SANDIA REPORT

SAND2004-5372
Unlimited Release
Printed January 2005

Evolutionary Complexity for Protection of Critical Assets

Michael E. Chandross and Corbett C. Battaile

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94AL85000.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865)576-8401
Facsimile: (865)576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800)553-6847
Facsimile: (703)605-6900
E-Mail: orders@ntis.fedworld.gov
Online order: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND 2004-5372
Unlimited Release
Printed January 2005

Evolutionary Complexity for Protection of Critical Assets

Michael E. Chandross and Corbett C. Battaile
Computational Materials and Molecular Sciences

Sandia National Laboratories
P. O. Box 5800
Albuquerque, NM 87185-1411 USA

Abstract

This report summarizes the work performed as part of a one-year LDRD project, "Evolutionary Complexity for Protection of Critical Assets." A brief introduction is given to the topics of genetic algorithms and genetic programming, followed by a discussion of relevant results obtained during the project's research, and finally the conclusions drawn from those results. The focus is on using genetic programming to evolve solutions for relatively simple algebraic equations as a prototype application for evolving complexity in computer codes. The results were obtained using the lil-gp genetic program, a C code for evolving solutions to user-defined problems and functions. These results suggest that genetic programs are not well-suited to evolving complexity for critical asset protection because they cannot efficiently evolve solutions to complex problems, and introduce unacceptable performance penalties into solutions for simple ones.

This Page
Involuntarily Left Blank

Contents

List of Figures and Tables

Introduction

The natural evolution of organisms has created remarkable systems that slowly change in response to sometimes harsh and unforgiving environments. This process shows that starting from the simplest one-celled bacteria, amazingly complex, well-adapted creatures that function on a high level can evolve in response to external stimuli. It is natural, therefore, to conclude that a similar paradigm might potentially be an extremely useful problem solving technique. Evolutionary methods for solving complex problems were introduced in 1975 by John Holland in his book, *Adaptation in Natural and Artifical Systems*. The field of Evolutionary Computing (EC) is generally divided into two major subfields, genetic algorithms (GA) and genetic programming (GP), which both use the concept of evolutionary methods although for different problems. We describe both GA and GP in detail, below.

Genetic Algorithms

With EC methods, in general, one starts with a population of trial solutions to a problem. As an example, we will demonstrate the use of a GA to find the minimum value of the function

$$f(x) = (x - 192)^2, \quad (1)$$

with an initial population consisting of a collection of random integers. Each organism in the population (*i.e.* each integer) can be assigned a fitness that describes how well it solves the problem. An example population with only four individuals is given in Table 1, below. The organisms in the population here are shown by their genome (*i.e.* their representation in binary notation) in order to simplify later discussions.

In this trivial example, the fitness for each organism *n* is the value *F(n)*. The population is ranked according to fitness (in this case, lower values being better), and then a certain number of the fittest individuals are replicated in the next generation. In this example, we keep organisms 2 and 3 for transmission to the next generation. This is referred to as reproduction, and is clearly asexual. The more fit individuals in the population also undergo sexual reproduction, which is referred to as crossover. Crossover begins with the selection of a subset of the more fit individuals, which are then randomly paired so that information can be exchanged between the genomes. In our example problem, the genome has been chosen to be the binary representation of the numbers, as shown in Table 1. A random point in the genome is chosen as the crossover point, and the bits to the right of this point are swapped between the parents to give two children, as shown in Fig. 1. After this operation, there are four members of the new population as shown in Table 2.

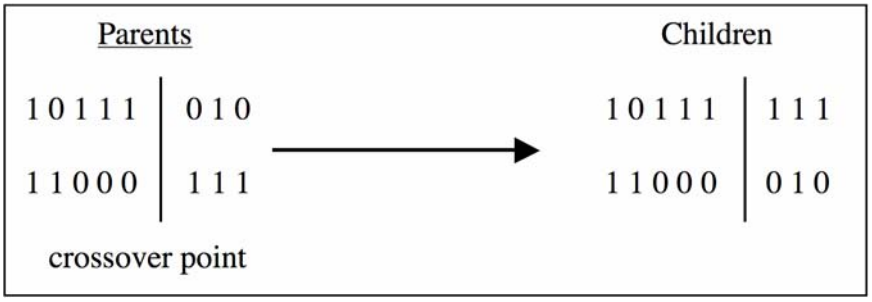


Figure 1. Example of crossover between two genomes.

Table 1. First generation of solutions to $f(x) = (x - 192)^2$.

Organism Number	Genome	Value	Fitness
1	01011100	92	10000
2	10111010	186	36
3	11000111	199	49
4	00011011	27	27225

Table 2. Second generation of solutions to $f(x) = (x - 192)^2$.

Organism Number	Genome	Value	Fitness
1	10111010	186	36
2	11000111	199	49
3	10111111	191	1
4	11000010	194	4

As is clear from Table 2, the overall fitness of the population has increased greatly. There are now two organisms that are very close to the correct answer of 192. The procedure of reproduction and crossover is continued until either a given number of generations has been reached, or an exact solution is found. There is one final mechanism available for modification of the genome that is distinct from the two forms of reproduction demonstrated above. This operation is the mutation operator, and for this example would consist of randomly flipping a bit in the genome (*i.e.* 0 becomes 1 or vice versa). As in biological evolution, the probability of a beneficial random mutation is small, and thus the rate of mutation in the algorithm must be kept correspondingly low.

Genetic Programming

The procedure for GP is essentially the same as that for GA in that a given population is evaluated for fitness, and the more fit individuals are chosen to propagate to the next generation through both reproduction and crossover. The essential difference is that GA seek potential solutions to a given problem (*e.g.* numbers, blackjack strategy tables, or electronic circuits), whereas GP evolves self-contained computer code whose fitness is determined by its output. There are a number of different methods that can generate, evaluate, and evolve a population of programs, but we will only describe two here. The first method is exemplified by the freely available Avida platform (http://dllab.caltech.edu/avida/). In this code, the genome of the organisms consists of programs in Avida's own stripped-down assembly language which runs on a virtual machine. Each organism contains code that allows it to replicate, and hence to reproduce. The programs compete for CPU time and resources that are allocated based on fitness. We determined that the Avida platform was not appropriate for this project. We instead used the common alternate paradigm developed by Koza [1], in which organisms are represented by snippets of Lisp code. To generate and execute codes we use the package lil-gp [2], which strictly adheres to Koza's methods. Before describing the method of GP with Lisp, however, it is first useful to explain the basics of Lisp itself.

Lisp is a simple language consisting of a small basic instruction set from which more complex instructions can be made. Operations are constructed in the form

$$(op\ A\ B), \qquad (2)$$

where *op* is an operator, also known as a terminal; and *A* and *B* are the arguments. For example, the expression (+ 3 5) would evaluate to 3+5, *i.e.* 8. Both *A* and *B* can be expressions of their own, so that the more complex expression (+ 3 (+ 3 2)) would also evaluate to 8. The nested operator (+ 3 2) is evaluated first, with the result passed up to the enclosing operator.

The structure of Lisp codes makes them ideal to represent as parse trees. The example above can be written as the tree shown in Fig. 2. More complex functions including variables can also be constructed in Lisp. For example, the function $f(x) = x^3+x^2+x$ can be represented by the Lisp expresion (+ (+ (* X (* X X)) (* X X)) X), or more conveniently as the parse tree shown in Fig. 3a. In Fig. 3b we show an alternate method of representing this parse tree that we will use below for longer, more complicated functions that would be excessively large if written as shown in Fig. 3a. It is precisely the method of representing Lisp programs shown in Fig. 3a that makes it ideal for GP. Each individual organism can be represented as a parse tree, and crossover can be achieved by selecting a random node in two organisms, and swapping the subtrees at that node. As an example, if we were to perform crossover between the organisms in Fig.s 2 and 3, with the nodes selected being the ones marked in red, the resulting organisms are those shown in Fig. 4.

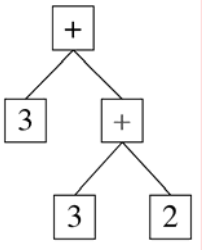


Figure 2. Example of a Lisp parse tree.

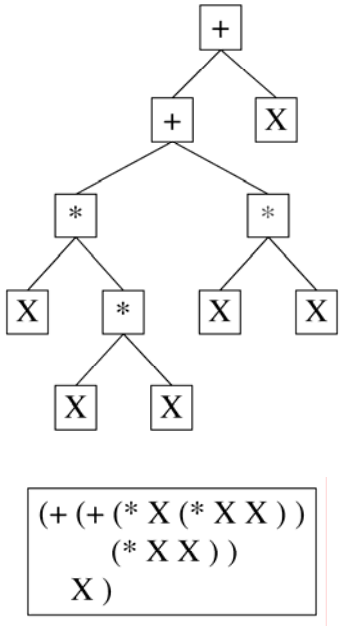


Figure 3. Example representations of Lisp parse trees for x^3+x^2+x.

