

## FEATURE

### CHALLENGES IN GETTING 'FORMAL' WITH VIRUSES

Arun Lakhotia and Prabhat K. Singh

University of Louisiana at Lafayette, USA

Is it a virus, a worm, a Trojan, or a backdoor? Answering this question *correctly* for any *arbitrary* program is known to be an undecidable problem. That is, it is impossible to write a computer program that will identify correctly whether an arbitrary program is a virus, a worm, etc. – no matter how much computing power is thrown at the problem.

With the emergence of polymorphic and metamorphic viruses, the anti-virus community is finally beginning to face this theoretical limit. Signature-based heuristics, whether dynamic or static, for detecting malicious code are no match for a program that modifies, encrypts, and decrypts its code as it propagates.

Researchers in academia and industry are beginning to develop anti-virus technologies founded on formal methods of analysing programs (Christodorescu and Jha 2003, 12th Usenix Security Symposium, 2003; Perriot, 13th Virus Bulletin International Conference 2003; Singh, Moinuddin et al., 2nd European Conference on Information Warfare and Security, 2003). These methods, with rigorous mathematical foundation, have mostly been developed for optimizing compilers and, more recently, for hardware and software verification.

The mathematical guarantees offered by these methods are a necessity for compilers and verifiers, for it is unimaginable that one would use a compiler or a verifier based on heuristics that give correct results only 90% of the time. The success of these techniques in compilers and verifiers has been extrapolated to offer promise in anti-virus technologies.

We argue that the formal methods for analysing programs for compilers and verifiers when applied in anti-virus technologies are likely to produce good results for the current generation of malicious code. However, this success will be very short-lived for it is extremely easy to 'attack' these analysers to make them produce incorrect results.

The fundamental basis of our observation is that: the formal methods designed for optimizing compilers assume that the compiler and the programmer are allies. In other words, the programmer does not stand to benefit from breaking the formal analysis. On the contrary, the compiler writer can assume that a knowledgeable programmer may be willing to reorganize a program to get optimal results from the compiler.

If there is one thing we can learn from polymorphic and metamorphic viruses it is that virus writers enjoy the challenge of beating the anti-virus technologies. Also, it would not be inaccurate to say that the virus writers, or say, the virus-engine writers are not just good programmers; they have a very good understanding of computer science. It is by no means a small feat to write a program that disassembles a host program; reorganizes the code; injects malicious code deep inside this reorganized code; reassembles the new program; and overwrites its disk image in the correct format. And this has not even touched the capabilities needed to encrypt, decrypt, and morph the malicious code.

It should, then, be safe to assume that virus writers can and will find ways to break the formal analysis techniques as well.

Are we saying that it is not worth using formal methods in anti-virus technologies? Not really. Just as signature-based heuristics (however limited) have offered effective defences by raising the bar for the virus writer, so will the use of formal methods. However, these methods must be adapted to the new scenario where the programmer (i.e., the virus writer) and the analyser have conflicting goals.

Rather than acting like an ostrich, it is important that we assess the use of these methods and their effectiveness against attack on the analysis mechanism itself. This is not an easy challenge, for as we show, it calls for a complete rethinking of all the underlying assumptions in all phases of analysing a program.

In this article we enumerate, using an example, the promises and pitfalls of formal analysis. We then outline the steps traditionally used in performing such analyses. Next we show how the known algorithms for each step can be broken. We conclude the article with a call to rethink the process of analysing binary executables.

### PROMISES OF FORMAL ANALYSIS

The methods for formal analysis of computer programs have mostly been motivated by one of the following: (1) improve runtime performance, (2) decrease code size, and (3) increase confidence in the correctness of a program – all with minimal, if any, intervention from the programmer.

The formal analyses may be classified into two categories: static analyses and dynamic analyses. A static analysis finds properties that hold for *all* executions of a program. Such an analysis is typically performed without executing the program, hence the qualifier *static*. A dynamic analysis finds properties true for a specific input. It is called *dynamic* because it is typically performed by executing or interpreting the program.

