Imposing Order on Program Statements to Assist Anti-Virus Scanners

Arun Lakhotia and Moinuddin Mohammed
University of Louisiana at Lafayette
arun@louisiana.edu

Abstract

A metamorphic virus applies semantics preserving transformations on itself to create a different variant before propagation. Metamorphic computer viruses thwart current anti-virus technologies that use signatures—a fixed sequence of bytes from a sample of a virus—since two variants of a metamorphic virus may not share the same signature. A method to impose an order on the statements and components of expressions of a program is presented. The method, called a "zeroing transformation," reduces the number of possible variants of a program created by reordering statement, reshaping expression, and renaming variable. On a collection of C program used for evaluation, the zeroing transformation reduced the space of program variants due to statement reordering from 10^{183} to 10^{20}. Further reduction can be expected by undoing other transformations. Anti-virus technologies may be improved by extracting signatures from zero form of a virus, and not the original version.

1. Introduction

A *metamorphic* computer virus transforms its code before propagating it to a new host [2, 4, 17, 18]. Win32/Evol, Win32/Zperm, and Win32/Bistro are some recent metamorphic viruses. Zperm carries with it the Real Permutating Engine (RPME), a metamorphic engine that can be combined with any virus to make it metamorphic [18]. There are other similar metamorphic engines available on websites catering to hackers.

Metamorphic viruses can escape signature-based anti-virus scanners [3]. The emergence of metamorphic viruses calls for new methods for anti-virus (AV) scanning. An ideal AV Scanner would be capable of detecting the different possible variants of a metamorphic virus by using signature from only one known variant.

This investigation has been motivated by a desire to create an ideal AV scanner. In this paper we investigate the question: Is it possible to transform a program to a canonical form such that two variants of the program have the same form? If such a transformation is feasible then, instead of the original virus, signatures may be extracted from its canonical form. When scanning, a suspect program may be converted to its canonical form and then checked for the existence of signature of known viruses.

As a step towards the larger goal, this paper presents a set of heuristics to impose order on the statements of a C-like program. By imposing such an order, it is expected that one can undo the effect of statement/instruction reordering transformation performed by metamorphic viruses. While the viruses posing greatest challenges are usually binary, an initial step is important. Besides, it is expected that for any significant analysis a virus will be first decompiled to a higher language. Thus, the method presented should be applicable for binary viruses.

Using GrammaTech's CodeSurfer™ we have implemented the proposed method in a prototype tool, C⊕. Empirical analysis of the method on real-world C programs show that our method is promising. For the programs studied, after fixing order of statements using our method, the number of possible permutations of the programs was reduced by a factor of 10^{163}. In the context of metamorphic viruses, this is the reduction in the number of signatures that may be needed if all possible reordered variants of a virus appear in the wild.

In this paper we summarize our method for fixing order and the results from an empirical analysis of the method. Section 2 presents some transformations used by metamorphic viruses. Section 3 outlines how to undo the transformations by mapping a program into a zero form. It presents our algorithm for imposing order on the statements of a program. Section 4 summarizes the results of an empirical evaluation of our algorithm. Section 5 contains a comparison with related works. Section 6 presents our conclusions.

2. Morphing transformations

A metamorphic virus carries with itself a morpher: a subprogram that transforms the structure of the virus

```
mov eax, V_S - 1
add eax, ecx
xor edx, edx
div ecx
mul ecx
push eax
```

```
mov eax, V_S - 1
nop
add eax, ecx
xor edx, edx
add eax, 0
div ecx
nop
mul ecx
push eax
```

(a) Deadcode insertion

```
push edx
mov edx, ecx
mov ebx, 000Ah
add edx, ebx
pop ebx
```

```
push eax
mov eax, ebx
mov edx, 000Ah
add eax, edx
pop edx
```

(b) Variable Renaming

```
statement-1
statement-2
statement-3
statement-4
statement-5
statement-6
```

```
goto L1
L3:
statement-3
goto L4
L1:
statement-1
statement-2
goto L3
L5:
statement-5
goto L6
L4:
statement-4
goto L5
L6:
statement-6
```

(c) Break and Join Transformation

```
if (i < a * b * c)
{
    a = x * 100 + 2;
    b = 10 * i;
    c = y * a + b;
    i = a + b + c;
}
```

```
if (i < a * b * c)
{
    a = 2 + x * 100;
    b = 10 * i;
    c = a * y + b;
    i = b + c + a;
}
```

(d) Expression Reshaping

```
x = 25;
y = x + get_index();
z= 100;
while (i < y + z)
{
    b = 200 * i;
    c = y * i;
    a = x * y + i * z;
    i = i + 1;
}
```

```
x = 25;
y = x + get_index();
z= 100;
while (i < y + z)
{
    a = x * y + i * z;
    b = 200 * i;
    c = y * i;
    i = i + 1;
}
```

(d) Statement Reordering

Figure 1 Morphing Transformations

without affecting its behavior. The morphing transformations used by recent metamorphic viruses include dead or irrelevant code insertion; register renaming; expression reshaping; break & join transformations; and statement reordering. Figure 1 enumerates these transformations using examples.

