University of Arizona

CSc 620

Security Through Obscurity

Christian Collberg
February 18, 2002

Code Obfuscation

University of Arizona

CSc 620

Security Through Obscurity

Introduction

A

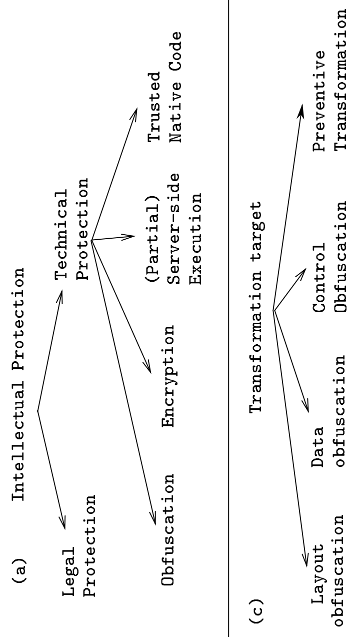
Malicious Reverse Engineering

- Given enough time, effort and determination, a competent programmer will always be able to reverse engineer any application.
- Tools: disassemblers, decompilers, slicers.
- Didn't use to be a big problem since most programs are large, monolithic, and shipped as stripped, native code.
- It is becoming more common to distribute software in forms that are easy to decompile and reverse engineer. Example: Java bytecode and the *Architecture Neutral Distribution Format* (ANDF).

Reverse Engineering Java Apps.

- Java applications are distributed over the Internet as Java *class files.*
- Java bytecode is a hardware-independent virtual machine code that retains virtually all the information of the original Java source.
- Hence, these class files are easy to decompile.
- Much of the computation in a Java app takes place in standard libraries. Hence, Java programs are often small in size. This makes them easier to reverse engineer.

Intellectual Protection of Software



Slide 13A–3

This Talk

- We will discuss the various forms of technical protection of intellectual property which are available to software developers.
- We will argue that the only cost effective approach to the protection of mobile code is *code obfuscation*.
- We will present a number of *obfuscating transformations*, classify them, and show the design of an automatic obfuscation tool.
- (1) Forms of technical protection against software theft, (2) Design of a code obfuscation tool, (3) Criteria for classifying and evaluating obfuscating transformations, (4) Catalogue of obfuscating transformations, (5) Algorithms, (6) Summary.

Slide 13A–4

University of Arizona

CSc 620

Security Through Obscurity

Overview

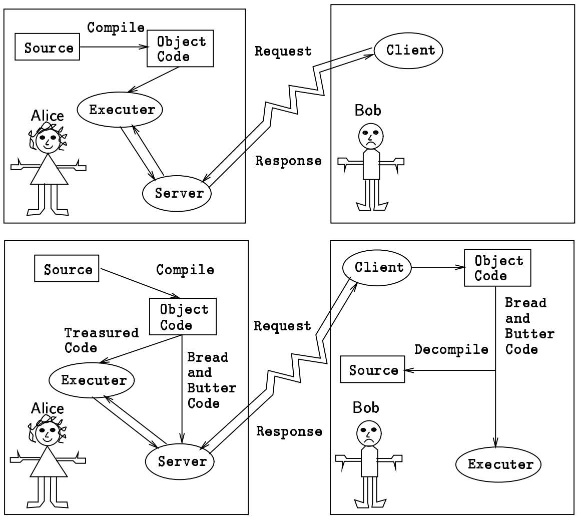
B

Scenario

- Alice is a small software developer who wants to make her applications available to users over the Internet, presumably at a charge. Bob is a rival developer who feels that he could gain a commercial edge over Alice if he had access to her application's key algorithms and data structures.
- Alice can protect her code from Bob's attack using either *legal* or *technical* protection.
- Economic realities make it difficult for a small company like Alice's to enforce the law against a larger and more powerful competitor.
- Alice can instead protect her code by making reverse engineering so technically difficult that it becomes economically inviable.

Slide 13B–1

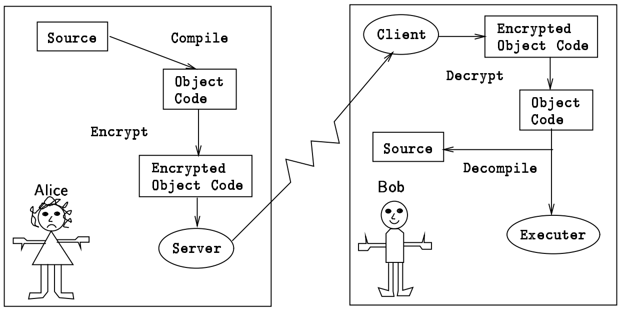
Server-Side Protection



- If Alice just sells the *services* of her application, Bob will never gain physical access to it and hence can't reverse engineer it.
- To improve performance, only the sensitive part is run remotely.

Slide 13B–2

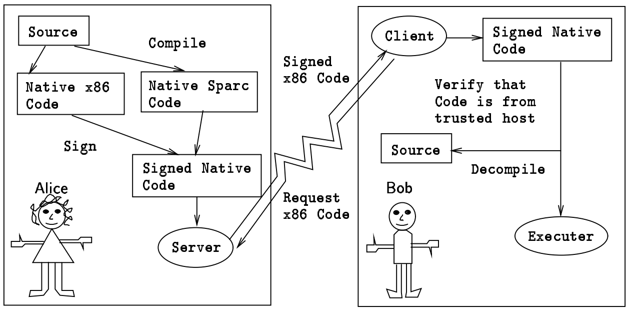
Protection by Encryption



- Alice can *encrypt* her code before it is sent off to the users. This only works if the entire decryption/execution process takes place in hardware.
- If the code is executed in software by a virtual machine interpreter, then it will always be possible for Bob to intercept and decompile the decrypted code.

Slide 13B–3

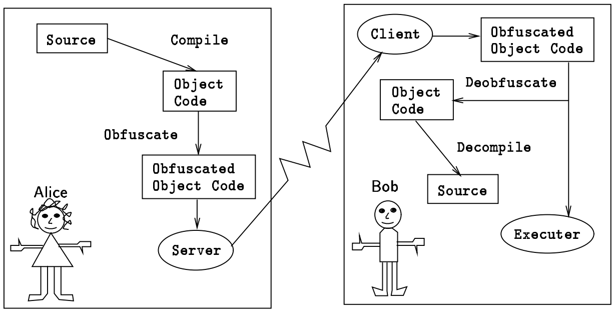
Protection By Native Code



- Alice could use of *just-in-time compilers* to create native code versions of her application for all popular architectures.
- When downloading the application, the appropriate version would be transmitted. Only having access to native code will make Bob's task more difficult, but not impossible.
- Alice must digitally sign the code, since (unlike Java bytecodes) it cannot be verified before execution.

Slide 13B–4

Protection By Obfuscation I



- Alice runs her application through an *obfuscator*, that transforms the application into one that is functionally identical to the original but which is more difficult for Bob to understand.
- Obfuscation can't completely prevent reverse engineering.
- Bob can use an automatic *deobfuscator* to undo the obfuscating transformations.

Slide 13B–5

Protection By Obfuscation II

- The level of security from reverse engineering that an obfuscator adds to an application depends on
  1. the sophistication of the obfuscating transformations,
  2. the power of the deobfuscator,
  3. the amount of resources available to the deobfuscator.
- Ideally, we would like to mimic the situation in current public-key cryptosystems, where there is a dramatic difference in the cost of encryption and decryption.
- There are obfuscating transformations that can be applied in polynomial time but which require worst-case exponential time to deobfuscate.

Slide 13B–6

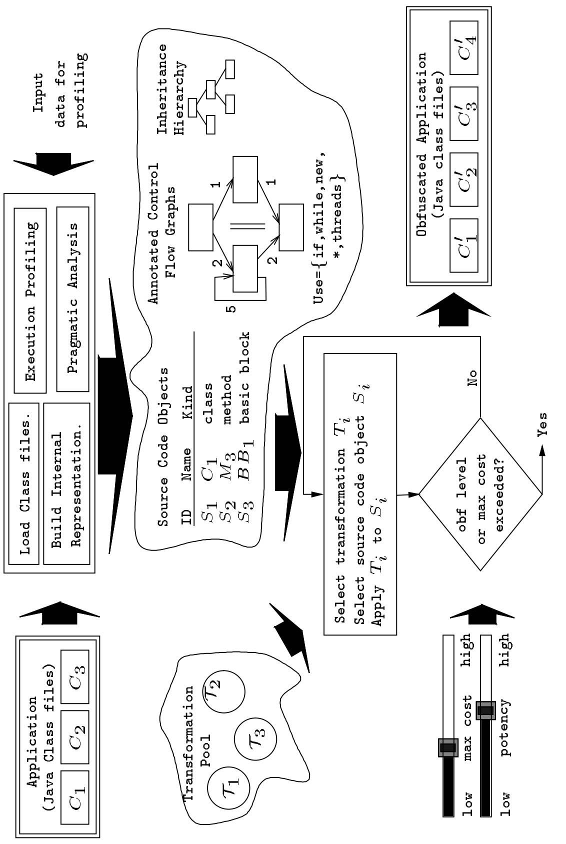
University of Arizona

CSc 620

Security Through Obscurity

Architecture

C



Slide 13C–1

A Java Obfuscator II

- **Input:** a set of Java class files, the required level of obfuscation, maximum execution time/space penalty, profiling data.
- **Output:** a new application given as a set of Java class files, annotated Java source for debugging.
- **Internal data structures:** symbol tables, inheritance tree, CFGs, data dependency graphs, etc.
- The profiling information can be used to guide the obfuscator so that frequently executed parts of the application are not obfuscated by very expensive transformations.
- The tool contains a large pool of code transformations.

Slide 13C–2

A Java Obfuscator III

- Pragmatic information: what kind of language constructs and programming idioms does the application contain. Used to select appropriate transformations.
- All types of language constructs can be obfuscated: classes can be split or merged, methods can be changed or created, new control- and data structures can be created and original ones modified, etc.
- New constructs added to the application are selected to be as similar as possible to the ones in the source application.
- The transformation process is repeated until the required potency has been achieved or the maximum cost has been exceeded.

