Static Analysis of Binary Executables

Steve Hanov
University of Waterloo
200 University Avenue West
Waterloo, Ontario, Canada N2L 3G1
steve.hanov@gmail.com

ABSTRACT

This paper is a survey of the use of static program analysis techniques on binary executables. Static analysis techniques are often used on a program's source code, which is usually a high level language. It is possible to apply them directly on the machine code of a compiled program. One of the challenges is building up a control flow graph of a procedure, since indirect branch instructions accept the contents of a register for the destination address. Program slicing techniques can be used to reduce the assembly code to the smallest possible program to compute the value of that register, and determine the range of values in the register.

Another problem is disassembly itself. On architectures with instructions of varying size, it is difficult to locate the start of the first machine code instruction in a section consisting of both code and data. Also, malicious code could take advantage of the difficulties in disassembly to hide its existence. Various static analysis techniques have been developed to analyze such programs, in order to build up a control flow graph and a call graph. Finally, type-state techniques have been developed to verify that machine code conforms to its interface, and does not alter areas of memory which it should not.

1. INTRODUCTION

A competent programmer can examine a program's source code and get a pretty good idea about what it does. Granted, he is aided by meaningful variable and procedure names. Even if these are stripped away, it is possible to determine facts about a program. Countless times in all parts of the world, some programmer has painstakingly checked over a program to find locking errors or memory that has not been freed, without first having to run the program in his head. It is logical, then that a computer can follow the same processes and make the same deductions about program behaviour, more accurately, and also without running the program. This is the realm of static program analysis.

Many static program analysis techniques are reduced to a data flow analysis problem (DFLAP) [8]. With a DFLAP, the program's data is modeled as mathematical set in some way, for example, a mapping from variable names to values. There is an instance of this set at each program point, summarizing all the information that can be deduced. The program's instructions are reduced to functions that operate on the set (for example, adding, modifying, or removing values). During the analysis, the functions are executed one at a time, each one feeding its output into the next. A program may contain loops, so the process is repeated until nothing changes. When that happens, we are left with a *fixed point*. Depending on the functions used, we can deduce information about the program by looking at the resulting sets at each program point. The seminal example of this process is constant propagation, where we can prove that a variable always has the same value at a particular point, and thus eliminate it from the program entirely.

The usefulness of static program analysis to program optimization is easy to see. It can be used to eliminate array bounds checks, for example. Outside of program optimization, it can be used to modify the source code to make it safer – adding type constraints to Java classes [13]. Static analysis techniques have been applied to the problem of program verification [6], and to detect buffer overflows in C [5]. However, all of these static analysis techniques operate on a high level language.

Computers do not directly execute code written in a high level language. Compilers transform the code into assembly language instructions, and then finally into the machine code compatible with the target processor. Since the machine code is being executed, there is value in applying static analysis techniques to a binary executable.

The analysis of binary executables is closely related to research into reverse engineering. A goal of reverse engineering is decompiling, or reconstructing the source text from the compiled machine code. This usually involves building a control flow graph for each procedure, and looking for idioms in the code. An idiom is a sequence of instructions that, individually, do not have meaning, but when taken together form a larger operation.

In this paper, we will examine how static analysis techniques have been applied to binary executables. We will look at what problems static analysis can solve, and also the chal-

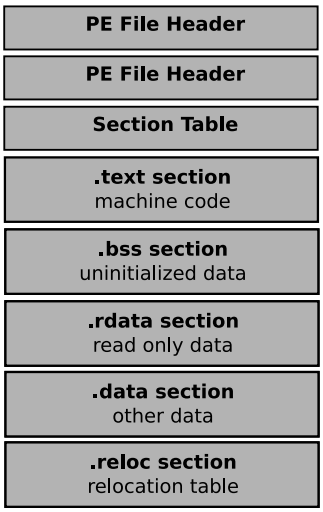

Figure 1: The PE format, used by Microsoft Windows executables.

lenges, and what static analysis techniques cannot solve.

2. DISASSEMBLY OF BINARY EXECUTABLES

An example of a binary executable file format is shown in Figure 1. In modern executable formats, the binary is split into several sections, which neatly separate the code from the data. However, it is not necessary to make this separation. As long as it is never accidentally executed, the data make be placed at any location within the .text section.