In GP, as in GA, it has been found that when solutions are generated, they are often exceedingly complex and difficult to understand [3]. As an example, consider the function $f(x) = x^3$, given by the simple Lisp expression (* (* X X) X). A GP used to evolve this function, however, arrives at the equally correct expression (* (+ (+ (/ X X) (* X X)) (- (* X X) (/ X X)) (- (/ (* X X) (+ X X)) (+ (- X X) (- X X)))). Clearly the evolved expression is more complicated to understand than the original one. It is, of course, possible to develop complicated expressions for $f(x) = x^3$ by hand, but it is unlikely that human-developed expressions will be as perverse as those derived by a program that evolves solutions. Just as in nature, where organisms evolve into complex and mysterious systems, the results of GP can be obfuscated through indirect and redundant methods resulting from the lack of human intervention. Here we will discuss attempts to exploit this aspect of GP in order to develop intentionally obfuscated code with the goal of protecting of critical intellectual property from reverse-engineering attempts.

Obfuscation

The goal of code obfuscation is to transform working source into code that is functionally identical, yet much more complex syntactically. Such a transformation is desirable for preventing reverse-engineering of concepts or algorithms that are important intellectual property or crucial for national security. The difficulty with code obfuscation is that, while in some cases it can be easy to identify code that is intentionally obfuscated as compared to code that is not, there is no clear way to quantify the obfuscation because, unlike in cryptography, there has not yet been a theory developed that allows such a measure [4]. Obfuscation differs from cryptography, however, in that once a cryptographic cipher is broken the code is no longer protected. With obfuscated code, the deobfuscation of one section of code, hopefully a time-consuming process, is of little to no use in attempts to deobfuscate other sections. In this sense, obfuscation is a complementary technique to cryptography. In general, actively preventing reverse engineering is a difficult prospect. The International Obfuscated C Code contest is a prime example of the lengths some will go to in order to hamper reverse engineering. As an example, consider the following code, one of the winners in 1998:

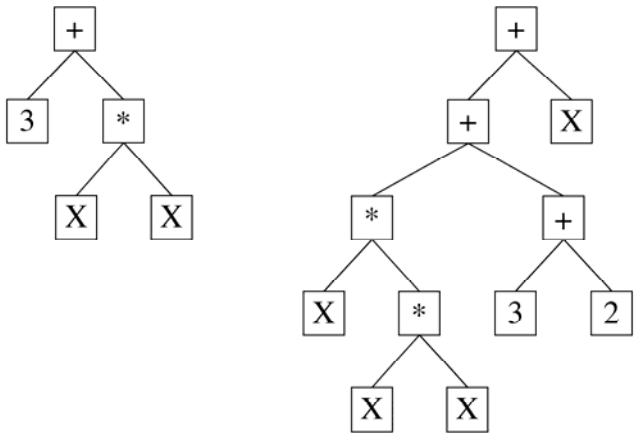


Figure 4. Example Lisp parse trees after crossover.

```
#include <stdio.h> main(t,_,a) char *a; { return!0<t?t<3?main(-79,-13,a+main(-87,1-
_,main(-86,0,a+1)+a)): 1,t<_?main(t+1,_,a):3,main(-94,-27+t,a)&&t==2?_<13?
main(2,_+1,"%s %d %d\n"):9:16:t<0?t<-72?main(_,t,"@n'+,#'/*{}w+/w#cdnr/+,
{}r/*de}+,/*{*+,/w{%+,/w#q#n+,/#{l+, /n{n+,/+#n+,/#\ ;#q#n+,/+k#;*+,/'r :'d*'3,}{w+K
w'K:'+}e#';dq#'l \ q#'+d'K#!/+k#;q#'r}eKK#} w'r}eKK{nl]'/#;#q#n')
{)#}w'){){nl]'/+#n';d}rw' i;# \ ){nl]!/n{n#'; r{#w'r nc{nl]'/#{l,+'K {rw' iK{;[{nl]'/
w#q#n'wk nw' \ iwk{KK{nl]!/w{%'l##w#' i; :{nl]'/*{q#'ld;r'} {nlwb!/*de}'c \ ;;{nl'-
{}rw]'/+,}##'*}#nc,', #nw]'/+kd'+e}+;#'rdq#w! nr'/ ') }+}{rl#'{n' ')# \ }'+}##(!!/") :t<-
50?_==*a?putchar(31[a]):main(-65,_,a+1): main((*a=='/')+t,_,a+1)
:0<t?main(2,2,"%s"):*a=='/'||main(0,main(-61,*a, "!ek;dc i@bK'(q)-
[w]*%n+r3#l,{}:\nuwloca-O;m .vpbks,fxntdCeghiry"),a+1); }
```

It is unlikely that anyone can decipher this code, which prints all of The Twelve Days of Christmas, without enormous effort. Similarly, however, writing this code was not a simple process. The emerging view in computer science is that automatic code obfuscation through the use of transformations similar to compiler optimizations is the most appropriate path for the prevention of reverse engineering. On the other hand, some experts feel that this type of obfuscation is impossible because a corresponding deobfuscator can always be devised [5]. This is partially the motivation for evolved complexity – human engineers and programmers, no matter how talented they may be, generally work within constrained mathematical models of idealized systems, and attack problems with particular, well-defined methods. This is why it is possible to create deobfuscators for human-developed obfuscation techniques. The enormous complexity of biological systems, and the correspondingly copious funding currently being allocated to researchers attempting to reverse engineer their functionality, demonstrates that evolution is the ideal technique for developing complex solutions that would never occur to human engineers. As Jostein Gaarder wrote, “If the human brain were simple enough for us to understand, we would still [sic.] be so stupid that we couldn’t understand it.” [6]

While it is indeed true that no theory of obfuscation has yet been developed, there have been a number of researchers who have studied various techniques used in code obfuscation and attempted to classify them. Such a classification is beyond the scope of this work, and the reader is referred to the excellent review by Campbell [7] for a more complete introduction to obfuscation and the attempts to quantify it.

The GP method will naturally lead to code that is large and difficult to understand even for extremely simple functions. For most practitioners this is an undesirable side effect referred to as “code-bloat,” and efforts have been made to try to understand its cause in order to prevention it. This is clearly antithetical to our purposes here, but understanding the cause of code-bloat can also potentially lead to methods for encouraging rather than discouraging it. The major effort in the GP community has been on the relationship of code-bloat and introns (*i.e.* sections of nonfunctioning code such as are found in DNA), although there is no consensus on which is the cause and which is the effect [8,9]. Simple examples of introns that occur in GP include multiplying or dividing large expressions by one [often in the form (/ *A A*)], adding or subtracting zero [often in the form (- *A A*) or a large, complex expression that is multiplied by this], and combinations of these. Introns are evolutionarily useful for the organism itself since they provide protection from crossover. The more introns that exist in a parse tree, the more likely it is that the subtree selected for crossover is useless to the overall function of that organism, and thus the more likely it is that the code will perform identically before and after crossover.

Introns are one of three types of obfuscation that occur naturally in code produced by GP. These types can be further classified according to the taxonomy of obfuscation due to Collberg *et al.* [4], but this detail is not necessary for our purposes here. The second trivial form of obfuscation arises from the overall allowed tree depth. Often the tree depth is constrained (generally to around 17 [1]) in order to prevent code bloat. Clearly the tree depth and introns are related in the production of code bloat. We have performed some experiments of induced obfuscation through tree depth manipulation which will be described below.

The third form of obfuscation from GP is algorithmic obfuscation. Algorithmic obfuscation is essentially using a complicated algorithm where a simpler one would do – in some sense it is the opposite of simplification of an equation. This form is arguably the most important for true code obfuscation, as introns can be easy to spot and ignore. This can greatly speed up understanding of a parse tree, particularly when large sub-trees (as occur with increased tree depth) can be ignored.

Results

Simple Functions

We begin by presenting results of GP runs to produce simple functions. The goal here is to study the method itself and to understand the types of obfuscation produced. To this end, we will show examples of the three types of obfuscation described above as produced by actual GP runs on a simple function. For this section we choose the trivial function $f(x) = 2x$.

We begin with an example of an intron. In Fig. 5 we show a successful run in lil-gp to generate $f(x) = 2x$. The code shown evaluates to X + [(X – X) + X], or 2X. The appearance of the intron (- X X) does little to obfuscate the code here, and it is clear that it can be ignored upon only cursory examination of the parse tree. It is not surprising that this example is trivial, however, since it was generated as one of the original random trees in the population, and just happened to be correct.

