

Software protection using Obfuscation

Aleksandar Ivanovski, Toni Stojanovski
aleksandar.ivanovski@live.com, toni.stojanovski@eurm.edu.mk

Abstract— Much of the intellectual property of successful firms is embedded in their software that is purchased or installed in many of their clients. Such software is increasingly exposed to attacks by those who use reversed engineering to try to understand the structure and source code of the program and thus to gain access and use foreign intellectual property for their own purposes. In this paper is reviewed software protection from reverse engineering attacks using method of obfuscation. Today it is accepted that automated obfuscation is most sustainable method for preventing reversed engineering. Obfuscator is based on the application of code transformations, which in most cases are similar to those used in the optimization of compilers. We are describing most famous obfuscation transformations, and classify them according to their potency- the extent to which man is confused when reading the code, resilience, resilience- resistance from attacks from automated deobfuscators and price- how much additional load is added to obfuscated code. Then we have explained some possible deobfuscation techniques. This paper discusses dynamic obfuscation in details, which is most popular and most successful method for the protection of intellectual property stored in software code, but also used by hackers to hide malicious code. Finally we recommend improvements to the methods of dynamic obfuscation.

Key words - obfuscation, deobfuscation, dynamic obfuscation, software security, obfuscation techniques, software security.

I. INTRODUCTION

SUPPOSE that a programmer develops a new piece of software which contains a unique innovative algorithm. If the programmer wants to sell this software, it is usually in the programmer's interest that the algorithm should remain secret from purchasers. What can the programmer do to stop the algorithm from being seen or modified by a client? A similar issue arises when a piece of software contains a date stamp that allows it to be used for a certain length of time. How can we stop people from identifying (and changing) the date stamp?

The method for protection of software that we are going to present in this paper is obfuscation. This technique includes lot of transformations, mostly from code optimization techniques. In this paper we put accent on dynamic obfuscation. Method for software protection that is more resistant from attacks by hacker. But also it is coming back like boomerang from hacker's side. It is also used to harm to users computers and valuable software. In the beginning we

define code obfuscation. Also we are explaining measures for obfuscation, potency- the extent to which man is confused when reading the code, resilience, resilience- resistance from attacks from automated deobfuscators and price- how much additional load is added to obfuscated code. Further, we describe dynamic obfuscation, methods and techniques for dynamic obfuscation. In the end, we give some examples and extend for dynamic obfuscation, and also implementation of dynamic obfuscation for malicious software, viruses etc.

II. DEFINING OBFUSCATION

We have chosen two popular definitions for obfuscation.

A. Collberg et al

First definition is from Collberg et al [1]:

Let $P \xrightarrow{\tau} P'$ be a transformation of a source program P into a target program P' . The transformation $P \xrightarrow{\tau} P'$ 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' .

B. Barak et al

Barak et al [2] takes a more formal approach to obfuscation — their notion of obfuscation is as follows. An obfuscator O is a “compiler” which takes as input a program P and produces a new program $O(P)$ such that for every P :

- Functionality — $O(P)$ computes the same function as P .
- Polynomial Slowdown — the description length and running time of $O(P)$ are at most polynomial larger than that of P .
- “Virtual black box” property — “Anything that can be efficiently computed from $O(P)$ can be efficiently computed given oracle access to P ” [2]

With this definition, Barak et al construct a family of functions which is unobfuscatable in the sense that there is no way of obfuscating programs that compute these functions. Thus their notion of obfuscation is impossible to achieve.

This definition of obfuscation, in particular the “Virtual Black Box” property, is too strong — obfuscators will still be of

practical use even if they do not provide perfect black boxes. This definition does not give an indication of the quality of a proposed obfuscation technique (the measure says whether the transformation completely obfuscates or not). After the publication of Barak et al [2], the focus of obfuscation research has changed to designing obfuscations that are difficult, but not necessarily impossible, for an attacker to undo.

III. OBFUSCATING TRANSFORMATIONS

In this part we have explained some obfuscation techniques and also extend some of them.

A. Opaque predicates

One of the most valuable obfuscation techniques is the use of opaque predicates. An opaque predicate P is a predicate whose value is known at obfuscation time — P^T denotes a predicate which is always *True* (similarly for P^F) and $P^?$ denotes a predicate which sometimes evaluates to *True* and sometimes to *False*.