Slide 13C–3

University of Arizona

CSc 620

Security Through Obscurity

Classification

D

Obfuscating Transformation

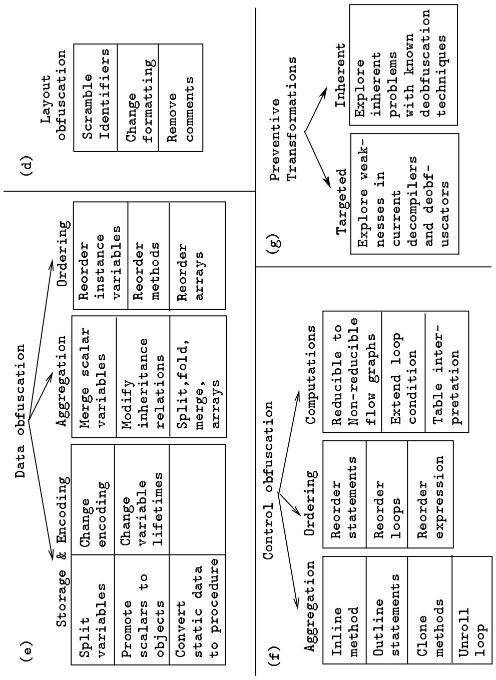
Let $P \xrightarrow{T} P'$ be a transformation of a source program P into a target program P'.
$P \xrightarrow{T} P'$ is an *obfuscating transformation*, if P and P' have the same *observable behavior*.
The following conditions must hold:

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

- Observable behavior is defined loosely as "behavior as experienced by the user."
- P' may have side-effects (creating files, sending messages) that P does not, as long as these side effects are not experienced by the user. P and P' don't have to be equally efficient.

Slide 13D–1

Classifying Obfuscating Transformations



(e) Data obfuscation

Storage & Encoding: Split variables; Promote scalars to objects; Convert static data to procedure; Change encoding; Change variable lifetimes

Aggregation: Merge scalar variables; Modify inheritance relations; Split, fold, merge, arrays

Ordering: Reorder instance variables; Reorder methods; Reorder arrays

(d) Layout obfuscation: Scramble Identifiers; Change formatting; Remove comments

(f) Control obfuscation

Aggregation: Inline method; Outline statements; Clone methods; Unroll loop

Ordering: Reorder statements; Reorder loops; Reorder expression

Computations: Reducible to Non-reducible flow graphs; Extend loop condition; Table interpretation

(g) Preventive Transformations

Targeted: Explore weaknesses in current decompilers and deobfuscators

Inherent: Explore inherent problems with known deobfuscation techniques

Slide 13D–2

Classifying Transformations

- We primarily classify an obfuscating transformation according to the kind of *information* it targets:

  layout The lexical structure of the application, such as source code formatting, names of variables, etc.

  data The types, declarations, data structures.

  control The flow of control.

  preventive

- Secondly, we classify a transformation according to the kind of operation it performs on the targeted information:

  Aggregation Break up or create user-defined abstractions.

  Ordering randomize the order of declarations or computations.

Slide 13D–3

University of Arizona

CSc 620

Security Through Obscurity

Quality

E

Quality

- We need to be able to evaluate the *quality* of an obfuscating transformation.
- We will attempt to classify each transformation $\mathcal{T}$ according to several criteria:

  potency how much obscurity $\mathcal{T}$ adds to the program,

  resilience how difficult $\mathcal{T}$ is for a deobfuscator to undo,

  stealth how well code introduced by $\mathcal{T}$ fits in with the original code,

  cost how much computational overhead $\mathcal{T}$ adds to the obfuscated application.

$\mathcal{T}_{\text{qual}}(P)$, the quality of a transformation $\mathcal{T}$, is defined as the combination of the potency, resilience, and cost of $\mathcal{T}$:

$$\mathcal{T}_{\text{qual}}(P) = (\mathcal{T}_{\text{pot}}(P), \mathcal{T}_{\text{res}}(P), \mathcal{T}_{\text{cost}}(P), \mathcal{T}_{\text{ste}}(P)).$$

Slide 13E–1

Measures of Potency

- What does it mean for a program P' to be more *obscure* (or *complex* or *unreadable*) than a program P?
- Any such metric must be rather vague, since it will be based on human cognitive abilities.
- We can draw upon the vast body of work in the *Software Complexity Metrics* branch of Software Engineering.
- Metrics are designed with the intent to aid the construction of readable, reliable, and maintainable software.
- We use metrics to construct un-readable, un-reliable, and un-maintainable software!
- The metrics count various source code properties combining these counts into a measure of complexity.

Slide 13E–2

METRIC	METRIC NAME	CITATION
μ_1	Program Length	Halstead
	$E(P)$ increases with the # of operators+operands in P.	
μ_2	Cyclomatic Complexity	McCabe
	$E(F)$ increases with the # of predicates in F.	
μ_3	Nesting Complexity	Harrison
	$E(F)$ increases with the nesting level of conditionals in F.	
μ_4	Data Flow Complexity	Oviedo
	$E(F)$ increases with the # of inter-basic block variable references in F.	
μ_5	Fan-in/out Complexity	Henry
	$E(F)$ increases with the # of formal parameters to F, and the # of global data structures referenced by F.	

Slide 13E–3

METRIC	METRIC NAME	CITATION
μ_6	Data Structure Complexity	Munson
	$E(P)$ increases with the complexity of the static data structures (arrays, records) declared in P.	
μ_7	OO Metric	Chidamber
	$E(C)$ increases with (μ_7^{a}) the # of methods in C, (μ_7^{b}) the depth (distance from the root) of C in the inheritance tree, (μ_7^{c}) the # of direct subclasses of C, (μ_7^{d}) the # of other classes to which C is coupled, (μ_7^{e}) the # 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.	

Slide 13E–4

Definition of Potency

- We will use *potency* to measure of the usefulness. of a transformation.
- A transformation is *potent* if it does a good job confusing Bob, by hiding the intent of Alice's original code.
- Potency measures how much more difficult the obfuscated code is to understand (for a human) than the original code.

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 some complexity metric.

$\mathcal{T}_{\mathrm{pot}}(P)$, the *potency* of $\mathcal{T}$ with respect to a program P, measures how $\mathcal{T}$ changes the complexity of P:
$\mathcal{T}_{\mathrm{pot}}(P) \overset{\mathrm{def}}{=} E(P')/E(P) - 1$.

Slide 13E–5

How Can We Increase Potency?

increase size increase overall program size (μ_1) and introduce new classes and methods (μ_7^{a}).

more decision points introduce new predicates (μ_2) and increase the nesting level of conditional and looping constructs (μ_3).

increase coupling increase the number of method arguments (μ_5) and inter-class instance variable dependencies (μ_7^{d}).

increase data structure complexity increase the number of array dimensions (μ_6).

increase inheritance increase the height of the inheritance tree ($\mu_7^{\mathrm{b,c}}$).

increase scope increase long-range variable dependencies (μ_4).

Slide 13E–6

Measures of Resilience

- It would seem that increasing $\mathcal{T}_{\text{pot}}(P)$ would be trivial. To increase the μ_2 metric we just add some arbitrary if-statements to P:

$$\begin{array}{l}\texttt{main() \{}\\ \quad S_1\texttt{;}\\ \quad S_2\texttt{;}\\ \texttt{\}}\end{array} \quad \overset{\mathcal{T}}{\Longrightarrow} \quad \begin{array}{l}\texttt{main() \{}\\ \quad S_1\texttt{;}\\ \quad \texttt{if (5==2) } S_1\texttt{;}\\ \quad S_2\texttt{;}\\ \quad \texttt{if (1>2) } S_2\texttt{;}\\ \texttt{\}}\end{array}$$

- Such transformations are useless, since they can easily be undone by simple automatic techniques.
- We introduce *resilience*, which measures how well a transformation holds up under attack from an automatic deobfuscator.

Slide 13E–7

Measures of Resilience

- The resilience of a transformation $\mathcal{T}$ is the combination of two measures:

Programmer Effort: the amount of time required to construct an automatic deobfuscator that is able to effectively reduce the potency of $\mathcal{T}$, and

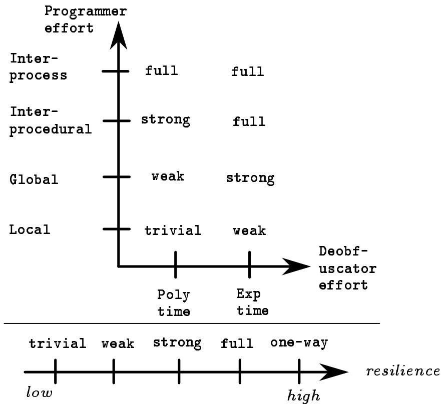
Deobfuscator Effort: the execution time and space required by such an automatic deobfuscator to effectively reduce the potency of $\mathcal{T}$.

- $\mathcal{T}$ is *potent* if it confuses a human reader.
- $\mathcal{T}$ is *resilient* if it confuses an automatic deobfuscator.

Slide 13E–8

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,



Slide 13E–9

Measures of Stealth

- While a resilient transformation may not be succeptible to attacks by automatic deobfuscators, it may still be succeptible to attacks by humans.
- Particularly, if a transformation introduces new code that differs wildly from what is in the original program it will be easy to spot for a reverse engineer. Such transformations are *unstealthy*. See example below.
- Stealth is a highly context-sensitive metric. Code may be stealthy in one program but extremely unstealthy in another one.

$$\begin{array}{l}\texttt{if IsPrime}\overbrace{(8375274\ldots3853347527)}^{\text{512-bit integer}}\texttt{ then}\\ \qquad \ldots\end{array}$$

Slide 13E–10

Measures of Execution Cost

- The *cost* of a transformation is the execution time/space penalty which a transformation incurs on an obfuscated application.

Let $\mathcal{T}$ be a behavior-conserving transformation. $\mathcal{T}_{\text{cost}}(P)$ is the extra execution time/space of P' compared to P:

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

Slide 13E–11

Layout Transformations

- Current Java obfuscators such as Crema usually only perform trivial *layout transformations*:
  1. Remove the source code formatting information sometimes available in Java class files: $\mathcal{T}_{\text{qual}}(P) = (\textit{low}, \textit{one-way}, \textit{free})$.
  2. Scramble identifier names: $\mathcal{T}_{\text{qual}}(P) = (\textit{high}, \textit{one-way}, \textit{free})$.
  3. Remove comments: $\mathcal{T}_{\text{qual}}(P) = (\textit{high}, \textit{one-way}, \textit{free})$.

Slide 13E–12

University of Arizona

CSc 620

Security Through Obscurity

Control Transformations

F

Control Transformations

- Next, we will present a catalogue of obfuscating transformations.
- First we'll look at transformations that obscure control-flow. There are three kinds:

  aggregation Break up computations that logically belong together or merge computations that do not.

  ordering Randomize the order in which computations are carried out.

  computations Insert new (redundant or dead) code, or make algorithmic changes to the source application.

