Detection of Malicious Code in COTS Software:
A Short Survey *

Jean Bergeron (Jean.Bergeron@ift.ulaval.ca)
Mourad Debbabi (Mourad.Debbabi@ift.ulaval.ca)
Jules Desharnais (Jules.Desharnais@ift.ulaval.ca)
Béchir Ktari (Bechir.Ktari@ift.ulaval.ca)
Martin Salois (Martin.Salois@ift.ulaval.ca)
Nadia Tawbi (Nadia.Tawbi@ift.ulaval.ca)
Département d’informatique, Université Laval
Québec QC G1K 7P4 Canada

Robert Charpentier (Robert.Charpentier@drev.dnd.ca)
Michel Patry (Michel.Patry@drev.dnd.ca)
DREV/CRDV, 2459 Boul. Pie XI nord
Val-Bélair QC G3J 1X5 Canada

Abstract
With the advent and the rising popularity of networks, Internet, intranets and distributed systems, security is becoming one of the focal points of research. A malicious code is a fragment of program that can affect, or let other programs affect, the confidentiality, the integrity, the data and control flow, and the functionality of a system. Therefore, the detection of malicious code is a major preoccupation within the computer science community. In this article, we describe the main characteristics of malicious code and propose a taxonomy for the existing varieties. We also describe several techniques that can be used to detect malicious code in Commercial-Off-The-Shelf (COTS) software products.

1 Introduction
From October 1st 1988 to December 30th 1995, 4299 incidents involving malevolent software were reported to the CERT®/CC[1][How97] and were fully documented by the organization. Of these, 450 were classified as Trojan horses, 567 as autonomous agents (e.g. viruses, worms), and 1948 were defined as having used some form of access vulnerabilities in commonly used programs.

These rather bleak statistics may only be the tip of the iceberg, since it could be argued that the number of events reported to the CERT®/CC is very limited. Because CERT®/CC's reported incidents are public and need a relatively large effort to report, many companies could hesitate before reporting a security breach, especially if they are able to cope with it by themselves and feel that a disclosure would be bad publicity. Therefore, the actual number of incidents reported can be considered as a minimum. There are other estimates that are orders of magnitude larger. Howard [How97] reports that the largest number suggested was 900 millions attacks for 1995 alone.

It is highly difficult to place a value of confidence on such numbers. All we need to realize is that the threat is real and that some form of protection against malicious code is needed.

*This research is supported by a contract from Defence Research Establishment, Valcartier (DREV), 2459 Boul. Pie XI Nord, Val-Bélair, QC, G3J 1X5, Canada.

In this document, we are concerned with malicious code that could exist in Commercial-Off-The-Shelf (COTS) software products. COTS qualifies any kind of commercially available software tool, usually protected by copyrights that prevent us from obtaining knowledge on its internal working. COTS products include, but are not limited to: code libraries, databases, complete applications, tools that automatically produce code, etc.

Why would a company want to use a COTS component if it is not 100% safe? Because the technology is getting bigger and bigger and it is becoming tougher and tougher, some would say impossible, to develop locally everything the users need. COTS packages provide an easy access to a huge number of functionalities at a cost much lower than local development because their development is shared among many users.

This is the good side of COTS. On the other side, there are major inconveniences to using them in a home system. Since COTS products, because of legal rights, are black boxes, they are hardly customizable, and the level of confidence that we have in them may be limited. Finally, and this is the major concern of this document, COTS software can have embedded security risks that could cause denial of service, access to privileged information, corruption of functionalities, etc.

In this document, we present many detection techniques that we have encountered in our study of the domain. Some of these methods were not designed directly for malicious code, but we feel they show promises toward that goal. The paper is divided as follows. In the next section, we give a more precise definition of what we call malicious code and present a taxonomy. Section 3 presents existing methods of detection. Then, we conclude.

2 Malicious Code

2.1 Definition

A malicious code is a fragment of program that can affect, or let other programs affect, the confidentiality, the integrity, the data and control flow, and the functionality of a system without the explicit knowledge and consent of the user. We distinguish intentionally malicious and unintentionally malicious code. The first type is generally due to malicious people that use programs to serve as vehicles to access, for example, confidential data. The second type is due to inadvertent human errors, especially during the development life-cycle of the software.

Covert channels, logic bombs, trapdoors, Trojan horses, viruses and worms, for example, are considered under the term malicious code.

