# An Obfuscation for Binary Trees

Stephen Drape*
*The University of Auckland, Auckland, New Zealand

***Abstract*— An obfuscation is a program transformation which aims to make a program "harder to understand" so that reverse engineering of that program becomes more difficult. This paper presents a fresh and promising approach to obfuscation by considering the obfuscation of objects, whose methods are modelled as functional programs. As an example of our approach, we concentrate on an object of binary trees.**

**We use the formal method approach to program correctness which enables us to establish a framework that produces obfuscations of tree objects that exploit properties of trees. Establishing the correctness of imperative obfuscations can be a challenging task but our approach enables this to be achieved easily for *all* our obfuscations.**

**Keywords:** Obfuscation, Refinement, Binary Trees

## I. Introduction and Related Work

An obfuscation is a behaviour-preserving program transformation which aims to make a program "harder to understand" so that reverse engineering of the program is more difficult. Our contribution to the study of obfuscation is to consider obfuscation as *data refinement* [5] and we propose a framework for objects which we view (for the purpose of refinement) as *data-types*, calling the methods *operations*. We consider abstract data-types and define obfuscations for the *whole* data-type. As an example of our approach, we present a simple data-type for binary trees and give obfuscations for this data-type.

Two current approaches to obfuscation are discussed in Collberg *et al.* [3] and Barak *et al.* [1]. Collberg *et al.* define metrics which try to qualify "harder to understand" and consider object-oriented obfuscations. Barak *et al.* take a formal approach to obfuscation and prove that their notion of obfuscation is impossible to achieve. Their definition of obfuscation is too strong for our purposes and so we consider a weaker notion. We will be content with obfuscations that are *difficult* (but not necessarily *impossible*) to undo. In this paper, we do not explicitly give a definition pertinent to our approach — a possible definition is given in [6]. This definition, called an "assertion obfuscation", requires that a list of assertions is supplied when data-types are defined. The definition is based on how much "harder" it is to prove these assertions for obfuscated operations. Thus when producing data-types obfuscations we could, for instance, ensure that our obfuscations have more clauses in their definitions or use more complicated data structures.

The current view of code obfuscation [3] concentrates on concrete data structures such as variables and arrays and considers obfuscations for object-oriented languages. With that view, we can represent trees imperatively using arrays — an example conversion is given in [4]. We could then use standard array obfuscations [3] to obfuscate our operations. Instead we consider abstract data-types — a local state accessible by only declared operations. Why do we go to the effort of using data-types rather than using arrays directly? Apart from making proofs of correctness easier, using data-types gives us an extra level in which to add obfuscations. Going immediately to arrays forces us to think in array terms and we would have only array obfuscations at our disposal. In particular, since we view a tree as an abstract data-type then the usual tree transformations (such as swapping two subtrees) are naturally available; they would be more difficult to conceive using arrays. As a result we have two opportunities for obfuscation; the first using our new data-types approach and the second using standard imperative methods.

We use the functional language Haskell [8] to provide a framework for specifying data-types and obfuscations (note that we are not aiming to obfuscate Haskell code but to use Haskell as a modelling language). Since we use Haskell for our new approach we have the benefits of the elegance of the functional style and the abstraction of side-effects.

$$
\begin{array}{r}
\hline
Tree\ \alpha ::= Null \mid Fk\ (Tree\ \alpha)\ \alpha\ (Tree\ \alpha) \\
\hline
\mathsf{flatten} :: Tree\ \alpha \rightarrow List\ \alpha \\
\mathsf{mem} :: \alpha \rightarrow Tree\ \alpha \rightarrow \mathbb{B} \\
\mathsf{mkT} :: List\ \alpha \rightarrow Tree\ \alpha \\
\hline
\end{array}
$$

Fig. 1. Data-Type for Binary Trees

## II. Binary Trees

Fig. 1 describes our binary tree data-type and for reasons of space we only consider three operations — a more detailed binary tree data-type is discussed in [6]. Our binary trees are finite with each node containing a well-defined value of type $\alpha$ (so we do not allow $\perp$) and for the rest of this paper we take $\alpha = \mathbb{Z}$. We now give definitions (taken from [2]) for each of the operations in our data-type. The flatten operation takes a tree and returns a list of values:

$$
\begin{aligned}
&\mathsf{flatten}\, Null &&= [\,] \\
&\mathsf{flatten}\,(Fk\, lt\, v\, rt) &&= (\mathsf{flatten}\, lt) \mathbin{+\!\!+} [v] \mathbin{+\!\!+} (\mathsf{flatten}\, rt)
\end{aligned}
$$