Inserting dead code (irrelevant/junk code) in a program has no effect on the results of the program. It is very effective in changing the program text. Dead code may be inserted by simply inserting the NOP instruction. There are other, more complex forms of dead code as well, such as inserting a PUSH instruction and a POP instruction at non-consecutive locations.

The register renaming transformation changes the register used in a computation. In order to preserve the behavior of the original virus, it is necessary that all related occurrence of the register also be renamed.

Break & join transformations break programs into pieces, select a random order of these pieces, and use unconditional branch statements to connect these pieces such that the statements are executed in the same sequence as in the original programs.

Expression reshaping involves generating random permutations of operands in expressions with commutative and associative operators. This changes the structure of expressions.

The statement reordering transformation reorders the statements in a program such that the behavior of the program does not change. It is possible to reorder statements if and only if there are no dependences [7] between the statements being reordered. If the virus signature includes bytes corresponding to a statement from this set of reorderable statements, then application of statement reordering transformation makes the original virus signature useless for as many different variants as possible.

(a) Fixing the statement order – Variant A

(b) Fixing the statement order – Variant B
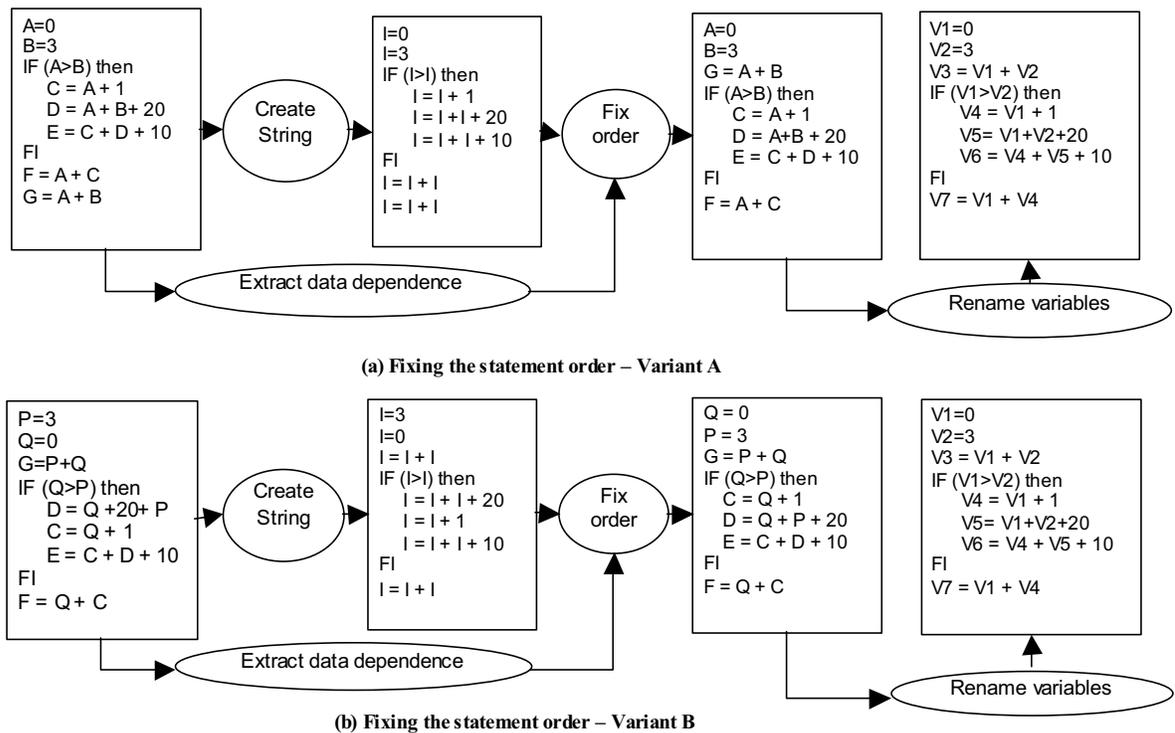
Figure 2 Transforming program variants to zero form

3. "Zeroing" transformation

The effect of a morphing transformation, described above, could be removed by performing its inverse transformation, if one exists. For instance, dead-code elimination, constant propagation [1], removal of redundant computations, and elimination of spurious unconditional branch statements may be used to undo the effects of (some types of) dead code or irrelevant code insertion, expression reshaping, and break & join transformations, respectively. The final outcome may depend on the specific order in which these transformations are applied. It is an open research question on how to choose an order for applying these transformations such that it yields the same canonical form for different variants.

The focus of our current investigation is on how to undo the effects of statement reordering, reshaping expressions, and variable renaming transformations. The inverse of these transformations have not been studied in the literature. We call our transformation the "zeroing" transformation, for its attempt to eliminate the effect of reordering, variable renaming, and expression reshaping. The program resulting from applying the zeroing transformation is called the zero form.