Here are some example predicates which are always *True* (supposing that x and y are integers):

$$x^2 \geq 0$$

$$x^2(x+1)^2 \equiv 0(\text{mod } 4)$$

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

Opaque predicates can be used to transform a program block B as follows:

- if (P^T) { B ; } This hides the fact that B will always be executed.
- if (P^F) { B' ; } else { B ; }. Since the predicate is always false we can make the block B' to be a copy of B which may contain errors.
- if ($P^?$) { B ; } else { B' ; }. In this case, we can have two copies of B each with the same functionality.

When creating opaque predicates, we must ensure that they are stealthy so that it is not obvious to an attacker that a predicate is in fact bogus. So we must choose predicates that match the “style” of the program and if possible use expressions that are already used within the program.

1) Extensions of Opaque Predicates

As mentioned in the previous section, one of the limitations with using opaque predicates is that often the value of an opaque predicate is true for all possible inputs.

This lead to developing dynamically opaque predicates. These are a set of predicates which all evaluate to the same result in any given run, but in different runs they may evaluate to different values. For example, if we had a block of three statements $S1$; $S2$; $S3$ and two linked predicate p and q (which evaluate to the same value in the same run) then we can obfuscate the block as follows:

```
if (p) S1; else {S1; S2; }
if (q) {S2; S3; } else S3;
```

This transformation is valid under certain conditions such as ensuring that the statements $S1$ and $S2$ do not change the

value of q . We can easily extend this to obfuscate larger blocks with a bigger set of dynamically opaque predicates but the conditions for ensuring that the transformation is valid become more complex.

B. Variable transformations

In this section, we show how to transform an integer variable i within a method. To do this, we define two functions f and g :

$$f :: X \rightarrow Y$$

$$g :: Y \rightarrow X$$

where $X \subseteq Z$ — this represents the set of values that i takes.

We require that g is a left inverse of f (and so f needs to be injective). To replace the variable i with a new variable, j say, of type Y we need to perform two kinds of replacement depending on whether we have an assignment to i or use of i . An assignment to i is a statement of the form $i = V$ and a use of i is an occurrence of i which is not an assignment. The two replacements are:

- Any assignments of i of the form $i = V$ are replaced by $j = f(V)$.
- Any uses of i are replaced by $g(j)$.

These replacements can be used to obfuscate a while loop.

C. Loops

In this section, we show some possible obfuscations of this simple while loop:

```
i = 1;
while (i < 100)
{
...
i++;
}
```

We can obfuscate the loop counter i — one possible way is to use a variable transformation. We define functions f and g to be:

```
f(i) = (2i + 3)
g(i) = (i - 3) div 2
```

and we can verify that $g \cdot f = \text{id}$.

Using the rules for variable transformation, we obtain:

```
j = 5;
while ((j - 3)/2 < 100)
{
...
j = (2 * (((j - 3)/2) + 1)) + 3;
}
```

With some simplifications, the loop becomes:

```
j = 5;
while (j < 203)
{
...
j = j + 2;
}
```

Another method we could use is to introduce a new variable, k say, into the loop and put an opaque predicate (depending on k) into the guard. The variable k performs no function in the loop, so we can make any assignment to k . As an example, our loop could be transformed to something of the form:

```

i = 1;
k = 20;
while (i < 100 && (k □ k □ (k + 1) □ (k + 1)%4 == 0))
{
...
i ++;
k = k □ (i + 3);
}

```

We could also put a false predicate into the middle of the loop that attempts to jump out of the loop.

Our last example changes the original single loop into two loops:

```

j = 0;
k = 1;
while (j < 10)
{
while (k < 10)
{
<replace uses of i by (10 □ j) + k >
k ++;
}
k = 0;
j ++;
}

```

D. Array transformations

There are many ways in which arrays can be obfuscated. One of the simplest ways is to change the array indices. Such a change could be achieved either by a variable transformation or by defining a permutation.

