Folklore Confirmed: Reducible Flow Graphs are Exponentially Larger

Larry Carter*
Computer Sci. & Eng. Dept.
UC San Diego
9500 Gilman Drive
La Jolla, CA 92093-0114 USA
carter@cs.ucsd.edu

Jeanne Ferrante
Computer Sci. & Eng. Dept.
UC San Diego
9500 Gilman Drive
La Jolla, CA 92093-0114 USA
ferrante@cs.ucsd.edu

Clark Thomborson
Dept. of Computer Science
University of Auckland
Private Bag 92019, Auckland
New Zealand
cthombor@cs.auckland.ac.nz

ABSTRACT

Many program analysis techniques used by compilers are applicable only to programs whose control flow graphs are *reducible*. Node-splitting is a technique that can be used to convert any control flow graph to a reducible one. However, as has been observed for various node-splitting algorithms, there can be an exponential blowup in the size of the graph.

We prove that exponential blowup is unavoidable. In particular, we show that any reducible graph that is equivalent to the complete graph on n nodes (or to related bounded-degree control flow graphs) must have at least 2^{n-1} nodes. While this result is not a surprise, it may be relevant to the quest for finding methods of obfuscation for software protection.

Categories and Subject Descriptors

D.3 [**Programming Languages**]: Processors

General Terms

Algorithms, Security, Languages, Theory

Keywords

Compilers, computational complexity, programming languages, safety/security in digital systems

1. INTRODUCTION

Control flow graphs [1] are frequently used in optimizing compilers as the basis of program analysis and optimization. Intuitively, a control flow graph of a program is a directed graph that represents the program's potential flow of control. *Reducible* flow graphs, defined in Section 2, are a subclass of control flow graphs that include all those derived from "structured" programs, i.e., programs generated using a restricted set of constructs such as **if-then-else**, **while-do**, **continue** and **break**. Many static analyses [1, 13] either require the code to have a reducible flow graph, or may be too costly to analyze when applied to a large, irreducible graph. As noted in [13], reducible flow graphs allow a "divide-and-conquer" approach (such as interval analysis) which reduces the complexity of the analysis. Nevertheless, *irreducible* flow graphs do arise, often through the use of **go to** constructs, and must be handled by optimizing compilers. They also appear in the machine code generated by some optimizing compilers.

The most commonly used technique[1] for handling irreducible flow graphs is node-splitting [1, 13, 2, 9]. This technique eliminates "unstructured" program constructs by making duplicate copies of selected nodes in the control flow graph. Figure 1 gives an example.

As has been observed in the literature that for various node splitting algorithms, there is an exponential blowup in the size of the program or flow graph.[2] However, there are different ways to choose which nodes to split, and to choose an order of splitting these nodes [14, 16]. This leaves open two questions, which we have been unable to find answered in the published literature.

- Is it possible that some (perhaps as yet undiscovered) node splitting algorithm avoids exponential blowup?
- Is it possible that for any flow graph, there is an equivalent, reducible flow graph (even one that might not be the result of a node splitting algorithm) that avoids exponential blowup?

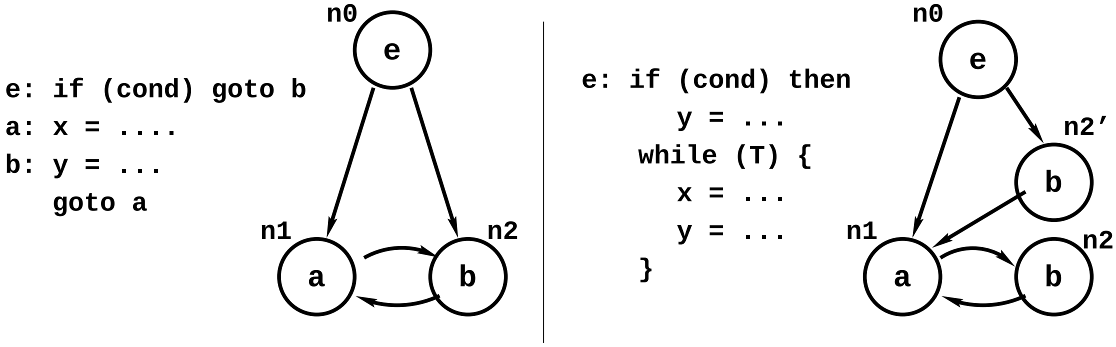
This paper answers both questions negatively. Although this will not come as a surprise to people familiar with the field, it fills in a gap that has been considered "folklore". Until now, there was little concern about the possible blowup, since it has often been observed that most programs written by humans are reducible [1]. However, we are motivated to

*Work performed while the first two authors were on sabbatical at University of Auckland.

[1] Another technique adds extra predicate variables to guard statements in a loop, e.g. [5, 6, 12, 4]. Although the resulting program is in some sense the same, it is not "equivalent" in the technical sense defined later in this paper. Thus, our results do not apply to such program restructuring techniques.

[2] See, for example, [1], p. 680 and [15], p. 197.

POPL'03 January 15–17, 2003, New Orleans, Louisiana, USA.
Copyright 2003 ACM 1-58113-628-5/03/0001 ...$5.00.



```
e: if (cond) goto b
a: x = ....
b: y = ...
   goto a
```

```
e: if (cond) then
    y = ...
  while (T) {
    x = ...
    y = ...
  }
```

Figure 1: On the left is an "unstructured" program and the corresponding irreducible control flow graph. The right figure is an equivalent "structured" program and its (reducible) flow graph.

take another look at this question by our interest in software obfuscation [10, 11].

A software obfuscator is an automatic technique that transforms programs into functionally equivalent variants that are more difficult to understand or analyze [10, 11]. Obfuscation can be shown to be theoretically "impossible" [7], given enough computational power and memory. This has not deterred attempts to construct obfuscators that have embedded instances of hard problems. For example, [8] shows that an instance of a PSPACE-hard problem may be embedded in the control flow of the obfuscated program. As another example, [17] introduce additional variable aliases by indirect addressing, making control flow difficult to determine. This latter work uses the fact that alias detection is an NP-hard problem.