The code shown in Fig. 6 in the alternative format is an excellent example of a more complicated intron that cannot easily be distinguished from important code. This code appeared in generation 10 of the run, and contains 47 nodes with a tree depth of 7. The code is equivalent to the expression

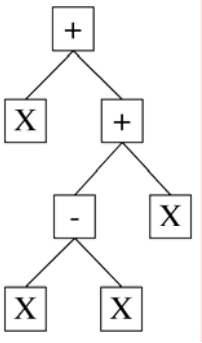


Figure 5. An example of an intron in a GP-evolved solution for $f(x) = 2x$.

$$(X+X)-\frac{(X-X)^2-(X^2-X^2)}{\frac{X-X}{X^2}+\frac{X}{X}\left(4X-\frac{X-X}{\frac{X}{X}}\right)}, \qquad (3)$$

where it can easily be understood that the third term is identically zero. It is, however, also more difficult to parse this expression than the trivial one shown in Fig. 5. This code shows how introns and increased tree depth lead to code bloat.

To give an example of algorithmic obfuscation we move to the slightly more complicated function $f(x) = x^3$. For this example we show in Fig. 7a and 7b two different results that evolved to find this solution. In Fig. 7a, the code shown evaluates to $[X^2 * X – (X – X)(X – X)] – [(X + X) – (X + X)] [X^2 * X^2]$, which simplifies to $X^3 + 0$. Although this code looks complicated, it is essentially obfuscation by introns, and is not particularly interesting. The purpose of showing this code, however, is that while it looks nearly identical to the code shown in Fig. 7b, there are distinct differences.

The code shown in Fig. 7b is an example of algorithmic obfuscation. Evaluation of the tree leads to the partially simplified expression $(X/X + X^2 + X^2 – X/X) [X^2/2X – (X – X) + (X – X)]$. Clearly there are introns in this expression, but after their removal further simplification steps give $(X^2 + X^2)(X^2/2X)$, then $2X^2 * (X/2)$, and finally X^3. The fundamental difference between the expressions in Fig. 7a and 7b is that in Fig. 7a after removal of the introns, one is left with simply X^3; whereas in Fig. 7b, removal of the introns leaves one with an expression that must be evaluated to give X^3. This, then, is what is meant by algorithmic obfuscation.

Polynomial Functions

Let us now turn our attention to the practical example of the obfuscation of the polynomial, $f(x) = x^3+x^2+x$. While it is clear from the preceding discussion that introns and algorithmic obfuscation tend to arise naturally during the course of GP optimization, we will further encourage obfuscation by incorporating not only the accuracy of the output but also the size of

```
(- (+ X X)
   (/ (- (* (- X X)
            (- X X))
         (- (* X X)
            (* X X)))
      (+ (/ (- X X)
            (* X X))
         (* (- (+ (+ X X)
                  (+ X X))
               (/ (- X X)
                  (/ X X)))
            (/ X X)))))
```

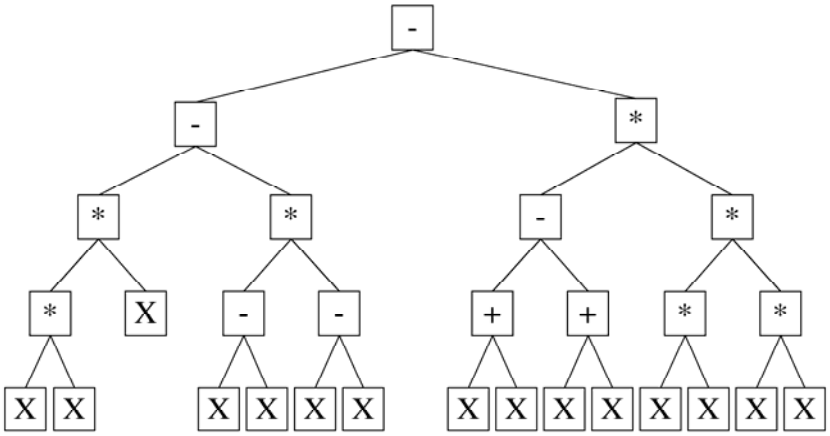
Figure 6. An expression for $f(x) = 2x$ that contains numerous introns. Everything after the first line equates to 0.

the evolved tree in the fitness criterion. Specifically, consider a GP in which the available terminals are +, -, *, and / and the arguments are functions of X. (This is the same sort of GP presented above.) In order to evolve a program that computes $f(x) = x^3+x^2+x$, it is natural to choose a fitness function like

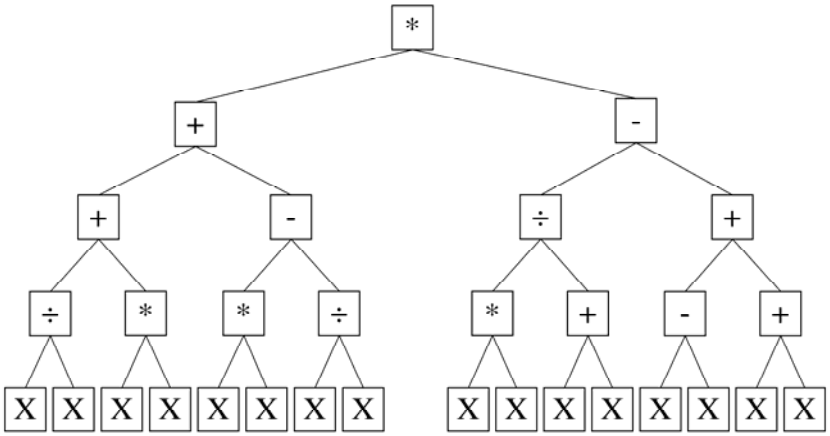
$$F = \sum_{trials} |f(x) - g(x)| \tag{4}$$

where the summation is performed over some predetermined set of fitness cases or "trials" (*i.e.* values of x), and $g(x)$ is the evolved GP that is attempting to evaluate to $f(x)$. Lower values of F represent a higher fitness. While the optimum fitness value, $F = 0$, might never be reached (in a tractable time) for complex target functions, something as simple as $f(x) = x^3+x^2+x$ is generally achievable in relatively few generations of the GP.

For example, we ran lil-gp with input values of 5120 trees (*i.e.* organisms), initial tree depth between 2 and 8, maximum depth of 32, a 9:1 ratio of crossover to reproduction rates, and 200 randomly-selected trial values for x between –1 and 1. This required only one generation to evolve the Lisp tree (+ (/ (* X X) (* (/ X X) X)) (* (+ X (* X X)) (+ X (- X X)))).



(a)



(b)

Figure 7. Evolved versions of $f(x) = x^3$ that show a) no evidence and b) clear evidence of algorithmic obfuscation.

Substituting (- X X) = 0, (/ X X) = 1, and (* X X) = X^2 yields (+ (/ X^2 (* 1 X)) (* (+ X X^2) (+ X 0))), which clearly reduces to (+ X (* (+ X X^2) X)) = X + X^2 + X^3. Thus, the evolved program contains introns, but is equivalent to the target function $f(x) = x^3+x^2+x$. For comparison, the fitness function in Eq. 4 was used to evolve solutions to both $f(x) = x^4+x^3+x^2+x$ and $f(x) = x^5+x^4+x^3+x^2+x$, with 10240 trees, initial tree depth between 2 and 10, and maximum depth of 64. The first accurate solution to the fourth-order polynomial lived in the 33rd generation and had the form (* X (+ (/ X X) (* (+ (* (+ (* X X) X) X) X) (/ X X)))), which contains two instances of the intron (/ X X). (The construct, X+1, is common in these examples for obvious reasons, and the only viable mechanism for generating a 1 is via (/ X X).) Surprisingly, for the fifth-order polynomial, the eighth generation contained the individual (- (* (+ (* (* X X) X) X) (+ X (* X X))) (- X (+ X X))), which is a match to the target function and contains no introns.

In order to see how quickly the GP can arrive at a completely unobfuscated solution, it is useful to modify the fitness function to discourage bloat, such that

$$F = \sum_{trials} N\left[\left| f(x) - g(x) \right| + 1\right], \tag{5}$$

where N is the number of nodes in the tree. While this is an overly simplified representation of obfuscation as applied to the fitness function, it is satisfactory for our present purpose. In this case, using lil-gp with the same input parameters as above, the GP arrived after only two generations at the solution (+ (* (+ (* X X) X) X) X), which is clearly $X^3 + X^2 + X$. For comparison, the fitness function in Eq. 5 was used as before to evolve solutions to both $f(x) = x^4+x^3+x^2+x$ and $f(x) = x^5+x^4+x^3+x^2+x$ over 256 generations, with 10240 trees, initial tree depth between 2 and 10, and maximum depth of 64. The best solution to the fourth-order polynomial lived in the sixth generation and had the form (* (+ (/ X X) X) (+ X (* (* X X) X))), which again contains the intron (/ X X). The GP could not evolve a match to the fifth-order polynomial within 256 generations, and ended up with the rather poor solution of X itself with a fitness of F = 97.6. (The strong bias in Eq. 5 against large trees is partly to blame for this.)

