *next* so that control will continue at the appropriate basic block

Protection of Software-based Survivability Mechanisms. Chenxi Wang et al 2001

Control Flow Flattening

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



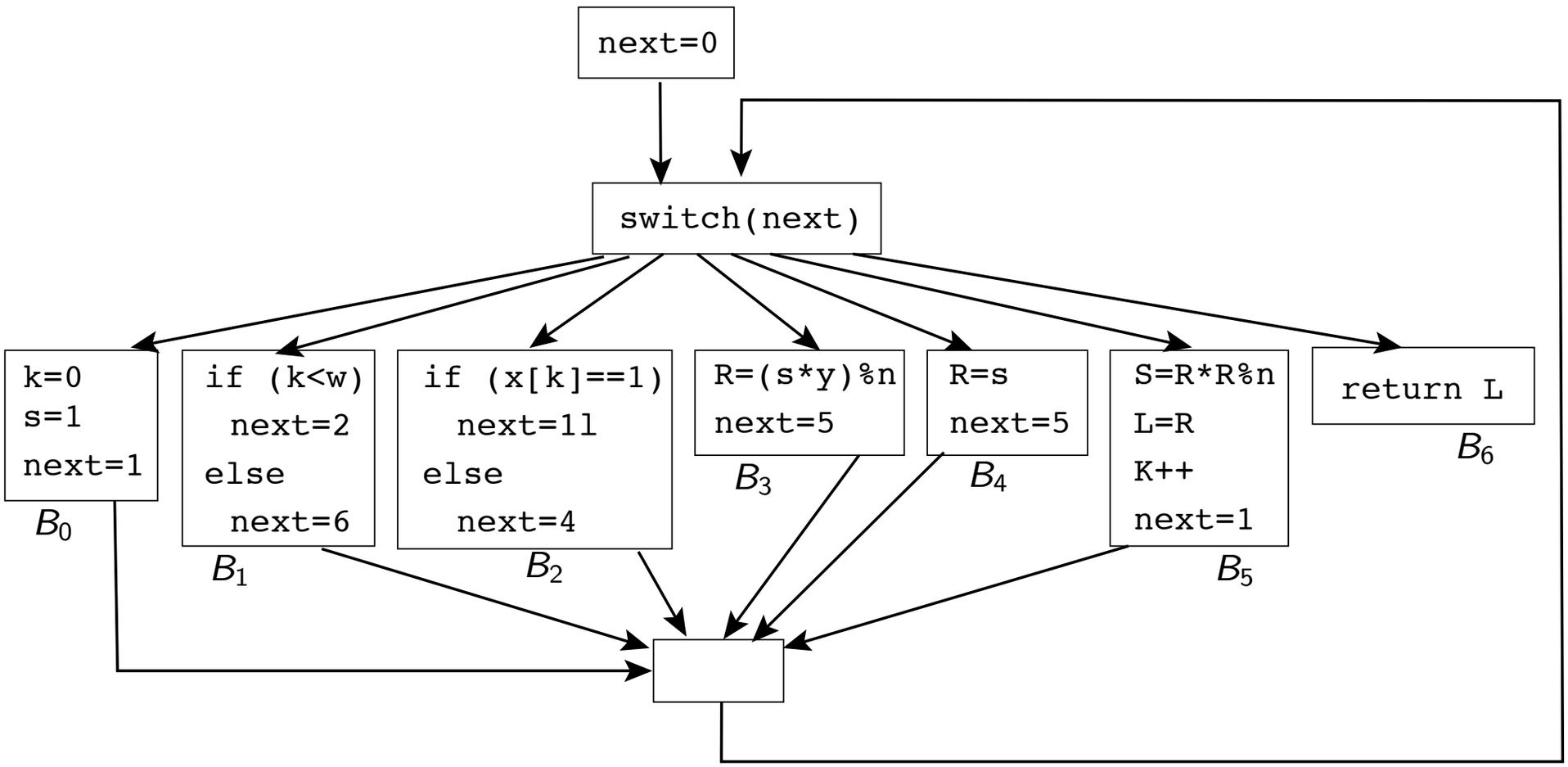
Protection of Software-based Survivability Mechanisms. Chenxi Wang et al 2001

Control Flow Flattening

```
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;
        }
}
```

Protection of Software-based Survivability Mechanisms. Chenxi Wang et al 2001

Control Flow Flattening



Protection of Software-based Survivability Mechanisms. Chenxi Wang et al 2001

Control Flow Flattening

- ✓ Very effective code obfuscation
- ✓ Very expensive code obfuscation 10x
- ✓ Apply to certain portions of the code

Protection of Software-based Survivability Mechanisms. Chenxi Wang et al 2001

Table II. Analysis of the strength of code obfuscation classes in different analysis scenarios (PM = Pattern Matching LD = Locating Data, LC = Locating Code, EC = Extracting Code, UC = Understanding Code).

In generale l'offuscamento funziona bene sull'analisi statica per questo è più sicuro sull'analisi statica

nella semantica scompare il rumore perché devo mantenere ciò che calcolo

Name	PM		Autom. Static				Autom. Dynamic				Human Assisted			
	LD	LC	LD	LC	EC	UC	LD	LC	EC	UC	LD	LC	EC	UC
Static code rewriting														
Replacing instructions	■	✓		✓		■								
Opaque predicates	■	■	■	✓	■	■			■	■			■	
Inserting dead code	■	■				✓			✓	✓				
Inserting irrelevant code	■	✓				■								
Reordering	■	■				■								
Loop transformations	■	■	■	■	■	■			■	■				
Function splitting/recombination	■	■				✓								
Aliasing	■	■	■	■	■	✓			✓	■			■	
Control flow obfuscation	✓	■	■	■	■	■			✓	✓		✓		✓
Parallelized code	■	■	■	■	■	■			■	■				
Name scrambling	■	■				✓						■	■	■
Removing standard library calls	■	■	■	■	■	■			■	■		■	■	✓
Breaking relations	■	■			■	■			■	■		■	■	■

Legend		
	■	obfuscation breaks analysis fundamentally
	■	obfuscation is not unbreakable, but makes analysis more expensive
		obfuscation only results in minor increases of costs for analysis
	✓	A checkmark indicates that the rating is supported by results in the literature
		Scenarios without a checkmark were classified based on theoretical evaluation

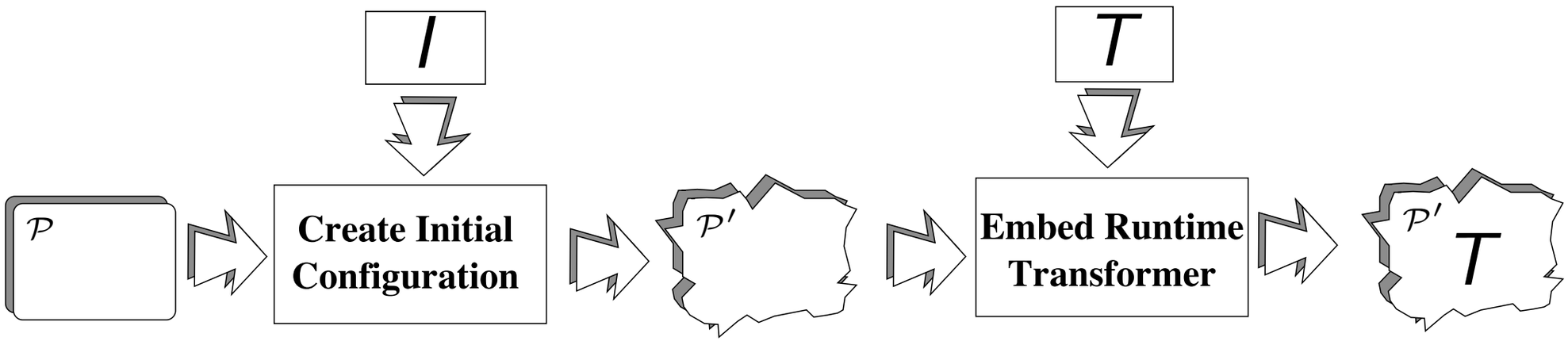
Protecting software through obfuscation: can it keep pace with progress in code analysis?
ACM Computing Survey 2016

Dynamic Obfuscation

Dynamic Obfuscation

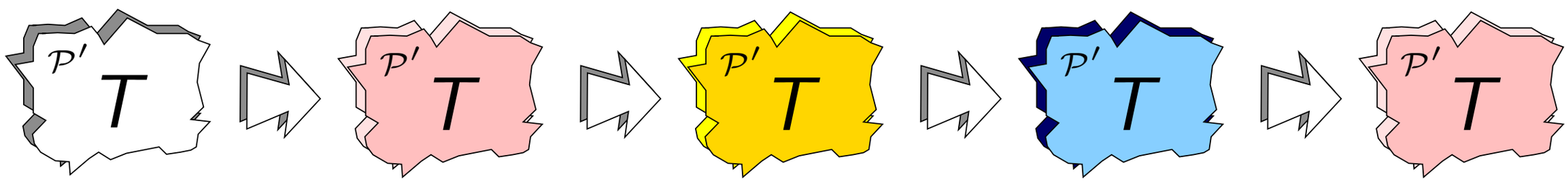
- ✓ Static obfuscation transforms the code prior to execution
- ✓ Dynamic obfuscation algorithms transform the program at runtime
- ✓ Static obfuscation counter attacks by static analysis
- ✓ Dynamic obfuscation counter attacks by dynamic analysis
- ✓ A dynamic obfuscatory runs in two phases:
  - At *compile time* transform the program to an initial configuration and add a runtime code transformer
  - At *runtime*, intersperse the execution of the program with calls to the code transformer
- ✓ A dynamic obfuscatory turns a normal program into a *self-modifying program*

Dynamic Obfuscation



- ✓ Transformer I creates the initial configuration P’
- ✓ T is the runtime obfuscation embedded in P’

Dynamic Obfuscation

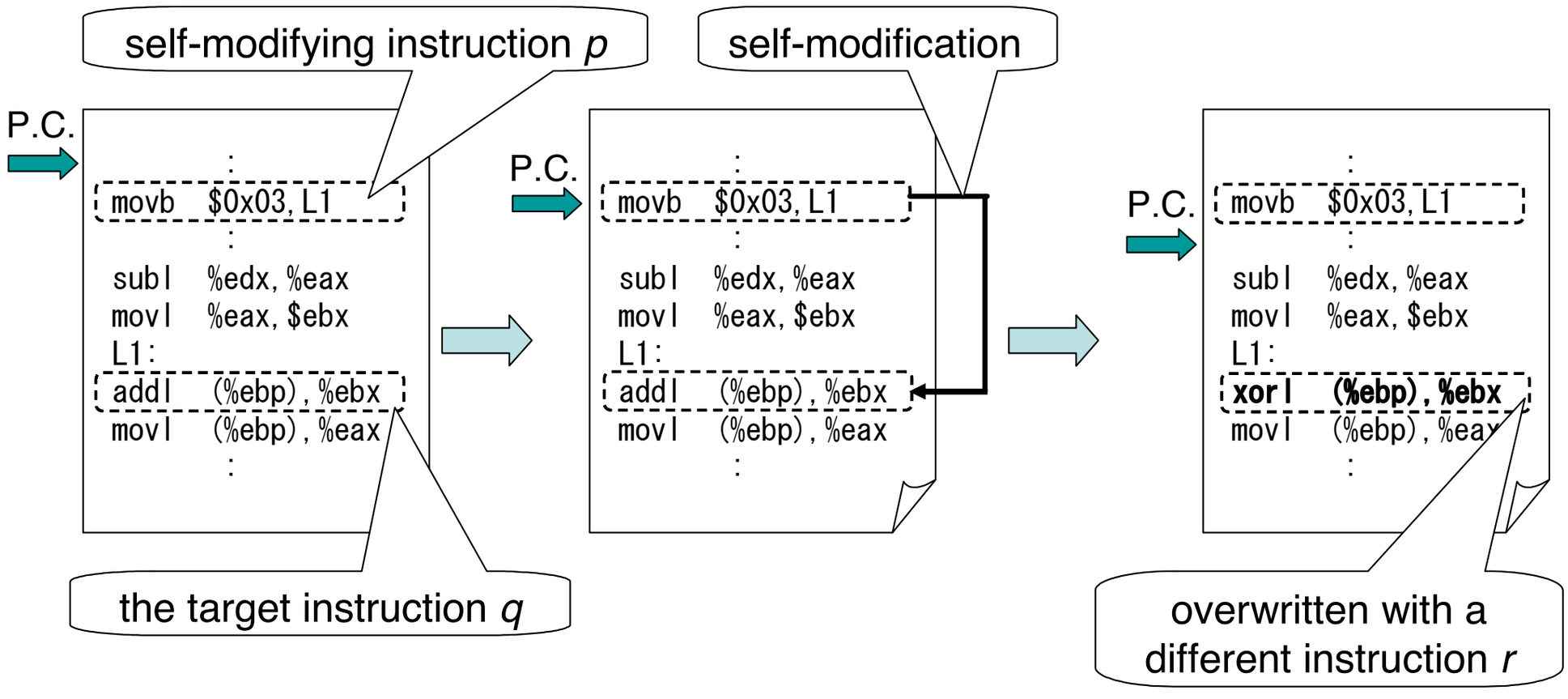


- ✓ Transformer T continuously modifies P’ at runtime
- ✓ We’d like an infinite, non-repeating series fo configurations
- ✓ In practice configurations repeat