We present the zeroing transformation top-down; presenting the high level steps first, followed by a description of the lower level steps.

The zeroing transformation has the following steps:

1. Create a Program Tree (PT) representation of the program.
2. Partition the PT nodes into reorderable sets, each set containing statements that may be mutually reordered without affecting the program's semantics.
3. Partition each reorderable set into a sequence of isomorphic sets, where every statement in an isomorphic set has the same 'string representation.' The representation does not depend on names of the variables in the program, order of variable in commutative operators, and order of the statements in the program,
4. Assign a number to each statement. The numbering is done using a depth-first traversal of the PT. Statements in a reorderable set are visited based on the order in the sequence of isomorphic set. Statements in an isomorphic set are visited in random order.
5. Create a new program by ordering the statements as per the numbers assigned in the last step. In each expression, replace each variable name by a new
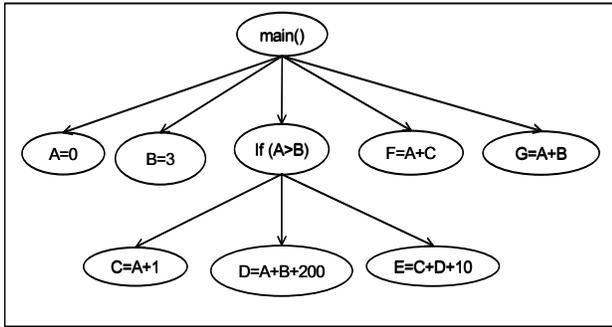
Figure 3 Program Tree for Example 1 (of Figure 2)

(a) Program   (b) CDG

Figure 4 Example 2

Figure 5 Program Tree for Example 2

variable name created using the number of the statement where it is first defined.

The following subsections describe the first three steps. The last two steps are self-explanatory, and are not elaborated any further.

3.1. Program tree (PT) representation

A program tree (PT) is a hierarchical ordering of statements in a program. While the abstract syntax tree (AST) of a program could serve as a PT, we do not use AST because its structure depends on the specific order of statements in the program. We use the control dependence sub-graph (CDG) [7] for constructing the PT. Construction of PT for structured programs, i.e., goto free programs, is straightforward. The nodes in the program tree are the statements of the program. We create an edge in the program tree from a node n_1 to node n_2 if there exists an edge from n_1 to n_2 in the corresponding CDG.

The PT of the code segment in Figure 2 is shown in Figure 3. For readability, we show the corresponding program statements for control predicates in the program tree. For example, in Figure 3, the control predicate *(A>B)* is shown as *If (A>B)*. The node *If (A>B)* in Figure 3 has an edge to the nodes *C=A+1, D=A+B+200, E=C+D+10* because of the control dependence relationship between the node *If (A>B)* and the nodes *C=A+1, D=A+B+200, E=C+D+10*.

In general, the control dependence graph may not be a tree. A CDG node may have multiple predecessors and the graph may have cycles. Consider the program in Figure 4(a). Its control dependence graph, shown in Figure 4(b), has a cycle. We create a PT by traversing a CDG in depth first order and terminating the traversal when a node in the ancestor list is visited again. To indicate the repetition we include a copy of that node in the tree. Figure 5 shows the PT for the CDG of Figure 4(b). Similarly, if a node in the CDG has multiple parents, we create a duplicate child node for each parent node.
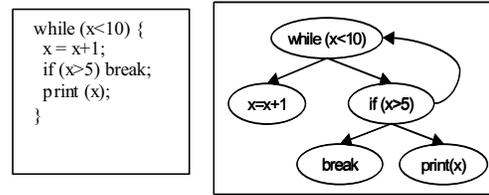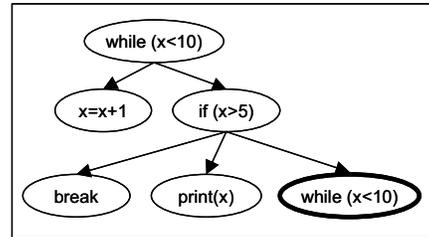
A program tree is constructed for each procedure of the program.

3.2. Partitioning PT into reorderable sets

This section describes an algorithm for partitioning PT nodes into reorderable sets. The aim of this step is to find statements that may be mutually reordered without changing the semantics of the program.

Definition: Reorderable set. A set of nodes in PT is reorderable if its nodes can be mutually reordered (swapped) without changing the semantics of the program represented by PT.

For a set of nodes to be reordered they must be siblings in the PT. A pair of sibling PT nodes can be reordered if one does not depend on the other, as per the following definition.

Definition: Tree dependence. Let n_1, $n_2 \in$ PT be sibling nodes. Node n_2 is tree-dependent on node n_1 if and only if there exists a path from n_1 to n_2 in the control flow graph of P and (1) a node in the sub-tree of PT rooted at n_2 is data dependent on a node in the sub-tree of PT rooted at n_1 or (2) there exist two nodes $n_{1'}$ and $n_{2'}$ in the sub-trees of PT rooted at n_1 and n_2, respectively, such that the intersection of the set of variables defined in $n_{1'}$ and $n_{2'}$ is non-empty.

