Static Detection of Malicious Code in Executable Programs *

J. Bergeron, M. Debbabi, J. Desharnais,
M. M. Erhioui, Y. Lavoie and N. Tawbi
LSFM Research Group
Département d'informatique
Université Laval
Québec, QC, G1K 7P4, Canada
{Jean.Bergeron, Mourad.Debbabi, Jules.Desharnais, Mourad.Erhioui, Yvan.Lavoie, Nadia.Tawbi}@ift.ulaval.ca

ABSTRACT

In this paper, we propose a new approach for the static detection of malicious code in executable programs. Our approach rests on a semantic analysis based on behaviour that even makes possible the detection of unknown malicious code. This analysis is carried out directly on binary code. Static analysis offers techniques for predicting properties of the behaviour of programs without running them. The static analysis of a binary executable is achieved in three major steps: construction of an intermediate representation, flow-based analysis that captures security-oriented program behaviour, and static verification of critical behaviours against security policies (model checking).

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: General—*Security and protection (e.g., firewalls)*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Model checking*; D.4.6 [**Operating Systems**]: Security and Protection—*Invasive software (e.g., viruses, worms, Trojan horses)*; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*Invasive software (e.g., viruses, worms, Trojan horses)*

General Terms

Security

Keywords

Detection of malicious code, binary code, static analysis, flow-based analysis, security policies, model checking

1. MOTIVATION AND BACKGROUND

With the advent and the rising popularity of networks, Internet, intranets and distributed systems, security is becoming one of the focal points of research. As a matter of fact, more and more people are concerned with malicious code that could exist in software products. Malicious codes are pieces of code that can affect the secrecy, the integrity, the data and control flow, and the functionality of a system. Therefore, their detection is a major concern within the computer science community as well as within the user community. As malicious code can affect the data and control flow of a program, static flow analysis may naturally be helpful as part of the detection process.

In this paper, we address the problem of static detection of malicious code in binary executables. The reason for targeting binary executables is that the source code of those programs where we need to detect malicious code is often not available. The primary objective of this research initiative is to elaborate practical methods and tools with robust theoretical foundations for the static detection of malicious code. The rest of the paper is organized in the following way. Section 2 is devoted to a comparison of static and dynamic approaches. Section 3 presents our approach to the detection of malices in binary executable code. Section 4 discusses the implementation of our approach. Finally, a few remarks and a discussion of future research are ultimately sketched as a conclusion in Section 5.

2. STATIC VS DYNAMIC ANALYSIS

There are two main approaches for the detection of malices : *static analysis* and *dynamic analysis*. Static analysis consists in examining the code of programs to determine properties of the dynamic execution of these programs without running them. This technique has been used extensively in the past by compiler developers to carry out various analyses and transformations aiming at optimizing the code [10]. Static analysis is also used in reverse engineering of software systems and for program understanding [3, 4]. Its use for the detection of malicious code is fairly recent.

Dynamic analysis mainly consists in monitoring the execution of a program to detect malicious behaviour.

Static analysis has the following three advantages over dynamic analysis:

- Static analysis techniques allow exhaustive analysis, because they are not bound to a specific execution of a

*This research is jointly funded by a grant from the Natural Sciences and Engineering Research Council (NSERC), Canada and by a research contract from the Defence Research Establishment (DREV), Valcartier, 2459, Pie XI Nord, Val-Bélair, QC, Canada, G3J 1X5

program and can give guarantees that apply to *all* executions of the program. In contrast, dynamic analysis techniques only allow examination of behaviours that correspond to selected test cases.

- A verdict can be given before execution, where it may be difficult to determine the proper action to take in the presence of malices.

- There is no run-time overhead.

However, it may be impossible to certify statically that certain properties hold (e.g., due to undecidability). In this case, dynamic monitoring may be the only solution.

Thus, static analysis and dynamic analysis are complementary. Static analysis can be used first, and properties that cannot be asserted statically can be monitored dynamically.

As mentioned in the introduction, in this paper, we are concerned with static analysis techniques. Not much has been published about their use for the detection of malicious code. In [8], the authors propose a method for statically detecting malicious code in C programs. Their method is based on so-called *tell-tale signs*, which are program properties that allow one to distinguish between malicious and benign programs. The authors combine the tell-tale sign approach with program slicing in order to extract from large programs small fragments that can easily be analyzed.