We, too, have explored making the control flow of a program harder to analyze. A resourceful attacker can be expected to attempt to automatically decompile the target code, and to perform other automated analyses. As noted in [8], such automatic analysis can statically obtain global information about the structure of the program, enabling reverse engineering. Thus, our hope is to defend against this attack by obfuscating the control flow (or data flow) of the original program. This can be done by adding extra edges to make the control flow graph irreducible. This can be accomplished by inserting extra conditional branches into the program. If the conditional branches use values not known until run time, or perhaps use mathematical identities that are beyond the scope of an optimizing compiler to verify, then static analysis would be unable to eliminate them.

While we were able to develop techniques for constructing a complete (irreducible) flow graph from the original flow graph of a program, thus confounding static analysis, we were unable to find techniques that couldn't easily be unscrambled by symbolic execution or by analyzing a runtime trace.

Our main result is that for any code whose flow graph is equivalent to the complete graph with n nodes, any equivalent reducible flow graph must have at least 2^{n-1} nodes. A reducible flow graph of exponential size is unlikely to be of much use to an attacker. Starting with a program of modest size, the reducible equivalent may not even fit in a computer's memory, and its sheer mass will frustrate any human's attempt to "understand" what the code is doing.

2. REDUCIBLE FLOW GRAPHS

The term *reducible* flow graph,[3] originally defined in [3], means that the graph can be reduced to a single node by a sequence of applications of the two transformations, $\mathbf{T}_1$ and $\mathbf{T}_2$, shown in Figure 2. In the context of data flow analysis, flow graphs are annotated with information on the nodes (such as "There is an assignment to `X` and a use of `Y` in this block of code.") Many data flow algorithms work by manipulating this information appropriately as the graph is reduced via $\mathbf{T}_1$ and $\mathbf{T}_2$, producing summary information about the program (such as "The assignment to `X` in Block 3 is used in Blocks 4 and 8," and "The use of `Y` in block 3 may be uninitialized".) The details of these algorithms need not concern us; however, they motivate the following definitions:

Definition 1. *A* **flow graph** *is a directed graph with a distinguished* **initial** *node n_0, together with a function L that assigns to each node of the graph a label chosen from an* **alphabet** *U. The labels need not be unique.*

Definition 2. *A finite string of labels s is said to be* **produced** *by a flow graph G if $s = L(n_0)L(n_1)L(n_2)\dots L(n_k)$ where $n_0, n_1, n_2..., n_k$ is a path in G beginning with the initial node. $L(G)$ denotes the* **language** *of G, i.e. the set of strings produced by G.*[4]

Definition 3. *Two flow graphs G_1 and G_2 are* **equivalent** *if $L(G_1) = L(G_2)$.*

For example, the two flow graphs of Figure 1 are equivalent. In general, node splitting techniques transform an arbitrary flow graph into an equivalent reducible flow graph. We note that flow graphs and the set of strings they produce are very similar to finite automata and regular languages — the difference is that flow graphs have labeled nodes, whereas finite automata have labeled edges. In fact, the language of a flow graph G is regular,[5] but flow graphs produce more

[3] There are several equivalent definitions of reducible flow graph; see [13, 1, 14].

[4] We "overload" the symbol L to mean, in different contexts, the label of a node, the language of a flow graph, or other ways of associating a set of strings to various objects (such as leftist trees or annotated flow graphs) that will come later.

[5] If the label of each node is put on each outgoing edge, the resulting finite automaton produces $L(G)$.

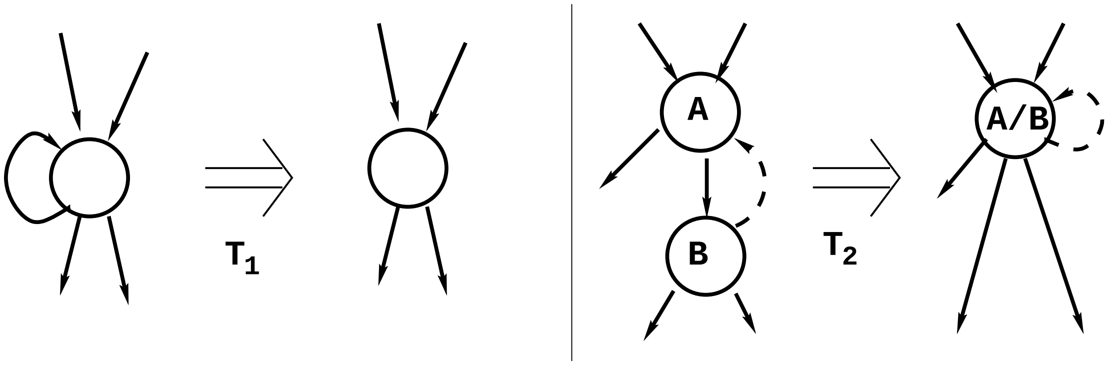


Figure 2: Two transformations used to simplify flow graphs. T_1 eliminates self-loops. T_2 merges a single-entry node into its parent. If there is an arc from the child back to the parent, it becomes a self-loop. A graph G is *reducible* if repeated applications of T_1 and T_2 reduce G to a single node.

restricted languages — for instance, every string in $L(G)$ must begin with the label of the initial node.

Before continuing with our study of flow graphs, we introduce leftist trees, which provide another way of producing a set of strings. We then prove a lower bound on the size of any leftist tree that produces the set of all strings over a given alphabet that begin with a given symbol. This result will allow us to prove a similar result for flow graphs.