Here is an example permutation for an array of size n :

$p = \lambda i . (a \times i + b \pmod{n})$ where $\text{gcd}(a, n) = 1$

Other array transformations involve changing the structure of an array. One way of changing the structure is by choosing different array dimensions. We could fold a 1-dimensional array of size $m \times n$ into a 2-dimensional array of size $[m, n]$. Similarly we could flatten an n -dimensional array into a 1-dimensional array.

Before performing array transformations, we must ensure that the arrays are safe to transform. For example, we may require that a whole array is not passed to another method or that elements of the array do not throw exceptions.

E. Control-Flow flattening

The idea of this transformation is to remove control-flow constructs such as while loops so that all basic blocks have the same predecessor and successor in the control-flow graph. So, for example, suppose we have this piece of code:

```

init;
while (cond)
{ loop body; }

```

Then we could remove the while loop and replace it with a switch block which loops until an end statement is reached:

```

var = 1;
switch (var)
{ case 1 :
init; var = 2; break;
case 2 :

```

```

if (cond) var = 3; else var = 4; break;
case 3 :
loop body; var = 2; break;
case 4 :
var = 1; end; }

```

The variable `var` acts as a dispatcher and effectively controls the execution of the blocks. Each case contains an assignment to `var` (and in the last case the assignment is a dummy one).

Here is a more complicated example which has conditional statements:

```

init;
while (cond)
{ loop body;
if (test) stat1; else stat2; }

```

Again, we can use a switch block to convert the program to:

```

var = 1;
switch (var)
{ case 1 :
init; var = 2; break;
case 2 :
if (cond) var = 3; else var = 7; break;
case 3 :
loop body; var = 4; break;
case 4 :
if (test) var = 5; else var = 6; break;
case 5 :
stat1; var = 2; break;
case 6 :
stat2; var = 2; break;
case 7 :
var = 1; end; }

```

It is quite easy to reconstruct the conditional statement and the while loop from the switch block. But there are some extra levels of obfuscation. First, some dummy cases can be added to the switch to obscure the control-flow — for instance, irreducible jumps could be added. Also, a global variable, such as an array, can be used to dynamically determine the values of the dispatcher variable. Finally, pointers can be used so that static analysis of the control-flow becomes difficult to perform.

F. Merging and Splitting

As well as obfuscating individual variables, we can also work with obfuscating a number of variables together. Suppose that we want to merge two variables x and y . If we know $0 \leq x < N$ and $y \geq 0$ then we can define z as follows:

$$z = N * y + x$$

As well as merging two or more variables, we can split up one variable in two (or more) other variables.

An integer variable x can be split into two variables a and b such that

$$a = x \text{ div } 10 \text{ and } b = x \text{ mod } 10$$

For example, under this transformation, the statement $x++$ is transformed to:

$$a = (10 * a + b + 1) \text{ div } 10;$$

$$b = (b + 1) \text{ mod } 10;$$

These assignments are equivalent to:

```

if (b == 9) {a = a + 1; b = 0; } else {b = b + 1; }

```

IV. DYNAMIC OBFUSCATION

A dynamic obfuscator typically runs in two phases. The first phase, at compile-time, transforms the program to an initial configuration and then adds a runtime code transformer. During the second phase, at runtime, the execution of the program code is interspersed with calls to this transformer. As a consequence, a dynamic obfuscator turns a “normal” program into a self-modifying one.

A. Differences between static and dynamic obfuscation

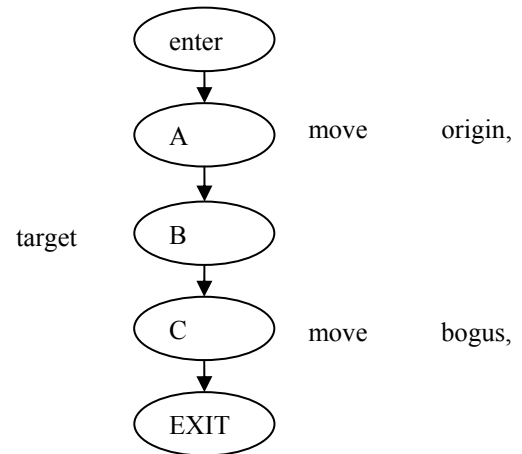
- Static obfuscations transform the code prior to execution.
- Dynamic algorithms transform the program at runtime.
- Static obfuscation counter attacks by static analysis.
- Dynamic obfuscation counter attacks by dynamic analysis
- A dynamic obfuscator runs in two phases:
 - At compile-time transform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

B. Dynamic obfuscation: replacing instructions

In following example we show how dynamic obfuscation is used to dynamic change the way of how program is obfuscated.