The data-dependence relation in Condition (1), above, is as traditionally used in compilers, and in the construction of program dependence graph (PDG). This condition propagates to the parents the data dependence relation between its (transitive, reflexive) children. Condition (2) relates statements that may become data-dependent if they were swapped.
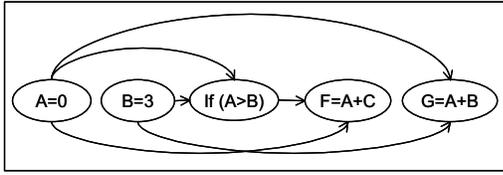
Figure 6 Tree-dependence for the children of "root" of Example 1

The children of each node in the PT are partitioned into a sequence of reorderable sets. The order between the reorderable sets is fixed, but the order of the elements within a reorderable set is not fixed. Given a program (P) and its corresponding PT, the PT nodes can be partitioned into reorderable sets using the partitioning algorithm in Figure 7.

The partitioning algorithm traverses the PT of P in the post order to find the nodes that can be reordered. It assigns a DG-Depth (dependence graph depth) value to each node. The DG-Depth is also the number of the reorderable set in which that node is placed. The DG-Depth of all nodes is initially set to '0'. The children of a node are processed in the control flow order. For every child node c of n, the partitioning algorithm finds the node c', having the maximum DG-Depth m, such that either c depends on c' or c' depends on c. The DG-Depth of c is set to $m+1$ implying that the child node c is placed in the reorderable set number $m+1$. The use of dependence graph depth for identifying order between statements assures that after the nodes are partitioned, the dependence order of the children is preserved.

Consider the Node *main ()* in Figure 2. Its root has the following children {*A=0, B=3, If (A>B), F=A+C, G=A+B*}. The dependence graph for this set of children is shown in Figure 6. The edge from the node *A=0* to the node *If (A>B)* corresponds to the dependence relation from node *A=0* to the node *If (A>B)*; i.e., there exists a path from the node *A=0* to the node *If (A>B)* in the control flow graph of P and a value defined by the node *A=0* is used by at least one node in the sub-tree starting with the node *If (A>B)*. Similarly, the edge from the node *If (A>B)* to the node *F = A+C* corresponds to a tree dependence relation from *If (A>B)* to *F=A+C*.

Figure 8 illustrates the working of the partitioning algorithm. The child nodes are processed in the control flow order. At each step, a node is picked and its DG-Depth computed. Initially, all the child nodes are given a DG-Depth of '0'. Figure 8(a) shows the processing of the child node *A=0*. The incoming/outgoing edges of the node *A=0* are directed from/to the nodes having a DG-

```
Partition-PT ( n : PT-Node ) {
    for each child c of n do
        Partition-PT ( c );
    for each child c of n
      (in control-flow order) do {
        m = 0;
        for each c' such that
                c tree-depends on c'
                or c' tree-depends on c do
            m = max ( DG-Depth(c), m);
    }
    // c is in reorderable set m+1
    DG-Depth (c) = m+1;
}
```

Figure 7 Algorithm to Partition PT Siblings

Depths as '0'. Hence, the node *A=0* is placed in the reorderable set numbered '1'. The DG-Depth of the node *A=0* is set to '1'. In Figure 8(d), the algorithm processes the child node *F=A+C*. The DG-Depths of the nodes that have a dependence relation with *F=A+C* are {2, 1}. The maximum value in this set is 2. Hence *F=A+C* is placed in the reorderable set numbered 3. The DG-Depth of the node *F=A+C* is set to '3'.

After partitioning, the reorderable sets will be as follows.
- Reorderable set–1: {*A=0, B=3*};
- Reorderable set–2: {*If (A>B), G=A+B*};
- Reorderable set–3: {*F=A+C*}.

Figure 8(f) shows the partitioned dependence graph.

We apply one more transformation to undo the effect of splitting a single assignment expression into multiple assignments. This is done by identifying chains in the tree-dependence graph of siblings and abstracting the chain as a single expression. A dependence chain is defined as follows.

Definition: *Dependence Chain.* A sequence of sibling nodes n_1 to n_k, $k > 0$, form a dependence chain iff (a) for all n_i, $1 \le i < k$ there is at most one node, n_{i+1}, tree-dependent on n_i, and (b) for all n_j, $1 < j \le k$, n_j is tree-dependent on at most one node, i.e., n_{j-1}.

Figure 9(b) shows the dependence chains formed for the dependence graph of Figure 9(a).

Definition of tree-dependence is extended to include dependence between chains, as follows.