3. LEFTIST TREES

A *leftist tree* T is simply a rooted binary tree (i.e., each node has either zero or two children), with labels on the leaves. $|T|$ denotes the number of nodes in T. We let T_l and T_r denote the left and right subtrees[6] of T. Note that T_l and T_r are either both empty or both non-empty. Define $n_l(T)$, $n_r(T)$ as follows: if T is a single node, both these denote that node. Otherwise, they are "first" (leftmost) leaf of T_l and T_r, respectively. Finally, let $s_l(T)$ and $s_r(T)$ be the labels of $n_l(T)$ and $n_r(T)$, respectively.

Whenever the tree T is understood from context, we may drop the "(T)" from our notation. Thus, for instance, T_l, n_l, and s_l denote the left subtree of T (or T itself if $|T| = 1$), the leftmost leaf of that subtree, and the symbol labeling that leaf.

We now give a method of associating a set of strings with a leftist tree. If $P = p_1p_2 \dots p_k$ is a non-empty sequence of leaves of tree T, then we say P is a *leftist leaf sequence* (LLS) of T if for all $i < k$, there exists a subtree T_i of T such that $p_i \in T_i$ and p_{i+1} is either $n_l(T_i)$ or $n_r(T_i)$. Intuitively, starting from p_i, you can move up the tree as far as you want, go down to either child, but then you must keep going left until you get to the leaf p_{i+1}. As an example, see Figure 3. Given a LLS $P = p_1p_2 \dots p_k$ of T, we say a string of symbols $s = s_1s_2 \dots s_k$ is *produced* by P if for $i = 1, \dots, k$, s_i is the label of p_i. Finally, we define the *language* of T, denoted, $L(T)$, to be the set of strings produced by LLS's in T that start with $n_l(T)$, the leftmost leaf of T.

The following three observations follow easily from the definition of leftist leaf sequence.

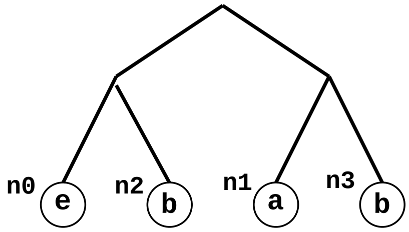


Figure 3: For this tree, the sequence $n_0n_2n_2n_1n_0n_1$ is a legal leftist leaf sequence, but n_0n_3 is not.

Lemma 1. *If P is a sequence of leaves of a subtree T' of T, then P is a LLS of T' if and only if it is a LLS of T.*

Lemma 2. *If $p_1 \dots p_j$ and $p_{j+1} \dots p_k$ are LLS's of T, and p_{j+1} is either n_l or n_r, then $p_1 \dots p_jp_{j+1} \dots p_k$ is a LLS of T.*

Lemma 3. *If $p_1 \dots p_jp_{j+1} \dots p_k$ is a LLS of T and p_{j+1} is in some subtree T' that doesn't include p_j, then p_{j+1} must be $n_l(T')$.*

We now can prove an alternate characterization of the set of strings that are produced by a tree T.

Theorem 1. *$L(T)$ can be expressed recursively by $L(T) = s_l(T)^+$ if $|T| = 1$ and $L(T) = L(T_l)(L(T_l) \cup L(T_r))^*$ otherwise.*[7]

Proof. By induction on $|T|$.

(Base case) If $|T| = 1$, i.e., if T is a single node n_l, then by letting $T_i = T$, we see that any sequence of one or more n_l's is a LLS starting with n_l. Together these LLS's produce the language $s_l(T)^+$.

(Inductive step) Assume that the theorem holds for T_l and T_r. Given any $s \in L(T_l)(L(T_l) \cup L(T_r))^*$, we can find a LLS in T_l or T_r for each of the substrings that are implied by this formula. By Lemma 1, these LLS's are also LLS's

[6] By a *subtree* we mean a node together with *all* of the nodes that it is an ancestor of.

[7] We use standard notation: if x and y are strings, then xy is their concatenation; if A and B are sets of strings, then AB represents the set $\{xy|x \in A, y \in B\}$, A^* is the set of strings that are the concatenation of zero or more strings in A, and $A^+ = AA^*$.

of T. Since each such LLS begins with either $n_l(T_l) = n_l$ or $n_l(T_r) = n_r$, their concatenation is an LLS by Lemma 2. It is also clear that this sequence begins with n_l. Thus, s is in $L(T)$.

Conversely, suppose $s \in L(T)$. Then s is generated by a LLS P of T that begins with n_l. We can decompose P into subsequences according to when P switches from T_l to T_r or back again. By Lemma 1, these subsequences are SSL's of T_l or T_r. By Lemma 3, each such subsequence begins with n_l or n_r. By the inductive hypothesis, they produce substrings in $L(T_l)$ or $L(T_r)$. Since T begins with n_l, the first substring is in $L(T_l)$. Thus s, which is the concatenation of the substrings, is in $L(T_l)(L(T_l) \cup L(T_r))^*$. □

A simple consequence of Theorem 1 is that for any leftist tree T, $L(T)^+ = L(T)$.

We need a few more definitions:

Definition 4. *We say that "a string t is a suffix of $L(T)$" if $\exists s \in U^*$ such that $st \in L(T)$.*

Definition 5. *Given an alphabet $\Sigma \subseteq U$, we say that "T is Σ-complete" if $\forall t \in \Sigma^*$, t is a suffix of $L(T)$.*

In our examples, the structure of a leftist tree T will be presented using parentheses recursively. We write $T = s$ if T is a single node labeled by s, and $T = (T_lT_r)$ if $|T| > 1$. For example, $T = (a(bc))$ is a tree whose left subtree is a and whose right subtree is (bc).

Example 1. Let $T = (ab)$. Then $L(T) = a\{a, b\}^*$, thus T is Σ-complete for $\Sigma = \{a, b\}$.

Example 2. Let $T = (a(bc))$. Then $L(T) = a(a^* \cup b\{b, c\}^*)^*$. The string ac is not a suffix of $L(T)$, so T is not complete for $\Sigma = \{a, b, c\}$. However T is both $\{a, b\}$-complete and $\{b, c\}$-complete.