Such neat separation in program executables is not always the case. The *.com* file format, used in MS-DOS, consists of data and executable instructions intermixed. The file is loaded at a fixed address, known in advance. Execution begins at the first byte in the file.

The first step in the static analysis must the be conversion of the machine code into assembly language instructions. However, this task itself is challenging, and has been the subject of much research. With Intel instructions [1], the opcode of the instruction can be between 1 and 3 bytes, and this can be followed by 1 to 8 bytes of immediate data and a scaling/offset byte (See Figure 2. The instructions are not required to follow any alignment rules in memory.

Because it is possible for control flow to jump around the program, it is difficult for a disassembler to find the alignment that yields the correct instructions. The disassembler must follow the path of execution in the same way as the processor, in order to skip around any data bytes in between the instructions. This may be as simple as performing a linear scan through the instructions, or as complex as interpreting the code to follow its path of execution.

3. PROGRAM SLICING

One of the earliest applications of static analysis techniques to binary executables is due to Cifuentes [4]. Cifuentes was researching into the decompilation of programs and needed

```
1: read(x,y);
2: total = 0;
3: sum = 0;
4: if (x <= 1) {
5:     sum = y;
} else {
6:     read(x);
7:     total = x * y;
}
8: print total, sum;
```

Figure 3: : Original Program, adapted from Weiser

to build the control flow graph. In order to determine the destination of an indirect branch instruction (jumping to the address stored in a register) she program slicing techniques. We first follow the development of program slicing techniques, and then show how it is applied to binary executables.

Program slicing was first developed in 1981 by Weiser [14], who noticed that when trying to understand programs, experienced programmers would create what he called a *slice* in their conceptual model of how the program worked. The program slice is an executable program, but with certain statements deleted. The resulting program contains only statements of interest to the problem at hand.

Slice on the value of *total* at the end of the program.

1: read(x,y);
2: total = 0;
4: if (x <= 1) {
} else {
7: total = x * y;
}

Weiser defines a slicing criterion $C(i, v)$. i is is the statement at which to observe, and v is the set of variable names to be observed. He defines a monotone data flow analysis framework:

$$R_{in}(n) = \begin{cases} R_{out}(n) - DEF(n) \cup REF(n) & if n \neq i \\ R_{out}(n) - DEF(n) \cup REF(n) \cup v & if n = i \end{cases}$$

$$R_{out}(n) = \bigcup_{m \in SUCC(n)} R_{in}(m)$$

where m, n are statements, $R_{in,out}$ are the sets in the framework, $DEF(n)$ is the set of variables defined at n, and $REF(n)$ is the set of variables referenced at statement n. The data flow step will remove all variables referenced or defined in a statement, unless a) they are referenced by the statement of interest, or b) a successor references them. When the analysis terminates, only variables which are referenced in the slicing criterion will remain in the sets. All other program statements may be removed, as long as care is taken not to violate the syntax of the language.

In the ensuing years, new ways of representing programs allowed Susan Horwitz [7] to extend Weiser's method interprocedurally. Horwitz describes two graphs: the Program Dependence Graph (PDG), which represents a single procedure, and the System Dependence Graph (SDG), which

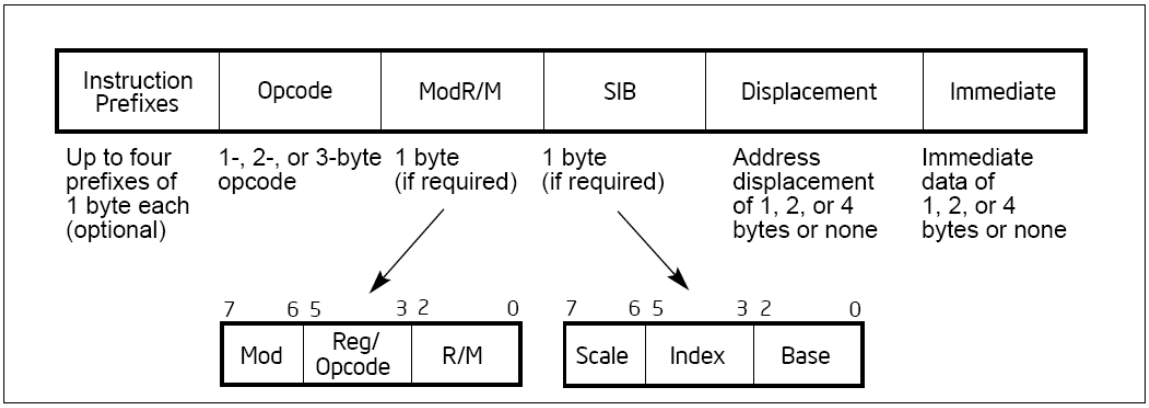

Figure 2: Intel Instruction format. Each opcode is of varying length.

connects procedures together.