The next section proposes a taxonomy that is useful for distinguishing the different types of malicious code during the detection process.

2.2 Taxonomy

To successfully detect malicious code in COTS software, one must be able to distinguish between all types of malicious code. Starting from the taxonomy proposed by [MC94], we define a new taxonomy that is oriented toward the goal of detecting malicious code. We classify malicious code according to three basic questions:

1. How does it enter the target or infect the system?
   - Intentionally: this class covers malicious code like Trojan horses, back doors, trap doors, covert channels, logic bombs and time bombs.
     - Random propagation: as an example, we can cite viruses. A virus infects a program by attaching itself to the program. Then it either destroys it or co-exists with it. Worms, bacteria, and rabbits are also in this category.
     - Driven propagation: logic torpedoes, crabs, creepers, and general intrusions on a specific system. The malicious attack has a specific target and performs no noticeable action until the target is reached.
     - During the development of the software: any malicious code that compromises the integrity of the files used during the development of a software system (system-development attack).
   - Unintentionally: bugs, vulnerable features, and other mistakes made by the developers,
When does it launch in the system?

- Logic events: this class covers actions that are triggered by a logical condition that is met. Evaluation copies of programs allowed to run $n$ times, a precise sequence of operations, access to a particular web site, all logic bombs, some types of Trojan horses, some types of trapdoors, and Easter eggs belong to this category.
  - Temporal events: these are logic events whose condition depends on a specific time or date. For instance, a virus wishing “Merry Christmas!”; evaluation copies stopping after 30 days, time bombs, and some types of Trojan horses are in this subcategory.
- Execution of the carrier: the malicious action is triggered when the carrier is run. Examples: viruses and Trojan horses.
- Continuous: malicious code that is always active or ready to go on command. Some actions of Back Orifice and Netbus, once activated, sniffers, bacteria, ...

What is the goal of the malice?

- Prevent access to system resources: bacteria, rabbits, overloads, any kind of denial of service, system-development attacks, ...
- Obtain protected information: covert channels, intrusions, sniffers, SecHole, Back Orifice, Netbus, ...
  - Gain access to the network, or get information from it: sometimes, the information is well protected on a machine, but this machine communicates with the network. Sniffers and covert channels belong to this class.
- Compromise system files: viruses, system-development attacks, SecHole, ...

Here, we have to note that the number of malevolent actions a malicious code can perform is virtually limitless. Thus, it is difficult, some would say impossible, to give an exhaustive list of all malices.

Detection Techniques

Binary Code vs Source Code

Given that COTS software is often produced in a very competitive environment, information concerning its design and implementation is being kept secret most of the time to help the company keep its software at the edge. For this reason, it is unreasonable to assume that we will have access to the source code when evaluating commercial software, or even have a clear and precise specification of the program’s features.

Consequently, the use of binary code as raw material for the detection of malicious behavior in commercial applications is more realistic than the use of source code. Even if this has the potential of making the problem even more difficult, we feel that any tool that proposes to solve the problem of detecting malicious code must take this into account.

On the other hand, companies also keep a tight grip on what can legally be done with their executable software, as we can see in the following:

"Restrictions. You agree not to modify, adapt, translate, reverse engineer, decompile, disassemble or otherwise attempt to discover the source code of the Software."

Quoted from Adobe Systems Incorporated Electronic End User License Agreement for Adobe Acrobat Reader.

This “legal” problem could be a major one with some of the techniques presented in this document because they are based on the analysis of the source code. Since we have already established that most often the only source code available is the actual binary program, even though it is not really source code, it follows that we would need to adapt the techniques to actual processor instructions, thus doing a form of disassembling. We feel that a certain leeway could be given to a technique that proposes only to protect the customer without any purposes whatsoever toward using the information to reverse engineer and reuse the technology of the studied software. However, the legal ramifications of this subject are out of the scope of this document.
Even if legal access to the assembly code is granted, not all problems are solved. Effective use of binary code as source code may be problematic. Here are some of the reasons:

- **Possible lack of information** — More information is generally available from source code in a high-level programming language than from binary code.

- **Possible loss of structural integrity** — Due to optimization, even if the source code is well structured, it may not be the case that the resulting binary code is. This also means that we have to deal with every kind of function calls (like C vs Pascal calls), data structure representations, and other particularities.