Obfuscate (P):

1. Select three points A, B, and C in P, such that:
 - a. A strictly dominates B,
 - b. C strictly post-dominates B, and
 - c. any path from B to A passes through C.
2. Let orig be the instruction at B.
3. Let orig be the instruction at B.
4. Let orig be the instruction at B.
5. At point A insert the instruction $move\ orig, v=B$ where v is an opaque expression that evaluates to the address of point B.
6. Similarly, at point C insert the instruction $move\ bogus, v=B$.

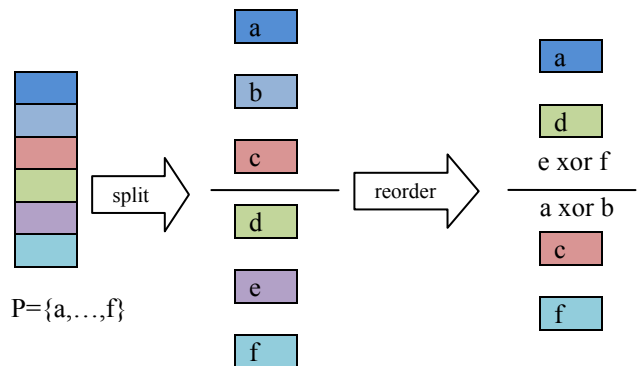


C. Dynamic obfuscation, aucsmith algorithm

The idea of this algorithm is following:

1. Split the program into pieces
2. xor them with each other
3. Add encryption to instructions

Illustration:



In general sense:

- $(A \oplus B) \oplus A = B$
- $(A \oplus B) \oplus B = A$

The idea is:

- Reorder blocks such that upper and lower blocks are alternated
- Execute a block and xor upper with lower if even iterate
- Execute a block and xor lower with upper if odd iterate

Obfuscation steps:

1. Obfuscate(P):
2. Split P into n pieces: P_0, \dots, P_{n-1} .
3. Let C_0, \dots, C_{n-1} be the memory cells in which the pieces will reside at runtime. The C_i 's are of equal size, large enough to fit the largest piece P_j .
4. Cells are, conceptually, divided into two spaces, upper ($C_0, \dots, C_{n/2-1}$) and lower ($C_{n/2}, \dots, C_{n-1}$). Each cell in the upper space partners with a cell in the lower space. Select a partner function $PF(c)$ that maps a cell number to the cell number of its partner, such as $PF(c) = c + K$, for some constant K. Let IV_0, \dots

. . . , IV_{n-1} be the initial values of cells C₀, . . . , C_{n-1}, respectively.

5. Initialize a set of equations $E1 = \{C_0 = IV_0, \dots, C_{n-1} = IV_{n-1}\}$ which expresses the current state of the memory cells as a function of their initial values.
6. Initialize a set of equations $E2 = \{\}$ which expresses how a piece P_i can be recovered in cleartext from the initial values IV_0, \dots, IV_{n-1} .
7. Initialize a table $next = \{P_0 = ?, \dots, P_{n-1} = ?\}$ which maps each subprogram P_i to the cell it should jump to in order to execute P_{i+1} .
8. make_obscure()

And the other function that is important for this algorithm is make_obscure(). So for this method we have:

For $p \in [0 \dots n-1]$ do

1. Select a cell C_c to hold piece P_p in cleartext.
2. Consult $E1$ to find the current contents V of C_c . Update $E2 := E2[P - p = V]$. Using Gaussian elimination, try to invert $E2$ (i.e. find values for all the IV_i 's). If there is no solution select another cell for P_p .
3. $next := next[P_{p-1} = C_c]$ For even (odd) p 's, simulate a mutation where every cell C_i in upper (lower) space is xor:ed with its partner cell $CPF(i)$ in lower (upper) space.
4. $E1 := E1[CPF(i) = CPF(i) \oplus C_i]$.

