The Goldan[sic] Obfuscator

Because sometimes readability is bad

By:
Boaz Goldstein
Or Kadouri
Guy Hizkiau

guided by:
Eitan Koch
Motivation

Bob has created a secret algorithm to draw funny cartoons. He wants to write a program based on his algorithm to sell. Users, like Alice, can now use it to create their own cartoons.
Motivation

Unfortunately, the evil Dalt Wisney company want to sell their own cartoon product. They hire corporate black-hats to get the secret algorithm for themselves.
Bob can protect his code, by obfuscating it. This means the black hats need to de-obfuscate it, *if they can.*
Do the bad guys always win?

Unfortunately, obfuscation is in essence not perfect. If the bad guys really want your secrets, they'll get them.
Do the bad guys always win?

Fortunately, we can make it take too much time, skill and money to be worth their while.
Bad guy tactic

Decompile, read the code.

```c
void shampoo (t_head head) {
  wash(head);
  rinse(head);
  // set repeate flag, if needed
  repeat(head);
}
```
Counter-measure

Strip comments, remove useful names

```c
void splurnge (blarg miffle) {
    klonk(miffle);
    goit(miffle);
    pleek(miffle);
}
```
Bad-guy tactic

Still, just read the code. The code still has structure, and can still be understood.

```c
void splurnge (blarg miffle) {
    klonk(miffle);
    goit(miffle);
    pleek(miffle);
}
```

CLEARLY BLARG IS NAVAJO FOR HEAD
Counter-measure

Hide the structure of the code.

```c
void splurnge (blarg miffle) {
    klonk(miffle);
    if(φ) {
        goit(miffle);
    }
    pleek(miffle);
}
```
Wait a moment.

Can't the bad guy just figure out that $\varphi$ is always true?

Beware, I know Boolean Algebra.
So what are we going to do about it?

This is where our implementation starts...
Introducing opaque predicates

They are predicates you can't see through.

- predicting them is NP-hard
- but we, the obfuscators, know their true identity.

It is as if they are super-predicates!
How do you create opaque predicates?

Our NP-hard problem is graph aliasing: given 2 pointers that travel along an imaginary graph, will they, at point P in the program, point to the same node?

To improve results, we use the same graph nodes for a few problems, such as connectivity and sibling-relations.
Obfuscate across functions

Spreading operations on the graph across functions makes the graph's state much harder to follow... For us as well.

Solution: pass the graph pointers back and forth between functions, all the way from their creation in the code's entry point.

Since the path involves branching, loops and recursion, the attacker must consider all possibilities, which takes waaayyyyy too much memory.
What have the good guys gained?

Obtaining any information by just reading the code is no longer an option, since it involves solving NP-hard problems.

*ASSUMING A GOOD RNG.*
Bad guy tactic

Run the code several times, see what runs and what doesn't

```java
void splurnge (blarg miffle) {
    klonk(miffle);
    if(φ) {
        goit(miffle);
    } else {
        pleek(miffle);
    }
}
```
Counter-measure

Don't always be deterministic. Make some of the predicates evaluate to either true or false.

You then have a branch, where both sides do the same thing, but differently. Each time, you effectively see different code.
What have the good guys gained?

The cost of finding which parts run and which don't is significantly increased, since various run paths can make some branches unlikely but possible, while others are still constant.
Bad guy tactic

Pay the increased cost, target "if" statements to improve detection.

```java
void splunrge (blarg miffle) {
    klonk(miffle);
    if(ϕ) {
        goit(miffle);
    } else {
        pleek(miffle);
    }
}
```
Counter-measure

Unfortunately "if" statements are as hard to hide as pink elephants.

We must use something that stands out less.
Introducing: opaque exceptions!

Place code that will result in Null Pointer Exceptions at well-planned points. Make some of them non-deterministic.

Now any line of code is suspect!
What have the good guys gained?

Locating branching points is now a hard task in its own right, in addition to predicting them.

Once again, the cost of understanding the code has skyrocketed.
So, my code is now safe, right?

No. It will always be in reach if someone wants it enough. It is, however, further away from the grasp of anyone who isn't a well-funded corporation.

We tested the obfuscated code against 2 de-obfuscators (JMD and JDO). They both failed.

To improve results, use multiple methods of obfuscation.
How good is it?

We used some standard software metrics to estimate how much more we complicated the code.

![Bar chart showing change ratio for test1, test2, test3, and test4 with categories for cyclomatic complexity, nested complexity, and code length.]

WHAT IS ARE THE METRICS?
A word about speed

In most of our tests the speed difference was insignificant compared to Java's load times.

We did however construct a specially designed edge-case where there was a X2000 drop in speed.

This type of obfuscation isn't designed for a whole program, but rather for a special component.
A word about stealth

Our code isn't inherently stealthy. However, we achieved stealth using a class-mangling obfuscater named Dash-O that merged our code with the target program's code.

A second method to hide our code was with the JDO de-obfuscator which left the code running but made it no longer decompile.
since this is the last slide would you like to ask any questions?

so that i can answer them