We can check whether a particular value is contained within a tree using the membership operation mem:

$$
\begin{aligned}
&\mathsf{mem}\ p\ Null &&= False \\
&\mathsf{mem}\ p\ (Fk\ lt\ v\ rt) &&= (v == p) \lor \mathsf{mem}\ p\ lt \\
& && \qquad\qquad\quad \lor \mathsf{mem}\ p\ rt
\end{aligned}
$$

To define $\mathsf{mkT}$ we have many choices of how to make a tree from a list. We adapt the definition in Bird [2, Page 183] to define a function which creates a binary tree of minimum height from a list of elements:

$$\begin{array}{l} \mathsf{mkT}\ [\,] \ = Null \\ \mathsf{mkT}\ xs = Fk\ (\mathsf{mkT}\ ys)\ z\ (\mathsf{mkT}\ zs) \\ \qquad \text{where } (ys, (z : zs)) \ = \ \mathsf{splitAt}\ (\mathsf{div}\ |xs|\ 2)\ xs \end{array}$$

This definition of $\mathsf{mkT}$ not only builds minimal height trees but also builds balanced trees. Note that we do not require that the trees in our data-type are balanced.

## III. Obfuscation as Data Refinement

Suppose that we have a data-type $D$ and we want to obfuscate it to obtain the data-type $O$. To provide a framework for obfuscating data-types (and establishing the correctness of the obfuscations) we view obfuscation as *data refinement* [5] and in particular we consider obfuscation as *functional refinement*. So for obfuscation we require an abstraction function $af :: O \rightarrow D$ and a data-type invariant $dti$ such that for elements $x :: D$ and $y :: O$

$$x \rightsquigarrow y \iff (x = af(y)) \wedge dti(y) \qquad (1)$$

The arrow $\rightsquigarrow$ is read as "…is data refined by …" (or in our case, "…is obfuscated by…") which expresses how the data-types are related. In our situation, it turns out that $af$ is a surjective function so we can find an obfuscation function $of :: D \rightarrow O$ that satisfies $of(x) = y \Rightarrow x \rightsquigarrow y$ and thus

$$af \cdot of = id \qquad (2)$$

Suppose that we have an operation $f :: D \rightarrow D$ defined in our data-type. Then to obfuscate $f$ we want an operation $f^O :: O \rightarrow O$ which preserves the correctness of $f$. In terms of data refinement, we say that $f^O$ is *correct* (with respect to $f$) if it is satisfies:

$$(\forall x :: D; y :: O) \ \bullet \ x \rightsquigarrow y \Rightarrow f(x) \rightsquigarrow f^O(y) \qquad (3)$$

If $f^O$ is a correct refinement (obfuscation) of $f$ then we write $f \rightsquigarrow f^O$. From (1) and (3), we have the following equation:

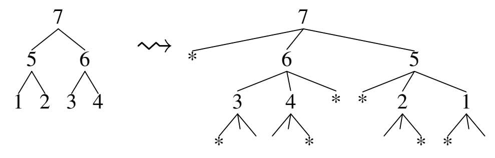
$$f \cdot af = af \cdot f^O \qquad (4)$$

where $\cdot$ means functional composition. Thus we can prove that a definition of $f^O$ is correct by using this equation.

## IV. Obfuscating Trees

We would like an obfuscation for our binary tree data-type that changes the structure of binary trees whilst allowing us to recover both the information and the structure if we so wish, with minimum overhead. Tree transformations such as rotations and reflections are suitable obfuscations, but for this paper, we consider converting a binary tree into a ternary tree. This conversion is chosen since it gives us the flexibility to add extra information as well as performing some tree transformations.

We convert a binary tree to a ternary tree by adding an extra subtree at every node of the binary tree. If a binary tree $t_2$ is represented by a ternary tree $t_3$, then by (1) we need an abstraction function $af$ and predicate $dti$ such that $t_2 = af(t_3) \wedge dti(t_3)$.



The symbol "*" stands for different arbitrary ternary trees.

Fig. 2. An example refinement

### A. Particular Representation

For our representation, we would like to convert the binary tree $Fk\ xt\ v\ yt$ into the ternary tree $Fk_3\ v\ lt\ ct\ rt$ so that the ternary trees $lt$, $ct$ and $rt$ depend on $v$, $xt$ and $yt$. Since a ternary tree can carry more information than a binary tree we can add "junk" to the ternary tree to aid the obfuscation. We then construct our abstraction function so that this junk is ignored — this gives us the opportunity to create random junk. The type definition for ternary trees is:

$$Tr_3\,\alpha ::= Null_3 \mid Fk_3\,\alpha\ (Tr_3\,\alpha)\,(Tr_3\ \alpha)\,(Tr_3\,\alpha)$$

We consider the following conversion:

$$Fk\ lt\ v\ rt \rightsquigarrow \begin{cases} Fk_3\ v\ lt_3\ rt_3\ junk_a & \text{if } v \text{ is even} \\ Fk_3\ v\ junk_b\ rt_3\ lt_3 & \text{otherwise} \end{cases}$$

and the tree $Null$ is converted to $Null_3$. An example is given in Fig. 2.