D. Ideas for dynamic obfuscation

Dynamic obfuscation is really great method in obfuscation theory for protection of program code. Our opinion is that in this stage dynamic obfuscation hasn't any good algorithm for code protection. Maybe the best combination of protection is to combine two areas of protection, and that is obfuscation and encryption.

We think that mixture of obfuscation and encryption can give really good results for code protection. Reasons for this are:

- o If we use AES to encrypt already obfuscate code, we can double protection level of a program
- o AES isn't cracked till today. In other words, there is no known methods that have cracked all rounds of Rijndael algorithm.

The main idea is not to obfuscate whole code that is executed, because that will significantly slow down all application. Code that needs to be transformed with transformer also should be encrypted and the transformer should also have encryption key so it can in same time decrypt obfuscated code. Maybe it will slow down code a little, but it doubles the protection of code.

It should be mentioned that in any case, dynamic obfuscation have higher level of protection that static obfuscation because it dynamically change program every time that is executed.

V. OBFUSCATING MALWARE

Today we face up with more and more sophisticated methods for hiding malicious or malware software, so it can penetrate into user OS and programs. Antivirus program constantly are trying to detect this kind of obfuscated viruses that are using dynamic obfuscation. We think that hackers are step forward.

Because fundamental theory says, we should have a reason so we can act. In our case, companies that produce antivirus programs need to detect that virus, and then build software or tools for removing viruses from user pc.

Dynamic obfuscation is a powerful tool in hands of hacker. And the second reason for this is that every time virus code is changed, dynamically, so it can be detect from antivirus program.

Because this paper is concentrated on methods for obfuscation, we will provide some basic methods for implementing dynamic obfuscation in malware software.

In the conclusion we will provide methods for deobfuscating and removal of malware software. However, only one solution for detecting malware software is deobfuscation and then providing unique removal software (antivirus) for that virus or worm etc.

A. Obfuscation methods for malware software

1) Trojan.Clampi – Virtualization

Trojan.Clampi used a commercial tool to obfuscate its code. Essentially this tool converts the existing instructions into an intermediate language. It also embeds a custom interpreter known as a virtual machine to interpret this custom language. Reverse engineering this code requires an understanding of how the virtual machine processes the custom code. While not impossible this can be a very time-consuming task. To understand Clampi one cannot simply rearrange blocks back into a readable order, but must decipher essentially an entirely new pseudo-language for each new sample.

2) Entry point obfuscation

Modifying an executable's start address, or the code at the original start address, constitutes extremely suspicious behavior for anti-virus heuristics. A virus can try to get control elsewhere instead; this is called entry point obfuscation or EPO.

3) Packers

Packers are a throwback to days of yore when the Internet was still a research toy and computer storage space was at a premium. System RAM and disks were much smaller in the 80's and early 90's. To keep the size of binary executables to an absolute minimum, so-called packing tools were popularized that encrypted and compressed files. While packers still have legitimate uses today (bundling executables with component files and commercial software protection), this technique was adopted and extended by malware authors to add polymorphism, armoring, metamorphism, EPO, and a host of other techniques aimed at evading AV scanners.

Packers offer powerful benefits to malware authors. When creating a new strain of an existing malware, if the malware author modifies most of the code but leaves parts of it intact (or picks and chooses pieces from other existing malware), the resultant executable will share patterns with its relatives. This means that if any signature exists for any piece of the antecedent, an AV scanner can match this pattern. However, packing the file with a packer means that just a tiny change in the source (for example, changing a register name) will result in a radically different binary executable. This effect is akin to how a single letter change in a lengthy document will result in a completely different cryptographic hash. There are literally

thousands of discrete packing tools out there that are used to compress, encrypt and armor malware.

4) Polypack

In 2009, University of Michigan PhD student Jon Oberheide debuted Polypack, a web-based automated file packing service. Dubbed by pundits as a "Crimeware as a Service" (CaaS) site, Polypack was in reality a service that evaluated the effectiveness of AV scanners when detecting packed malware. While it did show the ease with which someone could launch such a CaaS site, theirs is pro-bono only and password protected from the masses. What makes Polypack notably notorious is that it offers (registered) users automated access to a multitude of packers and AV scanners. The submitted file is packed by each packer and then scanned by each of the AV engines and the results displayed. It offers users a quick way to determine an effective evasive packing solution. Malware authors can use this model for obvious obfuscatory gain.