3. DESCRIPTION OF THE APPROACH

Static analysis techniques are generally used to operate on source code. However, as we explained in the introduction, we need to apply them to binary code, and thus, we had to adapt and evolve these techniques. Our approach is structured in three major steps: Firstly, the binary code is translated into an internal intermediate form (see Section 3.1) ; secondly, this intermediate form is abstracted through flow-based analysis as various relevant graphs (control-flow graph, data-flow graph, call graph, critical-API [1] graph, etc.) (Section 3.2); the third step is the static verification and consists in checking these graphs against security policies (Section 3.3).

3.1 Intermediate Representation

A binary executable is the machine code version of a high-level or assembly program that has been compiled (or assembled) and linked for a particular platform and operating system. The general format of binary executables varies widely among operating systems. For example, the Portable Executable format (PE) is used by the Windows NT/98/95 operating system. The PE format includes comprehensive information about the different sections of the program that form the main part of the file, including the following segments:

- .text, which contains the code and the entry point of the application,

- .data, which contains various types of data,

[1] API: Application Program Interface.

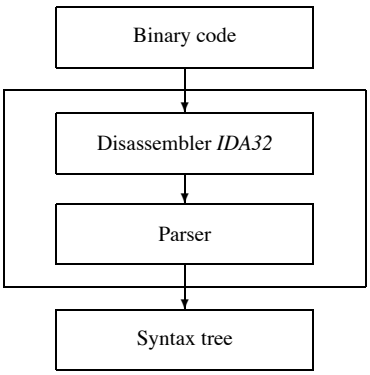

Figure 1: Generating the intermediate representation

- .idata and .edata, which contain respectively the list of imported and exported APIs for an application or a Dynamic-Linking Library (DLL).

The code segment (.text) constitutes the main part of the file; in fact, this section contains all the code that is to be analyzed.

In order to translate an executable program into an equivalent high-level-language program, we use the disassembly tool IDA32 Pro [7], which can disassemble various types of executable files (ELF, EXE, PE, etc.) for several processors and operating systems (Windows 98, Windows NT, etc.). Also, IDA32 automatically recognizes calls to the standard libraries (i.e., API calls) for a long list of compilers.

Statically analyzing a program requires the construction of the syntax tree of this program, also called intermediate representation. The various techniques of static analysis are based on this abstract representation. The goal of the first step is to disassemble the binary code and then to parse the assembly code thus generated to produce the syntax tree (Figure 1).

3.2 Flow-based Analysis

The aim of flow-based analysis is to generate informations about how control and data flow from one program point to another (Figure 2). The ultimate goal is to obtain an abstraction of the program behaviour. Control and data flow information is extracted from the intermediate form and represented as control and data-flow graphs.

- A control-flow graph is a directed graph where each node corresponds to a statement or block of statements of the program. An edge between two nodes represents a direct flow of control between them. A control flow-graph can be intra-procedural or interprocedural. In the first case, a procedure call is considered as an ordinary statement and the whole control-flow information consists of a set of control-flow graphs corresponding to the different procedures without any link between them. In the second case the control-flow graph links each call site to the corresponding procedure control-flow graph and each return to the point following immediately the call site.

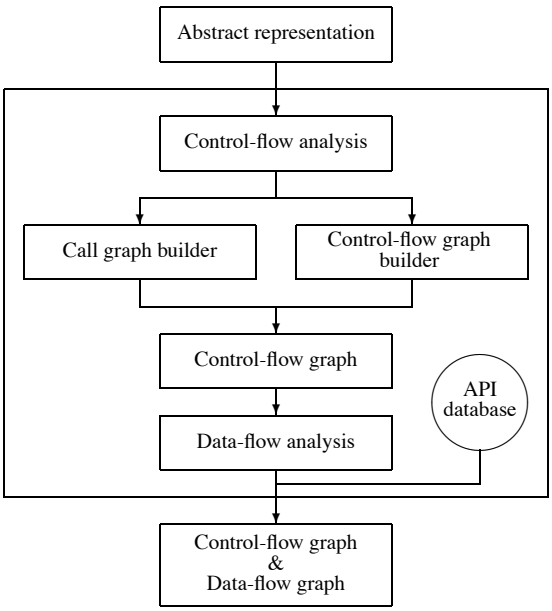

Figure 2: Flow-based Analysis

- A data-flow graph is a directed graph whose nodes correspond to operations in the program and whose edges represent various types of information flow between the nodes. This means that many different types of data-flow graphs are possible.