To construct the program dependence graph,

- Create an entry node. For every variable v, create a node labeled $FinalUse(v)$.
- Create a node for each program statement.
- Connect the entry vertex to all other vertexes which are not in a loop or conditional. These edges are control dependence edges.
- Connect control flow statements (such as if constructs) to their immediately nested contents, using control dependence edges.
- Connect together all nodes v_1 and v_2 such that v_1 defines a variable x, and there is a path through the program such that v_2 uses that same definition of x. This edge is a data dependence edge.

In Figure 4, we have applied the above procedure to the program in Figure 3. The result is the program dependence graph. The intraprocedural slice can then be found with a very simple algorithm given by Horwitz:

```
procedure MarkVerticesOfSlice(G,S)
declare
    G: a program dependence graph
    S: a set of vertices in G
    WorkList:a set of vertices in G
    v,w: vertices in G
begin
    WorkList := S
    while WorkList $\neq$ {} do
        Select and remove vertex $v$ from WorkList
        Mark $v$
        for each unmarked vertex w such that
            there is an edge $w \to v$ in G, do
            Insert $w$ into WorkList
         od
     od
end
```

This procedure begins at the nodes of interest, and works backwards to find all possible paths to those nodes from the entry point. By performing this procedure on one of the $FinalUse(v)$ nodes, we can create a program slice for any variable we please.

Horwitz also goes into into detail extending the method interprocedurally using the System Dependence graph to create function summaries. However, this area has not been applied to binaries, we omit it from this survey.

Thus far, the slicing method does have a problem: It handles only structured programs. If a goto or jump statement is inserted, the results are incorrect. Agrawal [2] proposes a method of handling them.

1. First, he constructs the program slice using the conventional algorithm, which excludes jump statements.
2. Then, he determines which jump statements to add back in. The criteria for this is explained below.
3. Finally, if the slice contains a jump, and the destination isn't in the slice, the target is pushed forward to the next executable statement.

Agrawal's algorithm depends on the control flow graph for the procedure. From the control flow graph, he derives two other types of graphs: The post dominator tree, and the lexical successor tree.

Figure 5 shows an example program containing unstructured goto statements. Figure 6 is its control flow graph. From the control flow graph, the program dependence graph in Figure 7 is calculated using the algorithm given by Horwitz. By examining the program dependency graph, we can follow the edges up from the node write(y). We see that it consists only of the nodes start, y = ..., and write(y). This is clearly not an accurate slice.

To add the jump statements, we first require the post dominator tree and the lexical successor tree. According to [**?**], a node p post dominates node i if every possible execution path from i to exit includes p. In other words, if i executes,