For these reasons, some analysis techniques are more difficult to apply to binary code, or at least would need some reworking. For example, a type-based analysis is easier to apply to source code than to binary code. This stems from the fact that, in the former, the information on types is already available whereas, in the latter, we need to devise a way to obtain this information.

At this point, the reader may have the impression that the situation is hopeless without access to source code, or that any tool would be a lot better if it had access to it. This may not necessarily be the case. In fact, there are some good reasons to use binary code as raw material. Two of the most important ones are:

- **Back-end code** — The binary code is the one that will actually be executed on the host machine, while the source code must be compiled before being executed. The compilation may introduce errors that are not present in the source code. Hence, the use of binary code for the analysis can raise the level of trust we have in the actual software that the users will run.

- **One tool for multiple platforms** — The binary code depends only on the processor family, not on the operating system, nor the compiler of a particular high-level language. This can lead to a more general tool that could be applied to many platforms.

The last point is particularly meaningful when considering the huge variety of programming languages that are available on a given platform. A tool working directly at the binary level would be machine dependent, but language independent. It could analyze a program coming from Ada, Basic, C, C++, Cobol, or even Java. In this last case, it would probably need to work at a different level by analyzing the virtual machine's binary code too, in order to prove it correct, according to some definition of correctness.

All in all, the use of binary code as a basis for the discovery of malicious code in a commercial application may not be the ideal working material, but it is the most realistic one.

### 3.2 Static Analysis

Static analysis offers techniques for predicting properties of dynamic executions of programs without running these programs. Traditionally, these techniques were used extensively to enable various optimizations and transformations in compilers. Among the newer applications is the detection of malicious behavior in software. In this section, we present three static techniques that can be used to detect malicious code in programs.

To apply these techniques to the detection of malicious behavior in binary code, the binary code must be converted to a more abstract form. The assembly language can be the first step in the translation. Once the code is disassembled, we still need to transform it to a more comprehensible form. For this purpose, reverse engineering techniques such as decompilation are required. For example, by applying data-flow analysis, sequences of flag-handling opcodes would be replaced by boolean conditional expressions, the arguments and the return value of the different subroutine calls would be computed, and dead-register and dead-condition code would be eliminated. The use of compiler and library signatures could reduce the number of subroutines to analyze. The aim of these transformations is to convert the machine opcodes to a more abstract form that is suitable for the application of the static analysis techniques presented in this section. Note that we have classified the technique of searching for virus signatures as an *ad hoc* technique (see Section 3.5), although it is applied
3.2.1 Flow-Based Analysis

The aim of flow-based analysis is to transform a source program into a more abstract representation that can be used to generate information about control and data flows. This abstract representation is generally a graph. We distinguish two sorts of graphs:

- **Control-flow graphs**: in a control-flow graph, each node corresponds to a statement or a block of statements of the program. An edge between two nodes represents direct flow of control between them.

- **Data-flow graphs**: a data-flow graph is computed from a control-flow graph. Its nodes correspond to an operation in a program and its edges represent various types of information flow between nodes.

Three techniques may be used to generate information that is useful for the detection of malicious code. The first technique, control-flow analysis, uses the control-flow graph to detect the use of malicious primitives. The second technique, data-flow analysis, uses the data-flow graph to detect dependencies between statements and variables. The third technique, slicing, uses both the control and data-flow graphs with the intention of extracting program statements relevant to some property. The aim of this technique is to focus the analysis only on some interesting statements rather than the entire program. As an example of a tool using these three techniques, we cite MCF (Malicious Code Filter) [LL095].

3.2.2 Type-Based Analysis

Type systems have always been used to statically guarantee some dynamic good-behavior properties of a program. The main property of such systems is type soundness, which can be summarized like this: “well-typed programs do not go wrong”. From this fact, many researchers have had the idea to use these systems to solve program security problems. The main idea in their research is to add security aspects in the type systems in order to guarantee that well-typed programs possess some security properties. More specifically, they first define an operational semantics that takes into account security errors. Then they define a type system in order to check if the program is well-typed. Finally, they prove a type soundness result which attests that well-typed programs never cause these security errors. See for instance [LR98].

Another approach used by researchers [DFT96, DAF97, Fao97] consists in enriching type systems with annotations, thus providing more information on the dynamic behavior of programs. Such systems are known as annotated type systems. The main idea following this approach is to modify existing type systems in order to process the annotations too. Once the annotated type is computed, it must be analyzed in order to detect some possibly malicious behavior.