In the first step, our control-flow graph generator constructs a control-flow graph for each procedure, and then it links each call site to the control-flow graph of the callee by two specific edges, namely a call edge and a return edge. The call edge links the call site to the first statement or block of the procedure and the return edge links the exit statement or block of the procedure to the program point that immediately follows the call site (see the top graph in Figure 3).

In our implementation, the nodes of the control-flow graph correspond to basic blocks, which are sequences of statements such that control starts at the first statement and leaves after the last statement with no possibility of halting or branching within the block. Using basic blocks as nodes is more appropriate than using single statements, since the number of statements in assembly code tends to be fairly large when compared to their high-level-language counterpart.

The inter-procedural flow graph is then abstracted in an API-graph (middle graph in Figure 3). In this graph, all the computation statements are ignored in the control-flow graph and the statements representing calls to APIs are kept. This API graph is then abstracted into a critical-API graph call (bottom graph in Figure 3), in which only calls to APIs that could compromise security are represented. The latter graph is then used in the verification step. Notice that the criticality of an API depends on the context and the application. This is why our implementation allows the user to determine the list of APIs that he or she considers relevant to the verification process. As an example, if a program reads a file and then sends its content over the net and if the APIs that access files are the only relevant APIs, then the critical-API graph shows that a read is followed by a write and ignores what happens between these two operations.

Static analysis can yield more precise information if data-flow information is extracted from the program. The situation depicted in the previous example would be considered as dangerous from a security point of view whatever the files read and sent over the net. A data-flow analysis can tell whether the data sent over the net actually depends on the information that is read in the confidential file, making the analysis and the warning messages more precise. In this case, a superfluous warning could be avoided if the data sent do not depend on information extracted from the confidential file. For instance, no warning is needed in the case of Figure 3, because the critical file that is read (ConfidFile) is not the one sent over the network (OtherFile).

By applying data-flow analysis, we can improve the analyzability of the assembly code. For example, if actual parameters and return values for different APIs and library subroutine calls involved in the program are computed, then it is possible to determine the kind of information transmitted through the network. As another example, consider the subroutine at instruction 2 in Figure 3: by applying data-flow analysis on registers, the actual parameters and the return value of the subroutine can be determined.

In a similar way, we can compute actual parameters for different API calls made by the program. The goal of these transformations is to get an imperative high-level representation which is more suitable for flow analysis. For example, the block

```
push var
push esi
push eax
push 0
call send
```

may be transformed as follows:

```
call send(var, esi, eax, 0).
```

3.3 Static Verification

A security policy is a set of rules that characterizes acceptable or unacceptable executions of a program. It might concern

- access control, and restrict what operations can be performed on objects;
- information flow, and restrict what can be inferred about objects from the observation of system behaviour;
- availability, and restrict principals from denying others the use of a resource.

The intent of the third step of our approach is twofold:

```
      0:  lea ebx, eax
      1:  mov ecx, ebx
      2:  call subA(ecx)
      3:  test ecx, ecx
      4:  jz short loc_7
      5:  call subB(ebx)
      6:  jmp loc_8
 loc_7:  mov esi, var3
 loc_8:  call MessageBox
      9:  call Send (var1, esi)

subA proc near
varx dword ptr 4
     10:  Call OpenFile (”security.txt”)
     11:  mov ebx, eax
     12:  Call ReadFile (ebx, varx)
     13:  ret

subB proc near
     14:  Call RegOpenKey
     15:  mov ebx, eax
     16:  Call RegSetValue
     17:  add edi, eax
     18:  Call RegCloseKey
     19:  ret
```