Example 3. $T = ((ab)(cb))$ is $\{a, b, c\}$-complete.

Another example of a Σ-complete tree is shown in Figure 4.

Lemma 4. *A string t is a suffix of $L(T)$ if and only if t is produced by a LLS of T (with no constraint on the first node of the LLS).*

The only non-trivial part of the proof is to show that you can construct a LLS from $n_l(T)$ to the first symbol in the string. This can be done by using the leftmost leaves of the trees rooted at the nodes that form a path from the root of T to the leaf corresponding the first symbol of t.

The next two lemmas provide insight into the structure of Σ-complete trees.

Lemma 5. *Given an alphabet Σ, if T is a Σ-complete tree of minimal size, then its right subtree T_r is $(\Sigma - s_l)$-complete.*

Proof. By contradiction. Assume T_r is not complete over the reduced alphabet $(\Sigma - s_l)$. Then there exists a string $u \in (\Sigma - s_l)^+$ such that u is not a suffix of $L(T_r)$. Because T is of minimal size, T_l is not Σ-complete, so there exists a shortest string t in Σ^+ that is not a suffix of T_l. Let j and k be the lengths of t and u. Because T is Σ-complete, we know that tu is a suffix of $L(T)$. Denote tu by the string $v = v_1v_2 \dots v_{j+k}$. By Lemma 4, there must be a LLS $p_1p_2 \dots p_jp_{j+1} \dots p_{j+k}$ of T that produces v.

We will derive a contradiction by considering where p_j, the leaf corresponding to the last symbol of t, is. First, suppose that $p_j \in T_r$. For all $i > j$, $v_i \neq s_l$, since v_i is a symbol in u and u contains no s_l's. Thus, applying Lemma 3 inductively tells us that for all $i > j$, p_i must be a leaf of T_r. But this would mean that the LLS $p_{j+1} \dots p_{j+k}$ produces u. This contradicts that we chose u to *not* be a suffix of $L(T_r)$.

On the other hand, suppose that $p_j \in T_l$. By construction, $p_1 \dots p_j$ is *not* a LLS of T_l (since t is not a suffix of T_l), so there must be some nodes of $p_1 \dots p_{j-1}$ in T_r. Choose m to be the largest integer $m < j$ such that $p_m \in T_r$. This ensures that $p_{m+1}p_{m+2} \dots p_j$ is a LLS of T_l. Furthermore, since $p_m \in T_r$ and $p_{m+1} \in T_l$, we must have $v_{m+1} = s_l$ by Lemma 3. Now we use the fact that t is a *shortest* non-suffix of T_l to conclude that there must be some LLS $p'_1p'_2 \dots p'_m$ in T_l that produces $t_1t_2 \dots t_m$. Now consider the sequence $p'_1 \dots p'_mp_{m+1} \dots p_j$. By construction, it is entirely in T_l and it produces t. It also is a LLS by Lemma 2. Thus, we conclude that t is produced by a LLS in T_l, which contradicts the fact that it is *not* a suffix of T_l. □

Lemma 6. *Given an alphabet Σ, if T is a Σ-complete tree of minimal size, then its left subtree T_l is $(\Sigma - s_r)$-complete.*

Proof. The proof is the same as for Lemma 5, with the roles of T_l and T_r reversed. The abbreviated proof follows.

Suppose there were a string $u \in (\Sigma - s_r)^+$ of length j that is not a suffix of $L(T_l)$. T_r is not Σ-complete, so there is a shortest string $t \in \Sigma^+$ that is not a suffix of T_l. Let $p_1p_2 \dots p_jp_{j+1} \dots p_{j+k}$ be a LLS of T that produces tu. p_j must be in T_r; otherwise, the remainder of the leaf sequence would be stuck in T_l and produce u. We find $m < j$ such that $p_{m+1} \dots p_j$ is a LLS of T_r. The initial substring $p_1 \dots p_m$, being shorter than t, has a LLS in T_r. Extending this LLS with $p_{m+1} \dots p_j$ gives a LLS in T_r that produces t — a contradiction. □

Lemmas 5 and 6 enable us to prove a lower bound on the size of Σ-complete trees.

Theorem 2. *Let S_n be the minimal number of leaves in a Σ-complete tree with $|\Sigma| = n$. Then $S_n \geq 2^{n-1}$.*

Proof. By induction on n. The induction basis $n = 1$ is established by noting that $T = a$ is the minimal-sized tree to generate the 1-symbol complete language $L(T) = a^+$. The inductive step $S_n \geq 2S_{n-1}$ is established by considering the subtrees T_l and T_r for a Σ-complete tree T of minimal size with $|\Sigma| = n$. By Lemma 6, the subtree T_l is $(\Sigma_T - s_r)$-complete, so by the inductive hypothesis it has at least 2^{n-2} leaves. By Lemma 5, the subtree T_r is $(\Sigma - s_l)$ complete, so by induction it has at least 2^{n-2} leaves. This gives us the desired inequality. □

Theorem 3. *$S_n = 2^{n-1}$.*

Proof. A recursive construction will match the bound of Theorem 2. Our basis, for alphabet $\Sigma = \{x_1\}$, is the one-node tree $T = x_1$. We build a tree T with 2^{n-1} leaves for alphabet $\Sigma = \{x_1, x_2, ..., x_n\}$ from a left subtree of size S_{n-1} with $L(T_l) = x_1(\Sigma - x_n)^*$ and a right subtree also of size S_{n-1} with $L(T_r) = x_n(\Sigma - x_1)^*$. The resulting language $L(T)$ is, by Theorem 1, $L(T_l)(L(T_l) \cup L(T_r))^*$, which is $x_1(\Sigma - x_n)^*(x_1(\Sigma - x_n)^* \cup x_n(\Sigma - x_1)^*)^* = x_1(\Sigma)^*$. Thus, T is Σ-complete and has 2^{n-1} leaves. □