The classic compiler optimizations, such as dead code elimination, constant folding, constant propagation, elimination of partial redundancies, loop unrolling, etc., are static analyses. Model checking, a technique for verifying the existence or non-existence of certain temporal properties in a program is also a static analysis. Analysing coverage achieved by a given set of test cases is a dynamic analysis. Profiling a program to identify code segments or functions consuming the most time is also a dynamic analysis.

We now take a common analysis performed by an optimizing compiler and assess how it may be applicable in anti-virus technologies. Consider the following code segment:

```
int offset, port;
offset = 2.5*2;
port = offset + 20;
```

Instead of generating code to multiply and then add the constants, as in the above code segment, most optimizing compilers will use constant folding to generate code equivalent to the following:

```
port = 25;
```

Quite coincidentally, metamorphic viruses apply transformations that go the other way around. They replace a constant by a sequence of steps that eventually produce the same constant. We call such a transformation *constant unfolding*. A metamorphic virus may apply constant unfolding randomly to different constants appearing in the program, thereby generating code that looks different from the original. Besides changing the signature of the program, this transformation also obfuscates the program, making its manual analysis harder.

Constant folding and constant unfolding are inverses. It stands to reason that an anti-virus technology may use constant folding to undo the obfuscation created by constant unfolding. For instance, in order to determine whether a program may be sending email, an anti-virus system may check whether a program calls the *connect* function so as to connect to port 25. If a virus writer, or a metamorphic engine, attempts to obfuscate the computation of the port number, the anti-virus system may apply constant folding to de-obfuscate it.

Like constant folding, other optimization transformations, offer similar promises. Dead code introduced by a metamorphic virus could be removed using dead-code elimination; reordering of instructions by introducing jump statements can be undone by reordering the instructions.

Though, at first glance, optimizing away the effects of metamorphic transformations looks like a promising technique, the real test of such analysis techniques depends

on how they can be attacked. The example used above can be optimized by constant folding because the computation generating the constant is in the same control-flow block. Consider the following code:

```
if (x < y) {
    x = 10;
    y = 15;
} else {
    x = 15;
    y = 10;
}
port = x + y;
```

In this program too the variable 'port' has the value 25 for all possible executions. However, constant folding will be unable to determine this fact. The reason is that 'port' depends on the value of variables 'x' and 'y', and neither of these variables holds a constant value at the point at which 'x+y' is computed. Hence, the analysis will incorrectly determine that 'port' is not a constant.

This limitation of static analysis should not come as a surprise. Determining whether a piece of code always produces a certain constant value is the same as determining program equivalence, which is an undecidable problem. Hence, we can never develop a method that always determines correctly that a variable is a certain constant for all possible programs in which the variable is really a constant.

One may argue that dynamic analysis may be used to make the determination needed in the above example. But then, dynamic analyses can also be fooled. For example, to avoid getting into an infinite loop, and thereby hanging a system, an anti-virus system that uses an emulator (or a sand box) would have to determine when to terminate the analysis. The virus could attack the analysis by exhausting the patience of such an analyser.

## PROCESS OF ANALYSING BINARIES

The real challenge facing the anti-virus community is in analysing binary viruses. The anti-virus community knows how to handle macro viruses pretty well. In this section we outline the steps commonly used in analysing binaries. In the next section we highlight some of the assumptions at each step and how they can be attacked.

Figure 1 gives a diagrammatic representation of the steps in analysing a binary statically for the presence of malicious behaviour. For the sake of our discussion we assume that the input binary is unencrypted. This constraint disqualifies the use of this analysis for polymorphic viruses. Even without polymorphism we have enough to worry about, hence we ignore this dimension.