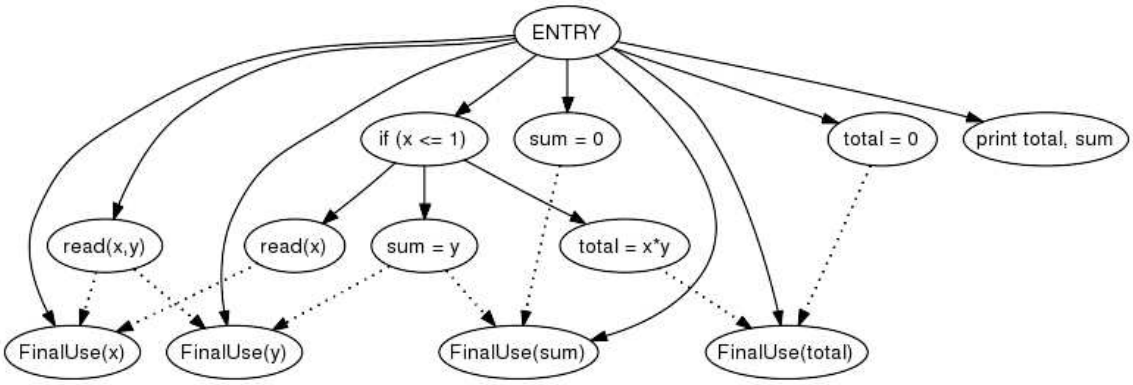

Figure 4: The program dependence graph, using Horwitz's algorithm on the example of Weiser in Figure 3.

```
1:      if ( C1 ) {
2:          goto L6;
3:          y = ...;
4:          goto L8;
        }
5:      z = ...;
6: L6:  x = ...;
7:      goto L3;
8: L8:  write(x);
9:      write(y);
10:     write(z);
```

Figure 5: from Agrawal. An unstructured program.

p must later execute. The tree is shown in figure 8. The lexical successor is simply the next statement in the program code, as written. In a compound statement, such as *if* or *while*, the successor is considered to be the statement after its body.

We traverse the post dominator tree using the preorder traversal, and for each jump statement encountered that is not already in the slice and whose nearest post dominator in slice is different from its lexical successor in the slice, we add it and the transitive closure of its dependencies.

In our example, we add the goto statements in line 2, 4, and 7, and the if statement in line 1 since line 2 is dependent upon it.

```
1:      if ( C1 ) {
2:          goto L6;
3:          y = ...;
4:          goto L8;
        }
7:      goto L3;
9:      write(y);
```

4. DISASSEMBLY USING STATIC SLICING

Cifuentes's 1997 paper [4] views the main problem of disassembly as the separation of instructions from data. The instructions can jump around the data, so it is not obvious to a disassembler which is which. A naive recursive dis-

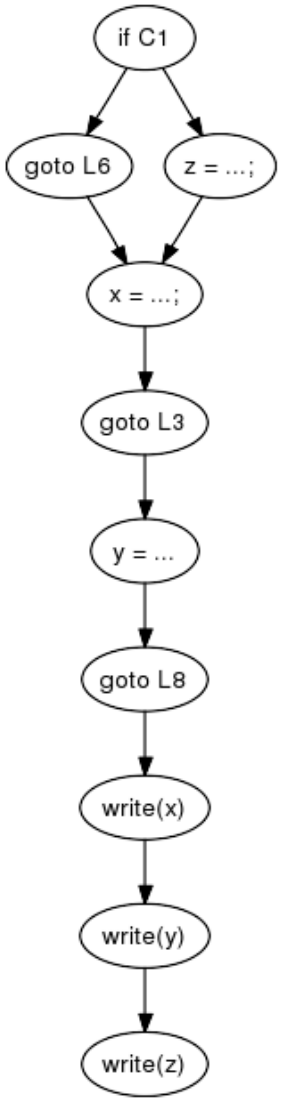

Figure 6: The control flow graph for the unstructured program in Figure 5.

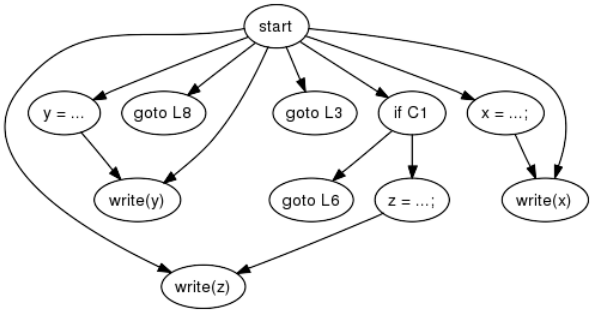

Figure 7: The program dependence graph for the program in Figure 5. This is the result of applying the algorithm of Horwitz to the example of Agrawal. For simplicity, the FinalUse() nodes have been omitted.

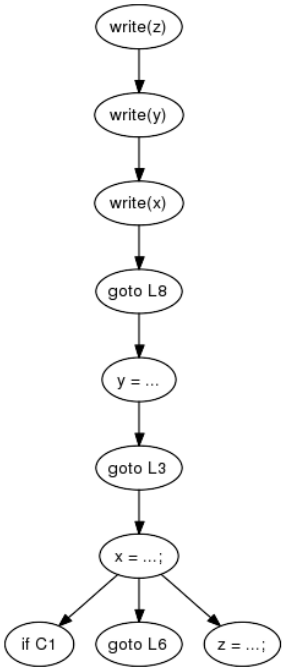

Figure 8: The post dominator tree for the program in figure 5.

assembler would not be able to work when it encountered indexed jump instructions, or an indirect procedure call to the contents of a register.