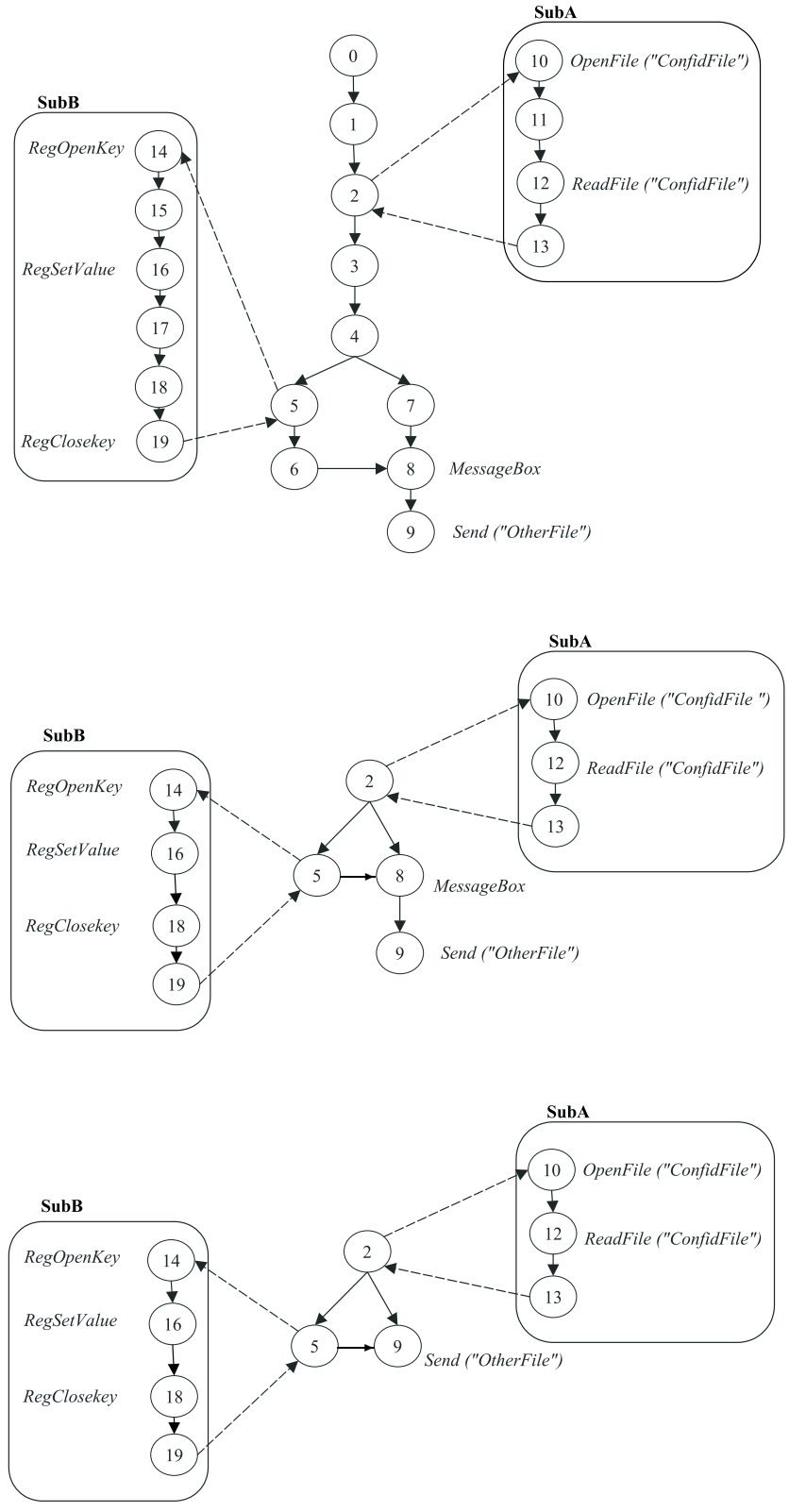

Figure 3: A fragment of code and its control-flow graph, API graph, and critical-API graph

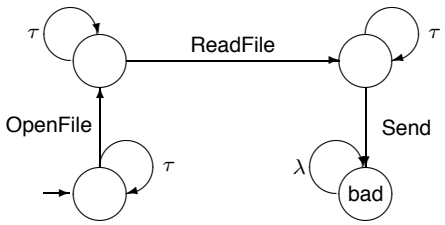

Figure 4: Security automaton

1. Expressing security policies: In our system, security policies are specified using security automata [11]. The transitions of these automata are labeled by actions that can be performed by the system. One of the states is designated as a bad state and and entry into this state is a violation of the security policy.

   We have chosen security automata because they are very expressive, able to encode any safety property (the labels of the transitions can be arbitrary computable functions). For example, Figure 4 depicts a security automaton that enforces the simple policy that programs must not send anything on the network after reading a file. Entrance into the bad state indicates that the security policy has been violated. In this automaton, τ denotes any action other than the relevant actions OpenFile, ReadFile and Send, whereas λ denotes any action.

2. Checking malicious behaviour against security policies (model checking): Once the critical-API graph is computed, it is subjected to verification against the security policy to statically determine whether it exhibits malicious behaviour or not.