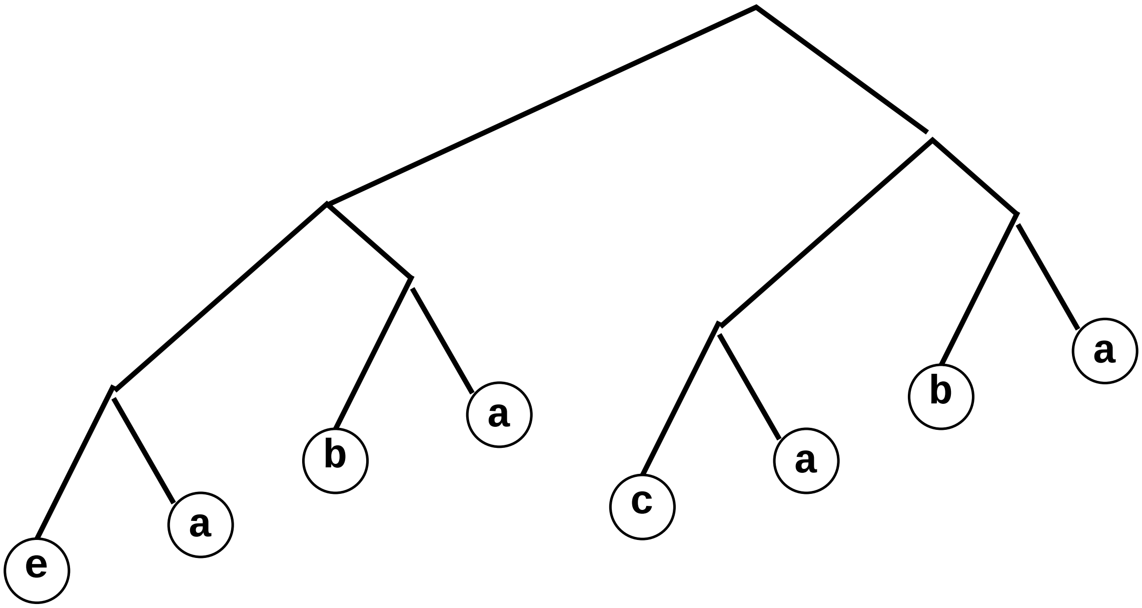


Figure 4: This tree is Σ-complete for $\Sigma = \{e, a, b, c\}$, and illustrates a recursive construction for such trees that have 2^{n-1} leaves when Σ is an alphabet of size n.

4. EXPONENTIAL BLOWUP

Given a finite alphabet Σ of size n and $e \in \Sigma$, let $K_{\Sigma,e}$ be the flow graph comprising the complete directed graph on n nodes, where each node is labeled with a distinct element of Σ and e is the label of the initial node. Let $L(K_{\Sigma,e})$ be the language produced by $K_{\Sigma,e}$. It's not hard to see that $L(K_{\Sigma,e})$ is the regular language $e\Sigma^*$, that is, the set of all strings over Σ that begin with "e". In this section, we will prove the main result of this paper:

Theorem 4. *If R is a reducible flow graph that is equivalent to $K_{\Sigma,e}$ (i.e. that is, if $L(R) = e\Sigma^*$), then R must have at least 2^{n-1} nodes, where n is the cardinality of Σ.*

Proof. We first show how to associate a leftist tree T with any reducible flow graph R. Each leaf of T will correspond to a distinct node of R, and $L(T)$ will be a superset of $L(R)$. We do this by annotating each node n_i of R with a leftist tree T_{n_i}. As we reduce R using the transformations $\mathbf{T}_1$ and $\mathbf{T}_2$, we will build up the trees that annotate the nodes. By the time R has been reduced to a single node, its associated tree will be the desired leftist tree. To each intermediate annotated flow graph there corresponds a language in the natural way — it is the set of strings of the form $s_1 s_2 ... s_k$, where $s_i \in L(T_{n_i})$ and $n_1, n_2, ...$ is a path through the flow graph.

Over the course of the construction, the following invariants will hold:

1. The language associated with the current annotated flow graph is a superset of $L(R)$.

2. There is a one-to-one correspondence between the nodes of R and the leaves in a forest formed of all the leftist trees in the annotations.

The first step of the construction is to annotate each node n_i of R with the leftist tree T_{n_i} consisting of a single (leaf) node labeled by $L(n_i)$, the label of n_i. It is easy to verify that properties (1) and (2) hold initially.

Whenever a $\mathbf{T}_1$ (self-loop elimination) transformation is applied to a node n_i of an annotated graph, we don't need to adjust the annotations. Property (1) will still hold since the only paths through the original flow graph that are no longer paths of the transformed one include repetitions of node n_i, but the fact that $L(T_n)^+ = L(T_n)$ ensures that arbitrary repetitions are represented even after eliminating the self-loop.

A $\mathbf{T}_2$ transformation can be applied to nodes n_1 and n_2 only when the edge (n_1, n_2) is the only edge into n_2. When a $\mathbf{T}_2$ transformation is applied, we fuse the two nodes into a single node and update its annotation as shown in Figure 5.

In any path through the pre-$\mathbf{T}_2$ flow graph, every occurrence of n_2 must be immediately preceded by an n_1. Thus, for each substring $s_{i-1}s_i$ where $s_{i-1} \in L(T_{n_1})$ and $s_i \in L(T_{n_2})$ that contributes to strings in the pre-$\mathbf{T}_2$ language, that same substring will be included in $L(T_{n_1/n_2}$ in the post-$\mathbf{T}_2$ flow graph. Thus, property (1) holds.[8] Property (2) also remains vaild throughout the construction because the $\mathbf{T}_1$ and $\mathbf{T}_1$ transformations do not affect the total number of leaves in the forest of annotations. For example, in Figure 6, the annotation forest always has four leaves (e, a, b, and c).

