Independence from Obfuscation
A Semantic Framework for Diversity

Riccardo Pucella

Joint work with Fred B. Schneider

CSFW
July 2006
Why Obfuscation?

Replicated server scenario:
- Attackers exploit implementation details.
- Defense: replica independence

Artificially create diversity:
- Relocate/pad runtime stack
- Rearrange basic blocks and code within basic blocks
- Change system calls or instruction opcodes
Our Goals

**Ultimate goal:** A precise characterization of obfuscation as a defense mechanism

**Realistic goals:**
- Develop models to understand obfuscation
- Determine effectiveness by comparing to other defenses
Obfuscators

Obfuscator $T$ transforms programs $P$ into *morphs* $T(P,K)$ using random key $K$:

- Source-to-source translation
- Object-level binary rewriting
- Compilation under different strategies

Semantics of morph $T(P,K)$: a set of possible execution histories
Attacks on Morphs

Attacks equated with inputs (non-assumption):

- **Interface attacks:** obfuscation cannot blunt attacks that exploit the semantics of that (flawed) interface
- **Implementation attacks:** obfuscation can blunt attacks that exploit implementation details

An input is a resistable attack relative to T and K₁,…,Kₙ if T(P,K₁),…,T(P,Kₙ) behave differently on that input

... Depends on what we mean by “differently”
Equivalence of Executions

“Differently” is in the eye of the beholder:
– Morphs can perform state changes differently
– Morphs can lay out memory differently
– Morphs can represent data differently

“Differently” captured abstractly using a relation $B$
– $(\sigma_1, \ldots, \sigma_n) \in B(P, K_1, \ldots, K_n)$ iff executions $\sigma_1, \ldots, \sigma_n$ have the same behavior
– $B$ need not be an equivalence relation(!)
How Effective is Obfuscation?

What attacks are blunted?
  – Nobody knows!

What attacks are blunted by typing?
  – Another commonly advocated defense

But, type systems and obfuscation seem to defend against the same kind of attacks…

Type systems =? obfuscation
An Exact Type System

For an obfuscator T and keys K1, ..., Kn:

- Nonstandard type system that exactly captures resistable attacks relative to T and K1, ..., Kn:
  - Before any output, execute the different morphs and compare outputs before proceeding

**Theorem**: Type error signaled if and only if resistable attack relative to T and K1, ..., Kn.
Dealing with Unspecified Keys

Don’t know in advance the set of keys, or the set might change (e.g. proactive obfuscation):

– Important to identify attacks relative to unspecified sets of keys

A resistable attack relative to T is a resistable attack relative to T and \textbf{some} finite set of keys
A Probabilistic Approximation

1. Choose keys K1, ..., Kn at random

2. Use exact type system with keys K1, ..., Kn
   - Identifies resistable attacks relative to T and K1, ..., Kn
   - May miss resistable attacks relative to T and other keys
   - Some probability of identifying a resistable attack relative to some finite set of keys

More precise type systems:
   language- and obfuscator-dependent!
Example Program:
Buffer Overflows

main(i:int) {
    var x : int;
    buf : int[3];
    x := 99;
    buf[i] := 42;
    print(x);
}

No checks on:
– Pointer arithmetic
– Array reference

On inputs 0,1,2
– Output is 99

On input -1
– Output is 42
Example Obfuscation:
Address Randomization

Ensure memory outside a buffer cannot be accessed reliably [Bhaktar et al. 2003]

Obfuscator $T_{addr}$ with keys ($l_0, d, \Pi, M_{init}$)

- $l_0$: start of stack
- $d$: padding size
- $\Pi$: permutations
- $M_{init}$: initial memory
Implementation of Calls

Usual stack:
- arg 1
- ...
- arg n
- Return address
- local 1
- ...
- local n

$T_{\text{addr}}$-morphs stack:
- d padding
- arg $\pi(1)$
- ...
- arg $\pi(n)$
- d padding
- Return address
- d padding
- local $\pi(1)$
- ...
- local $\pi(n)$
- d padding
Resistable Attacks for $T_{\text{addr}}$

\begin{verbatim}
main(i:int) {
    var x : int;
    buf : int[3];
    x := 99;
    buf[i] := 42;
    print(x);
}
\end{verbatim}

0,1,2 are not resistable attacks relative to $T_{\text{addr}}$

-1 is a resistable attack relative to $T_{\text{addr}}$
An Impossibility Result

**Earlier:** Type systems capture resistable attacks relative to $T_{\text{addr}}$ and a fixed set of keys.

**Theorem:** No computable dynamic type system can signal a type error for an input if and only if that input is a resistable attack relative to $T_{\text{addr}}$.

The best we can do is approximate
Approximation: Strong Typing

Cf. CCured, Cyclone

– Type of direct values (integer)
– Type of pointers (plus allowed range)

Type error: dereferencing a pointer out of range

Theorem: If resistable attack relative to $T_{addr}$, then strong typing signals a type error
main () {
    var a : int[5];
    x : int;
    x := a[10];
    print(x);
}
What Approximation Do We Get?

```java
main () {
    var a : int[5];
    x : int;
    x := a[10];
    print(x);
}
```

Appropriately signals a type error

Different morphs with different initial values in a[10] produce different outputs
What Approximation Do We Get?

```java
main () {
    var a : int[5];
    x : int;
    x := a[10];
    print(0);
}
```

Still signals a type error

The value read from `a[10]` has no observable effect!

But all morphs output 0, so no resistable attack present.
A More Accurate Type System

Track integrity of values by adding new type
- Type \textbf{low}: different value in different morphs
- When dereferencing a pointer out of range, value gets type \textbf{low}
- PC gets type \textbf{low} if control flow depends on value of type \textbf{low}

Type error: output depends on a value of type \textbf{low}

\textbf{Theorem}: If a resistable attack relative to $T_{\text{addr'}}$ then type system signals a type error
main() {
    var a : int[5];
    x : int;
    x := a[10];
    if (x=0) then
        print(1);
    else
        print(2);
}

What Approximation Do We Get?
What Approximation Do We Get?

```java
main() {
    var a : int[5];
    x : int;
    x := a[10];
    if (x=0) then
        print(1);
    else
        print(2);
}
```

Appropriately signals a type error at either of these points

Control flow depends on \( x \), which carries a value of type **low**
What Approximation Do We Get?

```cpp
main() {
    var a : int[5];
    x : int;
    x := a[10];
    if (x=x) then
        print(1);
    else
        print(2);
}
```

Now every morph outputs 1 because x=x is always true.

But signals a type error, even though no resistable attack occurs.

Presumably, we can take care of x=x as a special case...
What Approximation Do We Get?

... but undecidable in general whether \( f(x) \) always true

Just a special case of impossibility theorem

**Key point:** limited by how precisely can track information flow

```c
main() {
    var a : int[5];
    x : int;
    x := a[10];
    if (f(x)) then
        print(1);
    else
        print(2);
}
```
Conclusions

• Initiated a theoretical study of obfuscation as a defense mechanism
  – In particular, compared with type systems

• We have ignored the probabilities!
  – In practice, probabilities matter
    • What’s the probability that an attack is blunted?
  – Depend on how much diversity is introduced by obfuscation
  – Seem difficult to obtain
Type Systems vs Obfuscation

• Type systems:
  – Prevent attacks (always - not just probably)
  – If static, add no run-time cost
  – Not always part of the language

• Obfuscation:
  – Works on legacy code
  – Doesn’t always defend