The conventional approach is to compare the code to that output by a number of different compilers, in the hopes that the indirect jump is part of some larger idiom, such as a switch statement. However, this requires keeping a database of compiler data, and programs written in assembler would not be deciphered.

During disassembly, one could perform a reaching definitions analysis on the assembly code to determine the destination of an indirect branch instruction. However, more sophistication is needed to obtain a complete result. Cifuentes suggests using a static slice of the program to determine the register contents.

Cifuentes limits herself to the intraprocedural case, so she does not use the interprocedural system dependence graph of Horwitz [7]. However, the handling of the jump instruction due to Agrawal [2] is essential to the analysis, because machine code is unstructured.

One of the challenges faced by Cifuentes, and other researchers such as [9], is the complexity of the CISC Intel instruction set. Cifuentes deals with this complexity in two ways. First, the scope of the research is limited to the Intel 286 instruction set, although Pentium was available at the time. Secondly, by changing the form of some of the instructions. The DIV instruction, for example, stores its result and remainder in two different registers. For this purpose, the researchers convert the statement into three separate statements which use a fictional *tmp* register for its intermediate result. Such tricks mean the algorithm only needs to 110 instructions instead of the full set of 250 Intel 80286 instructions Such tricks mean the algorithm only needs to 110 instructions instead of the full set of 250 Intel 80286 instructions..

Cifuentes *et al.* were able to apply the static slicing algorithm to short snippets of assembly code without difficulty. However, they ignore the problem of stack variables and aliasing entirely, stating that it is beyond the scope of the analysis since they are only concerned with the intraprocedural case.

It is possible to perform an interprocedural analysis as well. Most research is done in the context of deciphering obfuscated binaries. Therefore, we will first give an example of obfuscation techniques, followed by ways of countering them.

5. OBFUSCATION TECHNIQUES

In 2003, Linn and Debray [10] proposed a number of ways to improve the resistance of an executable to static disassembly. There are two types of disassemblers. A linear sweep disassembler begins at the program start address and disassembles each instruction encountered until it gets to the end address. The best example of a linear sweep disassembler is *objdump*, part of the GNU binutils package.

To thwart a linear sweep disassembler, one merely has to insert junk bytes in areas where they will never be executed,

but will be seen by the disassembler. For example, junk bytes may be inserted after an unconditional jump statement, or before jump targets.

Recursive disassemblers work differently. They begin at the program's entry point and disassemble instructions until they reach a branch instruction. A branch has two targets – if a condition is true, it will jump to another location. If the condition is false, execution will fall through. The disassembler will continue from both of these paths recursively, and thus eventually reach all possible statements.

To defeat a recursive disassembler, Linn and Debray propose two methods. Taking advantage of the fact that a recursive disassembler follows both branch targets, they modify the program to include branch instructions that are always true or always false. When executed, the program will always take the same path. However, a disassembler will follow both paths, including the incorrect one, and become confused. This technique is called *opaque predicates*. The suitability of opaque predicates is questioned by [9], due to the difficulty in creating ones complex enough to fool a smart disassembler.

A second technique for thwarting a recursive disassembler is the use of *Branch functions*. An unobfuscated program contains many call instructions to different procedures. In the obfuscation step, all of theses calls are replaced to a single branch function. The branch function takes the return address, and using a perfect hash function stored in the data section, determines the correct address that was intended to be called. In order to determine the call graph, a disassembler would first have to work out the inner workings of the branch function. As an additional obfuscation, the caller may pass a parameter to the branch function which is an offset. When the function returns, it will not return to the next instruction after the *call*, which is the usual case. Instead, it will first apply the offset and return to the new address.

In their paper, Linn and Debray present a tool that processes a binary executable and obfuscates it. It is meant to be used for protecting commercially deployed products, so it includes a profiler component so that obfuscations are inserted only in places where they would not severly affect performance. It is on this obfuscation tool that our next paper focuses its attack.