The preceding examples demonstrate that the feasibility of evolving polynomial functions decreases rapidly with increasing complexity of the target function. However, for the purposes of the present study, it is useful to consider tractable functions and to examine how the GP evolves solutions for them when obfuscation is rewarded rather than penalized. To do this, we use the fitness function

$$F = \sum_{trials} \left[\left| f(x) - g(x) \right| + 1\right] \Big/ N \, . \tag{6}$$

and the same input parameters as above, to evolve $f(x) = x^3+x^2+x$. The fittest individual lived in generation 10 and is shown in Fig. 8. The tree in Fig. 8 simplifies to exactly $X^3 + X^2 + X$, but clearly has numerous introns and algorithmic obfuscations. As discussed above, while this tree can be simplified to reveal its functionality in a relatively short time, a more complex code containing many similarly obfuscated functions would be much harder to decipher.

However, it is apparent that the evaluation of the tree in Fig. 8 might require substantially more computer time than the evaluation of $X^3 + X^2 + X$ itself. To quantify this, we converted the tree into the C code shown in Fig. 9a, and compared its performance with the code in Fig. 9b. The codes were compiled using the GNU Compiler Suite's gcc 3.3.3 on an 800Mhz Intel Pentium III Xeon processor running Red Hat Fedora Core 2 (kernel 2.6.5 and glibc 2.3.3). The assembler

instructions for two main() functions from Fig. 9 are shown in Fig. 10. Each benchmark represents the average over ten executions of the code. Without any compiler optimizations, the target code (Fig. 9b) evaluated in 0.4 sec and the evolved code (Fig. 9a) in 18.90 sec. When compiled with compiler optimization at level 3 (*via* the –O3 flag), producing the assembler instructions in Fig. 11, each code completed in 0.15 sec. This is because, as is evident from Fig. 11, the compiler's optimizations have simplified the assembler instructions of the obfuscated code (Fig. 9a) so that they are identical to those of the target code (Fig. 9b). In an attempt to reduce the optimizer's ability to rearrange the code, we converted all the arithmetic operations into function calls, as reflected in Fig. 12. While this is in general a bad idea from the perspective of the code's performance, it is a useful exercise for the present purpose. Without compiler optimizations, the compiled codes in Fig. 12b and 12a executed in 4.41 and 54.28 sec, respectively, compared to 2.48 and 35.85 sec with level 3 optimization. In this case, the compiler's optimizer was unable to substantially simplify the code in Fig. 12a, as evidenced by the assembler instructions in Fig. 13.

```
( + X
  ( * X
    ( + ( * ( / X X )
        ( + X
          ( * ( - X X )
            ( - X X ) ) ) )
      ( + ( - ( / ( * ( * ( / X X )
                  ( - X X ) )
                ( - ( * X X )
                  ( * X X ) ) )
              ( * ( + ( + X X )
                  ( / X X ) )
                ( * ( * X X )
                  ( / X X ) ) ) )
            ( + ( + ( / ( * X X )
                  ( / X X ) )
                ( + ( + X X )
                  ( - X X ) ) )
              ( + ( - ( - X X )
                  ( + ( / ( * X X )
                      ( / X X ) )
                    ( + ( + X X )
                      ( - X X ) ) ) )
                ( - ( + X X )
                  ( + X X ) ) ) ) )
          ( - ( * X X )
            ( / ( - X X )
              ( + X X ) ) ) ) ) ) ) )
```

Figure 8. A solution to $f(x) = x^3+x^2+x$, evolved with a fitness function rewarding large trees.

```
int main (int argc, char **argv) {
       double X, Y;
       for (X = -100.0; X <= 100.0; X += 0.00001) {
              Y = ((((((((((X+((X-X))))*((X-((X-X))))))
              +((X*((((X/X))*((X/X)))))))+((((((X*((X-
              X))))*((((X*X))/((X*X))))))/((((((X-X))-
              ((X*X))))*((((X-X))*((X/X)))))))))/
              (((((((X/((X/X))))*(((((((X-X))/((X/X))))+
              ((((X+X))+((X-X))))))*((X-X))))))+
              ((((((((X+X))+((X-X))))/((((X+X))-
              ((X-X))))))+((X-X))))))/X))));
       }
       return(0);
}
```

(a)

```
int main (int argc, char **argv) {
       double X, Y;
       for (X = -100.0; X <= 100.0; X += 0.00001) {
              Y = (X*X*X) + (X*X) + X;
       }
       return(0);
}
```

(b)

Figure 9. C code representations of a) the tree in Fig. 8, and b) the function $X^3 + X^2 + X$.

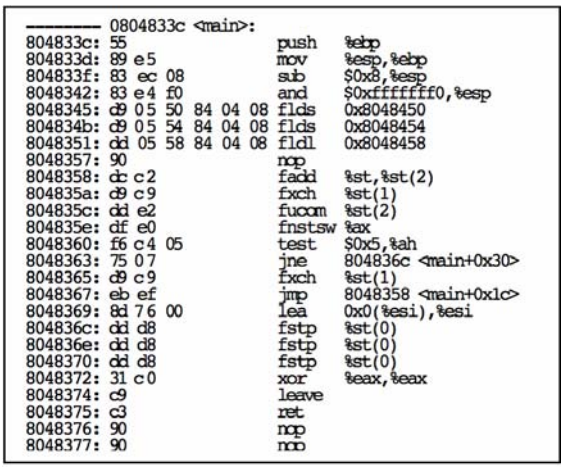
```
-------- 0804833c <main>:
804833c: 55                    push   %ebp
804833d: 89 e5                 mov    %esp,%ebp
804833f: 83 ec 18              sub    $0x18,%esp
8048342: 83 e4 f0              and    $0xfffffff0,%esp
8048345: b8 00 00 00 00        mov    $0x0,%eax
804834a: 29 c4                 sub    %eax,%esp
804834c: b8 00 00 00 00        mov    $0x0,%eax
8048351: ba 00 00 59 c0        mov    $0xc0590000,%edx
8048356: 89 45 f8              mov    %eax,0xfffffff8(%ebp)
8048359: 89 55 fc              mov    %edx,0xfffffffc(%ebp)
804835c: dd 45 f8              fldl   0xfffffff8(%ebp)
804835f: dd 05 28 85 04 08     fldl   0x8048528
8048365: da e9                 fucompp
8048367: df e0                 fnstsw %ax
8048369: f6 c4 05              test   $0x5,%ah
804836c: 74 05                 je     8048373 <main+0x37>
804836e: e9 da 00 00 00        jmp    804844d <main+0x111>
8048373: dd 45 f8              fldl   0xfffffff8(%ebp)
8048376: dc 65 f8              fsubl  0xfffffff8(%ebp)
8048379: dc 45 f8              faddl  0xfffffff8(%ebp)
804837c: dd 45 f8              fldl   0xfffffff8(%ebp)
804837f: dc 65 f8              fsubl  0xfffffff8(%ebp)
8048382: dd 45 f8              fldl   0xfffffff8(%ebp)
8048385: de e1                 fsubp  %st,%st(1)
8048387: de c9                 fmulp  %st,%st(1)
8048389: dd 45 f8              fldl   0xfffffff8(%ebp)
804838c: dc 75 f8              fdivl  0xfffffff8(%ebp)
804838f: dd 45 f8              fldl   0xfffffff8(%ebp)
8048392: dc 75 f8              fdivl  0xfffffff8(%ebp)
8048395: de c9                 fmulp  %st,%st(1)
8048397: dc 4d f8              fmull  0xfffffff8(%ebp)
804839a: de c1                 faddp  %st,%st(1)
804839c: dd 45 f8              fldl   0xfffffff8(%ebp)
804839f: dc 65 f8              fsubl  0xfffffff8(%ebp)
80483a2: dc 4d f8              fmull  0xfffffff8(%ebp)
80483a5: dd 45 f8              fldl   0xfffffff8(%ebp)
80483a8: dc 4d f8              fmull  0xfffffff8(%ebp)
80483ab: dd 45 f8              fldl   0xfffffff8(%ebp)
80483ae: dc 4d f8              fmull  0xfffffff8(%ebp)
80483b1: de f9                 fdivrp %st,%st(1)
80483b3: de c9                 fmulp  %st,%st(1)
80483b5: dd 45 f8              fldl   0xfffffff8(%ebp)
80483b8: dc 65 f8              fsubl  0xfffffff8(%ebp)
80483bb: dd 45 f8              fldl   0xfffffff8(%ebp)
80483be: dc 4d f8              fmull  0xfffffff8(%ebp)
80483c1: de e9                 fsubrp %st,%st(1)
80483c3: dd 45 f8              fldl   0xfffffff8(%ebp)
80483c6: dc 65 f8              fsubl  0xfffffff8(%ebp)
80483c9: dd 45 f8              fldl   0xfffffff8(%ebp)
80483cc: dc 75 f8              fdivl  0xfffffff8(%ebp)
80483cf: de c9                 fmulp  %st,%st(1)
80483d1: de c9                 fmulp  %st,%st(1)
80483d3: de f9                 fdivrp %st,%st(1)
80483d5: de c1                 faddp  %st,%st(1)
80483d7: dd 45 f8              fldl   0xfffffff8(%ebp)
80483da: dc 75 f8              fdivl  0xfffffff8(%ebp)
80483dd: dd 45 f8              fldl   0xfffffff8(%ebp)
80483e0: de f1                 fdivp  %st,%st(1)
80483e2: dd 45 f8              fldl   0xfffffff8(%ebp)
80483e5: dc 65 f8              fsubl  0xfffffff8(%ebp)
80483e8: dd 45 f8              fldl   0xfffffff8(%ebp)
80483eb: dc 75 f8              fdivl  0xfffffff8(%ebp)
80483ee: de f9                 fdivrp %st,%st(1)
80483f0: dd 45 f8              fldl   0xfffffff8(%ebp)
80483f3: dc 45 f8              faddl  0xfffffff8(%ebp)
80483f6: dd 45 f8              fldl   0xfffffff8(%ebp)
80483f9: dc 65 f8              fsubl  0xfffffff8(%ebp)
80483fc: de c1                 faddp  %st,%st(1)
80483fe: de c1                 faddp  %st,%st(1)
8048400: dd 45 f8              fldl   0xfffffff8(%ebp)
8048403: dc 65 f8              fsubl  0xfffffff8(%ebp)
8048406: de c9                 fmulp  %st,%st(1)
8048408: de c9                 fmulp  %st,%st(1)
804840a: dd 45 f8              fldl   0xfffffff8(%ebp)
804840d: dc 45 f8              faddl  0xfffffff8(%ebp)
8048410: dd 45 f8              fldl   0xfffffff8(%ebp)
8048413: dc 65 f8              fsubl  0xfffffff8(%ebp)
8048416: de c1                 faddp  %st,%st(1)
8048418: dd 45 f8              fldl   0xfffffff8(%ebp)
804841b: dc 45 f8              faddl  0xfffffff8(%ebp)
804841e: dd 45 f8              fldl   0xfffffff8(%ebp)
8048421: dc 65 f8              fsubl  0xfffffff8(%ebp)
8048424: de e9                 fsubrp %st,%st(1)
8048426: de f9                 fdivrp %st,%st(1)
8048428: dd 45 f8              fldl   0xfffffff8(%ebp)
804842b: dc 65 f8              fsubl  0xfffffff8(%ebp)
804842e: de c1                 faddp  %st,%st(1)
8048430: de c1                 faddp  %st,%st(1)
8048432: dc 75 f8              fdivl  0xfffffff8(%ebp)
8048435: de f9                 fdivrp %st,%st(1)
8048437: dd 5d f0              fstpl  0xfffffff0(%ebp)
804843a: dd 45 f8              fldl   0xfffffff8(%ebp)
804843d: dd 05 30 85 04 08     fldl   0x8048530
8048443: de c1                 faddp  %st,%st(1)
8048445: dd 5d f8              fstpl  0xfffffff8(%ebp)
8048448: e9 0f ff ff ff        jmp    804835c <main+0x20>
804844d: b8 00 00 00 00        mov    $0x0,%eax
8048452: c9                    leave
8048453: c3                    ret
```