Exploiting self-modification mechanism for program protection

Self-modification mechanism

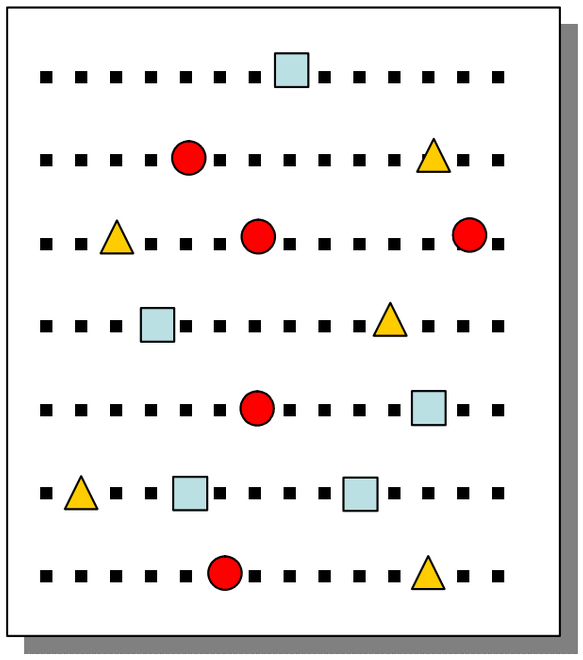
- ✓ Idea: add *a self- modification mechanism* to the original program, to increase the cost of understanding the original program.
- ✓ In the self-modification mechanisms, an instruction **p** in the program replaces another instruction **q** in the same program with a different instruction **r** at run-time.



Exploiting self-modification mechanism for program protection COMPAC 2003

Self-modification mechanism

- ✓ Our approach primarily consists of two parts:
  - ✓ Firstly, we camouflage many of the original instructions by *dummy instructions.*
  - ✓ Secondly, to assure the correctness of the original program, we add self-modifying instructions that *replace the dummy instructions with the original ones* within a certain period of execution.



- ● an instruction which is the target of camouflage
- □ a routine which writes the original instruction at run-time
- △ a routine which writes the dummy instruction at run-time

Self-modification mechanism

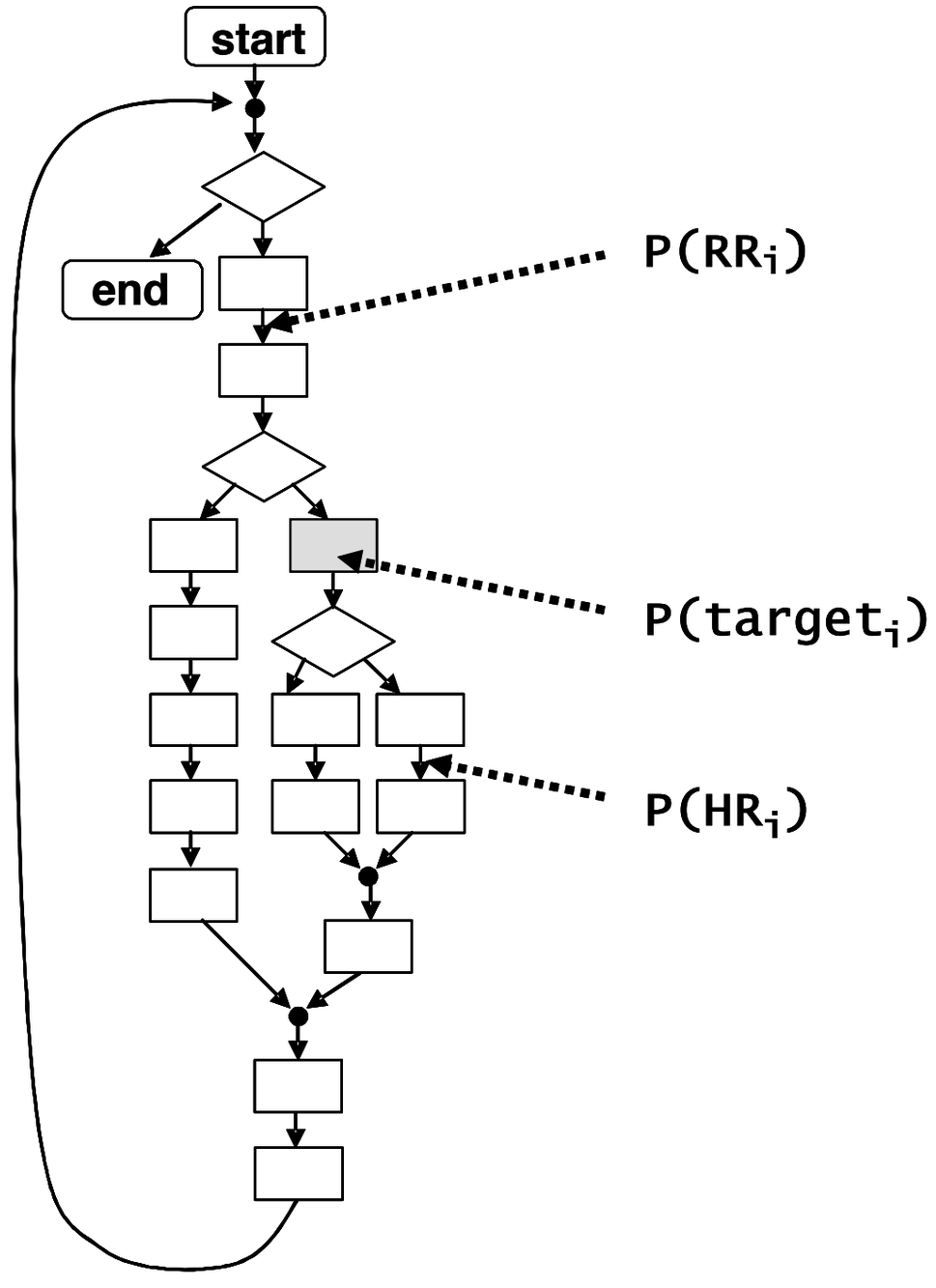
- ✓ Thus, by using the self-modification mechanism, we can *camouflage* the original instruction with a dummy instruction. If a cracker reads the camouflaged part as it is, he fails to understand the original instruction.
- ✓ The program protected by this method is quite hard to be understood by static analysis, and it is difficult for crackers to cancel the protection, since the dummy instructions are scattered over the program.
- ✓ A *target instruction* is an (original) instruction that we hide using the self-modification mechanism
- ✓ A *hiding routine (HR)* is a set of instructions which camouflage a target instruction with a dummy instruction.
- ✓ A *restoring routine (RR)* is a set of instructions which uncamouflage an original target instruction hidden by a dummy instruction.

Self-modification mechanism

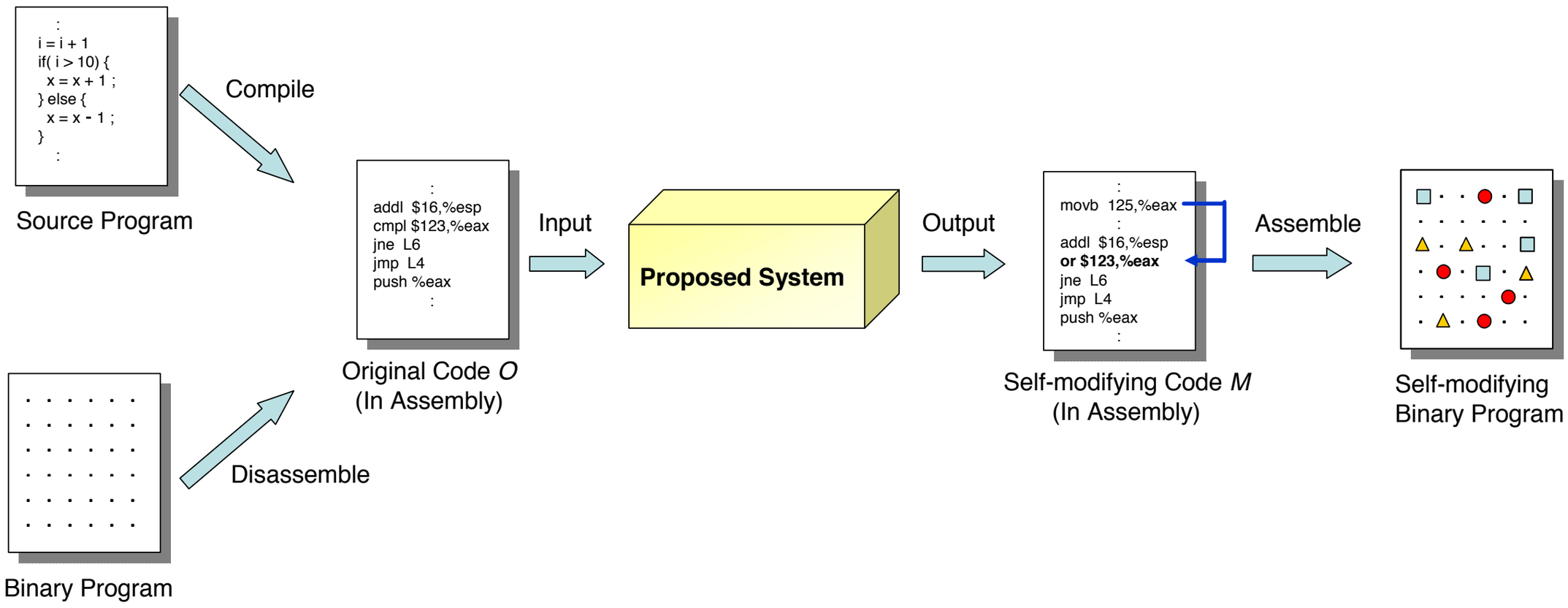
- ✓ A dummy instruction is obtained by changing the content of **`target`**$_i$, so that the *operation code* or the *operand* will be different
- ✓ Let us consider the following instruction as **`target`**$_i$
  - ✓ (Hex Representation) `03 5D F4`
  - ✓ (Assembly Representation) `addl -12(%ebp),%ebx`
- ✓ As for a possible candidate, we choose **`xorl`**, which is changed first byte from "`03`" to "`33`" , as shown below:
  - ✓ (Hex Representation) `33 5D F4`
  - ✓ (Assembly Representation) `xorl -12(%ebp),%ebx`

Self-modification mechanism

- ✓ Determine the positions of `targeti`, `RRi` and `HRi`.
- ✓ `P(targeti)` is randomly selected by the system.
- ✓ `P(RRi )` and `P(HRi )` are determined that they will satisfy the following three conditions, which are necessary for a program not to cause malfunction by adding a self- modifying function.
  - `P(RRi)` must exist on every control flow path from the program entry to the `P(targeti)`.
  - `P(HRi)` must not exist on every control flow path from `P(RRi)` to `P(targeti)`.
  - `P(RRi)` must exist on every control flow path from `P(HRi)` to `P(targeti)`.