3.2.3 Analysis by Abstract Interpretation

Abstract interpretation is a method for designing approximate semantics of programs which can be used to gather information about programs in order to provide sound answers to questions about their run-time behavior [CC92, CC77, JN95]. The primary goal in this approach is to perform the program’s computations using abstract values rather than actual computed values. Thus, the result of the analysis describes all possible program executions.

Abstract interpretation may play a major role in the detection of malicious code. First, it could be used as a theoretical framework for the expression of static analyses. Second, it allows to reason about their soundness and completeness with respect to well-defined semantic hierarchies. Third, it may be used to derive efficient approximation algorithms to compute security properties. As an example, we cite the abstract interpretation-based version of control-flow analysis. The latter could be used to track the call sites of API-functions that are critical from the security standpoint. Fourth, it may be used to tune, in a formal setting, the precision/cost ratio of these analyses.

The main problem that the static analysis techniques face is the complexity or even the undecidability of these analyses. This is due to the fact that we are considering all possible executions, which means that the whole input domains have
to be dealt with. These domains can be infinite or at least very large. With the approximations underlying the abstract interpretation framework, one can cope with this problem. Both data-flow analysis and control-flow analysis could profit from this framework. In the case of data-flow analysis based on abstract interpretation, for instance, one can get more powerful results when inferring properties based on the values of variables, the alias relation, relations between variables, etc. These properties can help determine whether a fragment of code may be executed or whether a function call could ever be triggered.

3.3 Dynamic Analysis

Dynamic analysis is concerned with the exploitation of knowledge that can be gained by running the program that we want information about. This knowledge can then be used to verify some properties or to limit some undesirable functionalities.

One very popular way to do dynamic analysis is to use monitors. We start this section with two examples that demonstrate variations on this approach. Then, we talk about testing techniques that can be used to detect malicious code. We follow this with a brief description of a complementary methodology to testing, which consists of voluntarily injecting errors in a program to observe its behavior under stress conditions. Finally, we have a look at what, in our opinion, is an unavoidable tool that will need to be used, namely wrappers.

3.3.1 Monitoring

Monitoring is the act of observing the behavior of one (or more) target program(s) with the help of one (or more) execution monitor program(s). Contrary to static analysis techniques, a monitor can have access to run-time actual values.

A monitor can present the information to the user as the target program executes, or show it later, after some post-analysis has been done. In the first case, the monitor can be fully automatic or interactive. With an automatic monitor, the user can only watch what is going on, whereas, in the interactive case, the user can have some control over the execution of the target program. A typical example of the latter is a debugger.

Hence, the foremost task of a monitor is to provide information about a target program in a compact and clear way. As if this task was not complicated enough by itself, three factors, inherently present in every monitor, come and complicate matters a little more. These factors are [Jef93]:

- **Volume** — A large quantity of information is available to an execution monitor and we must carefully select what is needed to avoid taking up too much resources and preventing the normal execution of the target program. This can bring false results or obscure useful information.

- **Intrusion** — An execution monitor inevitably influences the target program in some way, even if only in sharing the resources. This change is called intrusion. Two categories of intrusion exist:
  - **Passive** — Indirect data is used to monitor the target program.
  - **Active** — The environment of the target program is instrumented to facilitate the observation. There are two possible alternatives:
    * The source code can be modified, manually or automatically, so that it generates the needed information.
    * A virtual machine can be used to interpret the programming language and generate the information.

- **Access** — To be of any use, an execution monitor must have access to some kind of dynamic information on the target program. This is difficult in an active execution monitor since sharing of memory space is often prohibited by the operating system. Furthermore, this can cause more performance problems and, if one is not careful, one can modify the memory space of the target program and change its behavior, thereby falsifying any result.

These three problems are universal to all techniques based on monitoring. They are also interrelated on many levels. For example, an elegant way to solve the access problem is to have the execution monitor run the program in its own memory space. However, doing so augments intrusion.
Therefore, the major problem to solve in any monitoring tool is to minimize the volume level, the intrusion level, and the access level problems. This is not an easy equation to solve. This, combined with the very limited amount of computing resources of yesteryear, may explain why not much research has been done in this field. Following are two examples of monitoring frameworks that have been proposed, one using passive intrusion, and the other using active intrusion.