(a)

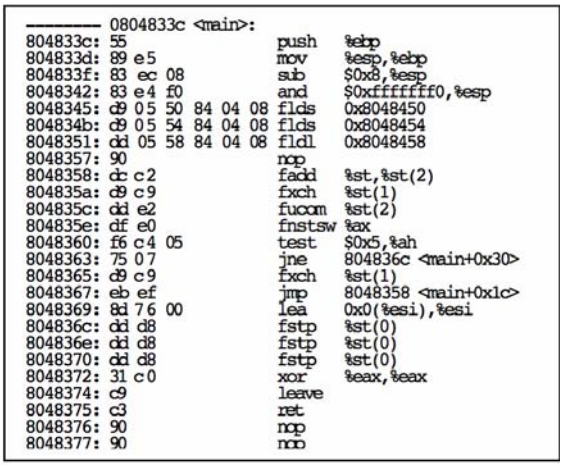
```
-------- 0804833c <main>:
804833c: 55                    push   %ebp
804833d: 89 e5                 mov    %esp,%ebp
804833f: 83 ec 18              sub    $0x18,%esp
8048342: 83 e4 f0              and    $0xfffffff0,%esp
8048345: b8 00 00 00 00        mov    $0x0,%eax
804834a: 29 c4                 sub    %eax,%esp
804834c: b8 00 00 00 00        mov    $0x0,%eax
8048351: ba 00 00 59 c0        mov    $0xc0590000,%edx
8048356: 89 45 f8              mov    %eax,0xfffffff8(%ebp)
8048359: 89 55 fc              mov    %edx,0xfffffffc(%ebp)
804835c: dd 45 f8              fldl   0xfffffff8(%ebp)
804835f: dd 05 78 84 04 08     fldl   0x8048478
8048365: da e9                 fucompp
8048367: df e0                 fnstsw %ax
8048369: f6 c4 05              test   $0x5,%ah
804836c: 74 02                 je     8048370 <main+0x34>
804836e: eb 27                 jmp    8048397 <main+0x5b>
8048370: dd 45 f8              fldl   0xfffffff8(%ebp)
8048373: dc 4d f8              fmull  0xfffffff8(%ebp)
8048376: dc 4d f8              fmull  0xfffffff8(%ebp)
8048379: dd 45 f8              fldl   0xfffffff8(%ebp)
804837c: dc 4d f8              fmull  0xfffffff8(%ebp)
804837f: de c1                 faddp  %st,%st(1)
8048381: dc 45 f8              faddl  0xfffffff8(%ebp)
8048384: dd 5d f0              fstpl  0xfffffff0(%ebp)
8048387: dd 45 f8              fldl   0xfffffff8(%ebp)
804838a: dd 05 80 84 04 08     fldl   0x8048480
8048390: de c1                 faddp  %st,%st(1)
8048392: dd 5d f8              fstpl  0xfffffff8(%ebp)
8048395: eb c5                 jmp    804835c <main+0x20>
8048397: b8 00 00 00 00        mov    $0x0,%eax
804839c: c9                    leave
804839d: c3                    ret
804839e: 90                    nop
804839f: 90                    nop
```

(b)

Figure 10. Assembler instructions for the C codes in Fig. 9, compiled without optimization.



(a)



(b)

Figure 11. Assembler instructions for the C codes in Fig. 9, compiled with level 3 optimization. Note that the two sets of assembler instructions are identical.

```
double a(double X, double Y);
double s(double X, double Y);
double m(double X, double Y);
double d(double X, double Y);

int main (int argc, char **argv) {
       double X, Y;
       for (X = -100.0; X <= 100.0; X += 0.00001) {
              Y =
              (d((a((a((m((a(X,(s(X,X)))),(s(X,(s(X,X))))))),(
              m(X,(m((d(X,X)),(d(X,X))))))))),(d((m((m(X,
              (s(X,X)))),(d((m(X,X)),(m(X,X))))))),(m((s((
              s(X,X)),(m(X,X)))),(m((s(X,X)),(d(X,X))))))))))
              ,(d((a((m((d(X,(d(X,X)))),(m((a((d((s(X,X)),(
              d(X,X)))),(a((a(X,X)),(s(X,X))))))),(s(X,X))))))
              ,(a((d((a((a(X,X)),(s(X,X)))),(s((a(X,X)),(
              s(X,X))))))),(s(X,X))))))),X))));
       }
        return(0);
}

double a(double X, double Y) { return(X+Y); }
double s(double X, double Y) { return(X-Y); }
double m(double X, double Y) { return(X*Y); }
double d(double X, double Y) { return(X/Y); }
```

(a)

