Slicing Obfuscations

Anirban Majumdar

University of Trento

Italy

anirban@disi.unitn.it

Obfuscation: A functionality-preserving and secrecy-enhancing transformation

- We doubt there is a precise and efficiently computable measure of the work required by a reasonably competent adversary to discover a secret protected by obfuscation.

- We treat obfuscation as a heuristic process, looking for transformations which are *difficult* but not *impossible* to reverse engineer.

- Thinking in terms of a *virtual black-box* an obfuscation function is a *failure* if an adversary could learn something from an examination of the obfuscated version of the program that cannot be learned (in roughly the same amount of time) by merely executing the program repeatedly.

A framework for obfuscation

- We restrict our attention to an imperative language which contains assignments, conditionals, loops, and compositions.
- We extend a previous work on using functional programs and data refinement for imperative programs.
- Benefits: we can generalise previous obfuscations, and we can also prove their correctness.

Attack model - Program Slicing

- A reverse engineering technique often used to aid program comprehension.
- A slice consists of the program parts that potentially affect the values computed at a particular point.
- We slice our unobfuscated program and use this information to create obfuscations that are targeted to restrict the effectiveness of slicing.
- We consider the nodes from the SDG that are left behind after slicing – we call such nodes the *orphans*.
- Add in obfuscations that create dependencies between the slicing variable and the variables contained within the orphans.

A Particular Example

As an example, consider the program *wc* which counts the number of lines (*nl*), words (*nw*) and characters (*nc*) in a file.

```
wc() {
  int c, nl = 0, nw = 0, nc = 0, in;
  in = F;
  while ((c = getchar())! = EOF) {
    nc ++;
    if (c ==' ' || c ==' \n' || c ==' \t')
      in = F;
    else if (in == F)
      {in = T; nw ++;}
    if (c ==' \n') nl ++;          }
  out(nl, nw, nc);   }
```

A Particular Example

As an example, consider the program *wc* which counts the number of lines (*nl*), words (*nw*) and characters (*nc*) in a file.

The backwards slice from *nl*.

```
wc() {
  int c, nl = 0, nw = 0, nc = 0, in;
  in = F;
  while ((c = getchar())! = EOF) {
    nc ++;
    if (c ==' ' || c ==' \n' || c ==' \t')
      in = F;
    else if (in == F)
      {in = T; nw ++;}
    if (c ==' \n') nl ++;          }
  out(nl, nw, nc);    }
```

A Particular Example

As an example, consider the program *wc* which counts the number of lines (*nl*), words (*nw*) and characters (*nc*) in a file.

The backwards slice from *nl* …

Our goal is to include these orphans in the slice.

```
wc() {
  int c, nl = 0, nw = 0, nc = 0, in;
  in = F;
  while ((c = getchar()) != EOF) {
    nc++;
    if (c == ' ' || c == '\n' || c == '\t')
      in = F;
    else if (in == F)
      {in = T; nw++;}
    if (c == '\n') nl++;
  }
  out(nl, nw, nc);
}
```

An Example Obfuscation

As an obfuscation, we add a bogus predicate (that is always false) to create dependencies.

```
wc-obf1() {
  int c, nl = 0, nw = 0, nc = 0, in;
  in = F;
  while ((c = getchar())! = EOF) {
    nc ++;
    if (c ==' ' || c ==' \n' || c ==' \t')
      in = F;
    else if (in == F)
      {in = T; nw ++;}
    if (c ==' \n') nl ++;          }
  out(nl, nw, nc);   }
```

An Example Obfuscation

As an obfuscation, we add a bogus predicate (that is always false) to create dependencies.

This predicate uses the invariant:

$$nc \geq nw \ \wedge \ nc \geq nl$$

```
wc-obf1() {
  int c, nl = 0, nw = 0, nc = 0, in;
  in = F;
  while ((c = getchar()) != EOF) {
    nc++;
    if (c == ' ' || c == '\n' || c == '\t')
      in = F;
    else if (in == F)
      {in = T; nw++;}
    if (c == '\n') nl++;
    if (nl > nc) nw = nc + nl;
    else {if (nw > nc) nc = nw - nl;} }
  out(nl, nw, nc);    }
```

An Example Obfuscation

As an obfuscation, we add a bogus predicate (that is always false) to create dependencies.

The backwards slice from nl.