Self-modification mechanism



Self-modification mechanism

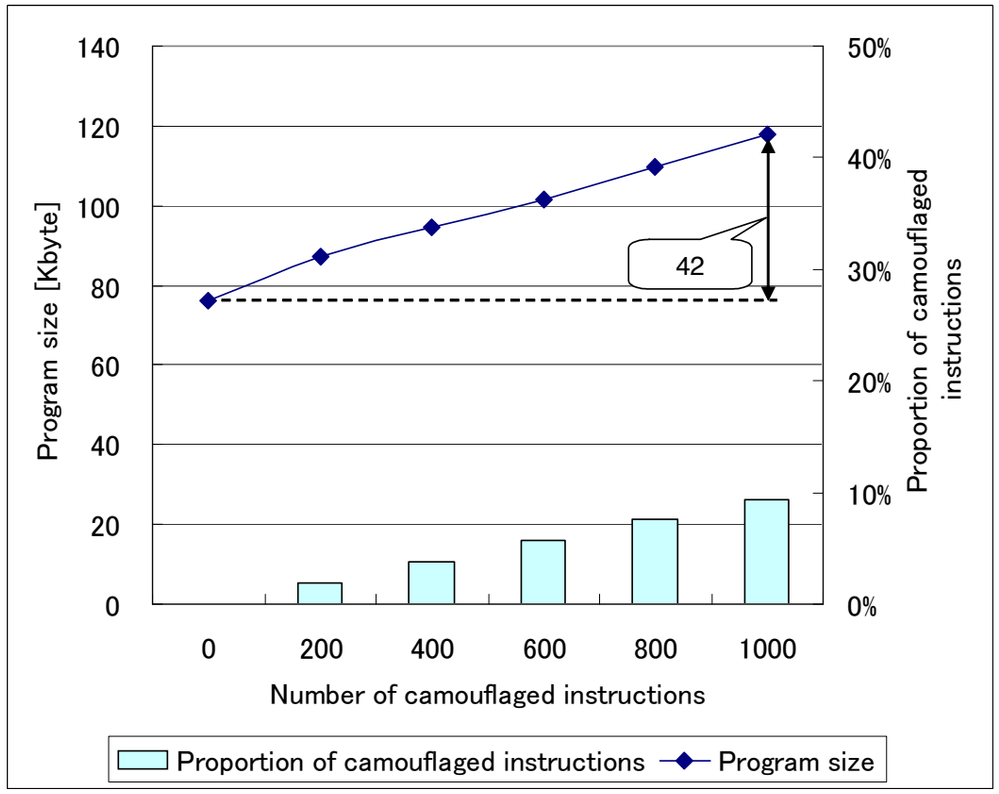


Figure 15. Impacts on program size

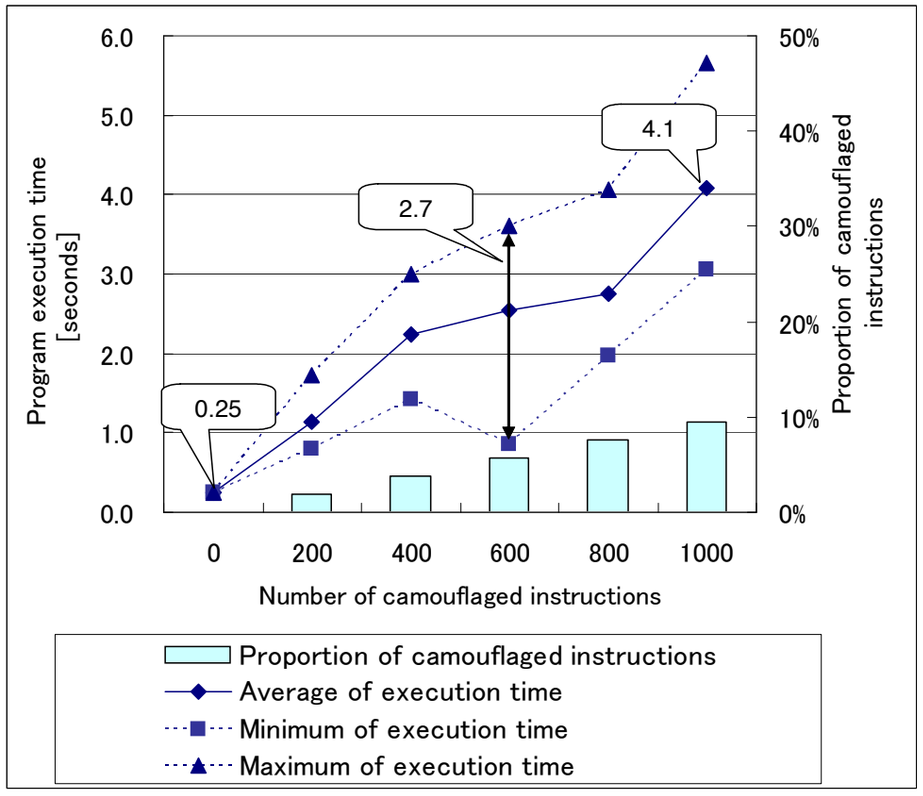


Figure 16. Impacts on program execution time

- ✓ The self-modification mechanism imposes a significant performance overhead compared with the size overhead.
- ✓ The proposed method should be applied with a careful consideration on the target program itself and the objective of the protection.

Exploiting self-modification mechanism for program protection COMPAC 2003

Virtualization

Virtualization

- ✓ Virtualization-obfuscation protects a program from manual or automated analysis by compiling it into bytecode for a randomized virtual architecture and attaching a corresponding interpreter.
- ✓ Hence, an equivalent but protected version of the original program can be generated by using just the interpreter as code and storing the translated bytecode as data.
- ✓ Static analysis appears to be helpless on such programs, where only the code of the interpreter is directly visible and the byte code is an incompressible data block.
- ✓ Very used by malware. Due to architecture randomization, the bytecode and interpreter can vary greatly from one protected instance to the other, preventing the generation of reliable malware signatures.

Virtualization

```
1  int code = { ... };
2  int data = { ... };
3
4  void interpret() {
5    int vpc = 0, op1, op2;
6    while (true) {
7      switch(code[vpc]) {
8        case 03: // increment
9          op1 = code[vpc + 1];
10         data[op1]++;
11         vpc += 2;
12         break;
13       case 08: // conditional jump
14         op1 = code[vpc + 1];
15         op2 = code[vpc + 2];
16         if (data[op1] <= 0)
17           vpc += data[op2]
18         else
19           vpc += 3;
20         break;
21       case 18: // call ext. function
22         op1 = code[vpc + 1];
23         apiCall(data[op1]);
24         vpc += 2;
25         break;
26       case 52: // assignment
27         op1 = code[vpc + 1];
28         op2 = code[vpc + 2];
29         data[op1] = data[op2];
30         vpc += 3;
31         break;
32       default: // halt
33         return;
34     } // end switch
35   } // end while
36 }
```

- ✓ Programs protected by virtualization-obfuscation all share the same general structure of looping over a large switch statement that distinguishes individual bytecode instructions

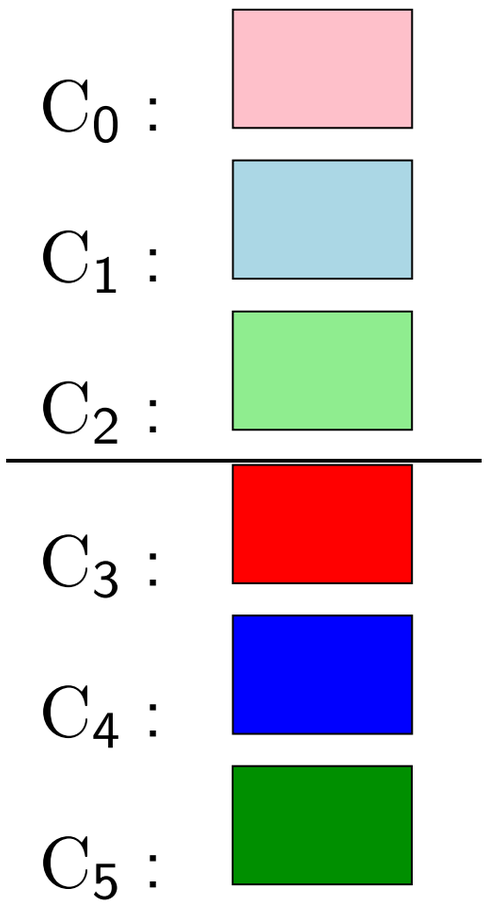
Aucsmith’s Algorithm

Aucsmith's Algorithm for dynamic obfuscation

- ✓ First serious attempt to software protection 1996
- ✓ The algorithm keeps the *program in constant flux* by splitting it up in pieces and cyclically moving these pieces around, xoring them with each other.
- ✓ An adversary that tries to examine the protected program under a debugger will find that *only some pieces of the program are in cleartext*, and that the program exhibits a very unusual address trace

D. Aucsmith. Tamper resistant software: an implementation, patent 5892899, 1999

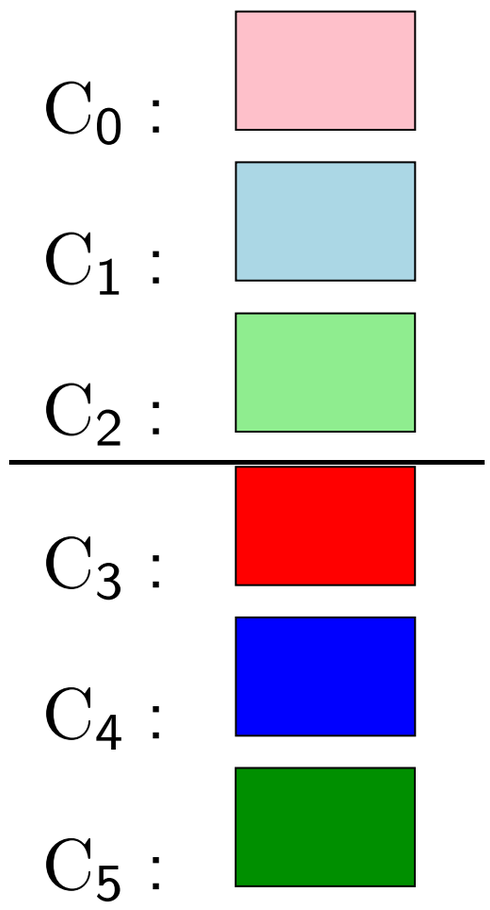
Aucsmith's Algorithm for dynamic obfuscation



- ✓ Consider a function **`f`** that has been splitted into six pieces (cells)
- ✓ The cells are divided into two regions in memory: upper and lower

D. Aucsmith. Tamper resistant software: an implementation, patent 5892899, 1999

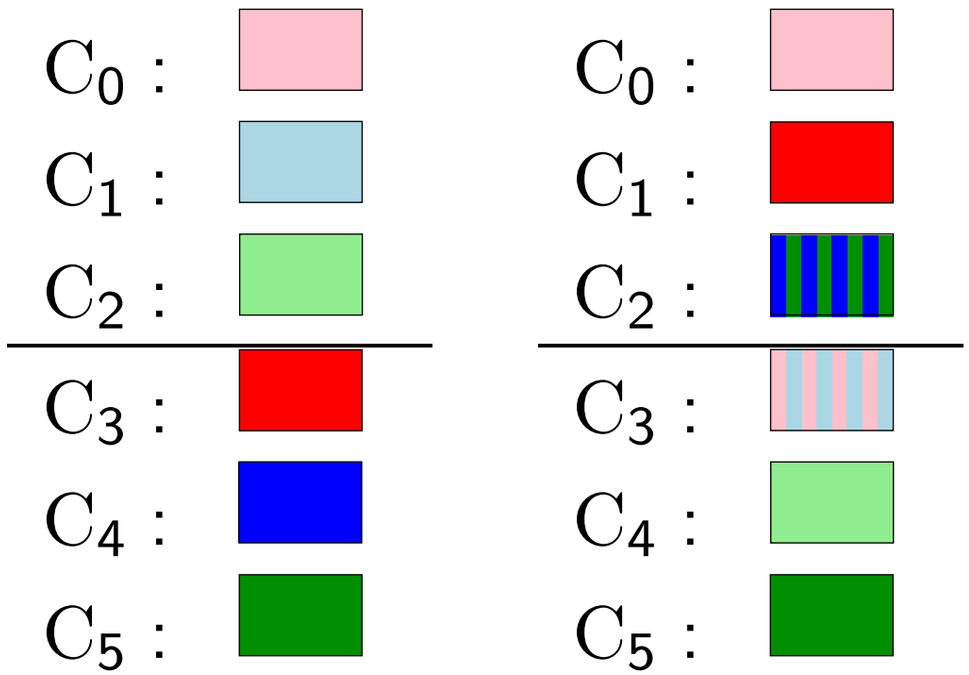
Aucsmith's Algorithm for dynamic obfuscation



