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.
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.

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.
**Protection by Server Execution**

- 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.

**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.

**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.

**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.
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.

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.
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.

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.
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.

Quality

- We need to be able to evaluate the quality of an obfuscating transformation.

- We will attempt to classify each transformation $T$ according to several criteria:
  - **potency** how much obscurity $T$ adds to the program,
  - **resilience** how difficult $T$ is for a deobfuscator to undo,
  - **stealth** how well code introduced by $T$ fits in with the original code,
  - **cost** how much computational overhead $T$ adds to the obfuscated application.

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

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

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.
## Definition of Potency

- We will use **potency** to measure 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 $T$ be a behavior-conserving transformation, such that $P \xrightarrow{T} P'$ transforms a source program $P$ into a target program $P'$. Let $E(P)$ be the complexity of $P$, as defined by some complexity metric.

$T_{pot}(P)$, the potency of $T$ with respect to a program $P$, measures how $T$ changes the complexity of $P$:

$$T_{pot}(P) \overset{\text{def}}{=} \frac{E(P')}{E(P)} - 1.$$
**Measures of Resilience**

- It would seem that increasing $T_{pot}(P)$ would be trivial. To increase the $\mu_2$ metric we just add some arbitrary if-statements to $P$:

  ```
  main() {
    S_1; 
    if (5==2) S_1; 
    S_2; 
    if (1>2) S_2; 
  } 
  ```

- 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.

**Measures of Stealth**

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

```plaintext
512-bit integer

if IsPrime(8375274...3853347527) then 
...
```
Measures of Execution Cost

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

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

$$T_{\text{cost}}(P) \overset{\text{def}}{=} \begin{cases} \text{dear} & \text{if executing } P' \text{ requires exponentially more resources than } P, \\ \text{costly} & \text{if executing } P' \text{ requires } O(n^p), p > 1, \text{ more resources than } P, \\ \text{cheap} & \text{if executing } P' \text{ requires } O(n) \text{ more resources than } P, \\ \text{free} & \text{if executing } P' \text{ requires } O(1) \text{ more resources than } P. \end{cases}$$

Slide 9E–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:
   $$T_{\text{qual}}(P) = (\text{low, one-way, free}).$$

2. Scramble identifier names:
   $$T_{\text{qual}}(P) = (\text{high, one-way, free}).$$

3. Remove comments:
   $$T_{\text{qual}}(P) = (\text{high, one-way, free}).$$

Slide 9E–12

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 9F–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 \textit{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 \( \{\text{trivial, weak, strong, full, one-way}\} \).
- We measure the added cost of an opaque construct on the scale \( \{\text{free, cheap, costly, dear}\} \).

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:
  - \( P^F \Rightarrow \) 's False.
  - \( P^T \Rightarrow \) 's True, sometimes False.
- Opaque predicates that are difficult for a deobfuscator to evaluate.

Trivial/Weak Opaque Constructs

- \textit{Trivial} opaque constructs can be cracked by a deobfuscator by a static local analysis:
  \[
  \begin{array}{l}
  \{ \text{int } v, a=5; b=6; \\
  \quad v^{-11} = a + b; \\
  \quad \text{if } (b > 5)^T \ldots \\
  \quad \text{if } (\text{random}(1,5) < 0)^F \ldots 
  \end{array}
  \]

- \textit{Weak} opaque constructs can be cracked using a static global analysis:
  \[
  \begin{array}{l}
  \{ \text{int } v, a=5; b=6; \\
  \quad \text{if } (\ldots) \ldots \\
  \quad \quad : (b \text{ is unchanged)} \\
  \quad \quad \text{if } (b < 7)^T \text{ a++;} \\
  \quad \quad v^{-36} = (a > 5)?v=b:b=v 
  \end{array}
  \]

Insert Dead/Irrelevant Code I

- The \( \mu_2 \) and \( \mu_3 \) metrics suggest 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:
  - \( \text{(a)} \) Insert an opaque predicate \( P^T \) into \( S \). \( P^T \) is \textit{irrelevant} code since it will always evaluate to \textbf{True}.
  - \( \text{(b)} \) Break \( S \) into two different obfuscated versions \( S^a \) and \( S^b \). \( P^T \) selects between them at runtime.
  - \( \text{(c)} \) As in \( \text{(b)} \), but introduce a bug into \( S^b \). \( P^T \) selects the correct version of the code, \( S^a \).
- We can obfuscate a loop by extending the loop condition with a \( p_x \) or \( p_y \) predicate which doesn’t affect the number of times the loop will execute.
- Here, \( p_x \) based on the fact that \( x^2 (x+1)^2 \equiv 0 \) (mod 4).

```latex
\begin{align*}
    &i=1; j=0; \\
    &\text{while } (i < 100) \{ \text{ } \\
    &\text{ } j = j + 1; \\
    &\text{ } \text{if } (j+1) \% 4 = 0 \text{ then } i++; \} \\
\end{align*}
```

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.
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:

  ```
  call m.P()
  ```

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

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 [...]  