Once we have reduced R to a single node via $\mathbf{T}_1$ and $\mathbf{T}_2$, the annotation of that node is T, the leftist tree we associate with R. As noted, $L(T)$ may contain strings not in $L(R)$. However, if R is equivalent to $K_{\Sigma,e}$, then its language already contains *all* strings starting with e. Thus, $L(T) = L(R) = e\Sigma^*$. By Theorem 2, T has at least 2^{n-1} leaves. Since there is a one-to-one correspondence between the leaves of T and the nodes of R, the proof is completed. □

[8] New strings may be have been added to the language; for instance, in the example there was no edge from n_1 to n_4 or from n_2 to n_3 before $\mathbf{T}_2$ was applied.

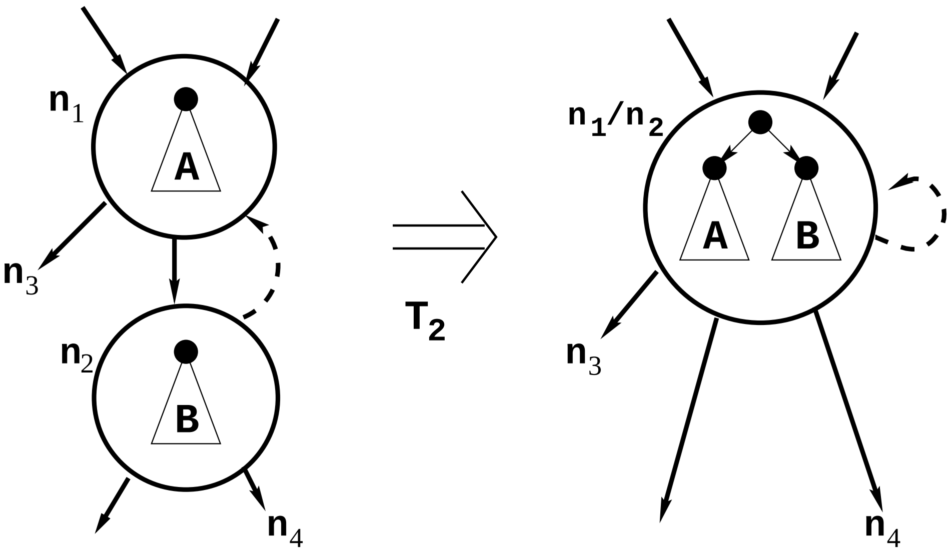


Figure 5: After a T_2 transformation, the annotation of the new fused node is a tree that has the old annotations as its left and right subtrees.

In Figure 6, we show the steps of transforming the example graph, resulting in its leftist tree.

Theorem 4 settles the two open questions posed in Section 1 negatively. In the next section, we will extend our results to show that even flow graphs which have outdegree 2 suffer from exponential blowup when converted to reducible flow graphs.

5. LIMITING OUTDEGREE

Suppose n_i is a node in a flow graph that has outdegree $k > 1$. We can replace n_i with a tree of $k-1$ nodes, where each node in the new tree has outdegree exactly 2. We begin by creating a new tree of $k-1$ nodes in which the outdegree of each node is at most 2, and each new node is given the same label as the original node, $L(n_i)$. We direct the edges of this new tree from its root towards its leaves. Now attach all of the edges that went into n_i to the root of this new tree, and attach the k outedges that originally came from n_i to nodes in the new tree in a way that makes each node has outdegree exactly 2. This construction is analogous to implementing a multi-way "case" statement as a sequence of binary branches. It doesn't matter for our purposes whether the tree is balanced or not.

Given $K_{\Sigma,e}$, the complete flow graph on n nodes used earlier, if we apply this procedure to every node, we will obtain a flow graph $J_{\Sigma,e}$ that has $n(n-1)$ nodes, each of outdegree 2. We will show that any reducible flow graph that is equivalent to $J_{\Sigma,e}$ has at least 2^{n-1} nodes. Figure 7 shows this construction for $\Sigma = \{e, a, b, c\}$.

The language of $J_{\Sigma,e}$ is related to Σ^* in that for each string $s = s_1 s_2 \dots s_k \in \Sigma^*$, there is a string $s' = s_1^{r_1} s_2^{r_2} \dots s_k r_k \in L(J_{\Sigma,e})$. In other words, the symbols of s can be duplicated a certain number of times[9] to produce s'.

We will show that any reducible flow graph that produces $L(J_{\Sigma,e})$ (or a superset thereof) must have at least 2^{n-1} nodes, where n is the size of Σ. We will do so by proving results similar to Lemmas 5 and 6 about the structure of any reducible graph producing such a language. First, we need some more definitions.

Given two strings u and u', we say that u is a *contraction* of u' if u is the result of eliminating some adjacent duplicates in u'. For instance, `abbac` is a contraction of `aaabbacc`. However, `abba` is not, since a contraction must contain at least one symbol from each string of duplicates.

Given a leftist tree T over an alphabet Σ, and a set of strings $S \subseteq \Sigma^*$, we say that T *covers* S if for every $s \in S$, there exists a suffix s' of T such that s is a contraction of s'.

LEMMA 7. *Given an alphabet Σ, if T is a minimum-size leftist tree that covers Σ^*, then its right subtree T_r covers $(\Sigma - s_l)^*$.*

PROOF. By contradiction. Assume T_r does not cover $(\Sigma - s_l)^*$. Then there exists a string $u \in (\Sigma - s_l)^+$ such that u is not a contraction of any suffix u' of $L(T_r)$. Because T is of minimal size, T_l does not cover Σ^*. Consequently, there exists a shortest string t in Σ^+ that is not a contraction of any suffix of T_l. Because T covers Σ^*, we know that tu is a contraction of some suffix v of $L(T)$. We can therefore write $v = v_1 v_2 \dots v_j v_{j+1} \dots v_{j+k}$, where t is a contraction of $v_1 v_2 \dots v_j$ and u is a contraction of $v_{j+1} \dots v_{j+k}$. By Lemma 4, there must be a LLS $p_1 p_2 \dots p_j p_{j+1} \dots p_{j+k}$ of T that produces v.