- ✓ The execution proceeds in *rounds* such that during each round every cell in upper memory is xored with a cell in lower memory
- ✓ At each round a new cell becomes in the clear and execution jumps there

D. Aucsmith. Tamper resistant software: an implementation, patent 5892899, 1999

Aucsmith's Algorithm for dynamic obfuscation

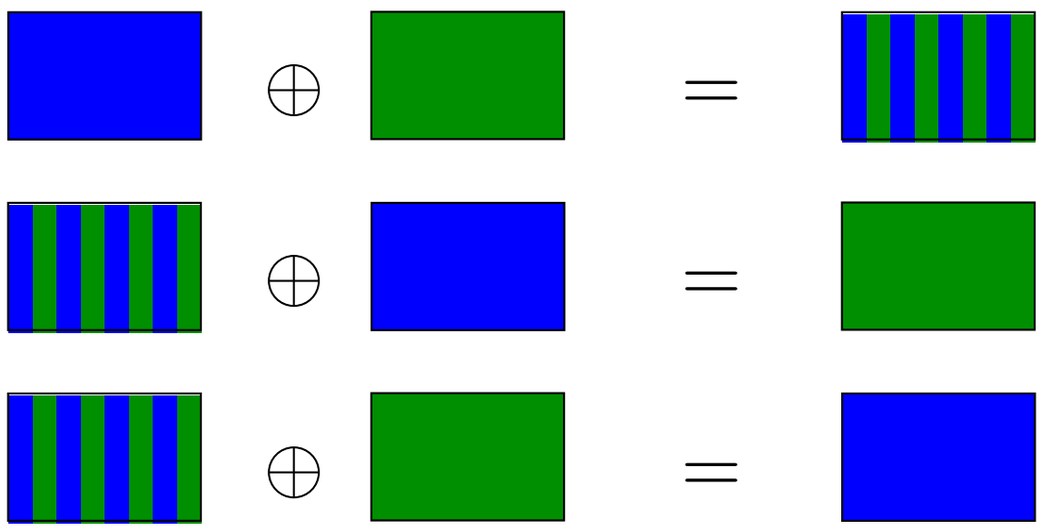


orig M_0

- ✓ During even rounds, the cells in upper memory are xored into cells in lower memory
- ✓ During odd rounds, lower cells are xored into upper memory cells
- ✓ After a certain number of rounds , the function returns to its initial configuration
- ✓ M_0 is the *initial obfuscated configuration*: C_0 is in clear text and is where the execution starts, C_1, C_4, C_5 are also in clear text but some of them are not in the same location as in the unobfuscated code. C_2 and C_3 are not in clear text.

D. Aucsmith. Tamper resistant software: an implementation, patent 5892899, 1999

XOR

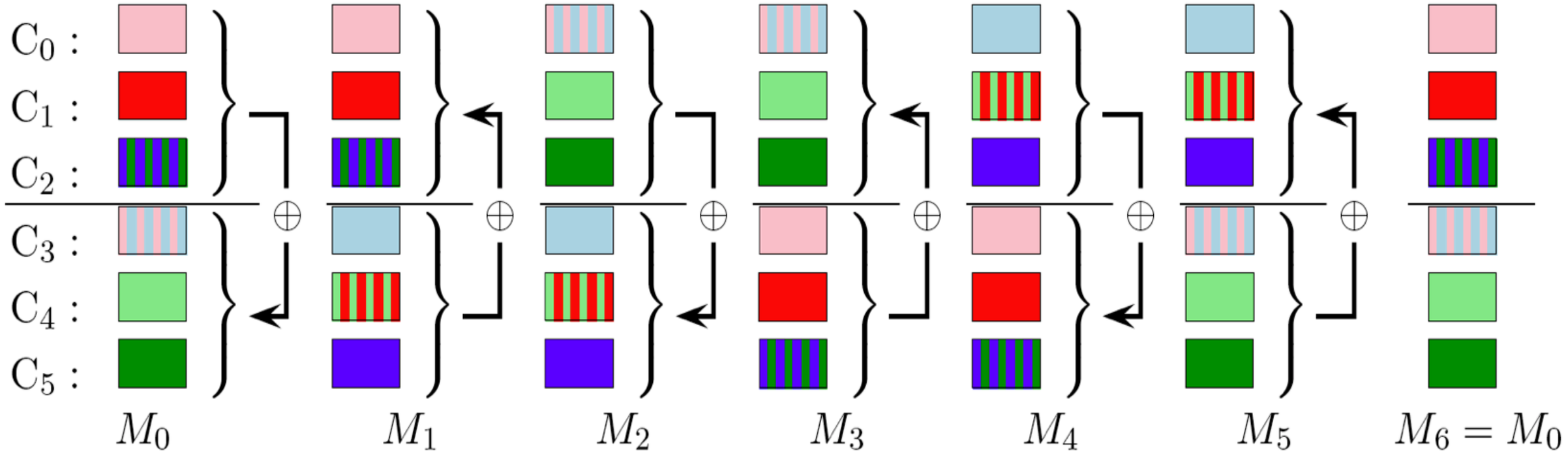


- ✓ (A XOR B) XOR A = B
- ✓ (A XOR B) XOR B = A

D. Aucsmith. Tamper resistant software: an implementation, patent 5892899, 1999

Aucsmith's Algorithm for dynamic obfuscation

- ✓When C_0 has finished executing every cell in upper memory is xored with every cell in lower memory, leading to configuration M_1
- ✓C_3 is now in clear text and execution proceeds from there
- ✓…



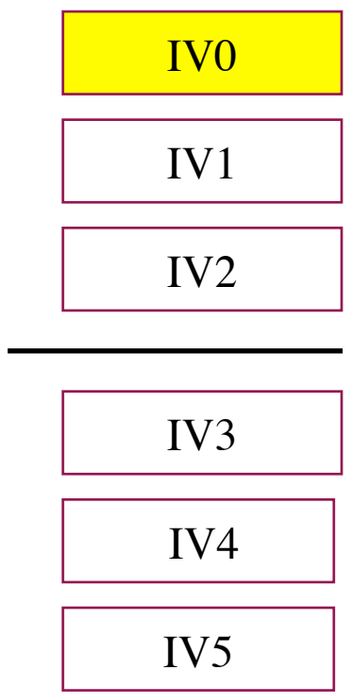
D. Aucsmith. Tamper resistant software: an implementation, patent 5892899, 1999

Aucsmith's Algorithm for dynamic obfuscation

✓At the beginning C_0 is in cleartext

initial configuration

Cell in execution



IVi are the initial values

E1

C0 = IV0
C1= IV1
C2 = IV2
C3 = IV3
C4 = IV4
C5 = IV5

Current state of the memory cells as a function of their initial value

E2

IV0 = P0

How to recover piece Pi of the original program from the initial values IVi

next

C0 —> ?
C1 —> ?
C2 —> ?
C3 —> ?
C4 —> ?
C5 —> ?

While reconstructing the cell we define the next function

D. Aucsmith. Tamper resistant software: an implementation, patent 5892899, 1999

Aucsmith's Algorithm for dynamics obfuscation

✓We consider a very simple mutation function:

During even rounds, each cell **`i`** in upper memory is xored with cell **`i + 3`** in lower memory, and during odd rounds, cell **`i`** in lower memory is xored with cell **`i-3`** in upper memory

D. Aucsmith. Tamper resistant software: an implementation, patent 5892899, 1999

Aucsmith's Algorithm for dynamics obfuscation

Round 1

	E1	E2	next
IV0			
IV1	C0 = IV0	IV0 = P0	C0 —> C3
IV2	C1= IV1 C2 = IV2	IV0 xor IV3 = P1	C1 —> ? C2 —> ?
IV0 xor IV3	C3 = IV0 xor IV3		C3 —> ?
IV1 xor IV4	C4 = IV1 xor IV4		C4 —> ?
IV2 xor IV5	C5 = IV2 xor IV5		C5 —> ?

D. Aucsmith. Tamper resistant software: an implementation, patent 5892899, 1999

Aucsmith's Algorithm for dynamics obfuscation

Round 2

	E1	E2	next
IV3			
IV4	C0 = IV3	IV0 = P0	C0 —> C3
IV5	C1= IV4	IV0 xor IV3 = P1	C1 —> ?
	C2 = IV5	IV4 = P2	C2 —> ?
IV0 xor IV3	C3 = IV0 xor IV3		C3 —> C1
IV1 xor IV4	C4 = IV1 xor IV4		C4 —> ?
IV2 xor IV5	C5 = IV2 xor IV5		C5 —> ?

D. Aucsmith. Tamper resistant software: an implementation, patent 5892899, 1999

Aucsmith's Algorithm for dynamics obfuscation

Round 3

	E1	E2	next
IV3			
IV4	C0 = IV3	IV0 = P0	C0 —> C3
IV5	C1= IV4	IV0 xor IV3 = P1	C1 —> C4
	C2 = IV5	IV4 = P2	C2 —> ?
IV0	C3 = IV0	IV1 = P3	C3 —> C1
IV1	C4 = IV1		C4 —> ?
IV2	C5 = IV2		C5 —> ?

D. Aucsmith. Tamper resistant software: an implementation, patent 5892899, 1999

Aucsmith's Algorithm for dynamics obfuscation

Round 4

	E1	E2	next
IV0 xor IV3	C0 = IV0 xor IV3	IV0 = P0	C0 —> C3
IV1 xor IV4	C1= IV1 xor IV4	IV0 xor IV3 = P1	C1 —> C4
IV2 xor IV5	C2 = IV2 xor IV5	IV4 = P2	C2 —> ?
---	C3 = IV0	IV1 = P3	C3 —> C1
IV0	C4 = IV1	IV2 xor IV5 = P4	C4 —> C2
IV1	C5 = IV2		C5 —> ?
IV2			

D. Aucsmith. Tamper resistant software: an implementation, patent 5892899, 1999

Aucsmith's Algorithm for dynamics obfuscation

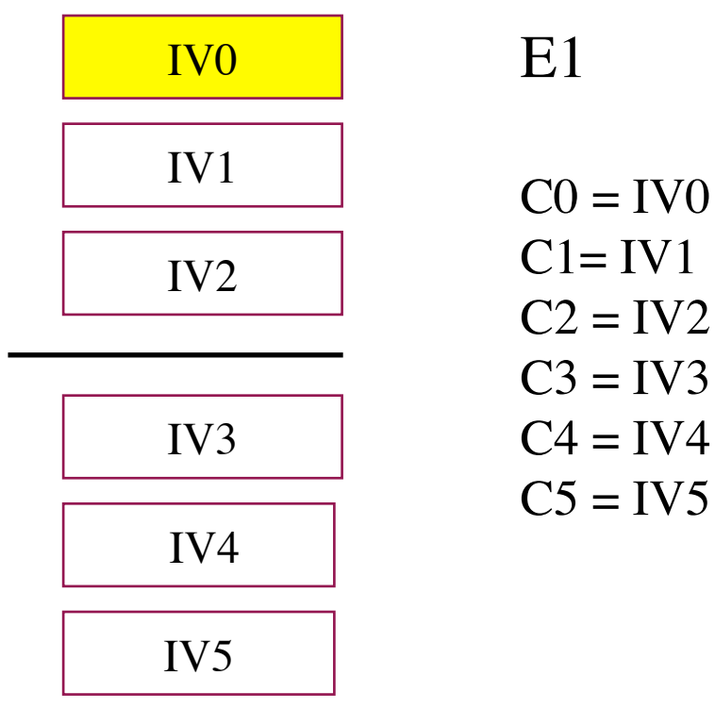
Round 5

	E1	E2	next
IV0 xor IV3	C0 = IV0 xor IV3	IV0 = P0	C0 —> C3
IV1 xor IV4	C1= IV1 xor IV4	IV0 xor IV3 = P1	C1 —> C4
IV2 xor IV5	C2 = IV2 xor IV5	IV4 = P2	C2 —> C5
IV3	C3 = IV3	IV1 = P3	C3 —> C1
IV4	C4 = IV4	IV2 xor IV5 = P4	C4 —> C2
IV5	C5 = IV5	IV5 = P5	C5 —> ret