- Control transformations will have some computational overhead: Alice may have to trade efficiency for obfuscation.

Slide 13F–1

Opaque Constructs I

- Many control-altering transformations rely on the existence of *opaque variables* and *opaque predicates.*
- A variable V is opaque if it has some property q which is known *a priori* to the obfuscator, but which is difficult for the deobfuscator to deduce.
- A predicate P is opaque if a deobfuscator can deduce its outcome only with great difficulty, while this outcome is well known to the obfuscator.
- The resilience of an opaque construct (i.e. its resistance to deobfuscation attacks) is measured on the scale ⟨*trivial,weak,strong,full,one-way*⟩.
- We measure the added cost of an opaque construct on the scale ⟨*free,cheap,costly,dear*⟩.

Slide 13F–2

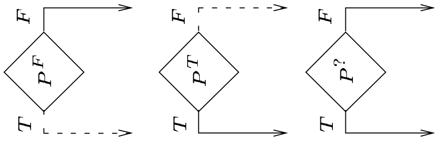
Control Transformations

- The complexity of a program grows with the number of predicates it contains.
- A predicate P is opaque if its outcome is known at obfuscation time:

  $P^F \Rightarrow P$ is always `False`.

  $P^T \Rightarrow P$ is always `True`.

  $P^? \Rightarrow P$ is sometimes `True`, sometimes `False`.
- Obfuscating control transformations insert opaque predicates that are difficult for a deobfuscator to evaluate.



Slide 13F–3

Simple Opaque Constructs

- *Trivial* opaque constructs can be cracked by a deobfuscator by a **static local** analysis:

```
{ int v, a=5; b=6;
  v$^{=11}$ = a + b;
  if (b > 5)$^T$ ···
  if (random(1,5) < 0)$^F$ ··· }
```

- *Weak* opaque constructs can be cracked using a **static global** analysis:

```
{ int v, a=5; b=6;
  if (···) ···
       ⋮ (b is unchanged)
  if ($b$ < 7)$^T$ a++;
  v$^{=36}$ = (a > 5)?v=b*b:v=b }
```

Slide 13F–4

Insert Dead/Irrelevant Code I

- The μ_2 and μ_3 metrics suggest that there is a strong correlation between the perceived complexity of a piece of code and the number of predicates it contains.
- Opaque predicates allow us to devise transformations that introduce new predicates into a program.
- Three cases:

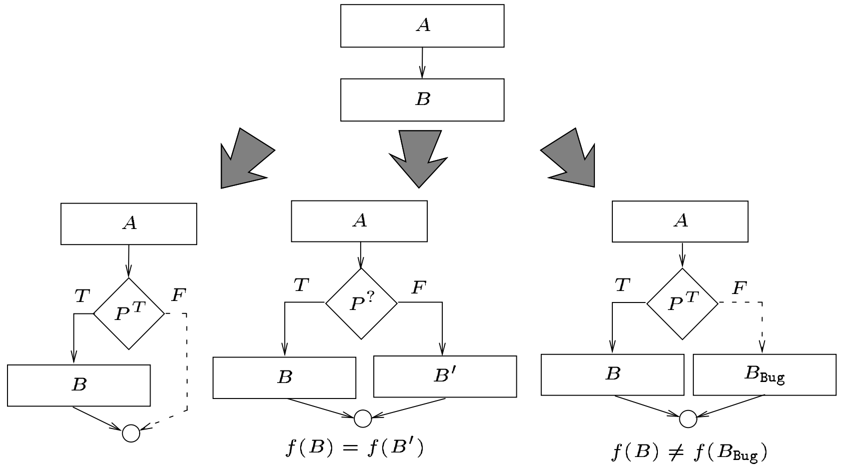
  (a) Insert an opaque predicate P^T into S. P^T is *irrelevant* code since it will always evaluate to `True`.

  (b) Break S into two *different* obfuscated versions S^a and S^b. $P^?$ selects between them at runtime.

  (c) As in **(b)**, but introduce a bug into S^b. P^T selects the correct version of the code, S^a.

Slide 13F–5

Insert Dead/Irrelevant Code II



Slide 13F–6

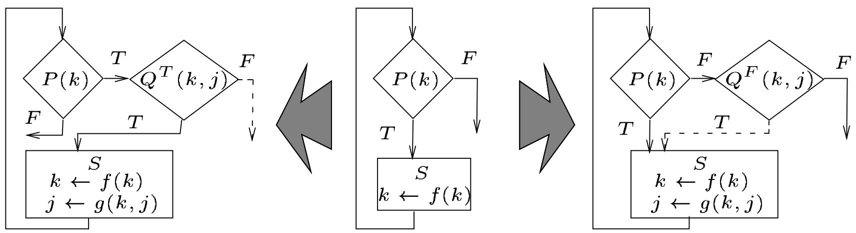
Extend Loop Condition I

- We can obfuscate a loop by extending the loop condition with a P^T or P^F predicate which doesn't affect the number of times the loop will execute.
- Here, P^T based on the fact that $x^2(x+1)^2 \equiv 0 \pmod 4$.

```
i=1;                       i=1; j=100;
while (i<100) {     T      while ((i<100) &&
      ...          ==>        (j*j*(j+1)*(j+1)%4 == 0)^T) {
  i++;                              ...
}                             i++;
                              j=j*i+3; }
```

Slide 13F–7

Extend Loop Condition II



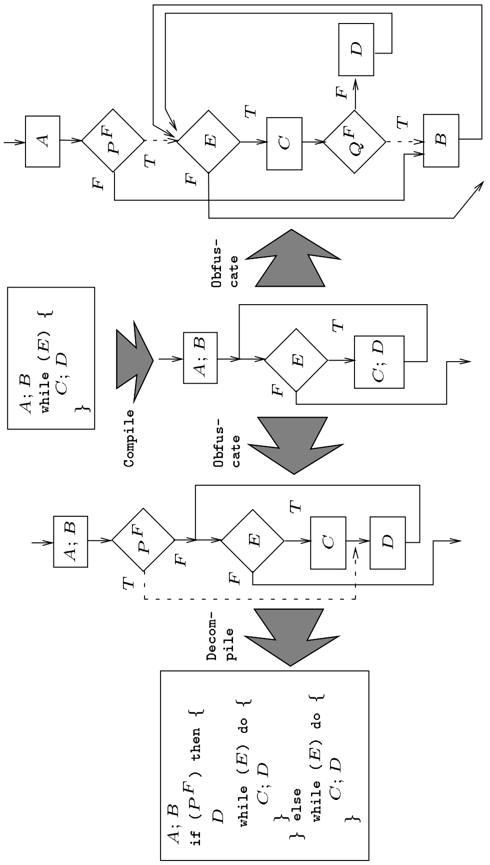
Slide 13F–8

Non-Reducible Flow Graphs I

- Often, a programming language is compiled to a object code which is more expressive than the language itself.
- This allows us to device *language-breaking* transformations. I.e. introduce virtual machine (or native code) instruction sequences which do not correspond directly to any source language construct.
- For example, the Java *bytecode* has a `goto` instruction while the Java *language* has no corresponding `goto`-statement.
- We construct a transformation which converts a reducible flow graph to a non-reducible one.
- This can be done by turning a structured loop into a loop with multiple headers.

Slide 13F–9

Non-Reducible Flow Graphs II



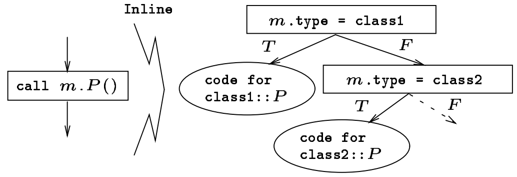
```
A; B
while (E) {
   C; D
}
```

```
A; B
if (P^F) then {
   D
   while (E) do {
      C; D
   }
} else
   while (E) do {
      C; D
   }
```

Slide 13F–10

Inlining & Outlining I

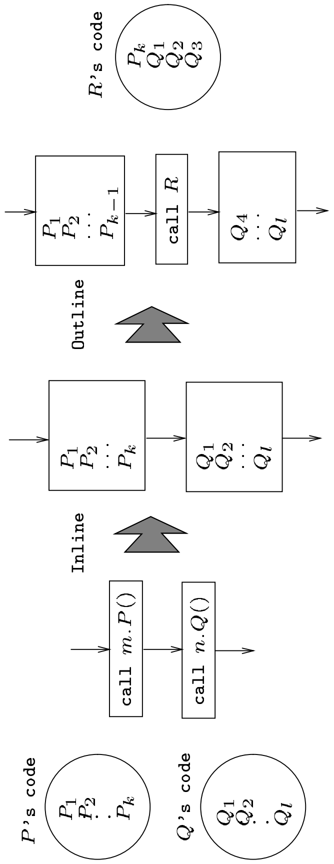
- Inlining is an extremely useful obfuscation transformation since it removes procedural abstractions from the program.
- Inlining is a highly resilient transformation (it is essentially *one-way*). This may not be true in OO languages, since inlining may leave a trace:



- Outlining (turning a sequence of statements into a subroutine) is a very useful companion transformation to inlining.

Slide 13F–11

Inlining & Outlining II



Slide 13F–12

Interleave Methods I

- The detection of *interleaved code* is an important and difficult reverse engineering task. Rugaber writes:

> One of the factors that can make a program difficult to understand is that code responsible for accomplishing more than one purpose may be woven together in a single section. We call this *interleaving* [$\cdots$]

- We can easily interleave two methods declared in the same class. The idea is to merge the bodies and parameter lists of the methods and add an extra parameter to discriminate between calls to the individual methods.
- Ideally, the methods should be similar in nature to allow merging of common code and parameters.

Slide 13F–13

Interleave Methods II

```
class C {
   A(T1 x){
      $a_1;a_2$;
   }
   B(T1 y;T2 z){
      $b_1;b_2$;
   }
}
{ C p=new C;
  p.A(x);
  p.B(y, z); }
```

$\overset{\mathcal{T}}{\Rightarrow}$

```
class C' {
   M(T1 xy;T2 y;int V){
      if ($V==p$) {$a_1;a_2$;}
      else        {$b_1;b_2$;}
   }
}
{ C' p=new C';
  p.M(x, c, $V$=$^p$);
  p.M(y, c, $V$=$^q$); }
```

Slide 13F–14

Clone Methods I

- To understand the behavior of a routine a reverse engineer would examine its call sites. We can obfuscate the call sites to make it appear that different routines are being called.
- We create several different versions of a method by applying different sets of obfuscating transformations to the original code. We use method dispatch to select between the different versions at runtime.
- Method cloning is similar to the predicate insertion transformations.
- The calls ⌜`x.m(5)`⌝ and ⌜`x.m1(7)`⌝ look as if they were made to two different methods. `C1::m` is a buggy version of `C::m` that is never called.