Now we've included all of the orphans in the slice for nl.

```
wc-obf1() {
  int c, nl = 0, nw = 0, nc = 0, in;
  in = F;
  while ((c = getchar())! = EOF) {
    nc ++;
    if (c ==' ' || c ==' \n' || c ==' \t')
      in = F;
    else if (in == F)
      {in = T; nw ++;}
    if (c ==' \n') nl ++;
    if (nl > nc) nw = nc + nl;
    else {if (nw > nc) nc = nw − nl;} }
  out(nl, nw, nc);   }
```

An Example Obfuscation

As an obfuscation, we add a bogus predicate (that is always false) to create dependencies.

We have also included the orphans of the slices for the other two output variables.

```
wc-obf1() {
  int c, nl = 0, nw = 0, nc = 0, in;
  in = F;
  while ((c = getchar())! = EOF) {
    nc ++;
    if (c ==' ' || c ==' \n' || c ==' \t')
      in = F;
    else if (in == F)
      {in = T; nw ++;}
    if (c ==' \n') nl ++;
    if (nl > nc) nw = nc + nl;
    else {if (nw > nc) nc = nw − nl;} }
  out(nl, nw, nc);   }
```

Orphans and Residues

For each $v_i \in V_O$ we can define:

$$residue(M, v_i) = M \backslash SL_i$$

So the *residue* of a slice is defined to be the set of points that are orphaned. Using this concept, we can define a slicing obfuscation as follows:

Definition An obfuscation $\mathcal{O}$ is a **slicing obfuscation** for a program P and a variable v_i if it decreases the size of the residue (the number of orphaned points), *i.e.*

$$|residue(P, v_i)| > |residue(\mathcal{O}(P), \mathcal{O}(v_i))|$$

Residue Metrics

Compactness Compactness measures the total number of orphaned points in relation to the size of the method.

$$C(M) = \frac{|RES_{un}|}{|M|}$$

MinDensity The minimum density is the ratio of the smallest residue in a method to the method length.

$$MinD(M) = \frac{1}{|M|} \min_i |residue(M, v_i)|$$

Density Density compares the average residue size to the method size.

$$D(M) = \frac{1}{|V_O|} \sum_{i=1}^{|V_O|} \frac{|residue(M, v_i)|}{|M|}$$

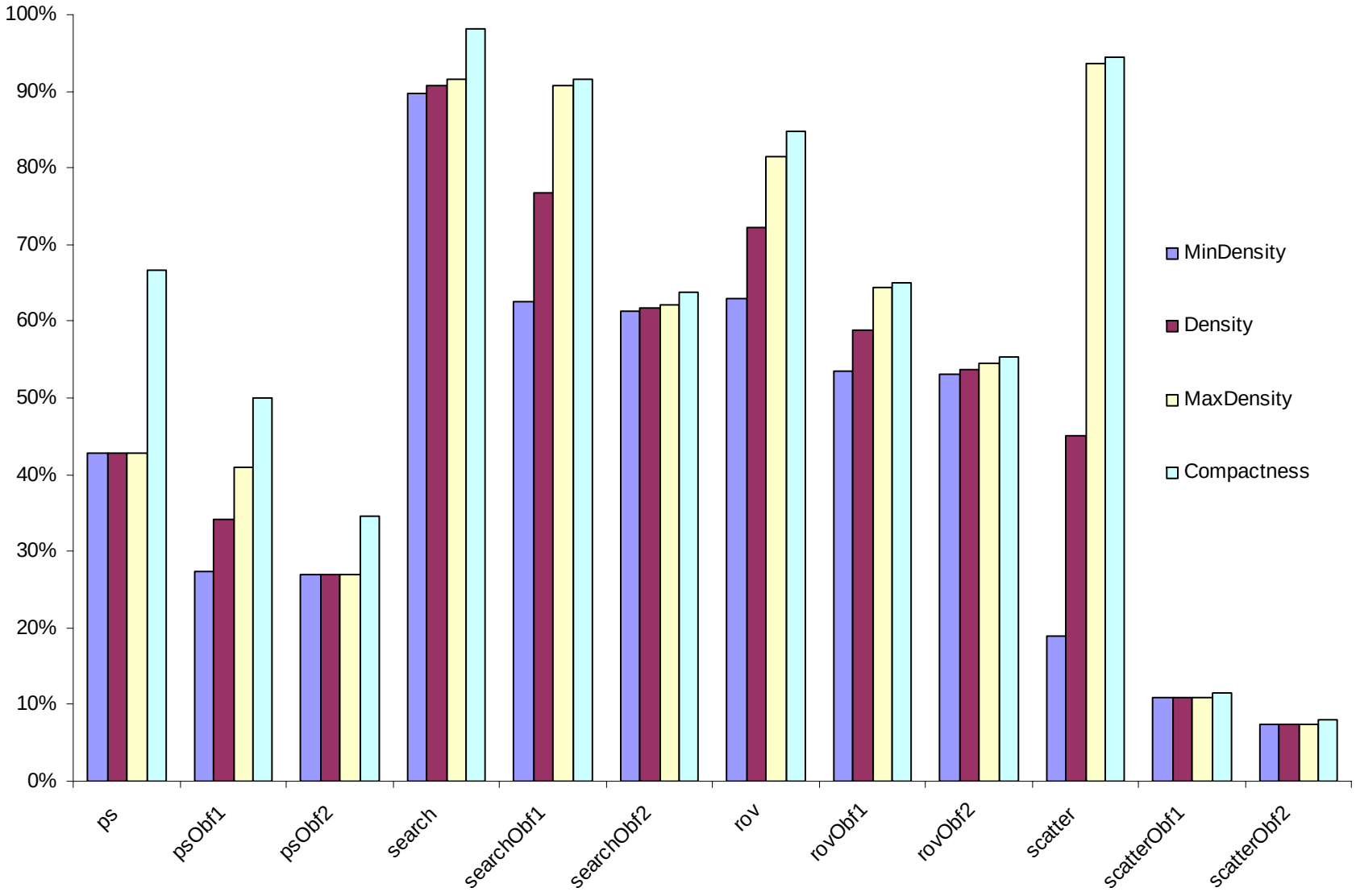
MaxDensity The maximum density is defined to be the ratio of the largest residue in a method to the method's length.