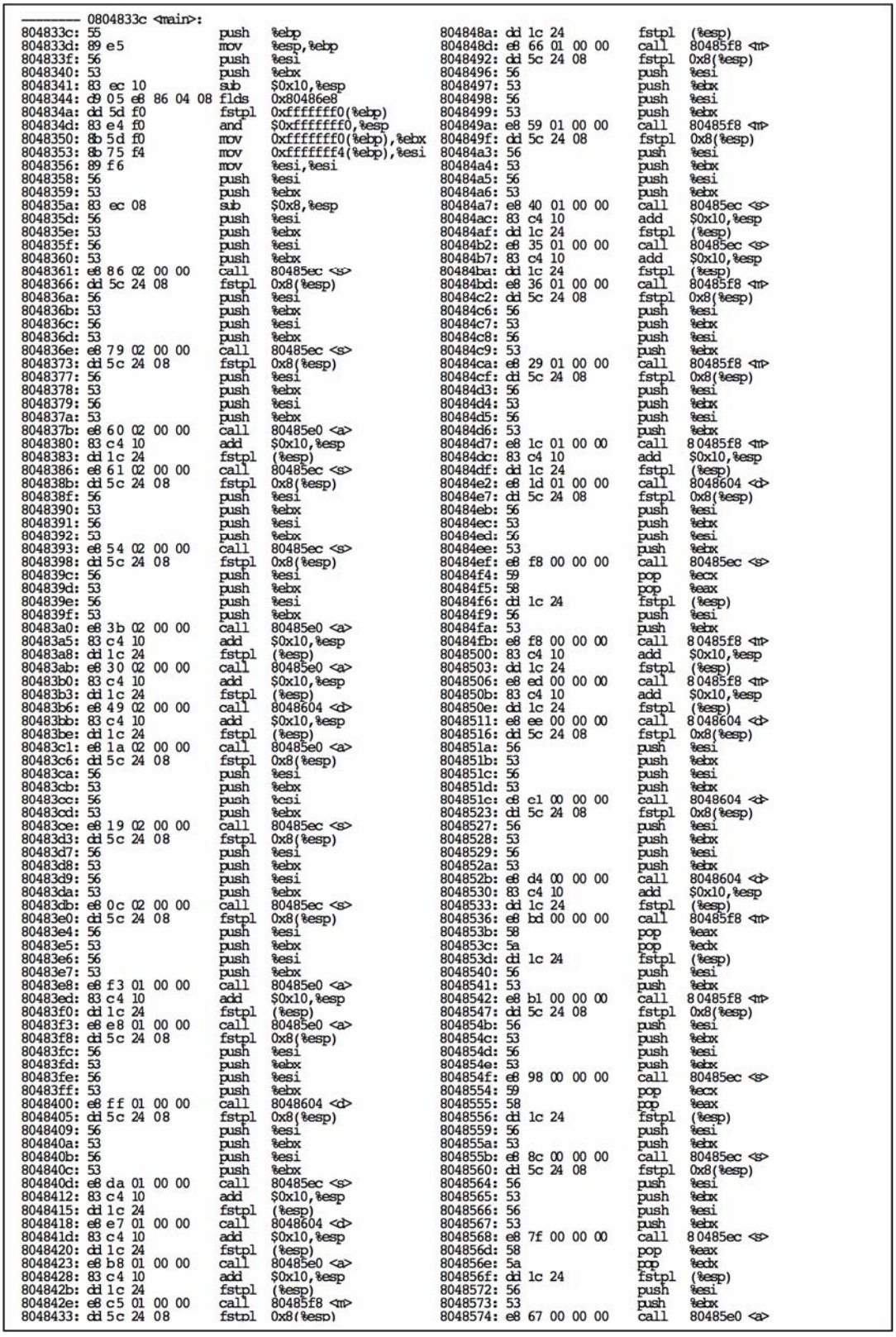
```
double a(double X, double Y);
double s(double X, double Y);
double m(double X, double Y);
double d(double X, double Y);

int main (int argc, char **argv) {
       double X, Y;
       for (X = -100.0; X <= 100.0; X += 0.00001) {
              Y = a(a(m(m(X,X),X),m(X,X)),X);
       }
       return(0);
}

double a(double X, double Y) { return(X+Y); }
double s(double X, double Y) { return(X-Y); }
double m(double X, double Y) { return(X*Y); }
double d(double X, double Y) { return(X/Y); }
```

(b)