Slide 13F–15

Clone Methods II

```
class C {
   m(int x)
      { $A;B$ }
}
{ C x = new C;
  x.m(5); ... x.m(7);
}
```

$\overset{\mathcal{T}}{\Rightarrow}$

```
class C1 {
   m(int x){ $A_1;B_1$ }
   m1(int x){ $A_2;B_2$ }
}
class C2 inherits C1 {
   m(int x){ $A_3;B_3$ }
}
{ C1 x ;
  if ($P^F$) x=new C1
  else x=new C2;
  x.m(5); ...; x.m1(7);}
```

Slide 13F–16

University of
Arizona

CSc 620

Security Through Obscurity

Data Transformations

G

Data Transformations

- Data transformations obscure the types and data structures used in the application.
- The transformations may affect the data structures'
  1. *storage*
  2. *encoding*
  3. *aggregation*
  4. *ordering*

Slide 13G–1

Split Variables I

- Boolean variables and other variables of restricted range can be split into two or more variables. We write this as $V = [p_1, \cdots, p_k]$.
- The potency of this transformation grows with k. So will the cost of the transformation, so we usually restrict k to 2 or 3.
- If $V = [p, q]$ we must provide:
  1. a function $f(p, q)$ that maps the p and q into the corresponding value of V,
  2. a function $g(V)$ that maps the value of V into values for p and q, and
  3. new operations (corresponding to the primitive operations on values of type T) cast in terms of operations on p and q.

Slide 13G–2

Split Variables II

		A			
	AND[A,B]	0	1	2	3
	0	3	0	0	0
B	1	3	1	2	3
	2	0	2	1	3
	3	3	0	0	3

$g(V)$		$f(p,q)$	
p	q	V	$2p+q$
0	0	False	0
0	1	True	1
1	0	True	2
1	1	False	3

		p	
	VAL[p,q]	0	1
q	0	0	1
	1	1	0

		A			
	OR[A,B]	0	1	2	3
	0	3	1	2	3
B	1	1	1	2	2
	2	2	2	1	1
	3	0	1	2	0

Slide 13G–3

Split Variables III

```
(1)  bool A,B,C;
(2)  A = True;
(3)  B = False;
(4)  C = False;
(5)  C = A & B;
(6)  C = A & B;
(7)  C = A | B;
(8)  if (A) ···;
(9)  if (B) ···;
(10) if (C) ···;
```

$\overset{\mathcal{T}}{\Longrightarrow}$

```
(1')  short a1,a2,b1,b2,c1,c2;
(2')  a1=0; a2=1;
(3')  b1=0; b2=0;
(4')  c1=1; c2=1;
(5')  x=AND[2*a1+a2,2*b1+b2]; c1=x/2; c2=x%2;
(6')  c1=(a1 ^ a2) & (b1 ^ b2); c2=0;
(7')  x=OR[2*a1+a2,2*b1+b2]; c1=x/2; c2=x%2;
(8')  x=2*a1+a2; if ((x==1) || (x==2)) ···;
(9')  if (b1 ^ b2) ···;
(10') if (VAL[c1,c2]) ···;
```

Slide 13G–4

Static to Procedural Data I

- Static data, particularly character strings, contain much useful pragmatic information to a reverse engineer.
- A simple way of obfuscating a static string is to convert it into *a program* that produces the string.
- The program – which could be a DFA, a Trie traversal, etc. – could possibly produce other strings as well.
- The function `G` obfuscates the strings `"AAA"`, `"BAAAA"`, and `"CCB"`.
- The values produced by `G` are `G(1)="AAA"`, `G(2)="BAAAA"`, `G(3)=G(5)="CCB"`, and `G(4)="XCB"`. For other argument values, `G` may not terminate.

Static to Procedural Data II

```
String G (int n) {
  int i=0,k;
  String S;
  while (1) {
    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 S;
    L9:  S[i++]="C"; goto L10;
    L10:  S[i++]="B"; goto L8;
    L11:  S[i++]="C"; goto L12;
    L12:  goto L10;
  }
}
```

Array Transforms I

- A number of transformations can be devised for obscuring operations performed on arrays: we can
  1. *split* an array into several sub-arrays,
  2. *merge* two or more arrays into one array,
  3. *fold* an array (increasing the number of dimensions),
  4. *flatten* an array (decreasing the number of dimensions).
- Array splitting and folding increase the μ_6 data complexity metric.
- Array merging and flattening *decrease* this measure. They are still obfuscating, though since they introduce structure where there was originally none or remove structure from the original program.

Array Transforms – Splitting

```
(1) int A[9];
(2) A[i] = ···;
```

$$\Downarrow \mathcal{T}$$

```
(1') int A1[4],A2[4];
(2') if ((i%2)==0) A1[i/2]=···
     else A2[⌊i/2⌋]=···;
```

	0	1	2	3	4	5	6	···	9
A :	A_0	A_1	A_2	A_3	A_4	A_5	A_6	···	A_9

$$\Downarrow \mathcal{T}$$

	0	1	2	3	4
A1 :	A_0	A_2	A_4	A_6	A_8

	0	1	2	3	4
A2 :	A_1	A_3	A_5	A_7	A_9

Array Transforms – Merging

```
(3) int B[9],C[19];
(4) B[i] = ···;
(5) C[i] = ···;
```

$$\Downarrow \mathcal{T}$$

```
(3') int BC[29];
(4') BC[3*i] = ···;
(5') BC[i/2*3+1+i%2] = ···;
```

	0	1	2	3	4	5	6	···	9
B :	B_0	B_1	B_2	B_3	B_4	B_5	B_6	···	B_9

	0	1	2	3	4	5	6	···	19
C :	C_0	C_1	C_2	C_3	C_4	C_5	C_6	···	C_{19}

$$\Downarrow \mathcal{T}$$

	0	1	2	3	4	5	6	···	29
BC :	B_0	C_0	C_1	B_1	C_2	C_3	B_2	···	C_{19}

Slide 13G–9

Array Transforms – Folding

```
(6) int D[9];
(7) for(i=0;i<=8;i++)
       D[i]=2*D[i+1];
```

$$\Downarrow \mathcal{T}$$

```
(6') int D1[1,4];
(7') for(j=0;j<=1;j++)
       for(k=0;k<=4;k++)
         if (k==4)
           D1[j,k]=2*D1[j+1,0];
         else
           D1[j,k]=2*D1[j,k+1];
```

	0	1	2	3	4	5	6	···	9
D :	D_0	D_1	D_2	D_3	D_4	D_5	D_6	···	D_9

$$\Downarrow \mathcal{T}$$

D1 :		0	1	2	3	4
	0	D_0	D_1	D_2	D_3	D_4
	1	D_5	D_6	D_7	D_8	D_9

Slide 13G–10

Array Transforms – Flattening

```
(8) int E[2,2];
(9) for(i=0;i<=2;i++)
      for(j=0;j<=2;i++)
        swap(E[i,j], E[j,i]);
```

$$\Downarrow \mathcal{T}$$

```
(8') int E1[8];
(9') for(i=0;i<=8;i++)
       swap(E[i], E[3*(i%3)+i/3]);
```

E :		0	1	2
	0	$E_{0,0}$	$E_{0,1}$	$E_{0,2}$
	1	$E_{1,0}$	$E_{1,1}$	$E_{1,2}$
	2	$E_{2,0}$	$E_{2,1}$	$E_{2,2}$

$$\Downarrow \mathcal{T}$$

	0	1	2	3	4	···	8
E1 :	$E_{0,0}$	$E_{0,1}$	$E_{0,2}$	$E_{1,0}$	$E_{1,1}$	···	$E_{2,2}$

Slide 13G–11

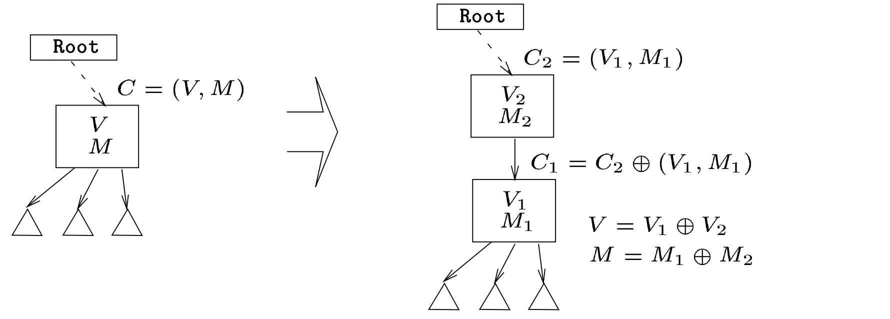
Modify Inheritance I

- According to metric μ_7, the complexity of a class C_1 grows with its
  1. *depth* (distance from the root) in the inheritance hierarchy, and
  2. the number of its direct descendants.
- There are two basic ways in which we can increase this complexity: we can
  1. split up a class, or
  2. insert a new, bogus, class
- We write a class as $C = (V, M)$, where V is the set of C's instance variables and M its methods.
- There is an arrow from class C_1 to C_2 if C_2 inherits from C_1.

Slide 13G–12

Modify Inheritance II – Splitting

- After splitting class C, all references to C in the program should be replaced by C_1.



$C = (V, M)$

$C_2 = (V_1, M_1)$

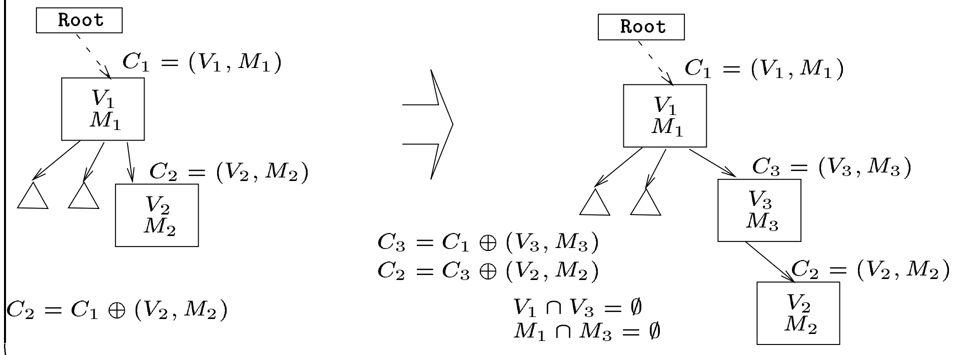
$C_1 = C_2 \oplus (V_1, M_1)$

$V = V_1 \oplus V_2$
$M = M_1 \oplus M_2$

Slide 13G–13

Modify Inheritance II – Insertion

- A random class C_3 is created and inserted into the inheritance hierarchy to increase its height.