**Passive Intrusion Monitoring**  The technique presented here is based on the work of Calvin Ko et al. [FKAL94, KFL94, Ko96]. It is a non-intrusive method which uses the audit trail of the Unix system to observe the behavior of the target program. In this case, the target programs are mainly small privileged utilities that are used in the Unix system to provide basic services, such as `rdist`, `fingerd`, `mail`, and `binmail`.

Using the definition given in [Ko96], the audit mechanism enables logging of security-relevant events that have occurred in the system. The audit trail is the listing of these events. Examples of such events are: opening and closing of files and communication ports, logging on and off, changing of access rights, etc. Since audit trails are produced by most major secure operating systems (e.g. Unix, Windows NT), it should be a reasonable tool to use as a basic block for a more thorough investigation of a particular program.

The method is able to detect the exploitation of a program’s vulnerabilities via the use of specifications which define the good (or bad) behavior of the program in question. The method uses the audit trail as its main tool; however, it also requires information that is not available in the audit trail of the Unix system. Thus, the technique presented can cope with a lack of information from the audit trail by adapting and fitting other means of getting the information.

This method has the advantage of being able to detect many violations at run-time without the need to modify in any way the program being monitored. Therefore, the intrusion and access problems are minimal. The only problematic factor can be the volume of information available from the audit trails. In most auditing mechanisms, this results in the turning off of some audit possibilities. For example, most systems record which files are being written to, but not what is being written because the volume of information is too large. However, the technique described does major filtering of the audit records because it is interested only in privileged programs. This is a major advantage over other intrusion detection systems which usually need to monitor non-privileged programs as well.

Many commercial versions of audit trail monitors, such as CMDS™ [Pro96], are available. Still, they seem to be more oriented toward detecting intrusion than general malicious code. Furthermore, they usually use statistical profiles of users and/or signatures of known attacks. This requires rather long training time for the tools, or a vast knowledge of current possible attacks.

**Active Intrusion Monitoring**  The technique presented here is based on the work of Clinton Lewis Jeffery and the team working on the Alamo project [BJ97, Jef93, TJ96, ZJ96]. The first part of the work was based on the interpreted language ICON. Jeffery instrumented the virtual machine to provide the necessary information. The second part, the major part of the Alamo project, is based on massive instrumentation of C source code to provide the same information that was available in the ICON version.

The first objective of the technique was not the detection of malicious code. It was an attempt to ease the visualization of the demeanor of the target program by creating a framework for the rapid development of various monitors. Nevertheless, some of the visualizations proposed by Jeffery clearly indicate ways to detect malicious code, whether intentional or not. For example, monitors are proposed to see if a memory access is out of bounds by drawing dots, representing memory access locations, over a line, representing allocated memory.

As we have said, the programming language ICON is an interpreted language. This means that a virtual machine is necessary to interpret the commands and feed them to the system. Jeffery presents an extension to the language, which he calls Multi-Tasking ICON (MT-ICON) [Jef93]. The extensions are aimed at creating a tool to rapidly and efficiently develop monitors. In addition to the modifications necessary to control the
flow of the program, four components have been added to the language:

1. **Dynamic loading** — The ability to load another program in one's memory space. This allows an execution monitor to run the target program with full access to its data (thus solving the access problem). This also accelerates the context switch between the two programs.

2. **Synchronous execution** — The ability to switch execution between the monitor and the target program. This gives greater control to the monitor.

3. **Run-time system instrumentation** — The virtual machine is modified to provide all available events to whomever is asking for them. This has the double advantage that the target program does not need to be altered in any way and it also offers better performance.

4. **Event masks** — To limit the number of context switches, which are time consuming, a filtering concept is added that allows monitors to indicate to the virtual machine which events it is interested in. This *mask* can be dynamically changed by the monitor at the end of every context switch. When events occur that are of no interest to any monitor, the virtual machine simply continues execution without any further delay.

   With this, an execution monitor can interrupt the target program when interesting events occur, examine the pertinent data and then give the control back to the virtual machine to run the program until another event of interest occurs.

   For the MT ICON language, examples of execution monitors, presented in [Jef93], include:
   - A monitor that dynamically constructs the activation (or calling) tree of the program.
   - Different monitors to *visualize* memory allocation.

   The Alamo Monitor Executive project extends the work of Jeffery to a more mainstream language, ANSI C. The primary problem in adapting the technique comes from the fact that, contrary to ICON, C is not usually an interpreted language. It is not possible to instrument the virtual machine since there is no virtual machine. Hence, a more direct (and consequently more intrusive) approach is needed: the source code has to be instrumented. To do this, an automatic tool has been developed and is presented in [TJ96]. The tool works as a pre-processor that modifies the source code and produces an intermediary output to be compiled by any ANSI C compiler.