For data refinement, we need to state an abstraction function that converts a ternary tree into a binary one — for our refinement, we call this function $\mathsf{to2}$:

$$\begin{array}{l} \mathsf{to2}\ Null_3 = Null \\ \mathsf{to2}\ (Fk_3\ v\ lt\ ct\ rt) \\ \qquad \left|\begin{array}{ll} \mathsf{even}\ v & = Fk\ (\mathsf{to2}\ lt)\ v\ (\mathsf{to2}\ ct) \\ \text{otherwise} & = Fk\ (\mathsf{to2}\ rt)\ v\ (\mathsf{to2}\ ct) \end{array}\right. \end{array}$$

This means that for our representation

$$t_2 \rightsquigarrow t_3 \iff t_2 = \mathsf{to2}\ (t_3)$$

with the data-type invariant $True$. We do not have a left inverse for this function as any binary tree can be represented by more than one ternary tree ($\mathsf{to2}$ is surjective but not injective). Using (2), we can construct a right inverse, $\mathsf{to3}$ (*i.e.* it satisfies $\mathsf{to2} \cdot \mathsf{to3} = id$).

To convert a binary tree into a ternary tree, we have many choices of how to create the junk to be inserted into the tree at the appropriate places. Thus we can construct many functions that are right inverses to $\mathsf{to2}$. The complexities of the binary tree operations defined earlier depends on the number of nodes or on the height. Thus if we want our ternary tree operations to have similar complexities then we should make sure that the height of a ternary tree is similar to the binary tree which it represents and that we do not introduce too many extra elements. As an example, we define

```
to3 $Null = Null_3$
to3 $(Fk\ lt\ v\ rt)$
  | even $v$   $= Fk_3\ v$ (to3 $lt$) (to3 $rt$) (to3' $lt$)
  | otherwise $= Fk_3\ v$ (to3' $rt$) (to3 $rt$) (to3 $lt$)
where
  to3' $Null = Null_3$
  to3' $(Fk\ lt\ v\ rt)$
     $= Fk_3\ (3 \times v + 2)$ (to3 $lt$) (to3' $lt$) (to3' $rt$)
```

This function keeps the height of the tree the same. The mapping $\lambda v.(3v+2)$ was chosen so that the odd and even numbers follow the same distribution and so that the junk values are not too large (and therefore do not "stand out" from the "real" values).

## *B. Ternary Operations*

Now, we need to define operations for our ternary tree representation — flatten$_3$ is straightforward:

```
flatten$_3$ $Null_3$ $= [\,]$
flatten$_3$ $(Fk_3\ v\ lt\ ct\ rt)$
  | even $v$   $=$ (flatten$_3$ $lt$) $+\!\!+ [v] +\!\!+$(flatten$_3$ $ct$)
  | otherwise $=$ (flatten$_3$ $rt$) $+\!\!+ [v] +\!\!+$(flatten$_3$ $ct$)
```

For mem$_3$, we can define the following:

```
mem$_3$ $p\ Null_3 = False$
mem$_3$ $p\ (Fk_3\ v\ lt\ ct\ rt) =$
   $(v == p)\ \vee$ (mem$_3$ $p\ ct$) $\vee$
   (if even $v$ then (mem$_3$ $p\ lt$) else (mem$_3$ $p\ rt$))
```

We can see that these operations have extra conditionals and, as these tests are not expensive, the obfuscated operations have the same complexity as the unobfuscated ones. To prove these obfuscations are correct we need to show that flatten$_3$ = flatten·to2 and mem$_3$ = mem·to2. The details of these proofs are omitted.