$C_1 = (V_1, M_1)$

$C_2 = (V_2, M_2)$

$C_2 = C_1 \oplus (V_2, M_2)$

$C_1 = (V_1, M_1)$

$C_3 = (V_3, M_3)$

$C_2 = (V_2, M_2)$

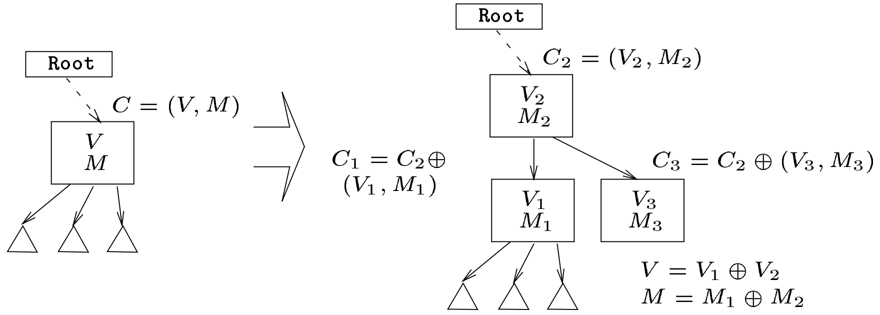
$C_3 = C_1 \oplus (V_3, M_3)$
$C_2 = C_3 \oplus (V_2, M_2)$

$V_1 \cap V_3 = \emptyset$
$M_1 \cap M_3 = \emptyset$

Slide 13G–14

Modify Inheritance II – Factoring

- A problem with class splitting is its low resilience; a deobfuscator can simply re-merge the split classes. Therefore, splitting and insertion are normally combined.



$C = (V, M)$

$C_1 = C_2 \oplus$
(V_1, M_1)

$C_2 = (V_2, M_2)$

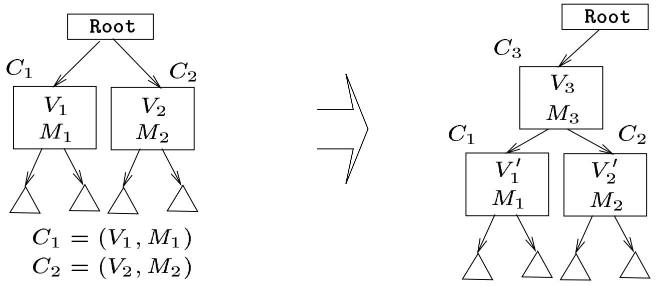
$C_3 = C_2 \oplus (V_3, M_3)$

$V = V_1 \oplus V_2$
$M = M_1 \oplus M_2$

Slide 13G–15

Modify Inheritance II – Refactoring

- False refactoring is performed on two classes C_1 and C_2 that have apparent but no actual common behavior. Features common to both classes are moved into a new (possibly abstract) parent class C_3.



$C_1 = (V_1, M_1)$
$C_2 = (V_2, M_2)$

$C_1 = C_3 \oplus (V_1', M_1)$

$C_2 = C_3 \oplus (V_2', M_2)$

$C_3 = (V_3, M_3)$

$V_3 = V_1 \cap V_2$

$V_1' = V_1 - V_3$

$V_2' = V_2 - V_3$

Slide 13G–16

University of Arizona

CSc 620

Security Through Obscurity

Opaque Constructs

H

Opaque Constructs

- Opaque predicates are the major building block in transformations that obfuscate control flow.
- We would like to be able to construct opaque predicates that require worst case exponential time to break but only polynomial time to construct.
- We will present two such techniques, based on aliasing and lightweight processes.

Slide 13H–1

Elementary Opaque Constructs

- Introduced predicates must be stealthy: ie. they cannot differ wildly from what is in the original program.
- Most predicates in real Java programs are simple (`p,q` are pointers; `n,m` are integers; `x,y` are reals):
  1. ⌜`if (p==null)`···⌝
  2. ⌜`if (p==q)`···⌝
  3. ⌜`if (n op` *`IntConst`*`)`···⌝, **op** $\in \{==, <, \leq, \cdots\}$
  4. ⌜`if (n op m)`···⌝, **op** $\in \{==, <, \leq, \cdots\}$
  5. ⌜`if (x op y)`···⌝, **op** $\in \{<, \leq, >, \geq\}$
  6. ⌜`if (x op` *`RealConst`*`)`···⌝, **op** $\in \{<, \leq, >, \geq\}$
- See elmentary number theory textbooks...

Slide 13H–2

#	Fact	Comments
i	$\forall x, y \in \mathcal{I}, 7y^2 - 1 \neq x^2$	
ii	$\forall x \in \mathcal{I}, 2 \mid (x + x^2)$	
iii	$\forall x \in \mathcal{I}, 3 \mid (x^3 - x)$	
iv	$\forall n \in \mathcal{I}^+, x, y \in \mathcal{I}, (x - y) \mid (x^n - y^n)$	
v	$\forall n \in \mathcal{I}^+, x, y \in \mathcal{I}, 2 \mid n \vee (x + y) \mid (x^n + y^n)$	
vi	$\forall n \in \mathcal{I}^+, x, y \in \mathcal{I}, 2 \not\mid n \vee (x + y) \mid (x^n - y^n)$	
vii	$\forall x \in \mathcal{I}^+, 9 \mid (10^x + 3 \cdot 4^{(x+2)} + 5)$	
viii	$\forall x \in \mathcal{I}, 3 \mid (7x - 5) \Rightarrow 9 \mid (28x^2 - 13x - 5)$	
ix	$\forall x \in \mathcal{I}, 5 \mid (2x - 1) \Rightarrow 25 \mid (14x^2 - 19x - 19)$	
x	$\forall x, y, z \in \mathcal{I}, (2 \not\mid x \wedge 2 \not\mid y) \Rightarrow x^2 + y^2 \neq z^2$	
xi	$\forall x \in \mathcal{I}^+, 14 \mid (3 \cdot 7^{4x+2} + 5 \cdot 4^{2x-1} - 5)$	

Slide 13H–3

#	Fact	Comments
xii	$\forall x \in \mathcal{I}, 2 \mid x \vee 8 \mid (x^2 - 1)$	
xiii	$\forall x \in \mathcal{I}^+, 64 \mid (7^{2x} + 16x - 1)$	
xiv	$\forall x \in \mathcal{I}^+, 24 \mid (2 \cdot 7^x + 3 \cdot 4^x - 5)$	
xv	$\forall x \in \mathcal{I}, \sum_{i=1,2\nmid i}^{2x-1} i = x^2$	The sum of the odd integers is a perfect square.
xvi	$\forall x \in \mathcal{I}^+, 8 \mid (7^{2x+1} + 17^x)$	
xvii	$\forall x \in \mathcal{I}^+, 2 \mid \lfloor \frac{x^2}{2} \rfloor$	The second bit of a squared number is always 0.

Slide 13H–4

Aliasing – Definitions

- Aliasing occurs when two variables refer to the same memory location.
- Aliasing occurs in languages with reference parameters, pointers, or arrays.
- In the general case alias analysis is undecidable. However, there exist many conservative algorithms that perform well for actual programs written by humans.

Slide 13H–5

Aliasing – Complexity Results

- Inter-procedural case is no more difficult than intra-procedural (wrt $\mathcal{P}$ vs. $\mathcal{NP}$).
- 1-level of indirection ⇒ $\mathcal{P}$; $\geq$ 2-levels of indirection ⇒ $\mathcal{NP}$.

Banning'79 Reference formals, no pointers, no structures ⇒ $\mathcal{P}$.

Horwitz'97 Flow-insensitive, may-alias, arbitrary levels of pointers, arbitrary pointer dereferencing ⇒ $\mathcal{NP}$ – hard.

Landi&Ryder'91 Flow-sensitive, may-alias, multi-level pointers, intra-procedural ⇒ $\mathcal{NP}$ – hard.

Landi'92 Flow-sensitive, may-alias, multi-level pointers, intra-procedural, dynamic memory allocation ⇒ Undecidable.

Slide 13H–6

Practical Shape Analysis

- Shape analysis requires alias analysis. Hence, all algorithms are approximate.

Ghiya'96a Accurate for programs that build simple data structures (trees, arrays of trees). Cannot handle major structural changes to the data structure.

Chase'90 Problems with destructive updates. Handles *list append*, but not *in-place list reversal.*

Hendren'90 Cannot handle cyclic structures.

various Only handle recursive structures no more than *k* levels deep.

Deutsch'94 Powerful, but large (8000 lines of ML) and slow (30 seconds to analyze a 50 line program).

Slide 13H–7

Using Aliasing I

- We will attempt to exploit the general difficulty of the alias analysis problem to manufacture cheap and resilient opaque predicates:
  1. Add to the obfuscated application code which builds a set of complex dynamic structures $S_1, S_2, \cdots$.
  2. Keep a set of pointers $p_1, p_2, \cdots$ into $S_1, S_2, \cdots$.
  3. The introduced code should update the structures, but must maintain certain invariants, such as "p_1 will never refer to the same heap location as p_3", "there may be a path from p_1 to p_2", etc.
  4. Use these invariants to manufacture opaque predicates when needed.

Slide 13H–8

Using Aliasing II

- This method is very attractive for three reasons:

Stealth

- The introduced code will closely resemble the code found in many real, pointer-rich, Java applications.

Resilience

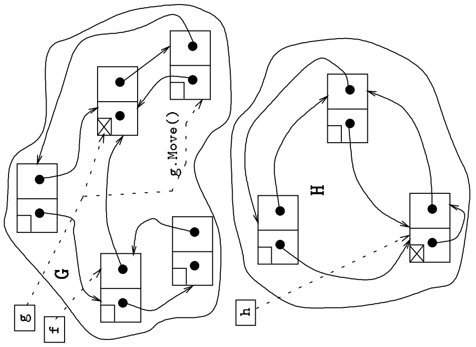
- We can construct 'destructive update' operations which current heap analysis algorithms will fail to analyze.