5.1 Disassembling Obfuscated Code

As the techniques of obfuscating code have advanced, so have the techniques of dissembling such code. In [9], Kruegel attempts to create a tool to combat the Linn and Debray's obfuscater. His disassembler make several assumptions. First, valid instructions must not overlap. It is extremely difficult to design a sequence of instructions that has meaning when executed at an offset to itself. The self-correcting property of Intel instructions works against this. Secondly, the algorithm assumes that opaque predicates do not exist. Although Linn and Debray point them out as a possible obfuscation, they do not actually use them. If they did, they would be easily detectable by the disassembler, Kruegel argues.

First, the start of each procedure in the binary must be found. To this end, the algorithm searches for function prologues (the first few instructions compilers used to prepare the stack upon entering a new procedure). This method is imprecise, but correct for a well-behaving compiler. The algorithm will also detect function entry points where none exist, because the data in the program or parts of other instructions may also look like a function prologue. These spurious parts will be discarded at a later step.

For the intraprocedural disassembly, Kruegel uses a statistical technique to build the control flow graph (CFG). Each procedure is analyzed as follows.

1. The procedure is disassembled using all possible alignments of instructions. At this point, certain alignments may already be discarded due to illegal instructions. Contiguous blocks of statements ending at a branching instruction become nodes in CFG. Note that the CFG may contain blocks that overlap in memory. For now, they are allowed.

2. For each branch statement, the source and destination nodes are connected. The destination node may be split, if the destination address is not at the start of the node.

3. At this point in the algorithm, the CFG is a superset of the real control flow graph of the procedure, because some of the blocks are overlapping. In the next step, Kruegel resolves these conflicts in a number of ways. He assumes that real basic blocks are more tightly integrated into the control flow than spurious ones. If two basic blocks conflict, then he chooses the one that is more tightly connected to others.

4. If there are remaining conflicting basic blocks, Kruegel randomly eliminates them and leaves other results up to future work.

At the end of the procedure, Kruegel's algorithm has disassembled at least the entire program, if indirect branch functions have not been used. Recall that Linn and Debray's branch function takes as input the return address with which it was called. It submits key to a perfect hash table, stored in the data section of the binary. The hash result is 1) the address of the function which to call, and 2) a new address to which to return (to confuse disassemblers that continue after the call instruction). Kruegel suggests interpreting the instructions of branch function in order to calculate the two results statically. This, of course, requires some knowledge of what the branch function does, and which function is the branch function. In a call graph, the identity of the branch function should be obvious – it will be the only function ever called. As for what it does, it is difficult to come up with a branch function that is complex enough to evade abstract interpretation, yet yields consistent results.

5.2 Bigram Analysis of Assembly Code

Instead of randomly choosing basic blocks to resolve conflicts, the authors in [9] suggest using another statistical technique: bigram analysis. Bigrams are widely used in natural language processing, for finding the best parse in

a probabilistic context free grammar [11]. In this analysis, for each set of two assembly language statements, s_1 and s_2, the probability that s_2 follows s_1 is stored in a table. This table can be derived by training the model – eg. count the frequency of co-occurring statement pairs in a large set of programs. After training, the table is normalized to obtain the probabilities. Then, the probability of an arbitrary sequence of instructions $s_1, s_2, s_3, \ldots, s_n$ occurring can be calculated as:

$$P(s_1, s_2) \times P(s_2, s_3) \times \cdots \times P(s_{n-1}, s_n)$$

Bigram analysis is applied to Java bytecode sequences in [12], a very short paper. The main result is that in the test benchmarks, usually only about 10% of the possible pairs of bytecode sequences are used. Because this number is so low, one can calculate with a high degree of accuracy the likelihood that a given sequence of bytes is a real Java program. If the result holds for Intel assembly language, this alone may help reveal correct alignments for disassembly. However, there has not been any work in this area. Also, an obfuscater that is aware of bigram analysis may attempt to further trick the disassembler by transform the instructions into an equivalent program with improbable statements.

In the next section, we examine such transformation techniques and how to detect them.

6. VIRUS DETECTION

Christodorescu's 2003 paper [3] applies static analysis techniques to the area of virus detection. Polymorphic viruses include self-modifying code, so that they are changed each time they spread themselves. They always have two parts – a section of encrypted code, and the decryptor. The encrypted part is easy to change by generating a new key. The decryptor has the task of modifying itself to that the new code is semantically equivalent to the old, yet any signatures that a virus scanner could look for would not match.