D. Aucsmith. Tamper resistant software: an implementation, patent 5892899, 1999

Aucsmith's Algorithm for dynamics obfuscation

Initial configuration



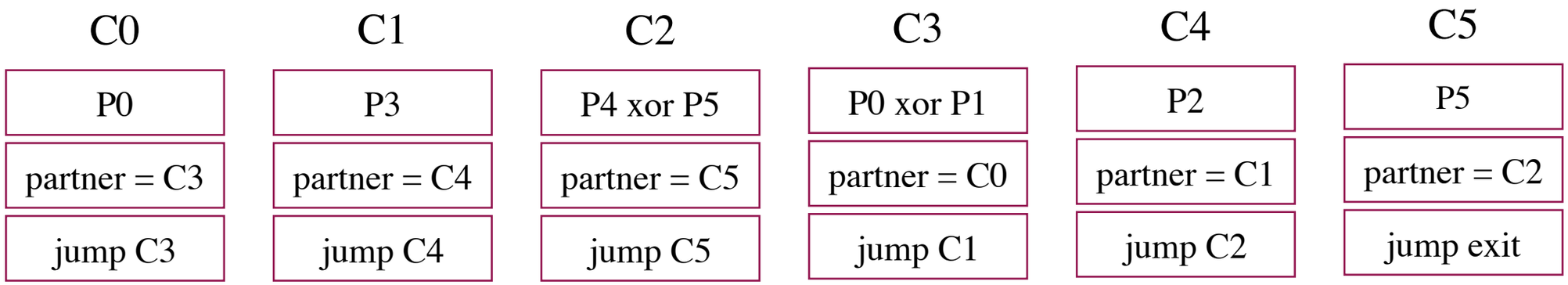
D. Aucsmith. Tamper resistant software: an implementation, patent 5892899, 1999

Aucsmith's Algorithm for dynamics obfuscation

- ✓ The purpose of this exercise was to construct the set of equations E2. Given E2 we can determine the initial values of the cells

IV0 = P0
IV0 xor IV3 = P1
IV4 = P2
IV1 = P3
IV2 xor IV5 = P4
IV5 = P5

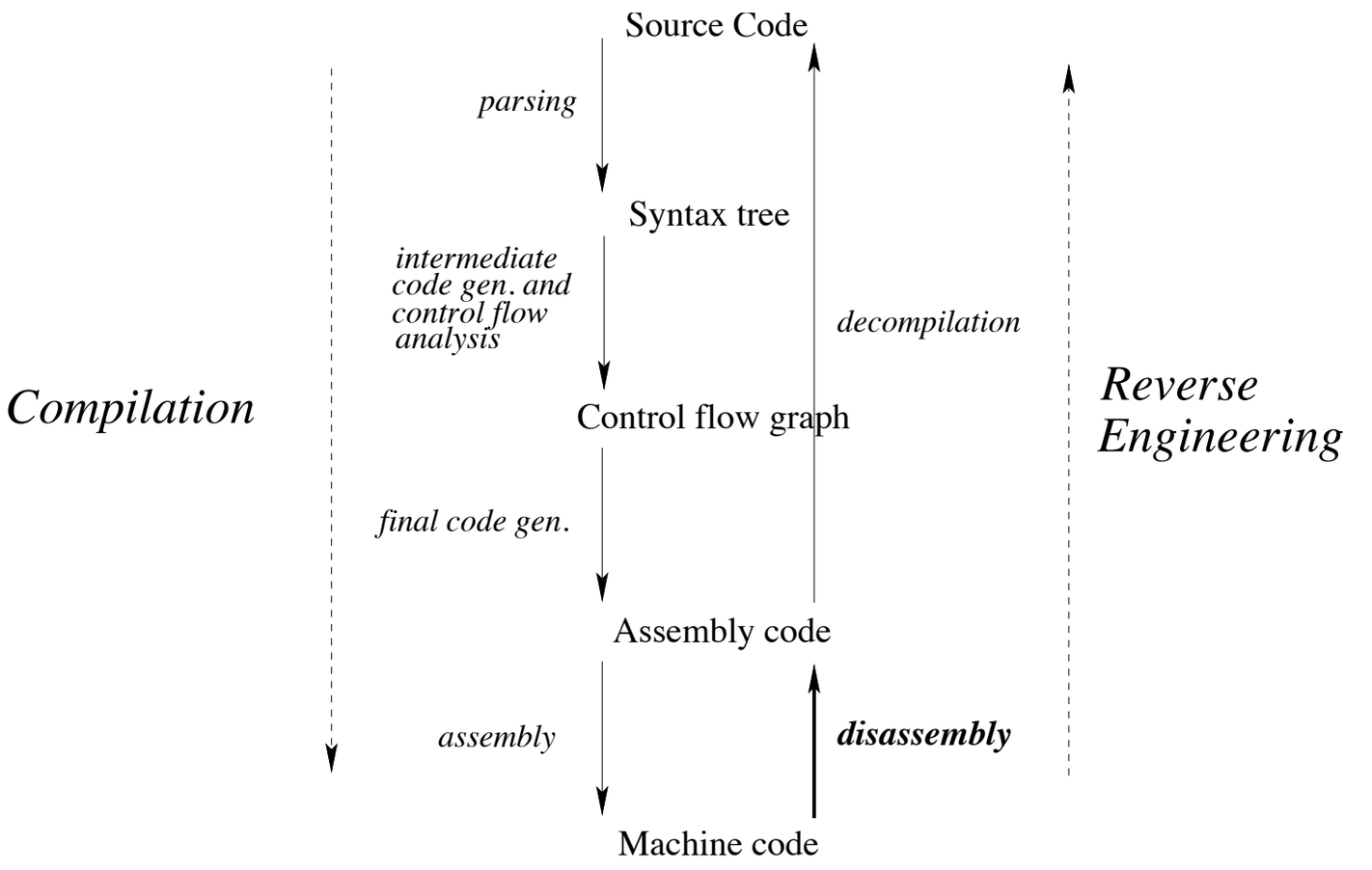
IV0 = P0
IV1 = P3
IV2 = P4 xor P5
IV3 = P0 xor P1
IV4 = P2
IV5 = P5



D. Aucsmith. Tamper resistant software: an implementation, patent 5892899, 1999

Thwarting Disassembly

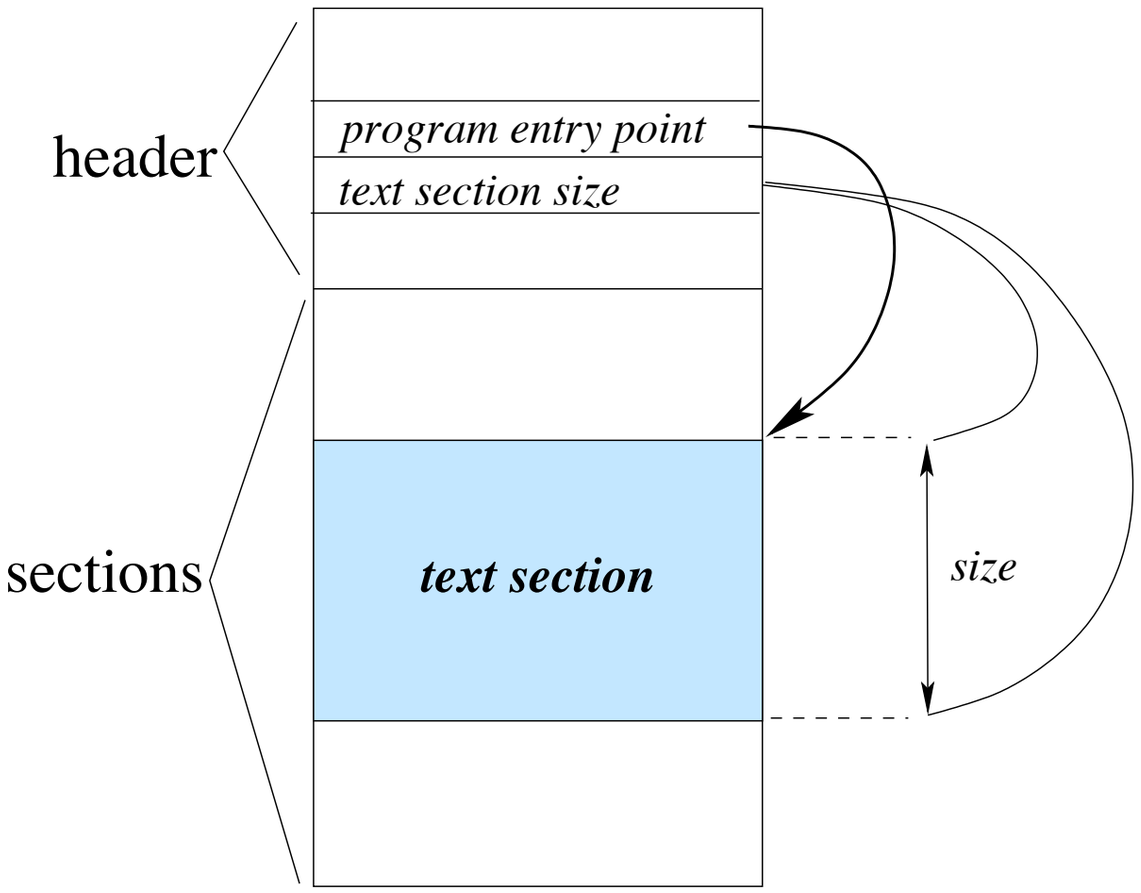
Thwarting Disassembly



- ✓ Most of the existing code obfuscation techniques focus on various aspects of decompilation
- ✓ Another possibility is to focus on the disassembly process

Obfuscation of Executable Code to Improve Resistance to Static Disassembly. Linn and Debray. CCS 2003

Thwarting Disassembly



✓ *machine code files*

consists of a number of different sections and the header

✓ the disassembler takes this file and reconstructs the corresponding assembly instructions

Obfuscation of Executable Code to Improve Resistance to Static Disassembly. Linn and Debray. CCS 2003

Thwarting Disassembly

- ✓ *Confuse* as much as possible the position of the *instruction boundaries* in a program
- ✓ Compare the set of instructions start addresses identified by a static disassembler and the actual instruction addresses encountered when the program is executed (*maximize this difference*)
- ✓ Self-reparing disassembly: even when a disassembly error occurs the disassembly ends up re-syncronizing with the actual instruction stream
- ✓ Delay the self preparing process as much as possible

Obfuscation of Executable Code to Improve Resistance to Static Disassembly. Linn and Debray. CCS 2003

Thwarting Disassembly

- ✓ Introduce disassembly errors by inserting *junk bytes* where the disassembler expects code:
  - Junk bytes are partial instructions
  - Junk bytes are never executed instructions at run time
- ✓ *Candidate block* is a basic block that can have junk bytes inserted before and we want to ensure that such junk bytes are *not reachable* during execution
  - A candiate block cannot have execution fall into it
  - The preceding BB ends with a direct jump or a function call
- ✓ Given a candidate block we have to determine the junk bytes to insert before it in order to *delay the self-repaving process* as much as possible (we insert the first k byte of a given instruction I where k maximizes the delay)

Obfuscation of Executable Code to Improve Resistance to Static Disassembly. Linn and Debray. CCS 2003

Linear Sweep

```
global startAddr, endAddr;
proc Linear(addr)
begin
    while(startAddr<addr<endAddr) do
      I := decode instr at address addr
      addr += length(I)
    od
end
```

```
proc main()
begin
    ep := program entry point;
    size := text section size;
    sratAddr := ep;
    endAddr := ep + size
    Linear(ep);
end
```

Begins disassembly at the *entry point* of the program and sweeps through the entire text section disassembling one instruction at the time

weakness: misinterpretation of data in the text section

Obfuscation of Executable Code to Improve Resistance to Static Disassembly. Linn and Debray. CCS 2003

Recursive Traversal

```
global startAddr, endAddr;
proc Recursive(addr)
begin
    while(startAddr<addr<endAddr) do
      if (addr has been visited) return;
      I := decode instr at address addr;
      mark addr as visited;
  if (I is branch or function call)
        for each possible target t do
            Recursive(t);
        od
        return;
      else addr += length(I);
od
end

proc main()
begin
    ep := program entry point;
    size := text section size;
    sratAddr := ep;
    endAddr := ep + size
    Recursive(ep);
end
```

Take into account the control flow.