$$MaxD(M) = \frac{1}{|M|} \max_i |residue(M, v_i)|$$

Table of Results

Method M	$\|M\|$	$\|V_O\|$	For each v_i the residue size $\|RES_i\|$						$\|RES_{un}\|$	$MinD(M)$	$D(M)$	$MaxD(M)$	$C(M)$
ps	21	2	$prod$	9	sum	9			14	42.9%	42.9%	42.9%	66.7%
$psObf1$	22	2	$prod$	6	sum	9			11	27.3%	34.1%	40.9%	50.0%
$psObf2$	26	2	$prod$	7	sum	7			9	26.9%	26.9%	26.9%	34.6%
$search$	107	2	n	98	$secs$	96			105	89.7%	90.7%	91.6%	98.1%
$searchObf1$	120	2	n	75	$secs$	109			110	62.5%	76.7%	90.8%	91.7%
$searchObf2$	127	2	n	78	$secs$	79			81	61.4%	61.8%	62.2%	63.8%
rov	124	2	$fuel$	101	$dist$	78			105	62.9%	72.2%	81.5%	84.7%
$rovObf1$	129	2	$fuel$	69	$dist$	83			84	53.5%	58.9%	64.3%	65.1%
$rovObf2$	132	2	$fuel$	70	$dist$	72			73	53.0%	53.8%	54.5%	55.3%
$scatter$	143	3	si	27	ru	32	i	134	135	18.9%	45.0%	93.7%	94.4%
$scatterObf1$	148	3	si	16	ru	16	i	16	17	10.8%	10.8%	10.8%	11.5%
$scatterObf2$	150	3	si	11	ru	11	i	11	12	7.3%	7.3%	7.3%	8.0%

Graph of Results



Future work

- Develop heuristics for combining and placing obfuscations in an order which maximises the difficulty of de-obfuscation.
- Remove some of our restrictions: *e.g.* allow non-exact arithmetic, pointers, and other imperative constructs.
- The abstraction and conversion functions can remain visible in the code and so we should try to combine these functions with surrounding statements.

Orthogonal Client Replacement Model

ALGORITHM:

- Orthogonal client generation
- **INPUT**
- C: Client, S: Server
- **OUTPUT**
- Ci: Next client, Si: next server
- **BEGIN**
- 1 Ci := C
- 2 While MaxSimilarity(Ci, {C1 ... Ci-1}) > SimThr
- 3 C' := RandomTransform(C)
- 4 C := C'
- 5 (Ci, Si) := MoveCompToServer(C')
- 6 End While
- 7 Output (Ci, Si)
- **END**

Conclusion

- Proposed a new approach of designing obfuscations by attacking a program first then defending against further attacks
- Created obfuscations that have false data dependencies. Aim was to include statements in the slice that are orphaned.
- We used data refinement and a state-function view of program semantics to create a framework in which we can specify data obfuscations.
- Our framework allows us to prove the correctness of obfuscations and to create generalised obfuscations.

Questions

Modelling statements as functions

We suppose that a statement takes an initial state as input and returns a new state.