   Dynamic loading is relatively easy to implement. All that is required is for the execution monitor to launch the target program. Synchronized execution and event simulation are implemented via the use of an EV() function that is added wherever it is necessary. Combined, these three devices allow just about the same functionalities for event watching as the MT ICON version.

   Active monitoring, as presented in this section, seems to be a great way to see what is going on inside a program while it runs. Of course, from our specific point of interest, namely the study of commercial software for which we do not have the source code, the technique can be problematic. It remains to be seen what can be done on binary code as basic material for analysis.

### 3.3.2 Testing

Testing can be performed in two ways: black box testing and white box testing [ZHM97]. The main idea underlying black box testing is to generate test cases from the specification, regardless of the implementation being tested. An interesting approach which could be used to test COTS software is based on the specification of behavioral properties using state transition systems followed by the generation of test suites from this model [FJJV97, FJJV96, FvBK+91, JPP+97].

The main advantage of this approach is the possibility to focus on the events of interest, those that indicate malicious behavior. Besides, more complex malicious behaviors could be checked using temporal logic to express them. Furthermore, some security properties are likely to be checked exhaustively.

The other approach is white box testing. In this approach the main idea is to devise test suites in order to respect some coverage strategy [ZHM97]. This coverage strategy is based on the structure of
the code. White box testing could be used if we were given trace-based specifications of the malicious behavior.

### 3.3.3 Injecting Faults

Fault injection is a useful method in elaborating reliable code [VMKV97]. It may be used to reveal malicious code fragments, thanks to extensive experiments that consist in producing anomalous behaviors of COTS components. It uncovers potential security vulnerabilities during the development cycle.

### 3.3.4 Wrapping

The term wrapper is relatively new in this domain. Accordingly, many definitions exist that are slightly different. Adams defines a wrapper as code that insulates programs from each other [Ada98]. Voas defines a wrapper as a software encasing that surrounds a component and limits what the component can do [Voa98].

If we use this last definition as a working definition, it follows that a wrapper can either capture inputs or outputs to verify that they are acceptable by the system, or by the wanted security specifications. As we can see in Figure 1, a wrapper can be added to any COTS component in a home-designed system. One of the biggest problems of wrappers is the fact that they need to be built on a one-to-one basis, according to the known interface of the COTS component. This means that a wrapper cannot protect against trap doors or covert channels that do not go through the interface.

Ergo, a wrapper would be used as a last recourse on a flawed product in which we have identified security risks, or bugs, that are accessible through the interface. This situation can realistically occur in real life because a perfect product is most often a myth. However, wrappers, as they are currently defined, are useless to protect against more clever kinds of malicious code.

Another use for wrappers is when a COTS product has too many unneeded functionalities. In this case, it is natural to use a wrapper to provide access to the needed functionalities only.

### 3.4 Certification

Certification is the process that aims to guarantee that COTS components are compliant with the functionality and security requirements. It can be addressed statically through conventional formal methods or dynamically through so-called proof-carrying code.

#### 3.4.1 Formal Verification

When the COTS components’ sources are available, either through decompilation or by the software provider, they can be reverse-reengineered into concrete specifications and then to abstract specifications, thanks to suitable abstraction techniques. These abstract specifications can be subjected to formal verification by using either deductive techniques (theorem proving) or semantic techniques (model-checking). The invariants should reflect the security properties that we would like the COTS component to fulfill.

#### 3.4.2 Proof-Carrying Code

The Proof-Carrying Code mechanism (PCC) allows a host to determine with certainty that a program supplied by an untrusted source can be safely executed [Nec97]. The particularity of this approach is that the producer supplies with the code a proof attesting that the code respects a formally defined security policy, publicized by the consumer. Once he gets the code with the proof, the host can validate the proof in order to execute the code safely.

Figure 2, taken from [Nec97], shows the typical process of generating and using proof-carrying code.