When it encounters a branch instruction it determines the control flow successors of that instructions and continues the disassembly from there

weakness: it may not be easy to determine statically the successors

Obfuscation of Executable Code to Improve Resistance to Static Disassembly. Linn and Debray. CCS 2003

Thwarting Linear Sweep

- ✓ ~~Attack limitation~~: unable to distinguish data embedded in the instruction stream
- ✓ Junk bytes insertion: 26% - 30% of instructions are incorrectly disassembled (candidate blocks are too far ~30 instructions)
- ✓ Increase the number of candidate blocks through branch flipping

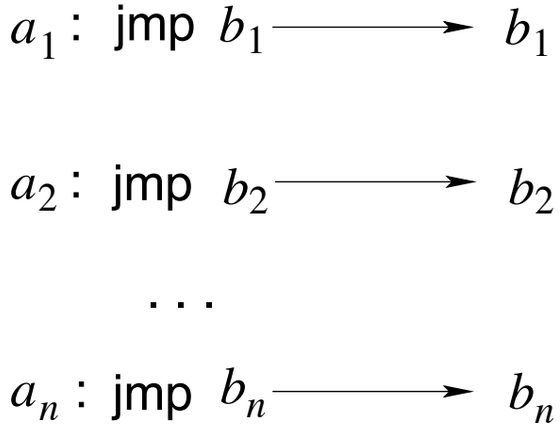
```
bccAddr          bcc L
L:…              jmpAddr
                 L:…
```

- ✓ **L** is now a candidate block, **cc** is the complementary condition of **cc**
- ✓ 70% of the instructions are incorrectly disassembled

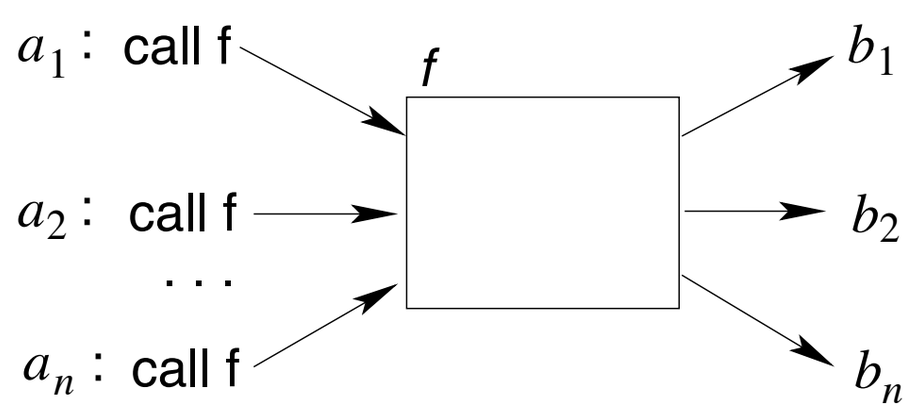
Obfuscation of Executable Code to Improve Resistance to Static Disassembly. Linn and Debray. CCS 2003

Thwarting Recursive Traversal

- ✓ Assumption: a function returns to the instruction following the call instruction
- ✓ A branch function is a function such that, whenever it is called from one of the locations **`ai`**, causes control to be transferred to the corresponding location **`bi`**



(a) Original code



(b) Code using a branch function

Obfuscation of Executable Code to Improve Resistance to Static Disassembly. Linn and Debray. CCS 2003

Thwarting Recursive Traversal

- ✓ We can replace *n* unconditional branches in a program by calls to the branch function

$$
\begin{array}{l}
a_1 : \texttt{jmp}\ b_1 \\
\ldots \\
a_2 : \texttt{jmp}\ b_2 \\
\ldots \\
a_n : \texttt{jmp}\ b_n
\end{array}
\quad \Rightarrow \quad
\begin{array}{l}
a_1 : \texttt{call}\ f_\varphi \\
\ldots \\
a_2 : \texttt{call}\ f_\varphi \\
\ldots \\
a_n : \texttt{call}\ f_\varphi
\end{array}
$$

- ✓ Branch functions do not behave like normal functions, in that they typically do not return to the instruction following the call instruction, but instead they branch to some other location in the program that depends, in general on where it was called from

Obfuscation of Executable Code to Improve Resistance to Static Disassembly. Linn and Debray. CCS 2003

Thwarting Recursive Traversal

- ✓ Branch functions allow us to:
  - *Obscure the control flow* of the original program
  - *Confuse disassembly* by inserting junk bytes at the points immediately after the call to the branch function
  - The resulting code is, more expensive (depending on the implementation of branch functions) and therefore such transformation is not applied to *hot code* (code frequently executed)
- ✓ Linear sweep: 75% of instructions are incorrectly disassembled
- ✓ Recursive traversal: 40% of instructions are incorrectly disassembled
- ✓ Average 52% speed penalty

Obfuscation of Executable Code to Improve Resistance to Static Disassembly. Linn and Debray. CCS 2003

Obstruct Static & Dynamic RE

- ✓ **Static Disassembly**
  - ✓ By inserting *indirect jumps* that do not reveal their jump target until runtime and utilizing the concept of a *branching function* we make *static control flow reconstruction more difficult*.
- ✓ **Dynamic Analysis**
  - ✓ IDEA: the information that an attacker can retrieve from the analysis of a single run of the program with certain input, is useless for understanding the program behaviour on other inputs
  - ✓ This concept can be considered as *diversification of the CFG*

Obstruct Static Disassembly

- ✓ The assembly code of the software is split into small pieces, which we call *gadgets*.
- ✓ At the end of each gadget we add a jump back to the *branching function*.
- ✓ At runtime, this function calculates, based on the previously executed gadget, the virtual memory address of the following gadget and jumps there.
- ✓ *The calculation of the next jump target depends on the current gadget and on the history of executed gadgets.*
- ✓ We achieve this requirement by assigning a *signature* to each gadget. At runtime, the signatures of executed gadgets are summed up and this sum is used inside the branching function as input parameter for a lookup table that contains the address of the subsequent gadget.

Obstruct Static Disassembly

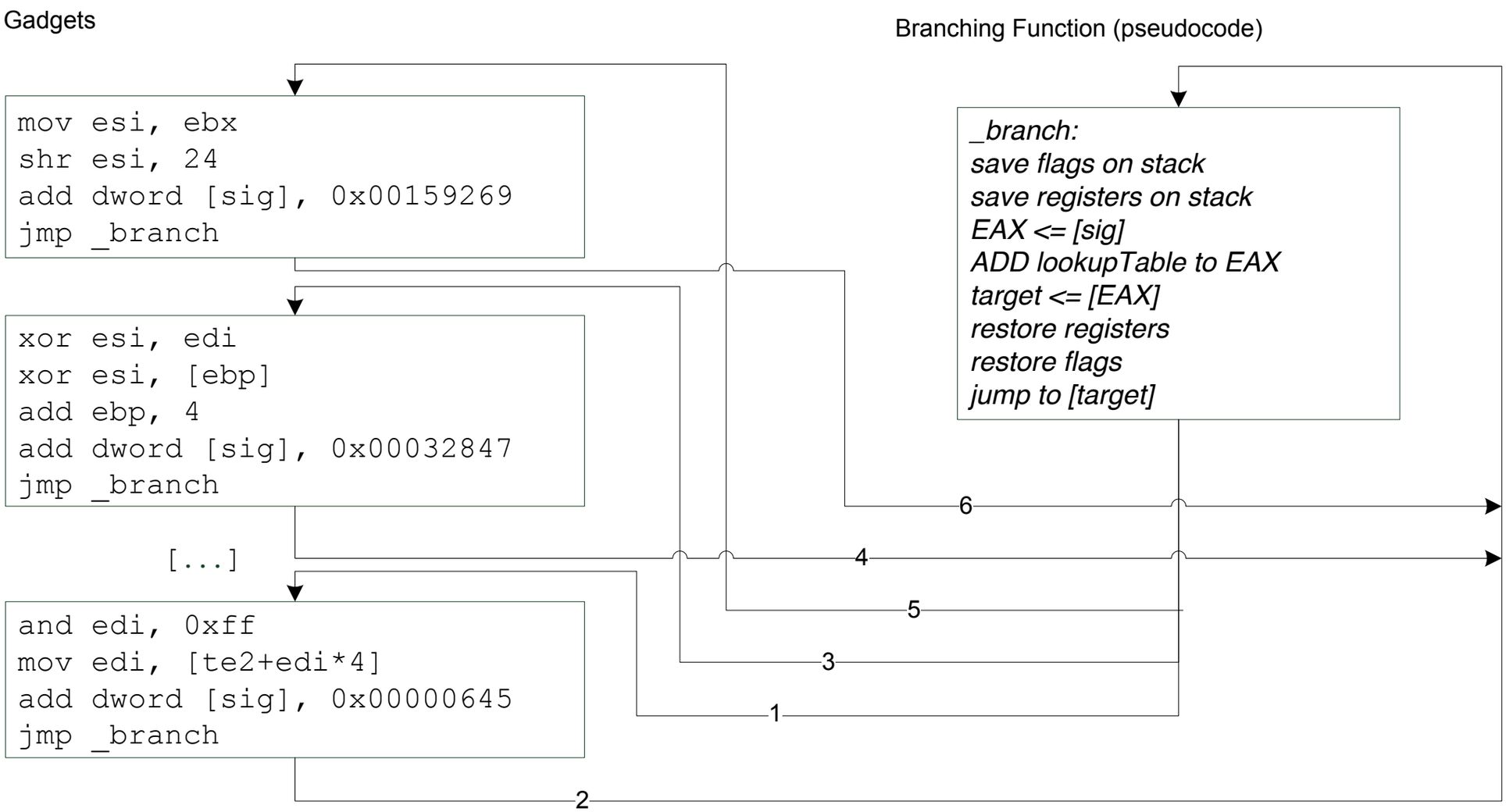


Fig. 1. Overall architecture of the obfuscated program: small code blocks (gadgets) are connected by a branching function.

Obstruct Dynamic Analysis

- ✓ Dynamic analysis, however, reveals all gadgets used in a single invocation of the software as well as their order.
- ✓ An attacker can easily remove the jumps to the branching function by just concatenating called gadgets in their correct order.
- ✓ By performing this task for several inputs, the attacker gets significant information on the software behavior.
- ✓ *To mitigate that risk, we diversify the control flow graph of the software so that it contains many more control flow paths than the original implementation*
- ✓ We diversify gadgets (i.e. add semantically identical but syntactical different gadgets to the code) and *add input dependent branches* so that different gadgets get executed upon running the software with different inputs.

Obstruct Dynamic Analysis

- ✓ We can symbolize this by a *gadget graph*, where the actual gadget code is stored in the edges that connect two nodes, which symbolize the state of a program.

- ✓ For every node, we create outgoing edges and fill them with gadgets. All outgoing edges of one node start with the same instruction and only differ in gadget length.