For example:

- assignment

$$(x := e)(\sigma_0) = \sigma_0 \oplus \{x \mapsto e[x_0/x]\} \text{ where } x \mapsto x_0 \in \sigma_0$$

- conditional

$$(\textbf{if } p \textbf{ then } T \textbf{ else } E)(\sigma_0) = \begin{cases} T(\sigma_0) & \text{if } p(\sigma_0) \\ E(\sigma_0) & \text{otherwise} \end{cases}$$

Abstraction and Conversion functions

We suppose that an obfuscation is a data refinement and so an obfuscation $\mathcal{O}$ will act on a state σ to produce a new state $\mathcal{O}(\sigma)$.

We require that $\sigma \rightsquigarrow \mathcal{O}(\sigma) \Leftrightarrow (\sigma = af(\mathcal{O}(\sigma))) \wedge I(\mathcal{O}(\sigma))$ where af is the abstraction function for the obfuscation and I is an invariant.

To perform the obfuscation, we need a conversion function cf which satisfies $cf; af \equiv skip$

Proving correctness

Using our framework, we can prove the correctness of our data obfuscations by establishing an equivalence of the form:

$$af;\ B \equiv \mathcal{O}(B);\ af$$

Our correctness proofs have four stages:

- Converting to simultaneous equations
- Substituting values
- Removing redundant definitions
- Converting back to code

Correctness equations

An obfuscated block of statements $\mathcal{O}(B)$ is said to be **correct** with respect to B if satisfies:

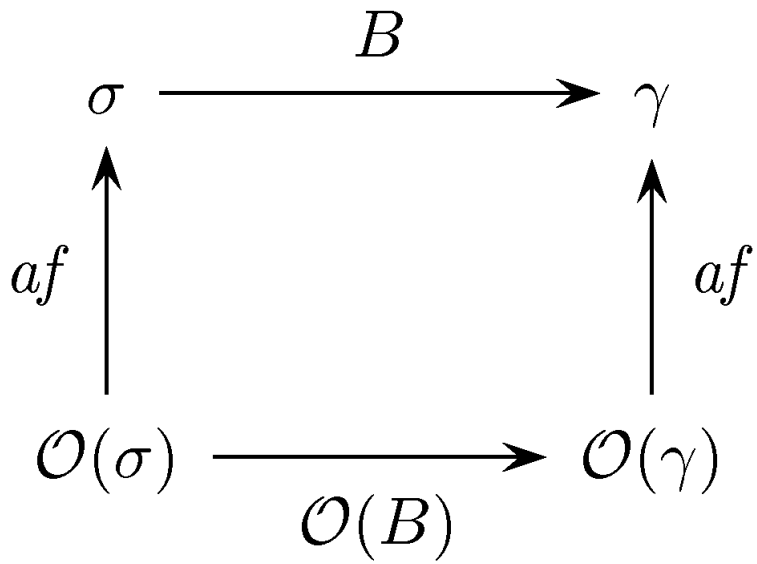
$$(\forall \sigma) \bullet \sigma \rightsquigarrow \mathcal{O}(\sigma) \Rightarrow B(\sigma) \rightsquigarrow \mathcal{O}(B)(\mathcal{O}(\sigma))$$

Using the commuting diagram:

$$af;\ B \equiv \mathcal{O}(B);\ af$$

Or:

$$B \equiv cf;\ \mathcal{O}(B);\ af$$



Slicing Metrics

Tightness measures the number of statements common to every slice: $T(M) = \frac{|SL_{int}|}{|M|}$

Minimum Coverage is the ratio of the smallest slice in a method to its length: $Min(M) = \frac{1}{|M|} \min_i |SL_i|$

Coverage compares the length of slices to the length of the entire method: $C(M) = \frac{1}{|V_O|} \sum_{i=1}^{|V_O|} \frac{|SL_i|}{|M|}$

Maximum Coverage is the ratio of the largest slice in a method to its length: $Max(M) = \frac{1}{|M|} \max_i |SL_i|$

Overlap is a measure of how many statements in a slice are found in all the other slices: $O(M) = \frac{1}{|V_O|} \sum_{i=1}^{|V_O|} \frac{|SL_{int}|}{|SL_i|}$

Results for *wordcount*

Method M	$\|M\|$	$\|Vo\|$	Size of nl	Size of nw	Size of nc	$\|SLint\|$
wc	36	3	15	20	10	7
wc-obf1	42	3	30	30	30	28

	$T(M)$	$Min(M)$	$C(M)$	$Max(M)$	$O(M)$
wc	19.4%	27.8%	41.7%	55.6%	50.6%
wc-obf1	66.7%	71.4%	71.4%	71.4%	93.3%

Measurements obtained from CodeSurfer