Definition: *Dependence between chains.* A dependence chain c_1 is dependent on a dependence chain c_2 if and only if a PT node in c_1 is tree-dependent on a PT node in c_2.
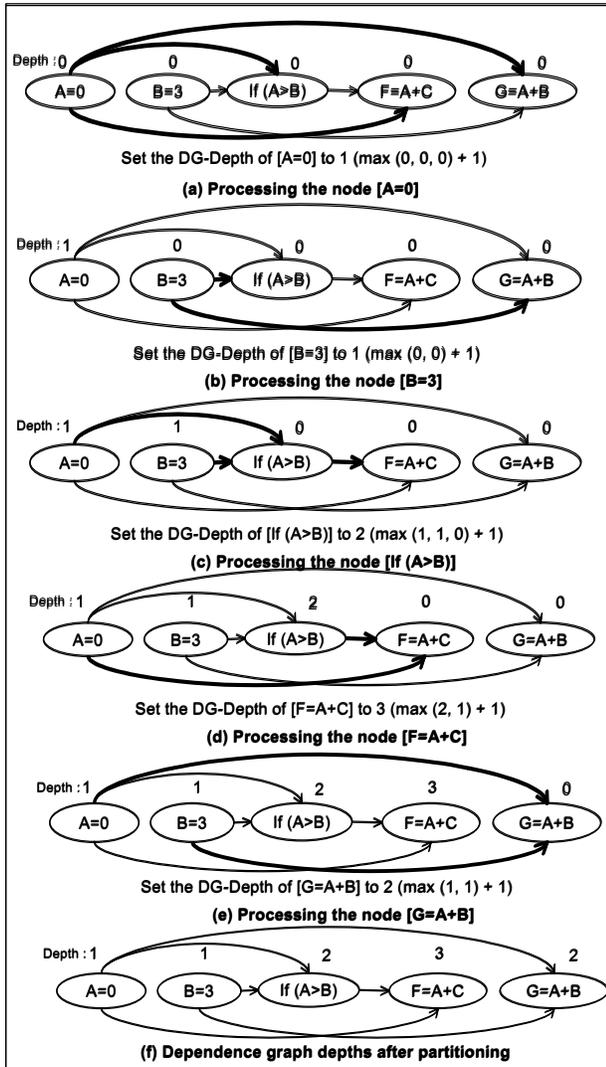
Figure 8 Working of Partitioning Algorithm

(a) Tree-Dependence Graph

(b) Dependence Chains

Figure 9 Examples of Dependence Chains

The partitioning algorithm treats a dependence chain like a PT node. The algorithm produces the same reorderable sets independent of statement reordering, expression reshaping, and variable renaming transformations. The next section describes a strategy to order statements in a reorderable set. The use of dependence chains instead of individual program statements improves the probability of imposing an order using the string representation of the dependence chains.

3.3. Partitioning reorderable sets

The statements in a reorderable set are partitioned into a sequence of isomorphic sets by associating a special string representation to each statement, sorting the statements based on the string representation, and placing the statements with identical string representation in the corresponding isomorphic set. We have experimented with six string representations, referred to as SR1 to SR6. String representation SRi is used to partition statements that could not be partitioned using SRi-1. Thus, the six string representations act as a succession of filters.

The string representations need not preserve the semantics of the original statement since they are used only for ordering statements. The string representations are designed so that they do not depend on the names of the variable in the program, or the order of statements in the program, or order of expressions in commutative operators.

Figure 11 shows an example of SR1 representations of two expressions. Figure 10 gives the algorithm for computing this SR1 form. The result of the algorithm is placed in the property *String* of the expression node. In the SR1 form every identifier is replaced by the same symbol, "I". The string representation of an operator is created by concatenating the string representations of its operands. To counter expression reshaping, the string representations of the operands of a commutative operator are sorted before concatenation. The algorithm assumes that nested sequences of commutative binary operators are represented using a single *n*-ary operator. In addition, other transformations may also be pre-applied on the AST, such as representing the subtraction operator as addition of a negative number.

The string representation of expressions in a PT is used to create the string representation of a PT (and its subtrees). Figure 12 presents the algorithm to compute

```
Stringify-Expression (e : Expression )
{
  if (e is a variable) e.String = "I";
  elseif (e is a constant)
      e.String = string rep of e;
  else {
    // e is an operator
    S = string rep of root operator of e;
    for each child c of e do
        Stringify-Expression (c);
    if (e represents a commutative operator)
        L = sorted list of string reps of children of e;
    else
        L = list of string reps of children of e;
    fi;
    S' = concatenate strings in L;
    e.String = concatenate S and S';
  }
```

Figure 10 Creating SR1 Strings of Expressions

Figure 11 SR1 Form of Expressions

string form of a PT node, and its children. The reorderable sets are processed in their sequence order.

The string representations of nodes are used to partition reorderable sets by sorting the nodes on their string representation. The statements with the same string representation are grouped into isomorphic sets. The isomorphic sets are sorted using the string representation of its member elements. For instance, the statements *A=B+20* and *C=20+D* have the same SR1s. If these statements are in the same reorderable set then they will also be placed in the same isomorphic set.

Statements that cannot be differentiated into singleton sets by SR*i* are attempted to partition using SR*i*+1 through SR6, until one differentiates them. These representations are summarized below.