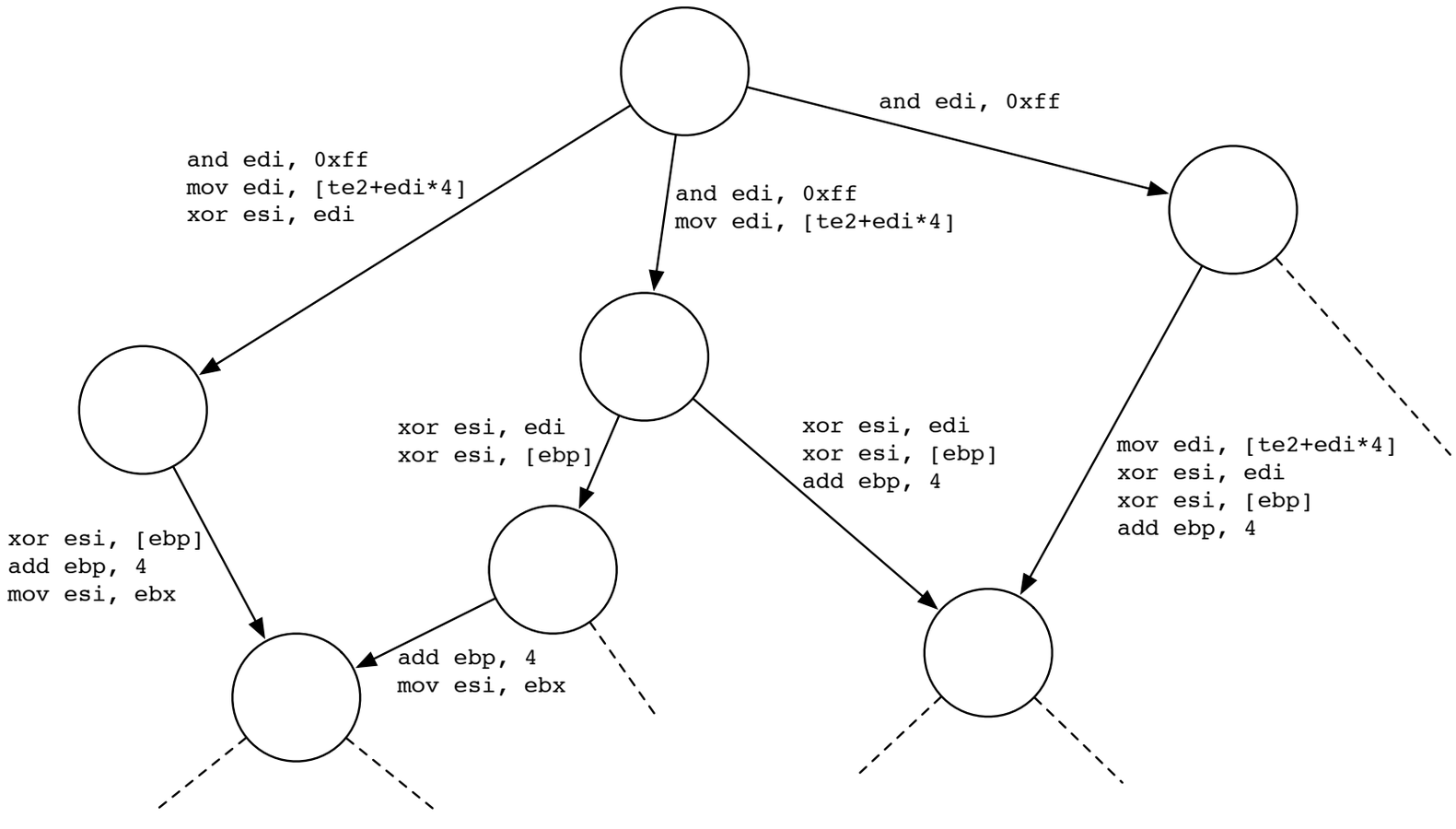


Fig. 2. Gadget graph.

Obstruct Dynamic Analysis

- ✓ Gadget diversification: Every path through the graph is a valid trace of the program. The branches are input dependent: based on the program's input the branching function decides which path through the graph has to be taken.
- ✓ For a logical connection between gadgets, we implement a path signature algorithm that uniquely identifies the currently executed node and all its predecessor
- ✓ In order to increase the security of the obfuscation, we prevent that a path that is valid for one input is also valid for other inputs.
  - ✓ We do this by modifying some instruction's operands and automatically compensate these modifications during runtime by corrective input data. (**specialize the path wrt the input**)
  - ✓ All paths through this graph are valid. However, because of the inserted modifications to operands, one specific path yields correct computation only for a specific input and fails otherwise

Obstruct Dynamic Analysis

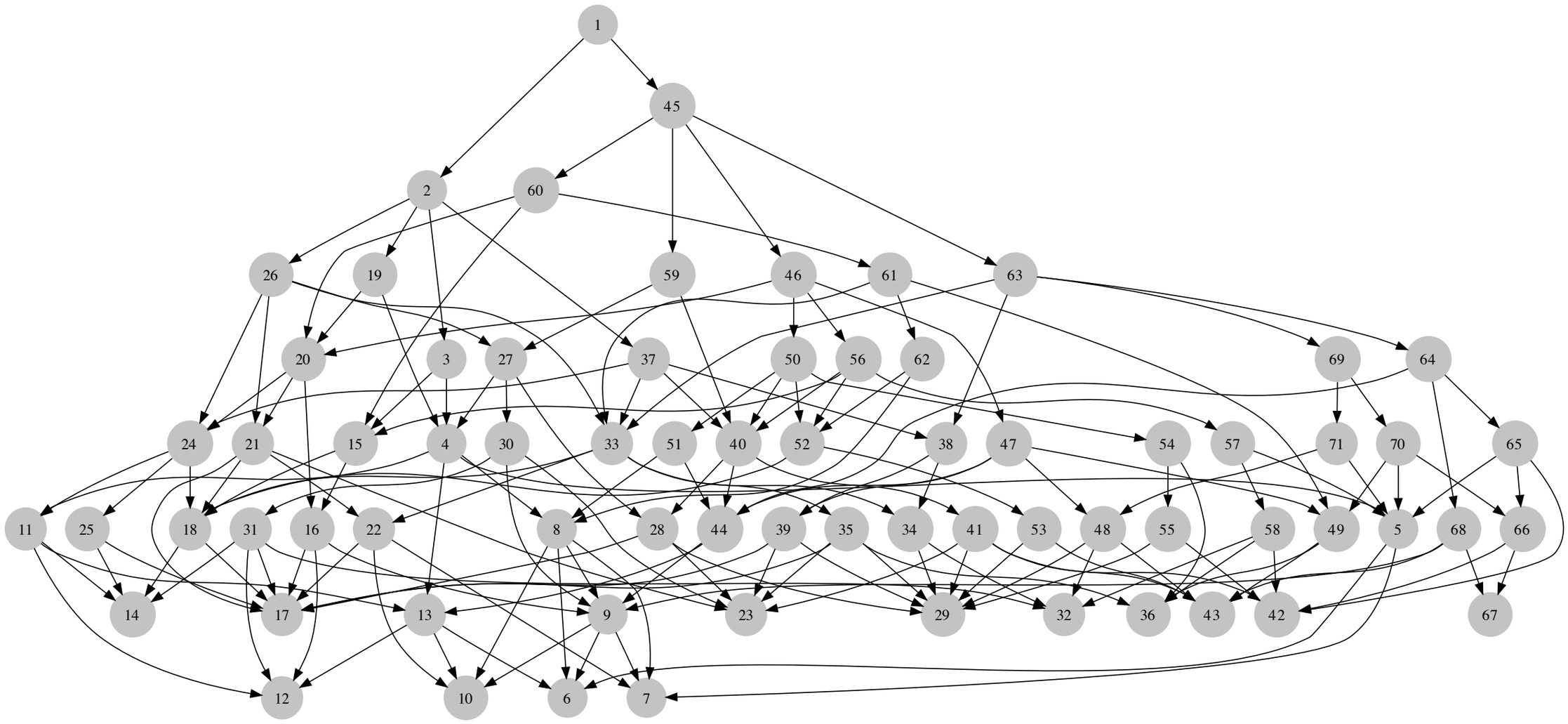


Fig. 3. Diversified control flow graph.

Gadget construction and diversification

- ✓ Automatic gadget construction
- ✓ Automatic gadget diversification
- ✓ Performances:
  - ✓ While the dynamic part of our approach accounts for an increase in memory space because of diversified copies of gadgets
  - ✓ Execution time heavily depends on the size and implementation of the branching function, as it inserts additional instructions.
  - ✓ The performance decreases with the number of gadgets, due to calls to the branching function, which are required to switch between gadgets.

Gadget construction and diversification

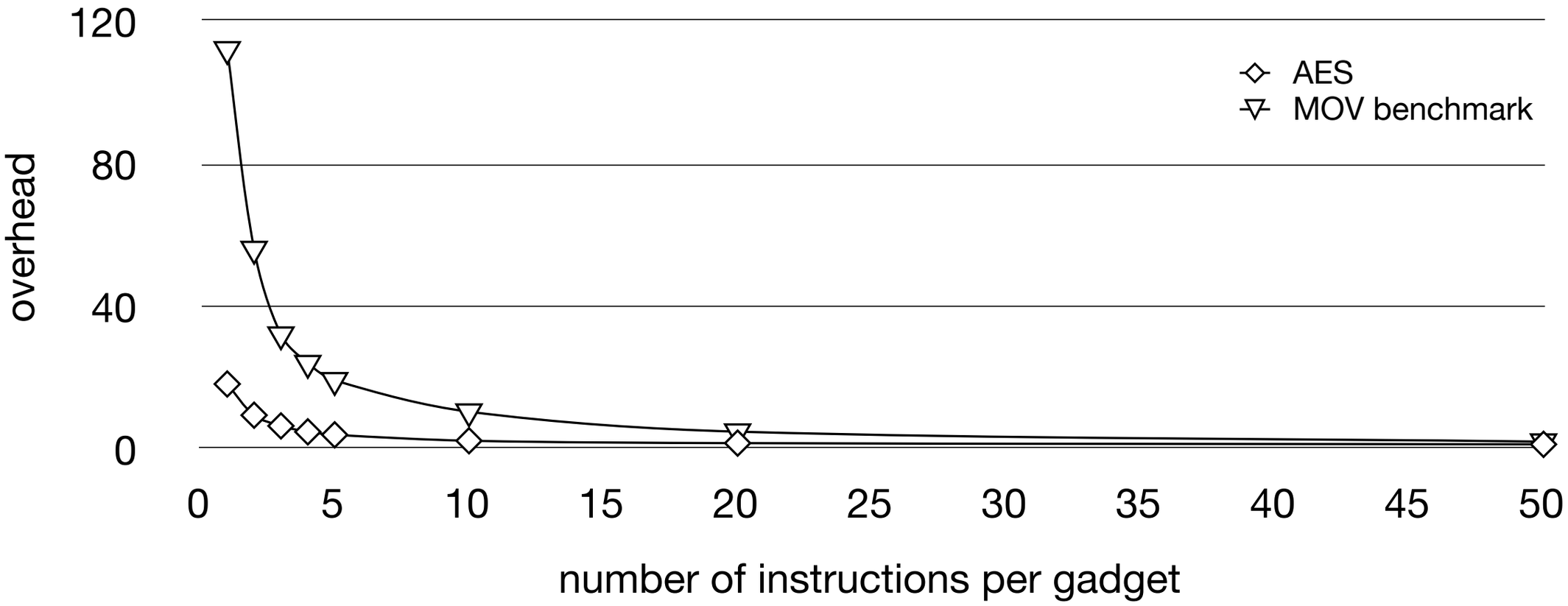


Fig. 6. Execution time for different gadget sizes.

Gadget construction and diversification

- ✓ The strength of the obfuscation is directly proportional to the number of gadgets
- ✓ The runtime of the deobfuscator grows exponentially with the size of the software and the branching level of the resulting graph, as a deobfuscator has to traverse through the entire graph to reconstruct the control flow.
- ✓ A trade-off between obfuscation strength and performance has to be made.

Table II. Analysis of the strength of code obfuscation classes in different analysis scenarios (PM = Pattern Matching LD = Locating Data, LC = Locating Code, EC = Extracting Code, UC = Understanding Code).