Christodorescu identifies several types of obfuscation techniques that are used by polymorphic viruses for the decrypting stub. The first, which worked for all commercial virus utilities at the time, was inserting nop instructions into the program text. Another technique is to rearrange the instructions. There may be two or more instructions whose order do not matter, so they can be safely transposed. Likewise, the virus may perform instruction substitution, so that one instruction is replaced by several that perform an equivalent function. A particularly insidious version is to weave the instructions into a host program, so that as the host is executing, it also executes the virus.

Christodorescu creates an analysis tool to detect the polymorphic techniques that he identified. The detector operates in three phases. First, the code is disassembled. Secondly, it is annotated, and thirdly, it is run through a detection module. The paper does not discuss disassembly, and presumes that the malicious programs have not made any attempt to confuse the disassembler.

To understand the goals of the method, it is best to first understand the detector portion. For each virus, Christodorescu manually builds a *malicious code automaton*. This automaton is a sequence of abstract statements, free of individual registers or machine code instructions. For example, a portion of the Chernobyl virus is reduced to:

```
Move(A,b)
Move(C,0d601h)
Pop(D)
Pop(B)
```

To detect the malicious sequence, the assembly code must be converted to a similar high level abstraction. Once the assembly code is in this form, all that remains to be done is to decide whether any path through the target program matches the path through the malicious code automaton. Christodorescu does this by viewing the control flow graph as a finite state machine (FSM) that produces a regular language. The FSM may include obfuscations that cause it to double back and execute other optional paths. However, the objective is to determine if the language produced by the two automatons share any common elements. If so, then at least one path through the program is the virus.

An additional complexity is that the detector must check for all permutations of the variables used. For this purpose, he borrows the concept of *unification* from the field of automated theorem proving. The Unify() function returns false if there is no binding of free variables that can be applied to the same two instruction sequences.

7. TYPE-STATE CHECKING OF MACHINE CODE

A data flow analysis can determine the range of values of a particular register at a program point, and so it can determine if, for example, a memory access is reading from the NULL pointer. However, it cannot answer questions such as, "is a particular variable always initialized before being read from?"

Type-state analysis is designed to this type of question. With a type-state analysis, one can assign rules to variables in a program. For example, in a higher level language, one can create a rule that lock() may never be called on a lock object that has already been locked. This may be verified at compile time. We will see how type-state analysis can be applied to machine code as well.

In Xu's PH.D thesis [15], he is concerned with verifying the safety of machine code. Most often, safety is determined through code signing. However, users often do not have the time or inclination to verify that they trust the signing authority. In addition, unsafe code may be signed. Xu's method is best suited for a binary plug-in distributed to a host program. The host program will be able to statically check for array bounds accesses, null pointer dereferences, and unaligned memory accesses before executing untrusted code.

The analysis operates in 5 phases: Preparation, Type-state propagation, annotation, local verification, and global verification. Xu assumes that the host program communicates

with the plug-in using a well defined interface. For example, consider a function call exported by a dynamic library. The function call takes as arguments an array, a pointer, and a length.

```
void foo( int array[], int length,
        const InfoStruct_t *readStruct );
```

The host program would like to verify that *foo()* does not write beyond the bounds of the array. In addition, it would like to verify that it does not write to the *readStruct_only* structure. Xu's algorithm can verify these rules, if they are encoded in an access policy.

An access policy is a set of 3-tuples. Each tuple consists of a region, category, and the type of access permitted. A region can be a range of addresses, or a single variable name. The category is a set of types, and the access field is, for example, a combination of read, write, followable, or executable flags. Together with the access policy, the host must provide the type-state for each variable and the invocation. In our example, here is the type-state:

Type-state
e: <int, uninitialized, rw>
array: <int [n], {e}, rw>
readStruct: <InfoStruct_t, initialized, r>

Access Policy
< e : int : rw >
< array : int [n]: rw >
< readStruct : InfoStruct_t : r >

Invocation
%o0 ← array
%o1 ← n
%o2 ← readStruct

The host type-state is the initial type-state of each variable. The access policy controls how the variables may be used. The invocation tells the algorithm what the initial state of the registers are. In this example, SPARC assembly is used, so the function parameters are passed in the registers.