- **SR1 representation:** Replace each variable in the statement by the symbol 'I', convert commutative (binary) operators to N-ary operators, and sort the operands of the N-ary commutative operators using the SR1 representation for the operands.
- **SR2 representation:** For each variable used in an expression count the number of reaching definitions for that expression. For each variable defined in the expression, count the number of statements using that definition. Create a string using the counts of the number of uses and definitions. Figure 13 contains an example of SR2 form. The string U{1,3} implies that of the two variables used in expression "A=C+D", one variable has '1' reaching definition and the other variable has '3' reaching definitions. The string "D{2}" in the SR2 form implies that the variable defined in this expression is used in two statements.
- **SR3 representation:** Concatenate the sorted sequence of strings representing the SR1
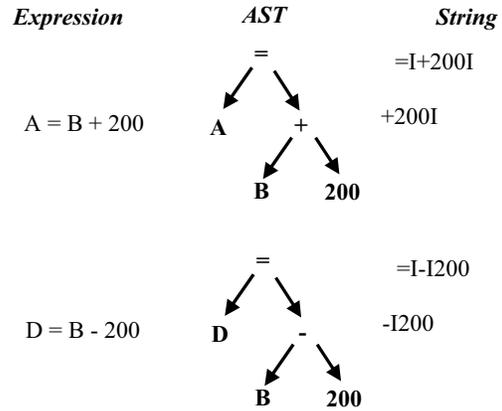
representations of all the data dependence predecessors of the statement.
- **SR4 representation:** Concatenate the sorted sequence of strings representing the SR1 representations of all the data dependence successors of the statement.
- **SR5 representation:** Replace each function call in the statement by the SR1 representation of the body of that function.
- **SR6 representation:** Concatenate the sorted sequence of strings representing the SR1 representations of all the statements in forward and backward slices of that statement.

Reorderable statements that yield the same string representation for SR1 through SR6 representations are placed in the same isomorphic set.

Let us return to Figure 2. All statements in the two programs in this example can be completely ordered using the SR1 representation. The box after 'Create String' gives the SR1 representation of the programs, with a caveat. The strings for the children have not been sorted. The box after 'Fix Order' shows the resulting order of statements. The result from renaming the variables is shown in the last step.

More details on the zeroing transformation may be found in [13].

4. Empirical analysis

We have developed a tool, C⊕, that implements the proposed algorithm for imposing order on the statements of C programs. It does not implement transformations to undo expression reshaping. C⊕ uses the Program Dependence Graph (PDG) [7], generated by CodeSurfer™ [8] of GrammaTech, to gather the control

```
Stringify-PT-Node ( n: PT-Node)
{
    if (n is a leaf node) {
        Stringify-Expression (expression of n);
        n.String = n.String;
    } else {
        Stringify-Expression (expression of n);
        S = (expression of n).String;
        for each reorderable set r of n do {
            for each child c in r do
                Stringify-PT-Node-SR1 (c);
            R = sorted list of SRs of nodes in r;
            r.String = concatenation of strings in R;
        }
        L = sorted list of SRs of reorderable sets of n;
        S' = concatenation of strings in L;
        n.String = concatenate S and S'
    }
}
```

Figure 12 Creating String Form of PT-Nodes

and data dependences needed to identify reorderable statements.

We analyzed a set of real-world C programs to study how well the proposed approach imposed order on their statements. Our test systems represent a 'best case' scenario. If the results for the best case are not satisfactory, then one cannot expect any significant returns from using the method for metamorphic viruses whose code is inherently obfuscated.

The test systems used in the experiments are described below:

• **Bison**: The Bison program is a LALR parser generator. It consists of 33 C files generating 10,718 nodes in the program dependence graph.

• **Cook**: The COOK system [6] is used for writing build scripts for projects. It is a powerful and easy to use replacement for make [14]. It consists of 39 files generating 9,147 nodes in the PDG.

• **SNNS**: The SNNS system is a neural network simulator for Unix® workstations. It consists of 11 files generating 8,835 nodes in the PDG.

• **Fractal**: The Fractal programs [15] create fractals using C programming language. It contains 36 files generating 8,856 nodes in the PDG.

• **Computer vision**: The Computer Vision programs is a collection of programs illustrating computer vision. It consists of 21 files generating 13,159 nodes in the PDG.

For each of these systems we measure the following:

1. *Reorderable Percentages*: The percentage of statements that can be reordered.
2. *Total Number of Permutations*: Product of the number of permutations of each reorderable set.

The two measures are computed for the original program, referred to as SR0 representation, and the

Expression	SR2 Form
A = C + D ⟶	U{1,3}:D{2}
B = D + K ⟶	U{2,3}:D{2}

Figure 13 SR2 Form of Expressions
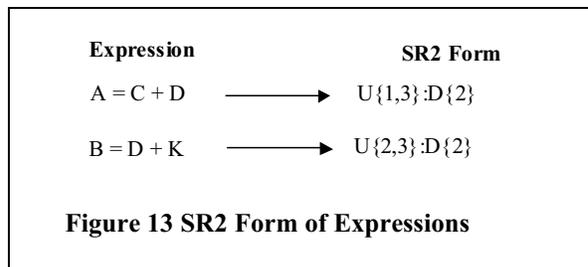
program resulting after fixing statement order using the filters SR1 through SR6, but without the transformation for expression reshaping.