Figure 12. C code representations of a) the tree in Fig. 8, and b) the function $X^3 + X^2 + X$, using function calls in place of arithmetic operators.



```
-------- 0804833c <main>:
804833c: 55                    push   %ebp
804833d: 89 e5                 mov    %esp,%ebp
804833f: 56                    push   %esi
8048340: 53                    push   %ebx
8048341: 83 ec 10              sub    $0x10,%esp
8048344: d9 05 e8 86 04 08     flds   0x80486e8
804834a: dd 5d f0              fstpl  0xfffffff0(%ebp)
804834d: 83 e4 f0              and    $0xfffffff0,%esp
8048350: 8b 5d f0              mov    0xfffffff0(%ebp),%ebx
8048353: 8b 75 f4              mov    0xfffffff4(%ebp),%esi
8048356: 89 f6                 mov    %esi,%esi
8048358: 56                    push   %esi
8048359: 53                    push   %ebx
804835a: 83 ec 08              sub    $0x8,%esp
804835d: 56                    push   %esi
804835e: 53                    push   %ebx
804835f: 56                    push   %esi
8048360: 53                    push   %ebx
8048361: e8 86 02 00 00        call   80485ec <s>
8048366: dd 5c 24 08           fstpl  0x8(%esp)
804836a: 56                    push   %esi
804836b: 53                    push   %ebx
804836c: 56                    push   %esi
804836d: 53                    push   %ebx
804836e: e8 79 02 00 00        call   80485ec <s>
8048373: dd 5c 24 08           fstpl  0x8(%esp)
8048377: 56                    push   %esi
8048378: 53                    push   %ebx
8048379: 56                    push   %esi
804837a: 53                    push   %ebx
804837b: e8 60 02 00 00        call   80485e0 <a>
8048380: 83 c4 10              add    $0x10,%esp
8048383: dd 1c 24              fstpl  (%esp)
8048386: e8 61 02 00 00        call   80485ec <s>
804838b: dd 5c 24 08           fstpl  0x8(%esp)
804838f: 56                    push   %esi
8048390: 53                    push   %ebx
8048391: 56                    push   %esi
8048392: 53                    push   %ebx
8048393: e8 54 02 00 00        call   80485ec <s>
8048398: dd 5c 24 08           fstpl  0x8(%esp)
804839c: 56                    push   %esi
804839d: 53                    push   %ebx
804839e: 56                    push   %esi
804839f: 53                    push   %ebx
80483a0: e8 3b 02 00 00        call   80485e0 <a>
80483a5: 83 c4 10              add    $0x10,%esp
80483a8: dd 1c 24              fstpl  (%esp)
80483ab: e8 30 02 00 00        call   80485e0 <a>
80483b0: 83 c4 10              add    $0x10,%esp
80483b3: dd 1c 24              fstpl  (%esp)
80483b6: e8 49 02 00 00        call   8048604 <d>
80483bb: 83 c4 10              add    $0x10,%esp
80483be: dd 1c 24              fstpl  (%esp)
80483c1: e8 1a 02 00 00        call   80485e0 <a>
80483c6: dd 5c 24 08           fstpl  0x8(%esp)
80483ca: 56                    push   %esi
80483cb: 53                    push   %ebx
80483cc: 56                    push   %esi
80483cd: 53                    push   %ebx
80483ce: e8 19 02 00 00        call   80485ec <s>
80483d3: dd 5c 24 08           fstpl  0x8(%esp)
80483d7: 56                    push   %esi
80483d8: 53                    push   %ebx
80483d9: 56                    push   %esi
80483da: 53                    push   %ebx
80483db: e8 0c 02 00 00        call   80485ec <s>
80483e0: dd 5c 24 08           fstpl  0x8(%esp)
80483e4: 56                    push   %esi
80483e5: 53                    push   %ebx
80483e6: 56                    push   %esi
80483e7: 53                    push   %ebx
80483e8: e8 f3 01 00 00        call   80485e0 <a>
80483ed: 83 c4 10              add    $0x10,%esp
80483f0: dd 1c 24              fstpl  (%esp)
80483f3: e8 e8 01 00 00        call   80485e0 <a>
80483f8: dd 5c 24 08           fstpl  0x8(%esp)
80483fc: 56                    push   %esi
80483fd: 53                    push   %ebx
80483fe: 56                    push   %esi
80483ff: 53                    push   %ebx
8048400: e8 ff 01 00 00        call   8048604 <d>
8048405: dd 5c 24 08           fstpl  0x8(%esp)
8048409: 56                    push   %esi
804840a: 53                    push   %ebx
804840b: 56                    push   %esi
804840c: 53                    push   %ebx
804840d: e8 da 01 00 00        call   80485ec <s>
8048412: 83 c4 10              add    $0x10,%esp
8048415: dd 1c 24              fstpl  (%esp)
8048418: e8 e7 01 00 00        call   8048604 <d>
804841d: 83 c4 10              add    $0x10,%esp
8048420: dd 1c 24              fstpl  (%esp)
8048423: e8 b8 01 00 00        call   80485e0 <a>
8048428: 83 c4 10              add    $0x10,%esp
804842b: dd 1c 24              fstpl  (%esp)
804842e: e8 c5 01 00 00        call   80485f8 <m>
8048433: dd 5c 24 08           fstpl  0x8(%esp)
804848a: dd 1c 24              fstpl  (%esp)
804848d: e8 66 01 00 00        call   80485f8 <m>
8048492: dd 5c 24 08           fstpl  0x8(%esp)
8048496: 56                    push   %esi
8048497: 53                    push   %ebx
8048498: 56                    push   %esi
8048499: 53                    push   %ebx
804849a: e8 59 01 00 00        call   80485f8 <m>
804849f: dd 5c 24 08           fstpl  0x8(%esp)
80484a3: 56                    push   %esi
80484a4: 53                    push   %ebx
80484a5: 56                    push   %esi
80484a6: 53                    push   %ebx
80484a7: e8 40 01 00 00        call   80485ec <s>
80484ac: 83 c4 10              add    $0x10,%esp
80484af: dd 1c 24              fstpl  (%esp)
80484b2: e8 35 01 00 00        call   80485ec <s>
80484b7: 83 c4 10              add    $0x10,%esp
80484ba: dd 1c 24              fstpl  (%esp)
80484bd: e8 36 01 00 00        call   80485f8 <m>
80484c2: dd 5c 24 08           fstpl  0x8(%esp)
80484c6: 56                    push   %esi
80484c7: 53                    push   %ebx
80484c8: 56                    push   %esi
80484c9: 53                    push   %ebx
80484ca: e8 29 01 00 00        call   80485f8 <m>
80484cf: dd 5c 24 08           fstpl  0x8(%esp)
80484d3: 56                    push   %esi
80484d4: 53                    push   %ebx
80484d5: 56                    push   %esi
80484d6: 53                    push   %ebx
80484d7: e8 1c 01 00 00        call   80485f8 <m>
80484dc: 83 c4 10              add    $0x10,%esp
80484df: dd 1c 24              fstpl  (%esp)
80484e2: e8 1d 01 00 00        call   8048604 <d>
80484e7: dd 5c 24 08           fstpl  0x8(%esp)
80484eb: 56                    push   %esi
80484ec: 53                    push   %ebx
80484ed: 56                    push   %esi
80484ee: 53                    push   %ebx
80484ef: e8 f8 00 00 00        call   80485ec <s>
80484f4: 59                    pop    %ecx
80484f5: 58                    pop    %eax
80484f6: dd 1c 24              fstpl  (%esp)
80484f9: 56                    push   %esi
80484fa: 53                    push   %ebx
80484fb: e8 f8 00 00 00        call   80485f8 <m>
8048500: 83 c4 10              add    $0x10,%esp
8048503: dd 1c 24              fstpl  (%esp)
8048506: e8 ed 00 00 00        call   80485f8 <m>
804850b: 83 c4 10              add    $0x10,%esp
804850e: dd 1c 24              fstpl  (%esp)
8048511: e8 ee 00 00 00        call   8048604 <d>
8048516: dd 5c 24 08           fstpl  0x8(%esp)
804851a: 56                    push   %esi
804851b: 53                    push   %ebx
804851c: 56                    push   %esi
804851d: 53                    push   %ebx
804851e: e8 c1 00 00 00        call   8048604 <d>
8048523: dd 5c 24 08           fstpl  0x8(%esp)
8048527: 56                    push   %esi
8048528: 53                    push   %ebx
8048529: 56                    push   %esi
804852a: 53                    push   %ebx
804852b: e8 d4 00 00 00        call   8048604 <d>
8048530: 83 c4 10              add    $0x10,%esp
8048533: dd 1c 24              fstpl  (%esp)
8048536: e8 bd 00 00 00        call   80485f8 <m>
804853b: 58                    pop    %eax
804853c: 5a                    pop    %edx
804853d: dd 1c 24              fstpl  (%esp)
8048540: 56                    push   %esi
8048541: 53                    push   %ebx
8048542: e8 b1 00 00 00        call   80485f8 <m>
8048547: dd 5c 24 08           fstpl  0x8(%esp)
804854b: 56                    push   %esi
804854c: 53                    push   %ebx
804854d: 56                    push   %esi
804854e: 53                    push   %ebx
804854f: e8 98 00 00 00        call   80485ec <s>
8048554: 59                    pop    %ecx
8048555: 58                    pop    %eax
8048556: dd 1c 24              fstpl  (%esp)
8048559: 56                    push   %esi
804855a: 53                    push   %ebx
804855b: e8 8c 00 00 00        call   80485ec <s>
8048560: dd 5c 24 08           fstpl  0x8(%esp)
8048564: 56                    push   %esi
8048565: 53                    push   %ebx
8048566: 56                    push   %esi
8048567: 53                    push   %ebx
8048568: e8 7f 00 00 00        call   80485ec <s>
804856d: 58                    pop    %eax
804856e: 5a                    pop    %edx
804856f: dd 1c 24              fstpl  (%esp)
8048572: 56                    push   %esi
8048573: 53                    push   %ebx
8048574: e8 67 00 00 00        call   80485e0 <a>
```

(a)

Figure 13. Assembler instructions for the C codes in Fig. 12 with level 3 optimization.

```
8048437: 56              push   %esi
8048438: 53              push   %ebx
8048439: 56              push   %esi
804843a: 53              push   %ebx
804843b: e8 c4 01 00 00  call   8048604 <d>
8048440: 58              pop    %eax
8048441: 5a              pop    %edx
8048442: dd 1c 24        fstpl  (%esp)
8048445: 56              push   %esi
8048446: 53              push   %ebx
8048447: e8 b8 01 00 00  call   8048604 <d>
804844c: 83 c4 10        add    $0x10,%esp
804844f: dd 1c 24        fstpl  (%esp)
8048452: e8 a1 01 00 00  call   80485f8 <m>
8048457: 83 c4 10        add    $0x10,%esp
804845a: dd 1c 24        fstpl  (%esp)
804845d: e8 7e 01 00 00  call   80485e0 <a>
8048462: 83 c4 10        add    $0x10,%esp
8048465: dd 1c 24        fstpl  (%esp)
8048468: e8 97 01 00 00  call   8048604 <d>
804846d: dd 5c 24 08     fstpl  0x8(%esp)
8048471: 56              push   %esi
8048472: 53              push   %ebx
8048473: 56              push   %esi
8048474: 53              push   %ebx
8048475: e8 8a 01 00 00  call   8048604 <d>
804847a: dd 5c 24 08     fstpl  0x8(%esp)
804847e: 56              push   %esi
804847f: 53              push   %ebx
8048480: 56              push   %esi
8048481: 53              push   %ebx
8048482: e8 65 01 00 00  call   80485ec <s>
8048487: 83 c4 10        add    $0x10,%esp

8048579: 83 c4 10        add    $0x10,%esp
804857c: dd 1c 24        fstpl  (%esp)
804857f: e8 74 00 00 00  call   80485f8 <m>
8048584: 83 c4 10        add    $0x10,%esp
8048587: dd 1c 24        fstpl  (%esp)
804858a: e8 51 00 00 00  call   80485e0 <a>
804858f: 83 c4 10        add    $0x10,%esp
8048592: dd 1c 24        fstpl  (%esp)
8048595: e8 46 00 00 00  call   80485e0 <a>
804859a: 83 c4 10        add    $0x10,%esp
804859d: dd 1c 24        fstpl  (%esp)
80485a0: e8 5f 00 00 00  call   8048604 <d>
80485a5: dd d8           fstp   %st(0)
80485a7: 89 5d f0        mov    %ebx,0xfffffff0(%ebp)
80485aa: 89 75 f4        mov    %esi,0xfffffff4(%ebp)
80485ad: dd 45 f0        fldl   0xfffffff0(%ebp)
80485b0: dc 05 f0 86 04 08 faddl  0x80486f0
80485b6: d9 05 ec 86 04 08 flds   0x80486ec
80485bc: dd e9           fucomp %st(1)
80485be: df e0           fnstsw %ax
80485c0: dd 5d f0        fstpl  0xfffffff0(%ebp)
80485c3: 83 c4 10        add    $0x10,%esp
80485c6: f6 c4 05        test   $0x5,%ah
80485c9: 8b 5d f0        mov    0xfffffff0(%ebp),%ebx
80485cc: 8b 75 f4        mov    0xfffffff4(%ebp),%esi
80485cf: 0f 84 83 fd ff ff je     8048358 <main+0x1c>
80485d5: 8d 65 f8        lea    0xfffffff8(%ebp),%esp
80485d8: 5b              pop    %ebx
80485d9: 31 c0           xor    %eax,%eax
80485db: 5e              pop    %esi
80485dc: c9              leave
80485dd: c3              ret
80485de: 89 f6           mov    %esi,%esi

-------- 080485e0 <a>:
80485e0: 55              push   %ebp
80485e1: 89 e5           mov    %esp,%ebp
80485e3: dd 45 10        fldl   0x10(%ebp)
80485e6: dc 45 08        faddl  0x8(%ebp)
80485e9: c9              leave
80485ea: c3              ret
80485eb: 90              nop
-------- 080485ec <m>:
80485ec:  55             push   %ebp
80485ed:  89 e5          mov    %esp,%ebp
80485ef: dd 45 10        fldl   0x10(%ebp)
80485f2: dc 4d 08        fmull  0x8(%ebp)
80485f5: c9              leave
80485f6: c3              ret
80485f7:  90             nop

-------- 080485f8 <s>:
80485f8:  55             push   %ebp
80485f9:  89 e5          mov    %esp,%ebp
80485fb:  dd 45 10       fldl   0x10(%ebp)
80485fe:  dc 6d 08       fsubrl 0x8(%ebp)
8048601:  c9             leave
8048602:  c3             ret
8048603:  90             nop
-------- 08048604 <d>:
8048604: 55              push   %ebp
8048605: 89 e5           mov    %esp,%ebp
8048607: dd 45 10        fldl   0x10(%ebp)
804860a: dc 7d 08        fdivrl 0x8(%ebp)
804860d: c9              leave
804860e: c3              ret
804860f: 90              nop
```