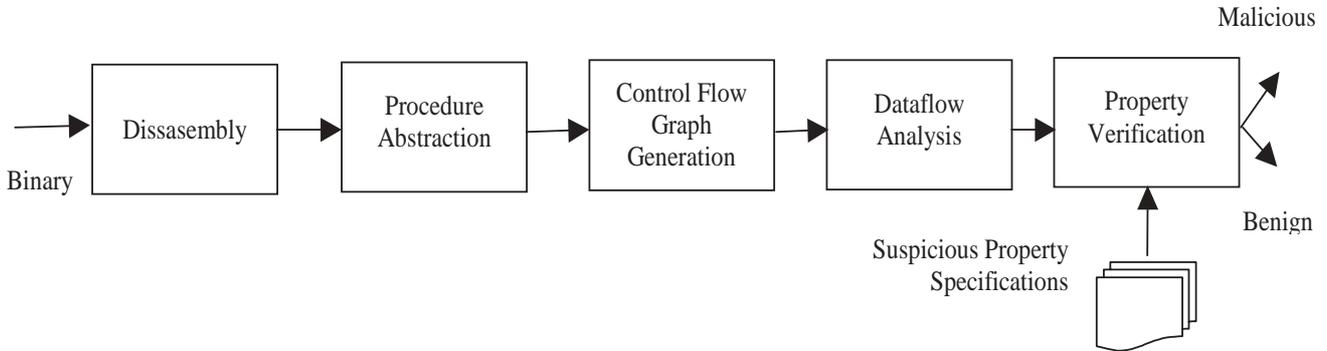


Figure 1. Stages in static analysis of binaries.

The various stages in the analysis and the key function performed at the stage is described as follows:

**Disassembly**

The broad role of disassembly is to create mnemonic representation of binary code. The mnemonic representation is useful for manual analysis. However, the mnemonics are not necessary for a complete automated analysis. The key work performed in this stage is to determine which bytes of the binary hold executable instruction and which hold data.

Since most recent architectures separate code and data, one may be tempted to believe the work at this stage to be trivial. That is indeed not the case because, while an architecture may enforce that a data segment does not contain code, there is no guarantee that the code segment does not contain data. Of course, since the code segment is write protected, the data in the code segment can only be constant. Nonetheless, there can be data, or simply garbage, in the code segment.

The common method of disassembly, referred to as the linear sweep method, assumes that all bytes starting from the entry point of a binary (or some start location) are instructions, and disassembles the entire code segment, following successive bytes. When the content of some location does not match a valid instruction, one may assume that it is data.

This method is acceptable in a disassembler designed for interactive analysis of binaries, such as IDA Pro. However, it has obvious shortcomings when used for automated analysis.

A recently reported algorithm, called recursive traversal, overcomes some of the shortcomings (Schwarz and Debray, 9th Working Conference on Reverse Engineering, 2002). In this method code is disassembled by tracing the flow of control in the program. Thus, whenever a branch instruction is encountered the disassembly continues simultaneously at both the address following the branch instruction and the

address that is the target of the branch instruction. Some targets that are reachable only through indirect control transfers are identified using a speculative disassembly process (Cifuentes and Emmerik, IEEE Computer 33(3), 2000).

**Procedure abstraction**

Once the executable instructions of a program have been identified they may be segmented into groups representing procedures (or functions). This is motivated from the notion of procedures (and functions) in high-level programming languages. Since most compilers compile a procedure into a contiguous set of instructions, the procedure boundaries could be determined by identifying the entry points of successive procedures. The entry points in turn could be identified by identifying the CALL instructions (in the disassembly stage).

Unlike its high-language counterpart, a binary program does not have any construct identifying the beginning and end of a procedure. This problem does not appear to have been discussed in the literature and may need attention.

**Control flow graph generation**

A control flow graph (CFG) is a directed graph. Its nodes represent statements (or blocks of statements). Edges in the graph represent flow of control from one statement (or block) to another. A CFG is commonly created for a single procedure. Since a procedure in a high-level language has a single entry and a single exit point, it is common for a CFG to have a unique entry and exit node as well. In a CFG the node representing a procedure call may be linked to the CFG of the called procedure, thereby creating *interprocedural* CFG.