We need to handle a messy detail. It could happen that for some i, $v_i = s_l$ and $p_{i+1} = n_l$.[10] Whenever this happens, we replace p_i by n_l. Notice that the modified string $\dots p_{i-1} n_l p_{i+1} \dots$ is still a LLS, since n_l is allowed to follow *any* node (and in particular p_{i-1}), and n_l to p_{i+1} is a valid

[9] The number of repetitions depends on the method used for expanding nodes of $K_{\Sigma,e}$ into binary trees.

[10] This condition means that v has two adjacent occurrences of the *label* of T's leftmost node, and the LLS that produces v uses that leftmost node to produce the second occurrence.

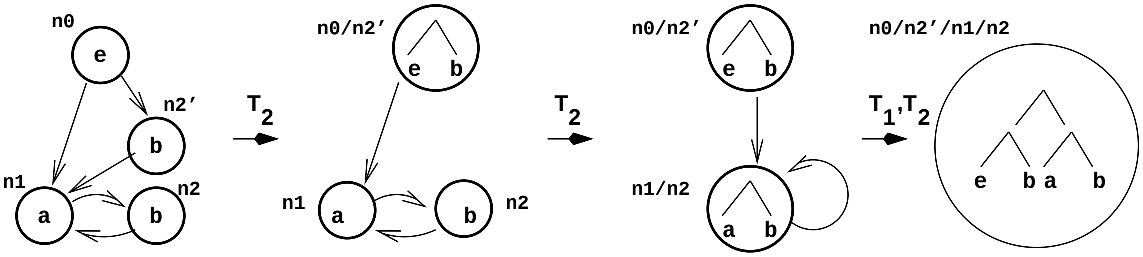


Figure 6: This example graph is successively transformed by T_2 and T_1 transformations, resulting in its leftist tree.

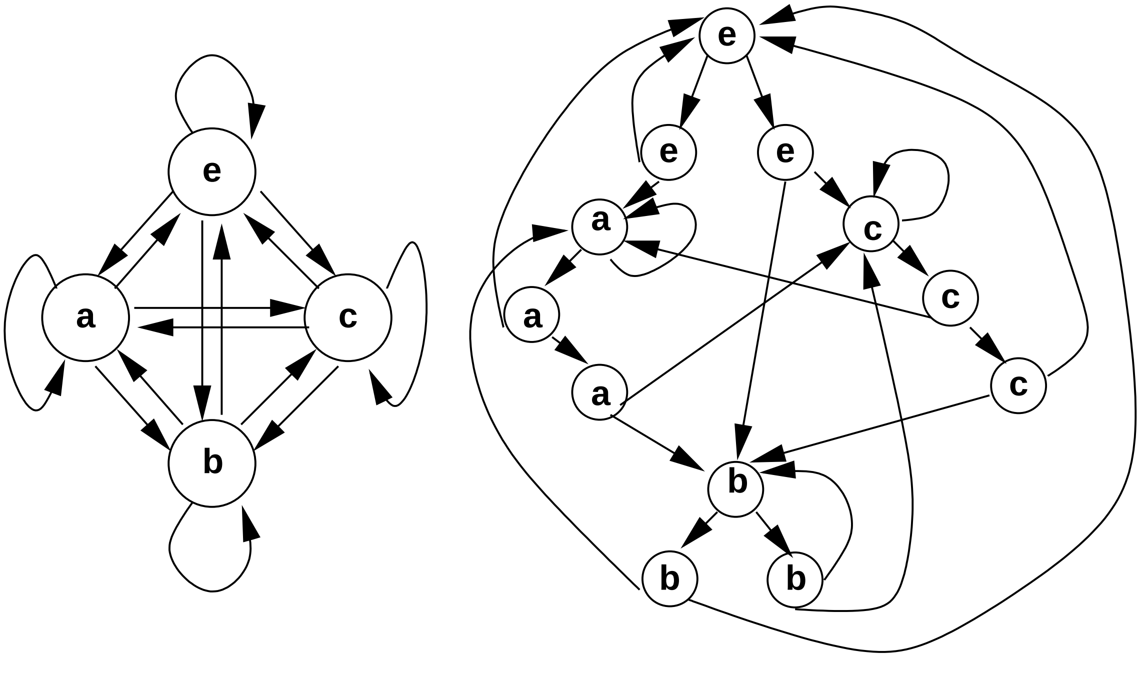


Figure 7: On the left is a flow graph with outdegree 3. The right figure is an equivalent flow graph of outdegree 2 that would result by applying our procedure.

transition since they are the same node. After making as many such substitutions as possible, we can assume without loss of generality that whenever the sequence $p_1p_2\dots$ moves from T_r to T_l, that is when $p_i \in T_r$ and $p_{i+1} = n_l$, then $v_i \neq s_l$. We will need this property in a moment.

We can now mimic the proof of Lemma 5 to derive a contradiction. Consider where p_j, a leaf of T corresponding to the last symbol of t, is in T. First, suppose that $p_j \in T_r$. Note that for all $i > j$, $v_i \neq s_l$, since v_i is a symbol in u and u contains no s_l's. Thus, applying Lemma 3 inductively tells us that for all $i > j$, p_i must be a leaf of T_r. But this would mean that $v_{j+1}\dots v_{j+k}$ is a suffix of $L(T_r)$. Summarizing, we know u is a contraction of the string $u' = v_{j+1}\dots v_{j+k}$, and that u' is a suffix of $L(T_r)$. This contradicts how we chose u.