- In the preparation stage, the algorithm takes the type-state, access policy, and invocation, applies the type-state to the invocation conditions, and creates initial constraints. The initial constraints are predicates on the values of the variables and the registers. For example, there may be a predicate that %o3 < n, if %o3 is used to address an array.
- In the type-state propagation step, the statements of the program are abstractly interpreted. At each program point, there is a copy of the memory state for each variable, register, or memory location, along with its type-state.
- In the previous stage, the type-states are known at each program point, but they cannot yet be verified. In the *annotation* step, the algorithm creates safety preconditions and assertions for each program statement. For isntance, it it determines that a variable is being used to index an array, it adds a safety precondition that the variable's value must lie within the bounds of the array. The two types of preconditions that it creates are local preconditions, which can be verified using typestate information alone, and global preconditions, which may require further analysis to verify. For example, a global precondition may require the calculation of the loop invariants to check.
- In the *Verification* step, the preconditions for each statement are checked. A local precondition may be, for example, that "*e* is initialized" and this can be checked immediately using the typestate calculated in the propagation step. However, a global precondition, such as an array bounds check, may require a range analysis on the values of the registers.

If all of the preconditions hold for the program, then it is deemed safe to execute.

Some of the limiations of the algorithm are its scalability and precision. The propagation step uses a flow sensitive interprocedural analysis which is quite slow [15]. Also, array elements are collapsed into a single element, so it is not possible to verify programs that are allowed to write to only certain portions of an array without creating a separate access policy for each element.

8. CONCLUSION

We have presented a survey of the use of static analysis techniques in binary executables. The techniques have been presented in the context of reverse engineering, disassembly, virus detection, and safety analysis. It is most striking not how far the research has advanced, but how far it must still go to do such basic tasks as disassemble a program. It is surprising that a virus can be constructed, and using simple technqiues, can disguise its source code from the best known disassembly techniques, such that it must be run in order to reveal its secrets. Researchers in program obfuscation and static analysis will perhaps fight a never ending battle with each other.

9. REFERENCES

[1] Intel 64 and IA-32 Architectures Software Developer's Manuals.

[2] H. Agrawal. On slicing programs with jump statements. *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 302–312, 1994.

[3] M. Christodorescu and S. Jha. Static Analysis of Executables to Detect Malicious Patterns. *Proceedings of the 12th USENIX Security Symposium*, pages 169–186, 2003.

[4] C. Cifuentes and A. Fraboulet. Intraprocedural static slicing of binary executables. *Software Maintenance, 1997. Proceedings., International Conference on*, pages 188–195, 1997.

[5] N. Dor, M. Rodeh, and M. Sagiv. CSSV: towards a realistic tool for statically detecting all buffer overflows in C. *Proceedings of the ACM SIGPLAN*

2003 conference on Programming language design and implementation, pages 155–167, 2003.

[6] S. Hallem, B. Chelf, Y. Xie, and D. Engler. *A system and language for building system-specific, static analyses*. ACM Press New York, NY, USA, 2002.

[7] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(1):26–60, 1990.

[8] J. Kam and J. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7(3):305–317, 1977.

[9] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static disassembly of obfuscated binaries. *Proceedings of the 13th USENIX Security Symposium (Security£04)*, 2004.

[10] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. *Proceedings of the 10th ACM conference on Computer and communications security*, pages 290–299, 2003.

[11] C. Manning and H. Sch'eutze. *Foundations of Statistical Natural Language Processing*. MIT Press, 1999.

[12] D. O'Donoghue, A. Leddy, J. Power, and J. Waldron. Bigram analysis of Java bytecode sequences. *Proceedings of the inaugural conference on the Principles and Practice of programming, 2002 and Proceedings of the second workshop on Intermediate representation engineering for virtual machines, 2002*, pages 187–192, 2002.

[13] F. Tip and D. Bäumer. Refactoring for generalization using type constraints. *Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 13–26, 2003.

[14] M. Weiser. Program slicing. *Proceedings of the 5th international conference on Software engineering*, pages 439–449, 1981.

[15] Z. Xu, B. Miller, and T. Reps. Safety checking of machine code. *ACM SIGPLAN Notices*, 35(5):70–82, 2000.