All static analysis techniques used in compilers and model checkers *assume* the existence of CFGs for the procedures of a program. Because a CFG is a language-neutral representation of the flow of control in a program, algorithms based on CFGs can be used for any (procedural) language.

An algorithm for constructing CFGs for high-level language programs is available in text books. The same method is adapted for assembly language programs.

### **Data flow analysis**

There are various analyses that qualify as data flow analysis. The most common analysis is data dependence analysis, which is to determine the instructions that use the variable (register or memory location) modified by another instruction. The analyses performed for optimizing transformations are all classified as data flow analyses, or flow analyses.

Besides assuming the existence of CFGs, these analyses also assume that the data area of a program is segmented into variables, where each variable has a unique name (modulo scoping rules). Also associated to a variable is its type and size. A binary program has no such segmentation. The data area is simply a continuous sequence of bytes. Though the address of a data area may be treated as its name, the type of data it holds and the size (number of such units) is not obvious from the binary. The problem of identification of variables in a binary does not appear to have been discussed in the literature, and deserves attention.

### **Property verification**

The property verification's phase determines the existence (or non-existence) of a property in a program. This phase takes two inputs: (1) a formal representation of the suspicious property (or properties) and (2) control flow graph and data flow information of the program. It outputs a determination whether or not the program satisfies the given properties. This answer is then translated into whether the program contains malicious behaviour.

For example, consider the property *SendsLotsOfMail* to be true if the following holds:

The program contains a loop containing *GetEmailAddress* and *SendMail*.

In order to determine whether this property holds, the analyser may create a compacted CFG that contains only calls to *GetEmailAddress* and *SendMail*. It would then determine whether the two functions are in a loop.

Over the last decade the use of model checking to verify the presence or absence of properties has gained prominence. Loosely speaking, model checking is a way to check for the existence of a finite state machine (specification) in another finite state machine (program).

The property to be checked is described as a finite state machine that transitions on *atomic* predicates, properties that can be identified by cursory look at the program. The program being checked is also converted to a finite state machine, created by abstracting away all the details except

the atomic predicates observed in the program. Model checking is then used to check whether a program has a given property.

## **PROBLEMS IN STATIC ANALYSIS OF BINARIES**

We now discuss how a virus writer may attack the various stages in the analysis of binaries described above.

### **Attack on disassembly**

In the von Neumann architecture there is no *definite* way to differentiate code and data that is resident in memory.

The linear sweep method can be fooled by introducing garbage data immediately after an unconditional jump instruction. The recursive traversal method can be fooled by placing garbage data after a conditional jump instruction and programmatically ensuring that the jump condition always succeeds. This will lead the recursive traversal algorithm to follow both the paths, the instruction immediately after the conditional jump instruction and the target of the jump instruction, potentially leading to an inconsistent state.

To throw the disassembly off, the garbage data may be crafted so that it matches a valid instruction, thus beating a heuristic that validates the instruction after the jump instruction. The target of the jump instruction itself may be hidden by computing its address in a register and using an indirect jump instruction to transfer control to the address in that register.

### **Attack on procedure abstraction**

Since a procedure is a basic unit of most analysis algorithms, a virus writer can defeat static analysis by making it harder to identify a procedure unit in binary program. The analysis can be attacked if one cannot determine the boundaries of a procedure. This stage can be attacked by obfuscating the call instruction – say, by using a combination of the push and jump instructions.

There is also no sanctity in the tradition of placing the code of a procedure in contiguous memory locations. In fact, this is just a tradition followed by compilers. There is no guarantee that hand-written assembly programs follow this tradition. The virus Win32.Evol is a classic example. Use IDA Pro to identify its procedures and you'd find that it beats the heuristic used by IDA Pro.