	PM		Autom. Static				Autom. Dynamic				Human Assisted			
Name	**LD**	**LC**	**LD**	**LC**	**EC**	**UC**	**LD**	**LC**	**EC**	**UC**	**LD**	**LC**	**EC**	**UC**
Dynamic code rewriting														
Packing/Encryption		✓		✓				✓	✓			✓		✓
Dynamic code modifications														
Environmental requirements														
Hardware-assisted code obfuscation											✓			
Virtualization			✓	✓				✓				✓		✓
Anti-debugging techniques						✓		✓	✓					✓

Legend		obfuscation breaks analysis fundamentally
		obfuscation is not unbreakable, but makes analysis more expensive
		obfuscation only results in minor increases of costs for analysis
	✓	A checkmark indicates that the rating is supported by results in the literature
		Scenarios without a checkmark were classified based on theoretical evaluation

Protecting software through obfuscation: can it keep pace with progress in code analysis? ACM Computing Survey 2016

Hybrid Obfuscation to Protect against Disclosure Attacks on Embedded Microprocessors

Marc Fyrbiak, Simon Rokicki, Nicolai Bissantz, Russell Tessier, Christof Paar, *Fellow, IEEE*

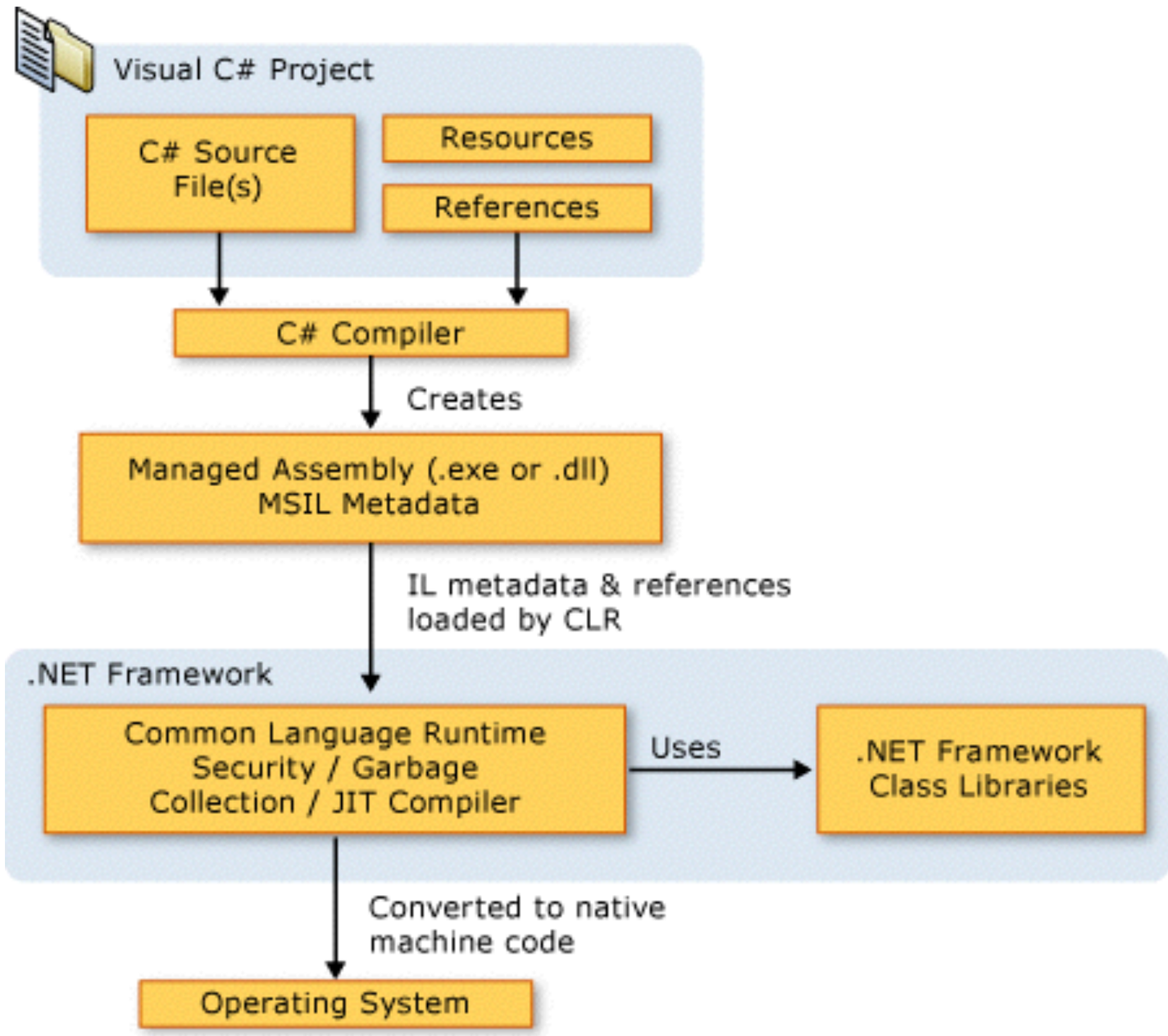
Abstract—The risk of code reverse-engineering is particularly acute for embedded processors which often have limited available resources to protect program information. Previous efforts involving code obfuscation provide some additional security against reverse-engineering of programs, but the security benefits are typically limited and not quantifiable. Hence, new approaches to code protection and creation of associated metrics are highly desirable. This paper has two main contributions. We propose the first hybrid diversification approach for protecting embedded software and we provide statistical metrics to evaluate the protection. Diversification is achieved by combining hardware obfuscation at the microarchitecture level and the use of software-level obfuscation techniques tailored to embedded systems. Both measures are based on a compiler which generates obfuscated programs, and an embedded processor implemented in an FPGA with a randomized ISA encoding to execute the hybrid obfuscated program. We employ a fine-grained, hardware-enforced access control mechanism for information exchange with the processor and hardware-assisted booby traps to actively counteract manipulation attacks. It is shown that our approach is effective against a wide variety of possible information disclosure attacks in case of a physically present adversary. Moreover, we propose a novel statistical evaluation methodology that provides a security metric for hybrid-obfuscated programs.

Index Terms—Computer architecture, ISA randomization, software obfuscation, reverse-engineering.

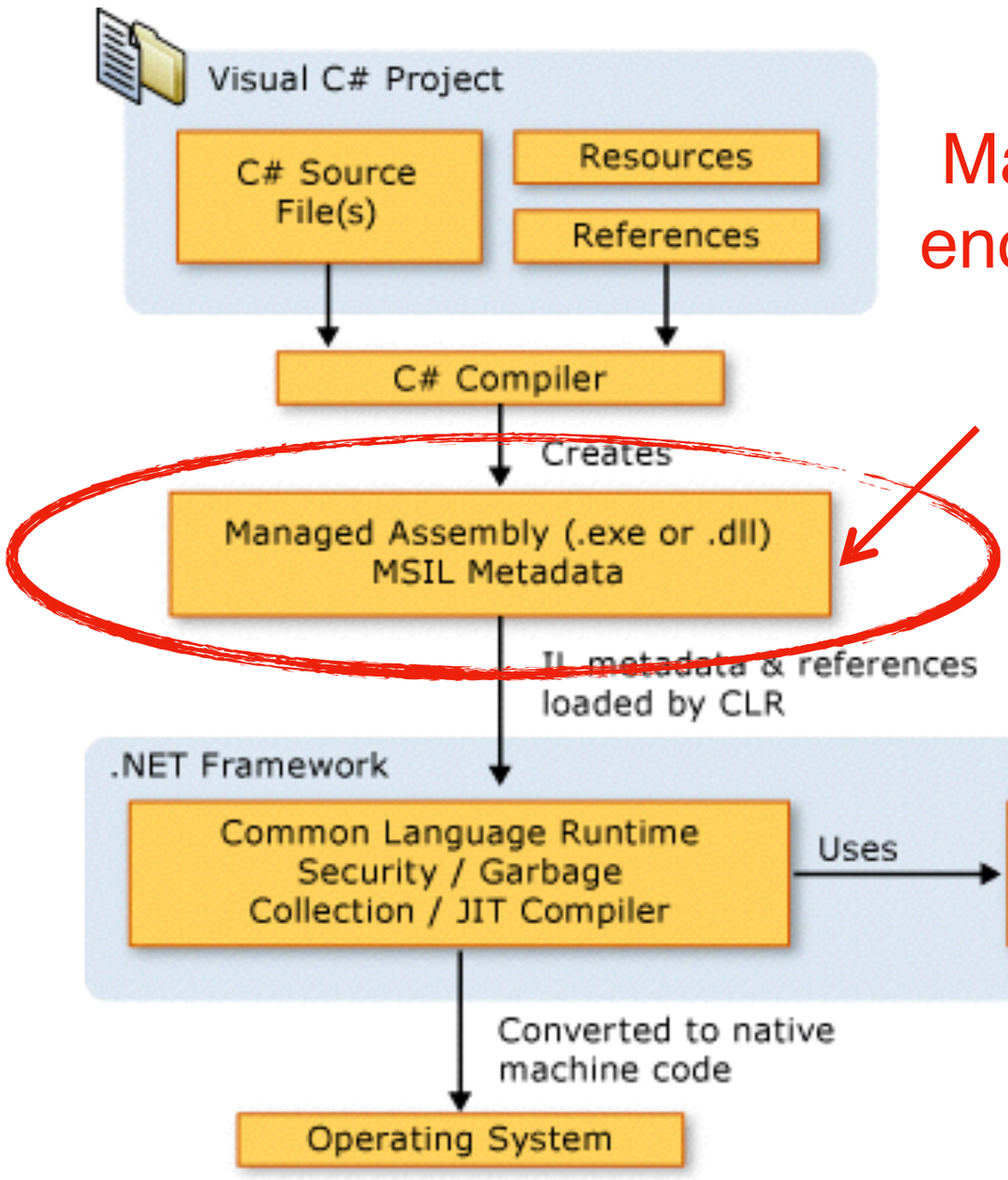
2018 IEEE Transactions on Computers

Obfuscation for .NET

.NET


Visual C# Project
C# Source File(s)
Resources
References
C# Compiler
Creates
Managed Assembly (.exe or .dll)
MSIL Metadata
IL metadata & references loaded by CLR
.NET Framework
Common Language Runtime Security / Garbage Collection / JIT Compiler
Uses
.NET Framework Class Libraries
Converted to native machine code
Operating System

.NET



Managed assembly contains enough information to recover the original source code

Managed assembly contains enough information to recover the original source code

Not a problem for web scenarios where the code resides on the server

Security issue for client scenarios that contain secrets or sensitive information in their algorithms, data structures or data.

CODE PROTECTION is NEEDED!!!!

The Market for .NET

	Obfuscar	Confuserex	Smart Assembly	Eazfuscator	.NetGuard	Spices.Net
Cost	free	free - disc	from 716$	399$	custom/jobs	399$
Mantained			✓	✓	✓	✓
Renaming	✓		✓	✓	✓	✓
Constant Encryption		✓	✓	✓	✓	✓
Resource Encryption		✓	✓	✓	✓	
Control Flow			✓	✓	✓	✓
Data Obf					✓	
Virtualization				✓		
Remove Useless			✓			✓
Anti Debugging/Profilers		✓			✓	
Anti-Decompilation		✓			✓	
Anti Disassembly		✓	✓		✓	✓
Anti memory dumping		✓			✓	
Asm merging/embedding			✓	✓		
Debugging Support				✓	✓	
Tamper Detection		✓	✓		✓	
Deobfuscator		NoFuserEx	DeSmart-2007	De4dot - 2015		De4dot - 2015
Deobfuscator			De4dot - 2015			
Summary	no	not maintained	very good	very good	**very good** obf as a service	very good recognized by microsoft

The Market for .NET

	.Net Reactor	Crypto	DeployLX	Babelor.Net	Skater.Net
Cost	179$	149$	299$ 899$	115 – 245Euro	80 – 299$
Mantained	✓	✓	✓	✓	
Renaming	✓	✓	✓	✓	✓
Constant Encryption	✓	✓			✓
Resource Encryption	✓	✓			
Control Flow	✓	✓		✓	✓
Data Obf					
Virtualization				✓	
Remove Useless				✓	
Code Encryption			✓	✓	
Anti Debugging/Profilers		✓	✓		
Anti-Decompilation	✓	✓	✓		
Anti Disassembly	✓	✓			
Anti memory dumping			✓		
Asm merging/embedding	✓			✓	✓
Debugging Support		✓		✓	
Tamper Detection	✓	✓	✓	✓	
Deobfuscator	DeReactor-2007			Babelor Deobf2016	De4dot-2015
Deobfuscator	De4dot - 2015	De4dot - 2015			
Summary	very good important customers	very good	mainly encryots for dynamic analysis	very good and supports development	fair