First of all, the consumer must define the security policy. All producers who want to send code to this consumer must satisfy this policy. In the next stage, called certification, the producer compiles its code and generates a proof that his source code respects the security policy. This proof is mixed with the object code to form the so-called binary PCC. In the next stage, called validation, the consumer validates the proof. Finally, the consumer executes the code safely.
Figure 1: A basic diagram for a wrapper

Figure 2: PCC’s architecture
3.5 Ad hoc Techniques

When the subject of malicious software code comes up, one generally thinks of its most common manifestation, namely the computer virus. A variety of anti-virus tools are available to help the user fix infected programs. These tools use a wide range of techniques to detect, identify, and, if possible, remove viruses. We refer to these techniques as ad hoc techniques. Their study can be useful for future research on more general malicious code detection.

Generally speaking, there are two basic methods to detect viruses: specific and generic.

By specific, we refer to methods, or anti-virus software, that use some pre-defined information, such as signatures, about the viruses to be detected. The class of tools that use signature-based analysis are known as scanners. In signature scanning, an executable is searched for selected binary code sequences, called virus signatures, which are unique to a particular virus, or a family of viruses. The virus signatures are generated by examining samples of the virus. Scanners are limited intrinsically to the detection of known viruses. Scanners also perform a more limited role as identification tools. They are primarily used to detect if an executable contains virus code, but they can also be used to detect resident viruses by scanning memory instead of executables. Single-point scanners add the concept of relative position to the virus signature. Here, the code sequence is expected at a particular position within a file. It may not even be detected if the position is wrong. As a result, these scanners can be more accurate than blind scanning without position.

Generic detection methods are based on universal characteristics of viruses, so they can detect more types of viruses, including the unknown ones. Instead of searching for a particular virus signature, behavior-based analysis [NN94b, NN94a] looks for more complex and virus-like behaviors. When such a pattern is found, the analyzer indicates to the user a possible infection. Since heuristic binary analyzers only flag potential infections, they rely heavily on the users' knowledge of the computer system to determine if a file is actually infected [Vel95].

4 Conclusion

As we have seen, program analysis can either be static, dynamic, or use some other kind of technique that can not clearly be classified as one or the other. In this section, we take a closer look at the advantages and disadvantages of static analysis over dynamic analysis.

First, using static analysis allows the detection of malicious code without actually running the program. Therefore, the malices discovered will never be executed. Also, static analysis gives a relatively precise idea of the program's behavior, for all possible executions. On the performance side, there is no overhead associated with a static analysis. After just one analysis, the program can run freely. In spite of these beneficial properties, there are some inconveniences. The main problem in using static analysis comes from the undecidability of many interesting properties. Finally, there is the problem that the analyzed code need not be the one that is actually run; some changes could be made between analysis and execution. Static analysis of source code is particularly vulnerable to this last problem because it needs to be compiled. For example, there is the possibility that a malevolent entity will not modify the source code directly but will rather modify the language libraries so that the changes are not apparent.

Dynamic analysis has, basically, the reversed pros and cons. Dynamically, you can not detect malicious code before execution, give or take a few commands. For example, imagine a five-instruction sequence that globally forms a malicious code. We could keep track of the last five instructions, or a list of suspicious instructions, and stop the execution of the fifth command. However, this method could be rather limiting on its own, because of the lack of a more global view. On the other hand, dynamic analysis does not suffer from the same undecidability problems as static analysis because all run-time values are, or can be made, available at some point in the program. Although dynamic analysis can have significant overhead in run-time performance when compared to static analysis, it has in the end one major advantage in that the analyzed code is the one that is actually run, without any further alterations.

Even if some detection techniques can not be
clearly defined as static or dynamic, most of them usually belong to one class or the other. Also, it could be argued that the methods we have termed *ad hoc* techniques have some kind of static or dynamic properties that would allow us to classify them in one category or the other.

Furthermore, as we have said earlier, some innovative techniques use clearly hybrid analysis. For example, Colby [Col96] proposes a way to statically define guards for loop expressions and see if they can be proven right statically, and if not, suggests to insert dynamic guards that would be checked at run-time, when the Boolean value of the expression can be computed.

So, it seems clear that some of the proposed tools, both in the static and dynamic sections of this document, could very well be combined to insure a much better rate of malicious code discovery. For example, we could do all that is possible with static analysis, identifying all its shortcomings precisely, and then use dynamic analysis to try to eliminate them. For example, we could pinpoint areas of code where we know (or find out) that static analysis will fail and then concentrate on these segments afterwards, using a dynamic method. Thus, we could greatly alleviate the overhead caused by any dynamic process that runs on top of a program. This would allow a greater surveillance to be made on an untrusted program without going over a tolerable level of intrusion.

References