- 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.
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.
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

Split Variables I

- Boolean variables and other variables of restricted range can be split into two or more variables. We write this as $V = [p_1, \ldots, p_k]$.
- 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 $q$.
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", "BBBBB", and "CCB".
- The values produced by G are G(1)="AAA", G(2)="BBBBB", G(3)=G(5)="CCB", and G(4)="XCB". For other argument values, G may not terminate.

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++,k<2) {S[i++]="A"; goto L5}
            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 $\mu_6$ 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 II – Splitting

\begin{align*}
    (1) \quad & \text{int } A[9]; \\
    (2) \quad & A[i] = \ldots; \\
    \quad \downarrow \quad & T \\
    (1') \quad & \text{int } A1[4], A2[4]; \\
    (2') \quad & \text{if } ((i*2)==0) A1[i/2]=\ldots \\
    \quad & \text{else } A2[i/2]=\ldots;
\end{align*}

\begin{tabular}{ccccccccccc}
    0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
    \hline
\end{tabular}

\begin{tabular}{cccc}
    0 & 1 & 2 & 3 \\
    \hline
    A1 & A0 & A2 & A4 & A6 & A8 \\
\end{tabular}

\begin{tabular}{cccc}
    0 & 1 & 2 & 3 & 4 \\
    \hline
    A2 & A1 & A3 & A5 & A7 & A9 \\
\end{tabular}
Array Transforms II – Merging

(3) int B[9], C[19];
(4) B[i] = ...;
(5) C[i] = ...;
\[ T \]

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

B: \[ B_0 \quad B_1 \quad B_2 \quad B_3 \quad B_4 \quad B_5 \quad B_6 \quad ... \quad B_9 \]
C: \[ C_0 \quad C_1 \quad C_2 \quad C_3 \quad C_4 \quad C_5 \quad C_6 \quad ... \quad C_{19} \]

\[ T \]
BC: \[ B_0 \quad C_0 \quad C_1 \quad B_1 \quad C_2 \quad C_3 \quad B_2 \quad ... \quad C_{19} \]

Array Transforms II – Folding

(6) int D[9];
(7) for(i=0; i<=8; i++)
\[ D[i]=2*D[i+1]; \]
\[ T \]
(6') int D1[i, 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
\[ D1[j,k]=2*D1[j,k+1]; \]

D: \[ D_0 \quad D_1 \quad D_2 \quad D_3 \quad D_4 \quad D_5 \quad D_6 \quad ... \quad D_9 \]
D1: \[ 0 \quad D_0 \quad D_1 \quad D_2 \quad D_3 \quad D_4 \quad 1 \quad D_5 \quad D_6 \quad D_7 \quad D_8 \quad D_9 \]

Array Transforms II – Flattening

(8) int E[2, 2];
(9) for(i=0; i<=2; i++)
for(j=0; i<=2; j++)
\[ \text{swap}(E[i, j], E[j, i]); \]
\[ T \]

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

E: \[ \begin{array}{ccc}
0 & E_{0,0} & E_{0,1} & E_{0,2} \\
1 & E_{1,0} & E_{1,1} & E_{1,2} \\
2 & E_{2,0} & E_{2,1} & E_{2,2} \\
\end{array} \]

\[ T \]
E1: \[ E_{0,0} \quad E_{0,1} \quad E_{0,2} \quad E_{1,0} \quad E_{1,1} \quad ... \quad E_{2,2} \]

Modify Inheritance I

- According to metric \( \mu_7 \), the complexity of a class \( C_1 \) grows with its
  1. \textit{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 \( C_1 \).
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.

\[ C = (V, M) \]

\[ C_1 = C_2 \oplus (V_1, M_1) \]

\[ C_2 = (V_2, M_2) \]

\[ C_3 = C_2 \oplus (V_3, M_3) \]

\[ V = V_1 \oplus V_2 \]

\[ M = M_1 \oplus M_2 \]

Modify Inheritance II – Splitting

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

\[ C = (V, M) \]

\[ C_1 = C_2 \oplus (V_1, M_1) \]

\[ C_2 = (V_2, M_2) \]

\[ V = V_1 \oplus V_2 \]

\[ M = M_1 \oplus M_2 \]

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 \( C_3 \).

\[ C_1 = (V_1, M_1) \]

\[ C_2 = (V_2, M_2) \]

\[ C_3 = C_3 \oplus (V_1', M_1) \]

\[ C_3 = C_3 \oplus (V_2', M_2) \]

\[ V_3 = V_1 \cap V_2 \]

\[ V_3' = V_1 - V_3 \]

\[ V_2' = V_2 - V_3 \]

Modify Inheritance II – Insertion

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

\[ C_1 = (V_1, M_1) \]

\[ C_2 = (V_2, M_2) \]

\[ C_3 = C_1 \oplus (V_3, M_3) \]

\[ C_3 = C_3 \oplus (V_3, M_3) \]

\[ V_1 \cap V_2 = \emptyset \]

\[ M_1 \cap M_3 = \emptyset \]

\[ C_2 = C_1 \oplus (V_2, M_2) \]

\[ C_2 = C_3 \oplus (V_2, M_3) \]
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.