On the other hand, suppose that $p_j \in T_l$. By construction, $p_1\dots p_j$ is *not* a LLS of T_l (since t is not a contraction of a suffix of T_l), so there must be some nodes of $p_1\dots p_{j-1}$ in T_r. Choose m to be the largest integer $m < j$ such that $p_m \in T_r$. This ensures that $p_{m+1}p_{m+2}\dots p_j$ is a LLS of T_l. Furthermore, since $p_m \in T_r$ and $p_{m+1} \in T_l$, we must have $p_{m+1} = n_l$ by Lemma 3. By the "messy detail" given above, we can assume that $v_m \neq s_l$. This ensures that the portion t' of t that is a contraction of $v_1\dots v_m$ is indeed *shorter* than t. Now we use the fact that t is a *shortest* non-suffix of T_l to conclude that there must be some LLS $p'_1p'_2\dots p'_{m'}$ in T_l that produces a string that covers t'. Now consider the sequence $p'_1\dots p'_{m'}p_{m+1}\dots p_j$. By construction, it is entirely in T_l and it covers t. It also is a LLS by Lemma 2 and the fact that $p_{m+1} = n_l$. Thus, we conclude that t is covered by a string produced by T_l, which contradicts how t was originally chosen. QED □

We can also prove this analog of Lemma 6

LEMMA 8. *Given an alphabet Σ, if T is a minimum-size leftist tree that covers Σ^*, then its left subtree T_l covers $(\Sigma - s_r)^*$.*

PROOF. The proof is the same as Lemma 7, with the roles of T_l and T_r reversed. We note that the corresponding "messy detail", where $L(n_i) = s_r$ and $p_{i+1} = n_r$ is handled by replacing p_i with n_r. The resulting leaf sequence is still an LLS since p_r can legally follow any note in T. □

We are now ready for our final result.

THEOREM 5. *If Σ is an alphabet of size n and R is a reducible flow graph that is equivalent to $J_{\Sigma,e}$ (or more generally, if $L(R)$ is a superset of $L(J_{\Sigma,e})$), then R has at least 2^{n-1} nodes.*

PROOF. Let T be the leftist tree associated with $J_{\Sigma,e}$ and T' the tree associated with R, using the construction of Theorem 4. We know that $L(T') \supseteq L(R) \supseteq L(J_{\Sigma,e})$. Further, we know that for every string $s \in \Sigma^*$, s is a contraction of a suffix of a string in $L(J_{\Sigma,e})$. Thus, T' covers Σ^*.

Using an induction proof as in Theorem 2, Lemmas 7 and 8 allow us to conclude that T' has at least 2^{n-1} leaves. Since R has as many nodes as T' has leaves, this completes the theorem. □

6. CONCLUSION

We have formalized and proven a portion of the folklore of compiler analysis. Some flow graphs, in particular the ones equivalent to the complete flow graph, have no equivalent *reducible* flow graph that is not exponential in size. In particular, our results show that no node splitting technique can avoid this exponential blowup.

While there has been little concern about such exponential blowup, since most programs written by programmers are reducible [1], automatic compiler analyses would find handling such large programs difficult. Such analyses either require the code to have a reducible flow graph, or would be very costly to analyze when applied to an irreducible graph of very large size. Thus, a line of defense against reverse engineering of programs is to distribute code whose flow graph is irreducible. Such code is unlikely to be easily automatically analyzable, and thus would be difficult to reverse engineer.

We have already noted that the technique of adding guard predicates to obtain a reducible flow graph from an irreducible one is not covered by our results. However, this technique may also considerably complicate program analysis. We leave it as an open question as to whether such analysis can be shown to be provably hard.

7. ACKNOWLEDGMENTS

We would like to thank the CSE 238 class at UCSD (in Fall, 2002) for their helpful comments.

8. REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986.

[2] F. Allen and J. Cocke. Graph-theoretic constructs for program control flow analysis. Technical Report RC-3923, IBM Research, 1972.

[3] F. E. Allen. Control flow analysis. *ACM Sigplan Notices*, 5(7):1–19, 1970.

[4] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures.* Morgan Kaufmann, 2001.

[5] R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Conference Record of the ACM Symposium on the Principles of Programming Languages (POPL)*, January 1983.

[6] Z. Ammarguellat. A control-flow normalization algorithm and its complexity. *IEEE Transactions on Software Engineering*, 18(3), Mar. 1992.

[7] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs. In *Proceedings of CRYPTO*, 2001.

[8] S. Chow, Y. Gu, H. Johnson, and V. Zakharov. An approach to the obfuscation of control-flow of sequential computer programs. In *Proceedings of ISC*, 2001. Published in Springer-Verlag LNCS 2200.

[9] J. Cocke and R. Miller. Some analysis techniques for optimizing computer programs. In *Proceedings of the Second Intl. Conf. of Systems Science*, January 1969.

[10] C. Collberg, C. Thomborson, and D. Low. Breaking abstractions and unstructing data structures. In *IEEE International Conference on Computer Languages (ICCL)*, May 1998.

[11] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient and stealthy opaque

constructs. In *Conference Record of the ACM Symposium on the Principles of Programming Languages (POPL)*, January 1998.

[12] A. Erosa and L. Hendren. Taming control flow: A structured approach to eliminating goto statements. In *International Conference on Computer Languages*, May 1994.

[13] M. S. Hecht. *Flow Analysis of Computer Programs*. North Holland, 1977.

[14] J. Janssen and H. Corporaal. Controlled Node Splitting. In *Proceedings of the 6th International Conference on Compiler Construction*, pages 44 – 58, Linkoping, Sweden, Apr. 1996. volume 1060 of Springer Lecture Notes in Computer Science.

[15] S. S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufmann, 1997.

[16] S. Unger and F. Mueller. Handling Irreducible Loops: Optimized Node Splitting vs. DJ-Graphs. Technical Report TR 01-146, Institut f. Informatik, Humboldt-University, Jan. 2001.

[17] C. Wang, J. Hill, J. Knight, and J. Davidson. Software tamper resistance: Obstructing static analysis of programs. Technical Report CS-2000-12, University of Virginia, 12 2000.