Figure 14 shows the reorderable percentages for our test programs. On an average 55% of the statements of the original program are reorderable. The number of reorderable statements is reduced to 6% after transformation. The Fractal programs have higher reorderable percentages than other systems. This is because the Fractal programs have computer graphics code has many code fragments with similar statement structures, but which compute on different coordinate axes.

Table 1 shows the total number of possible permutations for the test programs before and after applying zeroing transformations. The rows P1 to P6 are number of possible permutations that can be created by reordering statements whose order is not fixed by SR1 to SR6, respectively. P0 is the permutations for the original program.

The SR1 filter significantly reduces the number of possible permutations for each program. For Bison the reduction was from 10^{61} possible permutations to 10^{17}. The number of variants of the Cook system was reduced from 10^{47} to 10^{21}; for SNNS a reduction from 10^{184} to 10^{29}; for Fractal a reduction from 10^{88} to 10^{21}; and for Computer Vision a reduction from 10^{113} to 10^{7}. The sample programs exhibit significant variation on the effect of the successive filters SR2 to SR6. The SR2, SR4, and SR6 filters produce at least an order of magnitude reduction for Computer Vision and Fractal, whereas SR2 and SR6 produced similar reduction for SNNS. None of the other filters made any significant difference to Bison and Cook.

5. Related work

Besides making it difficult to detect a virus using signature-based analysis [3, 10], metamorphic transformations also obfuscate the code to make it difficult for an anti-virus analyst to analyze the program. Such obfuscations have legitimate uses too. Programs may be obfuscated to protect intellectual property, and to increase security by making it difficult for others to identify vulnerabilities [5, 11, 19]. The art of obfuscation is quite advanced. Collberg et al. [5] present a taxonomy
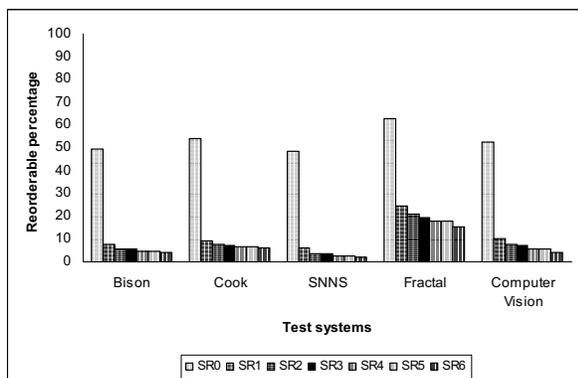
Figure 14 Reorderable percentages for test systems

Table 1 Number of permutations for test systems

	Bison	Cook	SNNS	Fractal	C/Vision
P0	1.8E61	2.8E47	1.4E184	9.1E88	1.3E113
P1	8.5E17	9.1E21	2.7E29	3.0E21	5.9E7
P2	8.5E17	9.1E21	4.9E9	3.0E21	15,156
P3	8.5E17	4.6E21	4.9E9	5.9E16	15,155
P4	8.5E17	4.6E21	3.0E9	1.8E9	1,591
P5	8.5E17	4.6E21	3.0E9	1.8E9	1,104
P6	8.5E17	4.6E21	1.3E6	6.6E8	203

of obfuscating transformations and a detailed theoretical description of such transformations.

The Bloodhound technology [16] of Symantec Inc. uses static and dynamic heuristic scanners. The static heuristic scanner maintains a signature database. The signatures are associated with program code representing different functional behaviors. The dynamic heuristic scanner uses CPU emulation to gather information about the interrupt calls the program is making. Based on this information it can identify the functional behavior of the program. Once different functional behaviors are identified, they are fed to an expert system, which judges whether the program is malicious or not. Static heuristics fail to detect morphed variants of the viruses because such variants may have different signatures. Dynamic heuristics consider only one possible execution of a program. A virus can avoid being detected by a dynamic scanner by introducing arbitrary loops.

Lo et al.'s MCF [12] uses program slicing and flow analysis for detecting computer viruses, worms, Trojan-horses, and time/logic bombs. MCF identifies telltale signs that differentiate between malicious and benign programs. MCF slices a program with respect to these telltale signs to get a smaller program segment representing the malicious behavior. This smaller program segment is manually analyzed for the existence of virus behavior.

Szappanos [17] uses code normalization techniques to remove junk code & white spaces, and comments in macro viruses before they generate virus signature. To deal with variable renaming, Szappanos suggests two methods: first, renaming variables using the order in which they appear in the program and second, renaming all the variables in a program with a same name. Former approach fails if the virus reorders its statements, and the later approach abstracts a lot of information and may lead to incorrect results. As our approach fixes the order of the statements in a program, the first approach suggested by

Szappanos for renaming the variable can be used in combination with our method.