Cost

- We can construct invariants which can be tested for in constant time.

Slide 13H–9

```
Node g, h;
method P(...,Node f) {
   /* 1 */  g = g.Move();
             h = h.Move();
   /* 2 */  h = h.Insert(new Node);
   /* 3 */  x.R(..., f.Move());
   /* 4 */  if (f == g)^? ...
   /* 5 */  if (g == h)^F ...
   /* 6 */  f.Token=False;
             g.Token=True;
   /* 7 */  if (f.Token)^? ...
   /* 8 */  f.Token=True;
             h.Token=False;
   /* 9 */  if (f.Token)^T ...
}
```



Slide 13H–10

```
public class Node {
  public Node car, cdr;

  public Node() {
    this.car = this.cdr = this; }

  /* addNode_i is a family of functions
    which insert a new node after 'this'.  */
  Node addNode_1() {
    Node p = new Node(); p.car = this.car;
    return this.car = p; }
  Node addNode_2() {
    Node p = new Node; p.cdr = this.car;
    return this.car = p; }

  /* selectNode_i is a family of functions
    which return a reference to a node
    reachable from 'this'.  */
  Node selectNode_1() { return this.car; }
  Node selectNode_2() { return this.car.cdr; }
```

Slide 13H–11

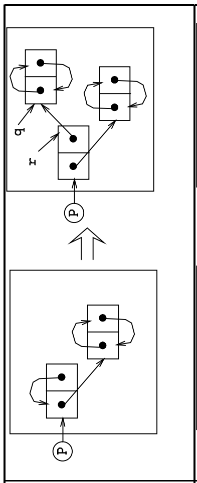
```
public Node selectNode3(int n) {
  return (n <= 0)?this:
    this.car.selectNode3b(n-1);
}
public Node selectNode3b(int n) {
  return (n <= 0)?this:
    this.cdr.selectNode3(n-1);
}

/* Return the set of nodes reachable
  from 'this'.  */
public Set reachableNodes()
  { return reachableNodes(new Set()); }
Set reachableNodes(Set reached) {
  if (!reached.member(this)) {
    reached.insert(this);
    this.car.reachableNodes(reached);
    this.cdr.reachableNodes(reached);
  }
  return reached;
}
```

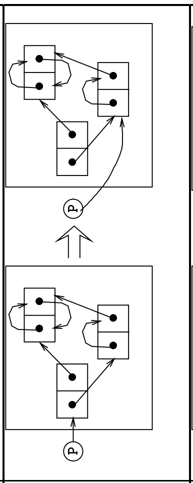
Slide 13H–12

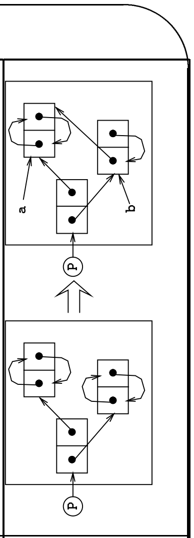
```
/* A and B are sets of graph nodes.
  Remove any references between nodes
  in A and B. */
private void splitComponent(
  Set R, Set A, Set B) {
  if (!R.member(this)) {
    R.insert(this);
    this.car.splitComponent(R, A, B);
    this.cdr.splitComponent(R, A, B);

    if (this.diffComp(this.car, A, B))
      this.car = this;
    if (this.diffComp(this.cdr, A, B))
      this.cdr = this;
}}

/* Returns true if the current node and */
  node b are in different components */
private boolean diffComp(
  Node b, Set A, Set B) {
  return (A.member(this) && B.member(b)) ||
    (B.member(this) && A.member(b));
}}}
```

Slide 13H–13

Code Pattern	Example

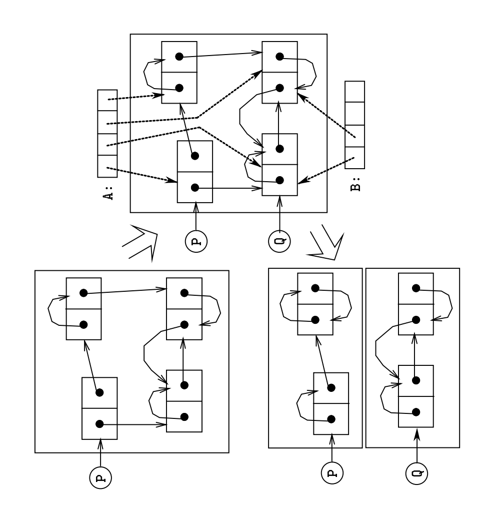
Node Insert (Node P) {
r = P.selectNode$_i$();
q = r.addNode$_j$();
return q; }
	

Node Move (node P) {
P = P.selectNode$_i$();
return P; }
	

void Link (Node P) {
a = P.selectNode$_i$();
b = P.selectNode$_j$();
b.car=(b.car==b)?
a:b.car;}
	

Slide 13H–14

Code Pattern	Example

void Split (Node P) {
Q=P.selectNode$_i$();
A=P.reachableNodes();
B=Q.reachableNodes();
P.splitGraph(A-B,B);
return Q;
}
	

Slide 13H–15

```
static void RayTrace (
 Vector scene, ViewDescr view) {
 Node p = new Node(), q = new Node();
 p.addNode1(); p.addNode2();
 for (int y = 0; y < view.height; y++) {
  if (y >= h - 10)
   p.selectNode3((int) (y*1.5)).p.addNode2();
  if (y == h - 10) {
   q = p.selectNode1;
   p.splitComponent(p.reachableNodes(),
    q.reachableNodes());}
  for (int x = 0; x < view.width; x++) {
   if ((y <= view.height - 10) &&
    (p.selectNode3(x) == q.selectNode3b))^F
     break;
   Ray theRay = view.pixelRay(y, x);
   SceneObject obj = hitObject(theRay, scene);
   if (obj != null) {
    Colour color = obj.surface.color(
     obj.hitPoint, obj.normal,
     view.eyePoint);
    Graphics.drawPoint(color, x, y);
}}}}
```

Slide 13H–16

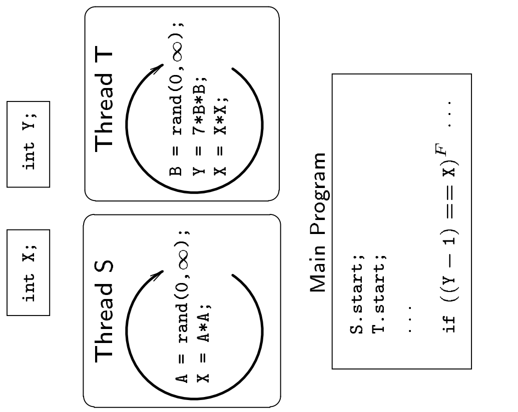
Concurrency

- Parallel programs are more difficult to analyze statically than sequential ones. The reason is their *interleaving* semantics: ⌜**PAR** S_1; S_2; ⋯; S_n; **ENDPAR**⌝ can be executed in $n!$ different ways.
- Java threads have two useful properties:
  1. their scheduling policy is not specified strictly by the language, and
  2. the actual scheduling of a thread will depend on asynchronous events, etc.
- A global data structure V is created and occasionally updated by concurrently executing threads, but kept in a state such that opaque queries can be made.

Slide 13H–17

Opaque Predicates by Concurrency

- Parallel programs are hard to analyze statically: **PAR**{$S_1; S_2; \cdots; S_n$} can be executed in $n!$ different ways.
- We create a set of threads that occasionally update a global data structure V.
- V is kept in a state such that opaque queries can be made.

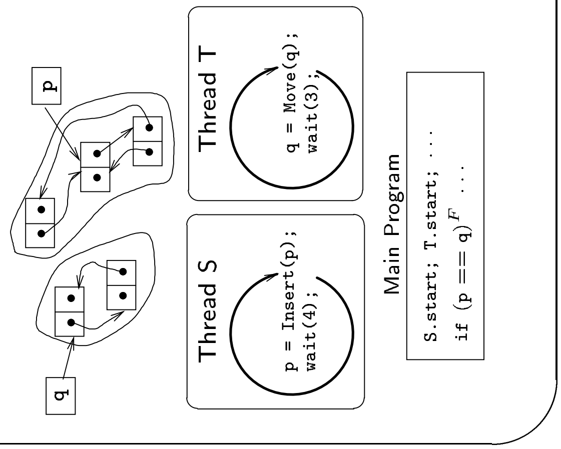


Slide 13H–18

```
class S extends Thread {
  public void run() {
    while (true) {
      int R = (int) (Math.random() * 65536);
      M.X = R*R; Thread.sleep(3);
}}
class T extends Thread {
  public void run() {
    while (true) {
      int R = (int) (Math.random() * 9300);
      M.Y = 7*R*R; Thread.sleep(2);
      M.X *= M.X; Thread.sleep(5);
}}}
public class M {
  public static int X, Y;
  public static void main(String argv[]) {
    S s = new S(); s.start();
    T t = new T(); t.start();
    if ((Y − 1) == X)^F ⇐ p
      System.out.println("Bogus code!");
    s.stop(); t.stop();
}}
```

Slide 13H–19

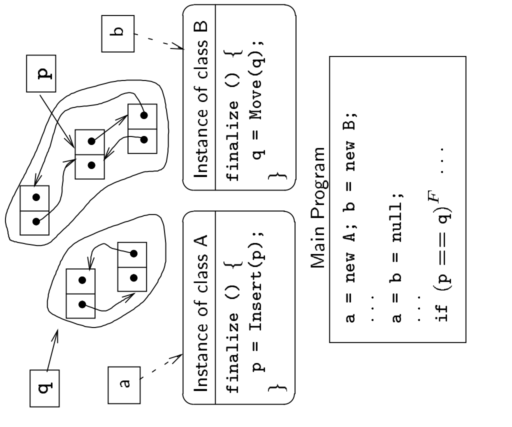
Concurrency & Aliasing



- If we let V be a dynamic data structure, we can combine interleaving and aliasing effects.
- The threads asynchronously move the global pointers p and q around in their respective components.
- This is quite resilient to deobfuscation attacks by static analysis.

Slide 13H–20

Opaque Predicates by Finalizers



- For inherently sequential applications opaque predicates based on bogus threads will be highly unstealthy.
- A similar effect can be achieved using Java's *finalizers*.
- Finalizers are invoked on an object at some (unspecified) time after it has become unreachable.