A program is a sequence of instructions that manipulate information: it creates, destroys, reads or writes information, computes new information, etc. The access to this information is done using the resources of a computer by the intermediary of the APIs of the system. By resources, we mean the internal memory, discs, files, networks, processes, clocks, etc. Thus, for example, the API Send is used to send on the network information from a disc or registry. A malicious program is characterized by how it handles critical resources. The security policy indicates if a combination of accesses to the resources is licit or not.

4. IMPLEMENTATION

We have built a prototype for the static detection of malices in binary code using the techniques described above. It takes as input an executable program and a security policy, and produces a report on the presence or absence of malicious code. The user has facilities for selecting a list of critical APIs from the complete list of APIs used in the program. See Figure 6 for a view of the user interface.

As mentioned in Section 3, the process starts with disassembly, using *IDA32*. The assembler program is then parsed to produce the syntax tree. The prototype then constructs the

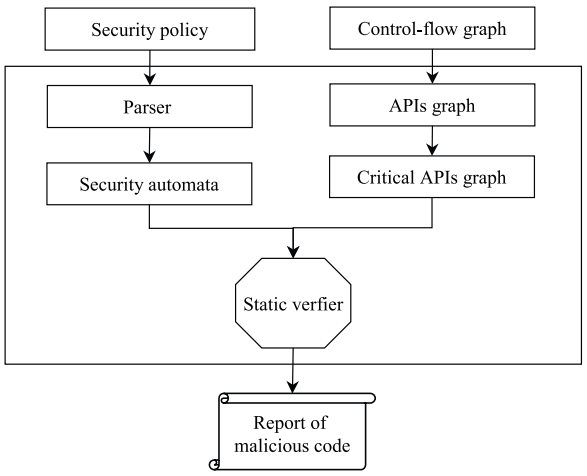

Figure 5: Static verification

control-flow graph for each subroutine by placing basic blocks on a list and then converting that list to a proper graph. While parsing, whenever an instruction marking the end of a basic block is met, the basic block is constructed, and the start and finish instructions are stored in the list of instructions for that subroutine. The control-flow graph is then constructed using standard techniques [1]. The security policy is entered manually as a list of transitions. Finally, the security policy is compared to the control-flow graph using a variant of Emerson's algorithm [5].

As a by-product of this analysis, the tool is also able to detect dead code, which shows up as a disconnected component in the control-flow graph.

We have tested our tool on known malicious programs, such as WINIPX.EXE, produced by the Win32/Semisoft virus [6]. This virus infects only EXE files in the Windows PE format. It contains specific code to infect NOTEPAD.EXE. The virus also creates several files containing just the body of the virus; these are WINIPX.EXE, WINIPXA.EXE, WINSRVC.EXE and EXPLORE.EXE. When the virus is operating, it sends information collected from the host machine to a few Internet addresses. analyzing the file WINIPX.EXE easily reveals that information is sent over the net and that the information sent comes from files on the disk. Figure 7 shows a zoom on the control-flow graph of WINIPX.EXE. In this figure, we see three basic blocks. The sets IN and OUT that appear in these blocks respectively contain the live registers on entry and exit of the blocks. The sets DEF and USE associated to a statement respectively contain the variables that are defined and used by that statement.

5. CONCLUSION

In this paper, we have presented an approach to the static detection of malicious code in executable programs. This is done in three steps: generating an intermediate representation, analyzing the control and data flows, and doing static verification. This last step consists in comparing a security policy to the output of the analysis phase. We have also briefly described