B. Protecting from obfuscated malware software

Deobfuscating simple obfuscated malware software is wary easy. Reason for this is because in obfuscated software:

- is added bogus code
- or maybe are added opaque predicates
- code is optimized, because lot of techniques for compiler optimization are used in obfuscation
- etc

In other words, till now every method that is implemented in static code obfuscation have corresponding deobfuscation method. That's the biggest and only one reason for wary low usage of static code obfuscation in malware software.

And the second obfuscation, dynamic obfuscation, is more used today in malware software, such are Trojans, worms etc. The reason for this is definition given in IV.

"The first phase, at compile-time, transforms the program to an initial configuration and then adds a runtime code transformer. During the second phase, at runtime, the execution of the program code is interspersed with calls to this transformer. As a consequence, a dynamic obfuscator turns a "normal" program into a self-modifying one."

The prevention for malware lies in runtime. Precisely in runtime code transformer. First step for program that detects malware software should be looking when in runtime will be added transformer, and the second step is to monitor execution of program code that is calling that transformer.

We think that this is the most secure way for preventing attacks from malware software.

Every algorithm till today, for preventing malware software, has global idea as it was described above.

VI. CONCLUSION

Obfuscation as a field of research has still a long way to go till find really stable methods for code protection. But in this moment it is the most sustainable method compared to watermarking, tampering and other techniques for software and intellectual protection.

Static obfuscation, as predecessor of dynamic obfuscation, provided excellent stable foundation for dynamic obfuscation.

As method we think that static obfuscation has still to be developed, but it has no shiny future. The key method for software protection lies in dynamic obfuscation. This method for protection can and will be more and more developed and updated in future. Reason for this is that in this moment it hasn't wary good level of security for protecting software from hackers, and reverse engineers.

This can be compared with software, or hardware. Every software and hardware in his early development is rudimentary, and as years are going it is upgraded and updated. That is same with dynamic obfuscation. In this moment it is a "baby" that needs to be upgraded.

Further development should be concentrated on bigger security level of converting dynamically obfuscated code in runtime.

Also further development should concentrate on:

- more methods and algorithms for dynamic obfuscation
- moving step forward from hackers and reverse engineers
- add more measures for dynamic obfuscation such as:
 - measuring incompleteness
 - measuring complexity of complete refinements

Finally, we conclude that at the moment obfuscation and its most promising variant - dynamic obfuscation are providing best protection of software.

VII. БИБЛИОГРАФИЈА

- [1] A. V. Aho, R. Sethi, and J. D. Ullman. Compilers: Principles, Techniques, and Tools. Addison-Wesley, 1986.
- [2] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology, pages 1–18. Springer-Verlag, 2001.
- [3] Jesse C. Rabeck Roger I. Khazan Scott M. Lewandowski Robert K. Cunningham. Detection of Injected, Dynamically Generated and Obfuscated Malicious Code. Massachusetts Institute of Technology.
- [4] Roberto Giacobazzi. CODE OBFUSCATION- DEFENSE STRATEGIES. Dipartimento di Informatica, Universit'a degli Studi di, Verona, Italy. 2009
- [5] William Feng Zhu, Concepts and Techniques in Software Watermarking and Obfuscation. The Department of Computer Sciences, The University of Auckland, New Zealand. August 2007
- [6] Stephen Drape, Obfuscation of Abstract Data-Types, St John's College, University of Oxford. 2004
- [7] Monirul Sharif, Andrea Lanzi, Jonathon Giffin, Wenke Lee. Impeding Malware Analysis Using Conditional Code Obfuscation. School of Computer Science, College of Computing, Georgia Institute of Technology, USA.
- [8] Haruaki Tamada, Masahide Nakamura, Akito Monden, Ken-ichi Matsumoto. INTRODUCING DYNAMIC NAME RESOLUTION MECHANISM FOR OBFUSCATING SYSTEM-DEFINED NAMES IN PROGRAMS. Proc. of IASTED International Conference on Software Engineering (IASTED SE 2008), Innsbruck, Austria, February 2008