(a) (cont'd)

```
-------- 0804833c <main>:
804833c: 55              push   %ebp
804833d: 89 e5           mov    %esp,%ebp
804833f: 83 ec 08        sub    $0x8,%esp
8048342: d9 05 d8 84 04 08 flds   0x80484d8
8048348: 83 e4 f0        and    $0xfffffff0,%esp
804834b: dd 5d f8        fstpl  0xfffffff8(%ebp)
804834e: 89 f6           mov    %esi,%esi
8048350: ff 75 fc        pushl  0xfffffffc(%ebp)
8048353: ff 75 f8        pushl  0xfffffff8(%ebp)
8048356: 83 ec 08        sub    $0x8,%esp
8048359: ff 75 fc        pushl  0xfffffffc(%ebp)
804835c: ff 75 f8        pushl  0xfffffff8(%ebp)
804835f: ff 75 fc        pushl  0xfffffffc(%ebp)
8048362: ff 75 f8        pushl  0xfffffff8(%ebp)
8048365: e8 72 00 00 00  call   80483dc <m>
804836a: dd 5c 24 08     fstpl  0x8(%esp)
804836e: ff 75 fc        pushl  0xfffffffc(%ebp)
8048371: ff 75 f8        pushl  0xfffffff8(%ebp)
8048374: 83 ec 08        sub    $0x8,%esp
8048377: ff 75 fc        pushl  0xfffffffc(%ebp)
804837a: ff 75 f8        pushl  0xfffffff8(%ebp)
804837d: ff 75 fc        pushl  0xfffffffc(%ebp)
8048380: ff 75 f8        pushl  0xfffffff8(%ebp)
8048383: e8 54 00 00 00  call   80483dc <m>

8048388: 83 c4 10        add    $0x10,%esp
804838b: dd 1c 24        fstpl  (%esp)
804838e: e8 49 00 00 00   call   80483dc <m>
8048393: 83 c4 10        add    $0x10,%esp
8048396: dd 1c 24        fstpl  (%esp)
8048399: e8 32 00 00 00   call   80483d0 <a>
804839e: 83 c4 10        add    $0x10,%esp
80483a1: dd 1c 24        fstpl  (%esp)
80483a4: e8 27 00 00 00   call   80483d0 <a>
80483a9: dd d8           fstp   %st(0)
80483ab: dd 45 f8        fldl   0xfffffff8(%ebp)
80483ae: dc 05 e0 84 04 08 faddl  0x80484e0
80483b4: d9 05 dc 84 04 08 flds   0x80484dc
80483ba: dd e9           fucomp %st(1)
80483bc: df e0           fnstsw %ax
80483be: 83 c4 10        add    $0x10,%esp
80483c1: f6 c4 05        test   $0x5,%ah
80483c4: dd 5d f8        fstpl  0xfffffff8(%ebp)
80483c7: 74 87           je     8048350 <main+0x14>
80483c9: 31 c0           xor    %eax,%eax
80483cb: c9              leave
80483cc: c3              ret
80483cd: 8d 76 00        lea    0x0(%esi),%esi

-------- 080483d0 <a>:
80483d0: 55              push   %ebp
80483d1: 89 e5           mov    %esp,%ebp
80483d3: dd 45 10        fldl   0x10(%ebp)
80483d6: dc 45 08        faddl  0x8(%ebp)
80483d9: c9              leave
80483da: c3              ret
80483db: 90              nop
-------- 080483dc <m>:
80483dc:  55             push   %ebp
80483dd:  89 e5          mov    %esp,%ebp
80483df:  dd 45 10       fldl   0x10(%ebp)
80483e2:  dc 4d 08       fmull  0x8(%ebp)
80483e5:  c9             leave
80483e6:  c3             ret
80483e7:  90             nop

-------- 080483e8 <s>:
80483e8:  55             push   %ebp
80483e9:  89 e5          mov    %esp,%ebp
80483eb:  dd 45 10       fldl   0x10(%ebp)
80483ee:  dc 6d 08       fsubrl 0x8(%ebp)
80483f1:  c9             leave
80483f2:  c3             ret
80483f3:  90             nop
-------- 080483f4 <d>:
80483f4: 55              push   %ebp
80483f5: 89 e5           mov    %esp,%ebp
80483f7: dd 45 10        fldl   0x10(%ebp)
80483fa: dc 7d 08        fdivrl 0x8(%ebp)
80483fd: c9              leave
80483fe: c3              ret
80483ff: 90              nop
```

(b)

Figure 13 (cont'd). Assembler instructions for the C codes in Fig. 12 with level 3 optimization.

Conclusions

Our results point to a few fundamental shortcoming inherent in GP, when applied to obfuscation for asset protection.

First, the very nature of evolution by reproduction and mutation makes it unlikely that a random population will converge to any complex solutions in a tractable time frame. In addition, the primary mechanism for obfuscation is complexity through introns, and the presence of introns will by definition increase the number of operations required to evaluate a function, for example. Therefore, using the tools and approaches detailed herein, it is practical to apply GP only to relatively simple functions and algorithms, and the natural process of obfuscation by introns leads to dramatic penalties in performance when compared to an unobfuscated solution, as evidenced by the benchmarks described above.

Second, introns are potentially very easy to identify, especially in algorithms that use basic operators like arithmetic. This means not only that a human might quickly simplify smaller algorithms obfuscated using a GP, but also that a computer can easily simplify the algorithm to its target form through compiler optimization, for example. While steps can be taken to mitigate this shortcoming, *e.g.* by replacing basic operations with function calls, this procedure is potentially impractical in both its logistical implications and its impact on the performance of the obfuscated algorithm.

We therefore conclude that GP is not an appropriate mechanism for the obfuscation of code because complex functions can not be reproduced exactly, and the obfuscation of multiple smaller functions will lead to unacceptable penalties in performance.

References

[1] John R. Koza, Genetic Programming (MIT Press, 1992).
[2] http://garage.cps.msu.edu/software/software-index.html#lilgp
[3] John R. Koza, Martin A. Keane, and Matthew J. Streeter, *Scientific American*, Februrary 2003.
[4]Christian Collberg, Clark Thomborson, and Douglas Low, Technical Report 148, Dept. of Computer Science, University of Auckland (1997).
[5] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang, Electronic Colloquium on Computational Complexity, Report No. 57 (2001).
[6] Jostein Gaarder, Sophie's World (Phoenix House, 1995) p. 256.
[7] Philip L. Campbell, SAN2004-2198 (2004).
[8] S. Luke, Proceedings of GECCO 2000, 228-235 (2000).
[9] Peter W.H. Smith, in Soft Computing and its Techniques, edited by R. John and R. Birkenhead. pp. 166-171. Physica-Verlag (1999).

Distribution

1	MS 0123	Donna L. Chavez (LDRD Office), 01011
1	MS 0455	Hamilton E. Link, 05632
1	MS 0785	Cheryl L. Beaver, 05614
1	MS 0785	Philip L. Campbell, 05616
2	MS 0899	Technical Library, 09616
1	MS 1205	James R. Gosler, 05004
2	MS 1411	Corbett C. Battaile, 01834
10	MS 1411	Michael E. Chandross, 01834
1	MS 1411	H. Eliot Fang, 01834
1	MS 9018	Central Technical Files, 08945-1