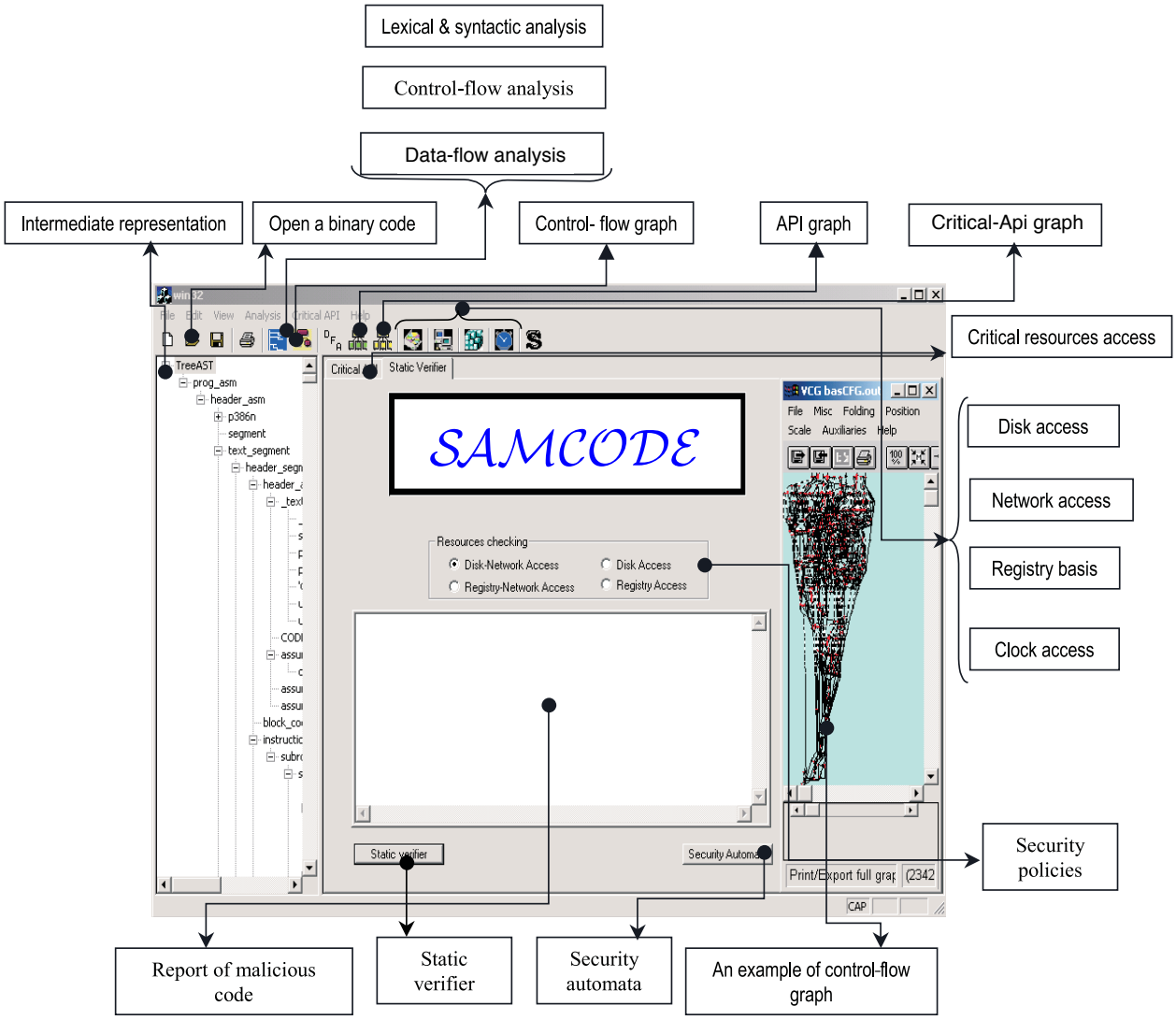

Figure 6: User interface of the prototype tool

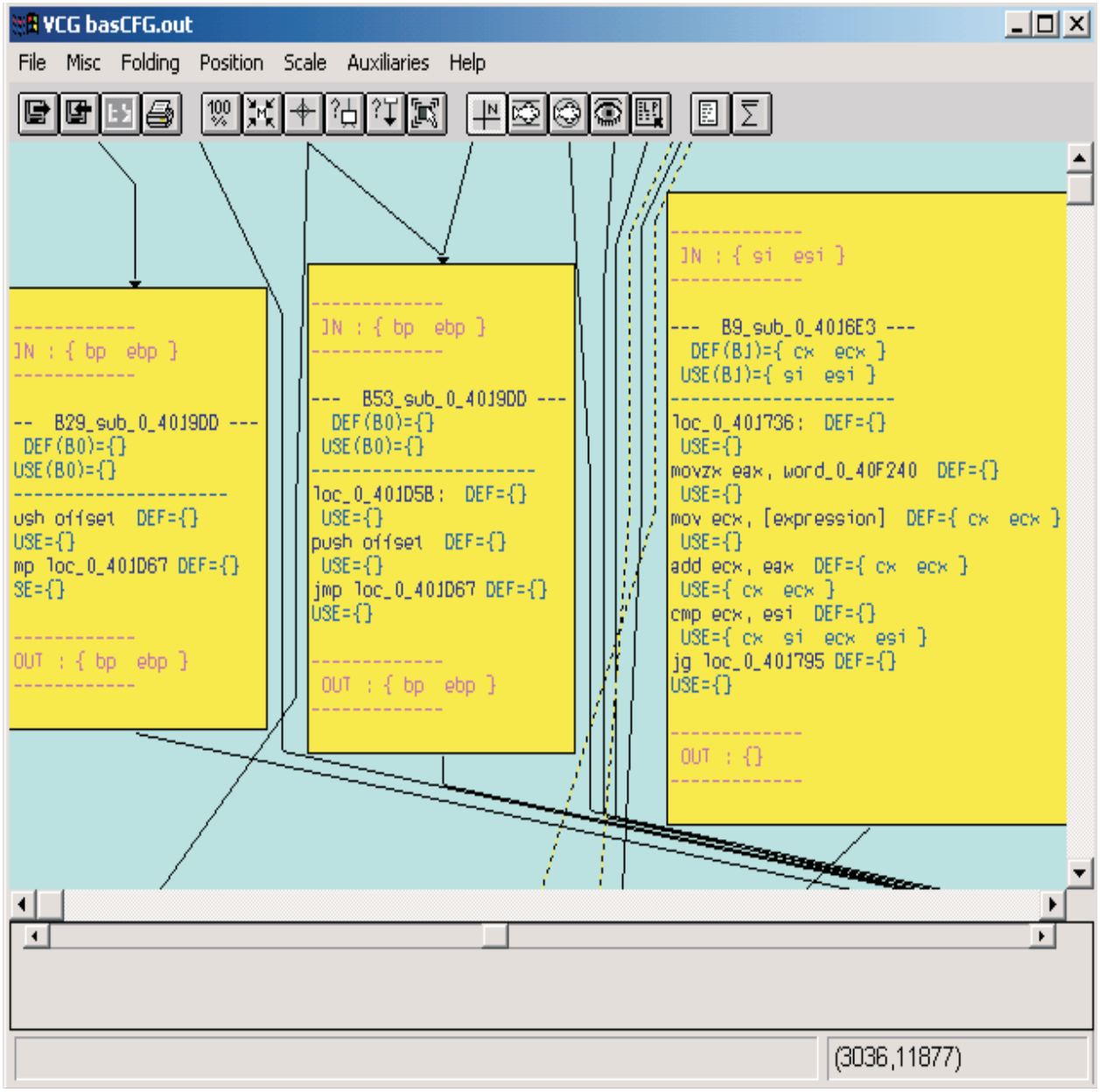

Figure 7: Zoom on a control-flow graph

our prototype tool.

As future work, we plan to

- improve the static verifier to take into account data-flow information,
- extend the analysis to the DLLs used in the program that is analyzed,
- make the tool portable on several platforms (e.g., Linux),
- and define a language for inputting security policies.

6. ACKNOWLEDGEMENTS

We thank the anonymous referees for their comments.

7. REFERENCES

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers – Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] Thomas Ball and Susan Horwitz. Slicing programs with arbitrary control-flow. In *Automated and Algorithmic Debugging, First International Workshop, AADEBUG'93*, volume 749 of *LNCS*, pages 206–222, 1993.

[3] Christina Cifuentes and K. John Gough. Decompilation of binary programs. *Software - Practice and Experience*, 25(7):811–829, July 1995.

[4] Cristina Cifuentes and Antoine Fraboulet. Intraprocedural static slicing of binary executables. In *Proceedings of the International Conference on Software Maintenance*, Bari, Italy, October 1997, pages 188–195, IEEE-CS Press.

[5] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[6] F-Secure Corporation. Semisoft. `http://www.europe.f-secure.com/v-descs/net666.htm`, 1998.

[7] Ilfak Guilfanov. An advanced interactive multi-processor disassembler. `http://www.datarescue.com`, 2000.

[8] Raymond W. Lo, Karl N. Levitt, and Ronald A. Olsson. MCF: A Malicious Code Filter. *Computers and Security*, 14(6):541–566, 1995.

[9] John P. McDermott and William S. Choi. Taxonomy of computer program security flaws. *ACM Computing Surveys*, 26(3):211–254, 1994.

[10] S. S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufman Publishers, San Francisco, CA, 1997.

[11] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, February 2000. Also Cornell University, `http://cs-tr.cs.cornell.edu:80/Dienst/UI/1.0/Display/ncstrl.cornell/TR99-1759`, 1998.