Christodorescu et al. [3] have developed a method for detecting patterns of malicious code in executables. They use abstraction patterns—patterns representing sequences of instructions. These patterns are parameterized to match different instructions sequences with the same instruction set but different operands. Their approach gives fewer false positives but the cost of creating and matching the abstraction patterns is high. They detect the virus variants created by performing dead code insertion, variable renaming in the absence of statement reordering, and break & join transformations. Our method, in addition to the above morphing transformations, can be used to detect the viruses that apply statement reordering and expression reshaping transformations.

The problem of detecting malicious code is related to that of detecting software clones. Current anti-virus technologies in essence check whether a given executable is a clone of a known, malicious executable. The primary difference is that an anti-virus software, running on a user's desktop, must make the determination without having access to a complete copy of the original, or else the anti-virus software itself may be distributing viruses. Our algorithm has some similarities to Komondoor and Horwitz's PDG based algorithm for detecting software clone [9]. While we create string representations using the data-dependence relations of a statement, Komondoor and Horwitz match theses statements to determine clones. They too compare statements by factoring out differences in the names of variables.

6. Conclusions

We are investigating methods for transforming a program into a canonical form. Such a method may be used to map different variants of a metamorphic virus to the same canonical form, thereby making it easier to detect such viruses. With this goal, we have developed transformations to undo the effect of statement reordering, expression reshaping, and variable renaming transformations. We call the resulting form a zero form. As our method is a heuristic, we may not always map all the variants to a single zero form. But the application of

zeroing transformations results in a significant decrease in the number of possible variants of a program. Our initial experiments show that on an average 55% of the statements of a program statements are reorderable. After imposing order on these statements using our zeroing transformation only 6% of the statements remained reorderable, that is, could not be ordered using our method. This is a significant reduction in the number of variants that can be created by statement reordering. To utilize the method in AV technologies, the method needs to be adapted for binary programs.

7. References

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

[2] V. Bontchev, "Macro and Script Virus Polymorphism," in *Proceedings of the 12th International Virus Bulletin Conference*, 2002.

[3] M. Christodrescu and S. Jha, "Static Analysis of Executables to Detect Malicious Patterns," in *The 12th USENIX Security Symposium (Security '03)*, Washington DC, USA, 2003.

[4] F. Cohen, "Computer Viruses-Theory and Experiments," *Computers and Security*, 6, 1984

[5] C. Collberg and C. Thomborson, "Watermarking, Tamper-Proofing, and Obfuscation - Tools for Software Protection," *IEEE Transactions on Software Engineering*, vol. 28, pp. 735-746, 2002.

[6] C. G. Davis, "Debian Cook Package," http://packages.debian.org/stable/devel/cook.html, Last accessed 08/29/2003.

[7] J. Ferrante, K. J. Ottenstein, and J. Warren, "The Program Dependence Graphs and Its Use in Optimization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, pp. 319-349, 1987.

[8] GrammaTech, "Codesurfer - Program Analysis Tool," http://www.codesurfer.com, Last accessed 08/29/2003.

[9] R. Komondoor and S. Horwitz, "Using Slicing to Identify Duplication in Source Code," in *Proceedings of the 8th International Symposium on Static Analysis*, Paris, France, 2001.

[10] A. Lakhotia and P. K. Singh, "Challenges in Getting Formal with Viruses," *Virus Bulletin*, 2003, http://www.virusbtn.com/magazine/archives/200309/formal.xml.

[11] C. Linn and S. Debray, "Obfuscation of Executable Code to Improve Resistance to Static Disassembly," in *Proceedings of the 10th ACM Conference on Computer and Communication Security 2003*, Washington D.C., USA, 2003.

[12] R. W. Lo, K. N. Levitt, and R. A. Olsson, "MCF: A Malicious Code Filter," *Computers & Security*, vol. 14, pp. 541-566, 1995.

[13] M. Mohammed, *Zeroing in on Metamorpic Computer Viruses*, Center for Advanced Computer Studies, University of Louisiana at Lafayette, M.S. Thesis, 2003.

[14] R. M. Stallman, R. McGrath, and P. Smith, "GNU Make, a Program for Directing Recompilation," 2002.

[15] R. T. Stevens, *Fractal Programming in C*: John Wiley and Sons, 1989.

[16] Symantec, "Understanding Heuristics: Symantec's Bloodhound Technology," http://www.symantec.com/avcenter/reference/heuristc.pdf, Last accessed July 1, 2004.

[17] G. Szappanos, "Are There Any Polymorphic Macro Viruses at All? (… and What to Do with Them)," in *Proceedings of the 12th International Virus Bulletin Conference*, 2002.

[18] P. Ször and P. Ferrie, "Hunting for Metamorphic," in *Proceedings of the 11th International Virus Bulletin Conference*, 2001.

[19] G. Wroblewski, *General Method of Program Code Obfuscation*, Institute of Engineering Cybernetics, Wroclaw University of Technology, PhD. Thesis, 2002.