## *C. Making Ternary Trees*

When we introduced binary trees we defined an operation mkT to convert a list into a binary tree. We can also define a corresponding operation mkT$_3$ that converts a list into a ternary tree and so satisfies the relationship: mkT = to2 · mkT$_3$. However as with the definition of an obfuscation function, we have many choices of mkT$_3$ that satisfy the above equation. For example, we can define mkT$_3$ so that mkT$_3$ = to3 · mkT using the definition of to3 from Section IV-A. We can use this equation to derive the following definition of mkT$_3$:

```
mkT$_3$ $[\,] = Null_3$
mkT$_3$ $xs$
  | even $z$   $= Fk_3\ z$ (mkT$_3$ $ys$)(mkT$_3$ $zs$)(mkT'$_3$ $ys$)
  | otherwise $= Fk_3\ z$ (mkT'$_3$ $zs$)(mkT$_3$ $zs$)(mkT$_3$ $ys$)
     where $(ys, (z : zs))$ = splitAt (div $|xs|$ 2) $xs$
```

and

```
mkT'$_3$ $[\,] = Null_3$
mkT'$_3$ $rs$ =
     $Fk_3\ (3 * q + 2)$ (mkT$_3$ $ps$) (mkT'$_3$ $ps$) (mkT'$_3$ $qs$)
      where $(ps, (q : qs))$ = splitAt (div $|rs|$ 2) $rs$
```

(the details of this derivation are omitted). Note that if too much "junk" is created then this will have an adverse effect on the efficiency of the obfuscated operations.

## V. Conclusions

In this paper we have discussed a new, promising approach to obfuscation by considering abstract data-types and refinement. In particular, a refinement suitable for creating tree obfuscations was developed. The examples given have been made simple in order to demonstrate the techniques involved. We can create more complicated abstraction functions by using a function which partitions our value type $\alpha$ into $n$ different partitions. In Section I we discussed the definition of an *assertion obfuscation*. We have seen that our obfuscated tree operations use a more complicated data structure and the definitions contain extra tests — thus proving properties about these obfuscations should be harder and so will be "more obfuscated" according to the definition of [6].

We have demonstrated how to obfuscate binary trees by representing them as ternary trees. This representation has allowed us to add bogus elements and so hide the "real" information. We must ensure that we do not adversely affect the efficiency of our operations by adding too much junk which would increase the size of the trees. However, these extra values give us further scope for obfuscation. For example, if we had an operation that inserted an element in a binary tree then, as an obfuscation, we could try to delete this element from the junk part of a ternary tree. We can actually specify a set of operations that represent the same function and we could choose one operation randomly. In fact, our approach gives plenty of opportunities for randomization. On each execution, we can have a different abstraction function (by choosing a suitable partition function), different junk values in a ternary tree representation and a different definition for each operation. Each of these choices can be made randomly and so we can create different program traces. This is a very important benefit of our approach as this randomness provides additional confusion for an adversary and helps to keep the unobfuscated operations secret.

For our tree obfuscation, we have a trade-off between how much junk we place in a ternary tree and the complexity of the obfuscated operations. Putting in too much junk makes building the ternary trees expensive and operating on the junk can have a severe impact on the efficiency. However, we should ensure that our operations act on the junk in some way (so that it is not obvious that the junk is not significant). Thus we should aim to keep the height and size of the ternary tree roughly the same as that of the binary tree it represents.

In [6] an imperative program that deletes an element from a binary tree is given and it is shown how this program can be obfuscated using the techniques described in this paper. One area for future work is to explore how we can use these techniques to automatically obfuscate imperative programs. Using imperative programs gives us different opportunities for obfuscation. For instance, trees can be implemented using pointers and we can exploit the difficulty in performing pointer analysis to construct obfuscations. For example, we could change binary trees into ternary trees and in the junk we could add in bogus pointers that point back up the tree. Obfuscations can often be used to help disguise watermarks. A software watermark is a "secret" (such as a piece of code or a data structure) embedded in a program which can be used to prove software ownership. A watermark called a PPCT (*Planted Plane Cubic Tree*) is described in Palsberg *et al.* [7] — this particular watermark is a special kind of binary tree. The obfuscation described in this paper could be used in conjunction with the PPCT watermark to make the watermark more obscure. In fact, the "junk" part of a ternary tree could itself be used as a watermark.

## References

[1] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. P. Vadhan, and K. 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.

[2] R. Bird. *Introduction to Functional Programming in Haskell*. Prentice Hall, 1998.

[3] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Science, University of Auckland, July 1997.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (Second Edition)*. The MIT Press, 2001.

[5] W.-P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1998.

[6] S. Drape. *Obfuscation of Abstract Data-Types*. DPhil thesis, Oxford University Computing Laboratory, 2004.

[7] J. Palsberg, S. Krishnaswamy, K. Minseok, D. Ma, Q. Shao, and Y. Zhang. Experience with software watermarking. In *Proceedings of the 16th Annual Computer Security Applications Conference, ACSAC '00*, pages 308–316. IEEE, 2000.

[8] S. Peyton Jones. The Haskell 98 language and libraries: the revised report. *Journal of Functional Programming*, 13(1), January 2003.