Slide 13H–21

```
class A {
  private Node p;
  public A(Node p, Node q) {
    this.p = p;
    p.splitComponent(p.reachableNodes(),
        q.reachableNodes());
  }
  public void finalize() { p.addNode_1(); }
}
class B {
  private Node p; private int i;
  public B(Node p,int i){this.p=p; this.i=i;}
  public void finalize() {p.selectNode_3(i);}
}
public class Main {
  public static void main(String argv[]) {
    Node p = new Node(), q = new Node();
    p.addNode_2(); p.addNode_1();
    A a = new A(p, q);
    B b = new B(q, 5); ···
    a = b = null; ··· ⇐ 1
    if (p == q)^F ··· ⇐ 2
}}
```

Slide 13H–22

University of Arizona

CSc 620

Security Through Obscurity

Deobfuscation

I

Copyright © 2002 C. Collberg

Obfuscate/Deobfuscate/Anti-Deobfuscate/···

Slide 13I–1

Deobfuscation

- How do we evaluate the resilience of obfuscating transformations? We must know what tools are available to an automatic deobuscator.
- So far we have assumed only static deobfuscation techniques.
- An obfuscated program can, for example, be instrumented to analyze the outcome of all predicates. Any predicate that always returns `true` over a large number of test runs may warrant further study.

Slide 13I–2

Counter-Measures Against Dynamic Analysis

- We can design opaque predicates such that several predicates have to be cracked at the same time.

$$\left\{\begin{array}{l} S_1; \\ \quad\dots \\ S_2; \end{array}\right\} \overset{\mathcal{T}}{\Rightarrow} \begin{array}{l} \texttt{int k=0;} \\ \texttt{bool } \mathtt{Q}_1\texttt{(x) \{} \\ \quad \texttt{k+=}2^{31}\texttt{; return } (P_1^T)\} \\ \texttt{bool } \mathtt{Q}_2\texttt{(x) \{} \\ \quad \texttt{k-=}2^{31}\texttt{; return } (P_2^T)\} \\ \\ \{\texttt{if } (\mathtt{Q}_1\texttt{(j)}^T)\ S_1; \\ \qquad \dots \\ \texttt{if } (\mathtt{Q}_2\texttt{(k)}^T)\ S_2;\} \end{array}$$

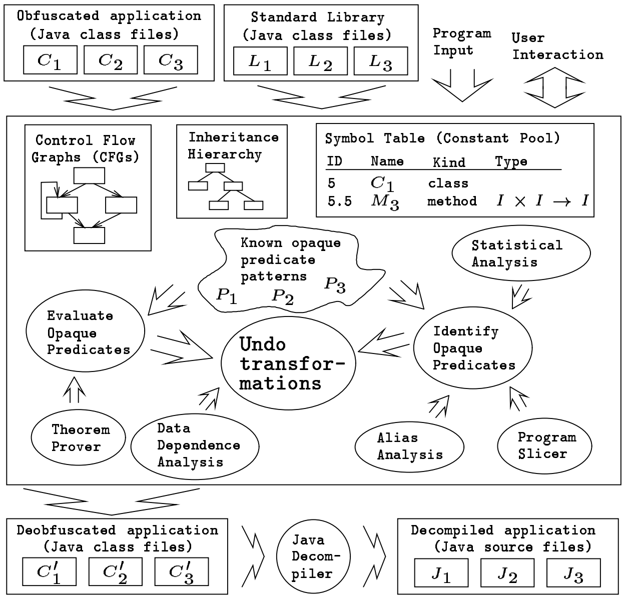
Slide 13I–3

Obfuscation vs. Deobfuscation

$$\left\{\begin{array}{l} \boxed{S_1}; \\ \boxed{S_2}; \\ \boxed{S_3}; \end{array}\right\} \overset{(a)}{\Rightarrow} \begin{array}{l} \{\texttt{if } (P^?) \\ \quad \boxed{S_1} \\ \texttt{else} \\ \quad S_1'; \\ \texttt{if } (Q^T) \\ \quad \boxed{S_2} \\ \texttt{else} \\ \quad S_2^{\text{bug}}; \\ \texttt{if } (R^F) \\ \quad S_3^{\text{bug}}; \\ \boxed{S_3};\ \} \end{array} \overset{(b)}{\Rightarrow} \begin{array}{l} \{\texttt{if } (P^?) \\ \quad \boxed{S_1} \\ \texttt{else} \\ \quad S_1'; \\ \texttt{if (True)} \\ \quad \boxed{S_2} \\ \texttt{else} \\ \quad S_2^{\text{bug}}; \\ \texttt{if (False)} \\ \quad S_3^{\text{bug}}; \\ \boxed{S_3};\ \} \end{array} \overset{(c)}{\Rightarrow} \begin{array}{l} \{\boxed{S_1}; \\ \texttt{if } (P^?) \\ \quad ; \\ \texttt{else} \\ \quad ; \\ \texttt{if (True)} \\ \quad \boxed{S_2} \\ \texttt{else} \\ \quad S_2^{\text{bug}}; \\ \texttt{if (False)} \\ \quad S_3^{\text{bug}}; \\ \boxed{S_3};\ \} \end{array} \overset{(d)}{\Rightarrow} \left\{\begin{array}{l} \boxed{S_1}; \\ \boxed{S_2}; \\ \boxed{S_3}; \end{array}\right\}$$

Slide 13I–4

A Deobfuscation Tool



Obfuscated application (Java class files) C_1 C_2 C_3

Standard Library (Java class files) L_1 L_2 L_3

Program Input

User Interaction

Control Flow Graphs (CFGs)

Inheritance Hierarchy

Symbol Table (Constant Pool)

ID	Name	Kind	Type
5	C_1	class	
5.5	M_3	method	$I \times I \rightarrow I$

Known opaque predicate patterns P_1 P_2 P_3

Statistical Analysis

Evaluate Opaque Predicates

Undo transformations

Identify Opaque Predicates

Theorem Prover

Data Dependence Analysis

Alias Analysis

Program Slicer

Deobfuscated application (Java class files) C'_1 C'_2 C'_3

Java Decompiler

Decompiled application (Java source files) J_1 J_2 J_3

Slide 13I–5

Preventive Transformations

- A preventive transformation
  - makes known automatic deobfuscation techniques more difficult (*inherent* preventive transformations), or
  - explores known problems in current deobfuscators or decompilers (*targeted* preventive transformations).
- We can prevent *slicing* by adding parameter aliases or creating large slices:

```
main() {
    int x=1;
    x = x * 3;
}
```

$\xRightarrow{\mathcal{T}}$

```
main() {
    int x=1;
    if (P^F) x++;
    x = x + V^0;
    x = x * 3;
}
```

Slide 13I–6

University of Arizona

CSc 620

Security Through Obscurity

Algorithms

J

Algorithms

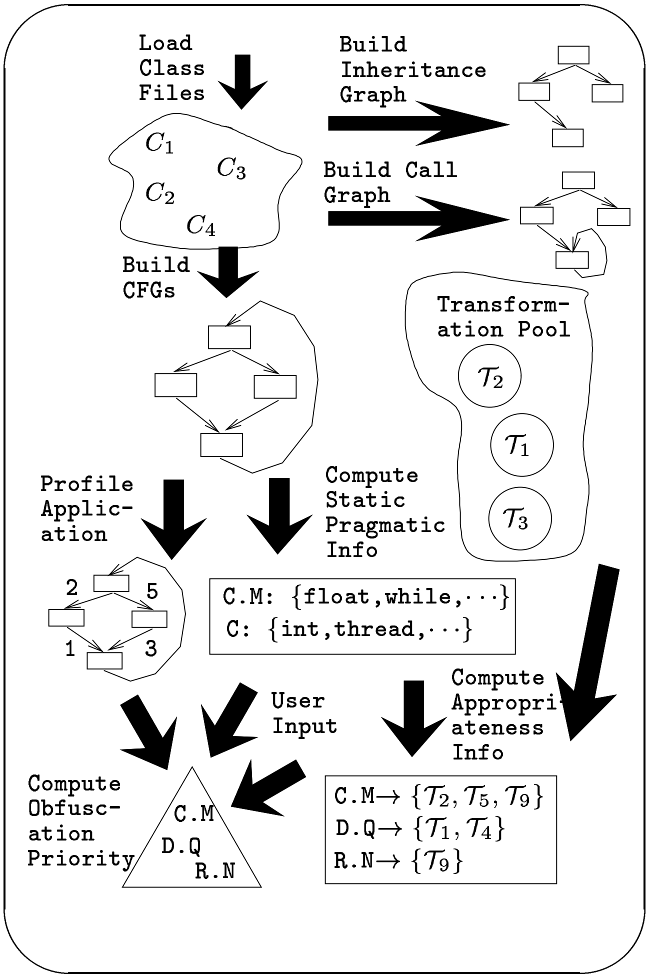
Phase1

1. Read class files and uild CFGs, a call graph, and an inheritance graph.
2. Profile the application.
3. Compute static pragmatic information (which source code objects use which Java features?).
4. Compute appropriateness information (which transformations can stealthily be applied to which source code objects?).
5. Compute obfuscation priority information (how important is it to obfuscate a particular source code object?).

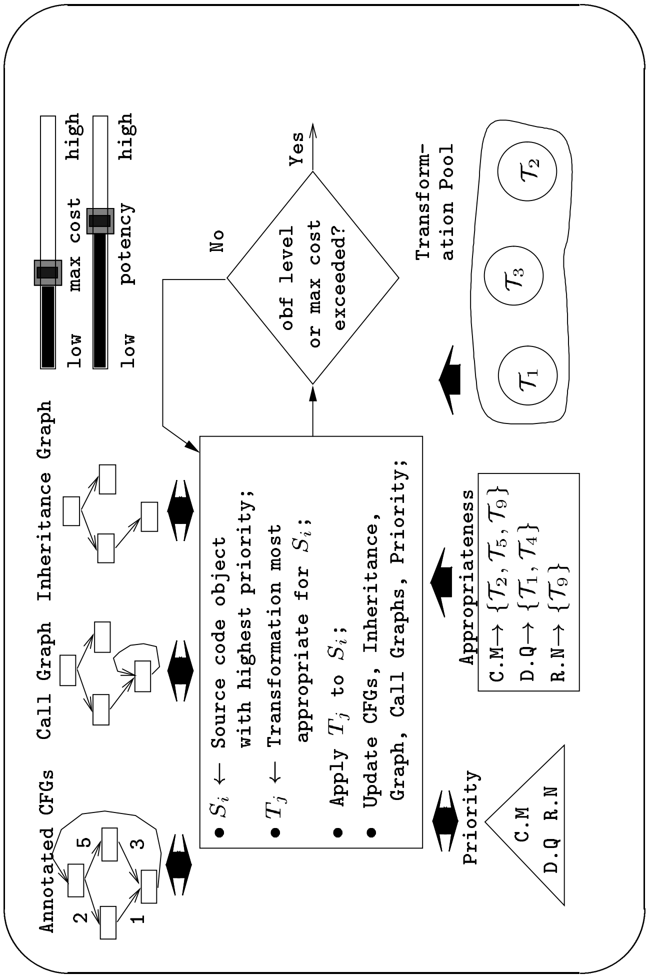
Phase2

- Apply transformations until either the max cost is exceeded or the required obfuscation level has been attained.