Elementary Opaque Constructs

- Introduced predicates must be stealthy: i.e. 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)...`, $\mathbf{op} \in \{==, <, \leq, \geq\}$
  4. `if (n op m)...`, $\mathbf{op} \in \{==, <, \leq, \geq\}$
  5. `if (x op y)...`, $\mathbf{op} \in \{<, \leq, >, \geq\}$
  6. `if (x op RealConst)...`, $\mathbf{op} \in \{<, \leq, >, \geq\}$
- See elementary number theory textbooks...
**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.

**Aliasing – Complexity Results**

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

<table>
<thead>
<tr>
<th>Fact</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \forall x \in T, 2x \leq 8 \left( x^2 - 1 \right) )</td>
<td>The sum of the odd integers is a perfect square.</td>
</tr>
<tr>
<td>( \forall x \in T, 64 \left( 2 \left( x^2 - 1 \right) \right) - 5 )</td>
<td>The second bit of a squared number is always 0.</td>
</tr>
<tr>
<td>( \forall x \in T, 64 \left( 2 \left( x^2 - 1 \right) \right) + 1 )</td>
<td></td>
</tr>
</tbody>
</table>

**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).
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 $S_1, S_2, \ldots$.
  2. Keep a set of pointers $p_1, p_2, \ldots$ into $S_1, S_2, \ldots$.
  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 $p_2$”, “there may be a path from $p_1$ to $p_2$”, etc.
  4. Use these invariants to manufacture opaque predicates when needed.

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.

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

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

    /* addNode; is a family of functions which insert a new node after 'this'. */
    Node addNode1() {
        Node p = new Node();
        p.car = this.car;
        return this.car = p;
    }
    Node addNode2() {
        Node p = new Node();
        p.cdr = this.car;
        return this.car = p;
    }

    /* selectNode; is a family of functions which return a reference to a node reachable from 'this'. */
    Node selectNode1() { return this.car; }
    Node selectNode2() { return this.car.cdr; }
}
```
public Node selectNodes3(int n) { 
    return (n <= 0) ? this: 
    this.car.selectNodes3(n-1); 
}
public Node selectNodes3b(int n) { 
    return (n <= 0) ? this: 
    this.cdr.selectNodes3(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.carreachableNodes(reached); 
    this.cdrreachableNodes(reached); 
} return reached; 
}

/* 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)); 
}}
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.selectNode8))
      break;
    Ray theRay = view.pixelRay(y, x);
    SceneObject obj = hitObject(theRay, scene)
    if (obj != null) {
      Colour color = obj.surface.color(
              obj.bitPoint, obj.normal,
              view.eyePoint);
      Graphics.drawPoint(color, x, y);
    }
  }
}

Concurrence

- Parallel programs are more difficult to analyze statically than sequential ones. The reason is their interleaving semantics: PAR $S_1; S_2; \cdots; S_n$; ENDPAR can be executed in all 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.

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)
      System.out.println("Bogus code!");
    s.stop(); t.stop();
  }
}
For inherently sequential applications, obfuscation is used to make it difficult to understand the program's functionality, making it harder to debug and maintain.

A similar effect can be achieved using Java's finalizers. Finalizers are invoked on all objects when they are garbage collected. By carefully placing code in finalizers, obfuscation can be achieved.

```java
class A {
    private Node p;
    public A(Node p, Node q) {
        this.p = p;
        p.splitComponent(preachableNodes(),
                        qreachableNodes());
    }
    public void finalize() { p.addNode1(); }
}
class B {
    private Node p; private int i;
    public B(Node p, int i){this.p=p; this.i=i;}
    public void finalize() { p.selectNode3(i); }
}
public class Main { 
    public static void main(String argv[]) { 
        Node p = new Node(), q = new Node();
        p.addNode2(); p.addNode1();
        A a = new A(p, q);
        B b = new B(q, 5); ... 
        a = b = null; ... \( \text{if } (p == q) \) \( \text{if } (p == q) \) \( \text{if } (p == q) \) \( \text{if } (p == q) \) \( \text{if } (p == q) \) \( \text{if } (p == q) \) \( \text{if } (p == q) \)
    }
}```
- How do we evaluate the resilience of obfuscating transformations? We must know what tools are available to an automatic deobfuscator.
- 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.

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

\[
\begin{align*}
\text{int } & k=0; \\
\text{bool } & Q_1(x) \{ \\
& \text{return } (P_1) \}
\end{align*}
\]

\[
\begin{align*}
\text{bool } & Q_2(x) \{ \\
& \text{return } (P_2) \}
\end{align*}
\]

\[
\begin{align*}
\text{if } (Q_1(i)) & S_1; \\
& \ldots \\
\text{if } (Q_2(k)) & S_2;
\end{align*}
\]
**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;
      int y=1;
      if (P \land x++)
      x = x * 3;
  }
  ```

---

**Algorithms**

**Phase 1**

1. Read class files and build 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?).

**Phase 2**

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

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! Afterwards 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!