There is no necessity for such mangled code to be hand-written. It will not be too hard to modify a compiler such as gcc so that it mangles the code for a procedure into non-contiguous blocks. Or such mangling can be performed automatically after an executable is created.

**Attack on control flow graph generation**

The construction of CFG can be attacked by fooling the CFG generation algorithm in creating redundant edges in the CFG. The creation of extra edges may jeopardize the precision of the analysis in the later stages. The CFG generation process can also be attacked by obfuscating the assembly code such that one cannot determine the correct target of a jump instruction, such as using a jump through register.

One method for resolving the targets is to assume that the jump will transfer control to every instruction (or to every block). This assumption, though safe for various static analyses for compiler optimizations, could spell doom for an anti-virus technology by increasing the time and space costs of the analyses.

**Attack on data flow analysis**

An example of attack on the flow analysis stage was discussed in the section entitled 'promises of formal analyses'. Most flow analysis algorithms propagate sets of data from one node of a CFG to another. When data flows into a node from two different predecessors, the two sets of data are merged to create a single set. In the process of merging the data some information is lost, leading to an imprecise analysis.

The data flow analysis stage can be attacked by moving some computation from a block (of a CFG) into the preceding nodes and obfuscating the computation along each path. The analysis can also be attacked by using data that resides outside the scope of the program, such as in the areas managed by operating system. The known constant values in these areas may be used in the program for computation, thus thwarting accurate analysis. For instance, Win32.Evol and other viruses utilize knowledge of the specific address where kernel32.dll is loaded. The same addresses could also be used to access constant data from kernel32.dll code.

**Attack on property verification**

Property verification is carried out using theorem proving systems. Typically, these systems depend on human guidance to prevent them from getting into infinite loops. Completely automated theorem provers, such as model checkers, operate on an abstraction of the problem. Sometimes the abstraction itself may be so large that the theorem prover may take an inordinate amount of time and resources to complete the proof. Or else, the abstraction may throw away so much information that the theorem prover may yield results that are correct for the abstraction, but incorrect for the program being analysed.

A virus writer can attack the mechanism using the knowledge of how the theorem prover used in an anti-virus

technology determines the existence of the *atomic properties* and how it composes atomic properties into more complex properties.

For instance, consider an anti-virus tool that uses the presence of calls to system library as atomic properties. This AV tool may look at the address of the target of a call instruction to determine whether a system library function is being called. Win32.Evol will escape such a virus detector because it obfuscates calls to system library functions. Instead of using the call instruction, this virus uses the return instruction to make the call. Before the return instruction is executed, though, the address of the function to be called is pushed on the stack. If the analyser cannot determine whether a virus calls a specific library function, the analyser has two choices. One, assume that the program actually does not call the Win32 API, thus letting the virus pass through. Two, assume that the program does call the library, thus generating false positives for programs that actually do not call the API.

**CONCLUSIONS**

We have argued that formal analysis methods used by optimizing compilers and other programming tools, though they appear promising in the detection of metamorphic viruses, are not directly suitable for use in anti-virus technologies.

The common approach for analysing a binary consists of the following stages: assembly, procedure abstraction, control flow graph generation, data flow analysis, and property verification. Compiler optimization-based methods for each of these stages can easily be attacked. Thus, even if the processing in each stage was correct 90 per cent of the time, the overall system will be correct only 59 per cent of the time, which is pretty close to the results offered by flipping a coin.

In order to use these formal analysis methods in anti-virus technologies, we may have to rethink the whole process. Unlike in the context of optimizing compilers and other similar tools, the analysis tool and the programmer (virus writer) do not have a common objective. Hence, assumptions made by analysis methods used by such compilers can be exploited by the virus writer.

The good news is that a virus writer is confronted with the same theoretical limit as are anti-virus technologies. To keep ahead of anti-virus technologies, a metamorphic virus writer has to address some of the same challenges that AV technologies face. This is likely to have an effect on the pace at which new metamorphic transformations can be introduced. It may be worth contemplating how this could be used to the advantage of AV technologies.