Slide 13J–1



Slide 13J–2



- $S_i \leftarrow$ Source code object with highest priority;
- $T_j \leftarrow$ Transformation most appropriate for S_i;
- Apply T_j to S_i;
- Update CFGs, Inheritance, Graph, Call Graphs, Priority;

Slide 13J–3

University of Arizona

CSc 620

Security Through Obscurity

Summary & Discussion

K

Summary

- It may under many circumstances be acceptable for an obfuscated program to behave differently than the original one.
- In particular, most of our obfuscating transformations make the target program slower or larger than the original.
- In special cases we even allow the target program to have different side-effects than the original, or not to terminate when the original program terminates with an error condition.
- Our only requirement is that the *observable behavior* (the behavior as experienced by a user) of the two programs should be identical.

Slide 13K–1

Obfuscation vs. Encryption

- Encryption and program obfuscation bear a striking resemblance.
- Both try to hide information and purport to do so for a limited time only.
- An encrypted document has a limited shelf-life.
- The same is true for an obfuscated application; it remains secret only for as long as sufficiently powerful deobfuscators have yet to be built.
- For quickly evolving applications this will not be a problem.
- However, if an application contains trade secrets that can be assumed to survive several releases, then these should be protected by means other than obfuscation.

Slide 13K–2

Obfuscation vs. Steganography

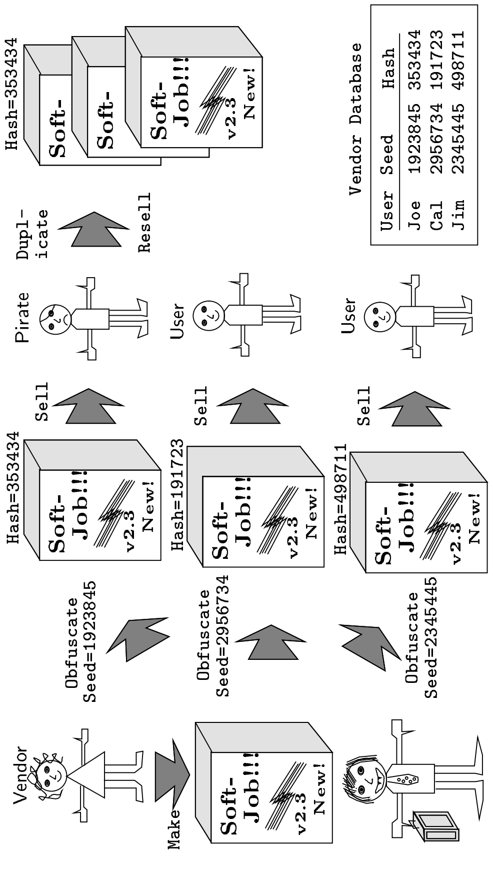
- Obfuscation is more like *steganography* than encryption. Everything is readable, but the real content is hidden within irrelevant text.

A "null cipher"

Archibald is fine. Theo and him went fishing yesterday. They caught two bass! Aftwerwards they skinned and cooked the fish, but they spit it out because it tasted mostly mud. Can you imagine! Kodak moment, if I ever saw one! Archie is such a cad, anyway. Twice this week I got a note home from school saying he can't keep his hands off the girls. Definitely Daddy's boy! Academically he takes after you, as well. Whenever I tell him to do his homework he runs and hides. No, I'm going to have to stop now!

Slide 13K–3

Watermarking by Obfuscation

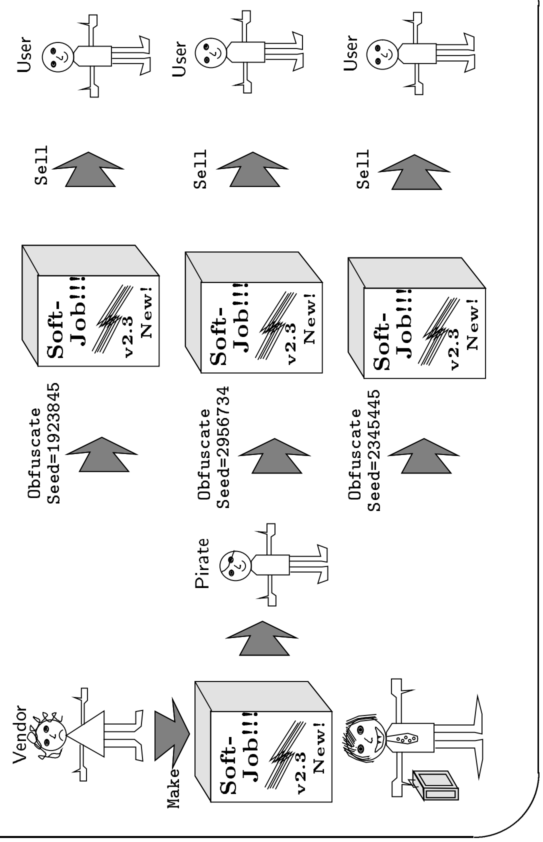


Vendor Database

User	Seed	Hash
Joe	1923845	353434
Cal	2956734	191723
Jim	2345445	498711

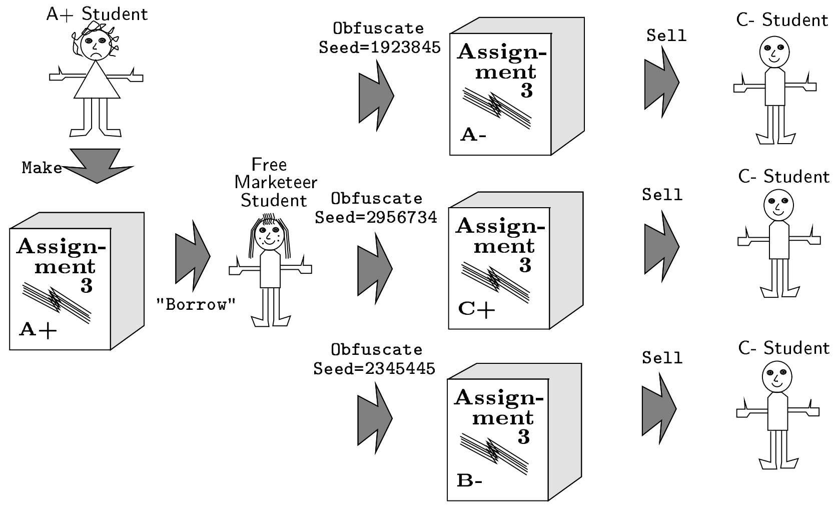
Slide 13K–4

Software Piracy by Obfuscation



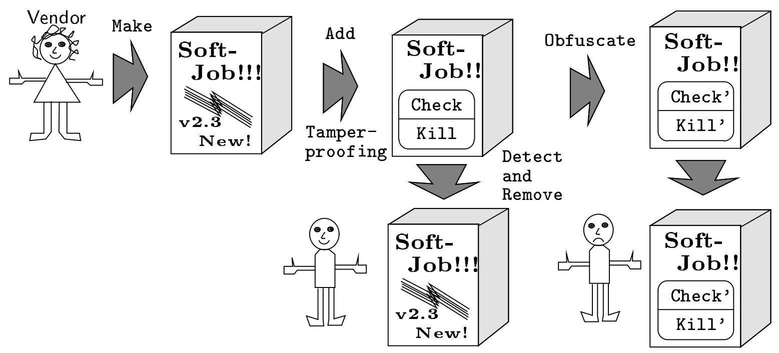
Slide 13K–5

Plagiarism by Obfuscation



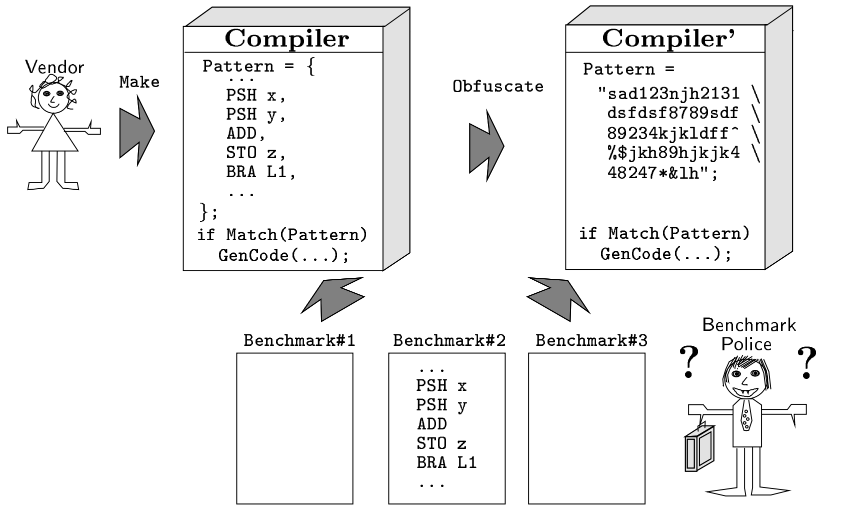
Slide 13K–6

Tamperproofing by Obfuscation



Slide 13K–7

Benchmark-Cheating by Obfuscation



```
Pattern = {
    ...
    PSH x,
    PSH y,
    ADD,
    STO z,
    BRA L1,
    ...
};
if Match(Pattern)
  GenCode(...);
```

```
Pattern =
  "sad123njh2131 \
   dsfdsf8789sdf \
   89234kjkldff^ \
   %$jkh89hjkjk4 \
   48247*&lh";

if Match(Pattern)
  GenCode(...);
```

```
...
PSH x
PSH y
ADD
STO z
BRA L1
...
```

Slide 13K–8