Static Analysis of Binary Code to Isolate Malicious Behaviors*

J. Bergeron & M. Debbabi & M. M. Erhioui & B. Ktari
LSFM Research Group,
Computer Science Department,
Science and Engineering Faculty,
Laval University,
Quebec, Canada
{bergeron,debabi,erhioui,ktari}@ift.ulaval.ca

Abstract

In this paper, we address the problem of static slicing on binary executables for the purposes of the malicious code detection in COTS components. By operating directly on binary code without any assumption on the availability of source code, our approach is realistic and appropriate for the analysis of COTS software products. To be able to reason on such low-level code, we need a suite of program transformations that aim to get a high level imperative representation of the code. The intention is to significantly improve the analysability while preserving the original semantics. Next, we apply slicing techniques to extract those code fragments that are critical from the security standpoint. Finally, these fragments are subjected to verification against behavioral specifications to statically decide whether they exhibit malicious behaviors or not.

1. Motivation and Background

Nowadays, many are the information infrastructures that are based on the so-called commercial off-the-shelf (COTS) components. Actually, many organizations are undergoing a remarkable move from legacy systems towards COTS-based systems. The main motivation underlying such a migration is to take advantage of cutting-edge technologies and also to lower the program life-cycle costs of computer systems. Nevertheless, this migration phenomenon poses severe, and very interesting, challenges to the currently established computer system technologies in terms of security, reliability, integration, interoperability, maintenance, planning, etc.

*This research is jointly funded by a research grant from the Natural Sciences and Engineering Research Council, NSERC, Canada and also by a research contract from the Defense Research Establishment, Valcartier, DREV, Quebec, Canada.

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 COTS software products. Malicious code are fragments of code that can affect the secrecy, the integrity, the data and the control flow, and the functionality of a system. Therefore, their detection is a major concern within the computer science community [2]. As these malicious code can affect the data and control flow of a program, static flow analysis may naturally be required as part of the detection process.

In this paper, we address the problem of static slicing on binary executables for the purposes of the malicious code detection in COTS components. Static slicing is useful to extract those code fragments that are critical from the security standpoint. Once computed, these fragments are subjected to verification against behavioral specifications to statically decide whether they exhibit malicious behaviors or not.

The rest of the paper is organized in the following way. Section 2 reports a methodology for malicious code detection in binary executable code. Section 3 discusses in details different issues regarding the translation of assembly code to a high-level imperative representation. Section 4 is dedicated to the presentation of a slicing algorithm for a binary code which has been previously translated to an abstract representation. Section 5 is devoted to the related work. Finally, a few concluding remarks and a discussion of future research are ultimately sketched as a conclusion in Section 6.

2. Methodology

In this section, we report a methodology for malicious code detection in binary executable code. Our approach is

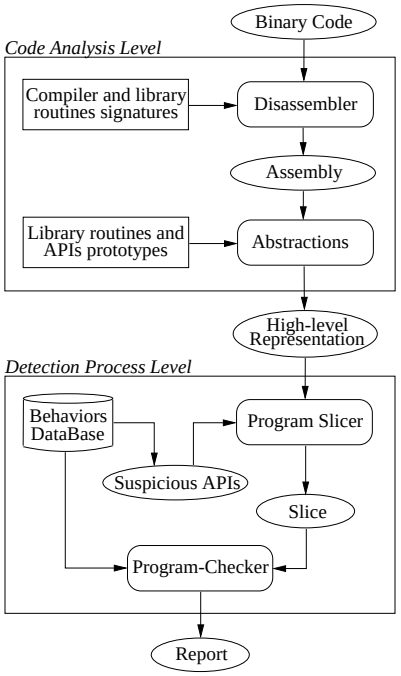

Figure 1. Proposed Architecture.

structured in three major steps. The first step consists in disassembling the binary code, which yields an assembly version of the code. The intent of the second step is twofold: First, we need a suite of program transformations that aim to get a high level imperative representation of the code. The intention is to significantly improve the analysability while preserving the original semantics. As an example of program transformations, we mention stack elimination, recovering of subroutine parameters, recovering of return results, etc. For that, the assembly version is subjected to extensive data and control flow analysis. Second, we aim to decrease the complexity of the detection problem. This is achieved by extracting a potentially malicious fragment instead of considering the entire code. This is achieved by applying slicing techniques. In section 4, we present an algorithm to achieve this goal. The third and last step is dedicated to the detection process and is based on program checking. The overall architecture of our methodology is reported in Figure 1.

3. High-level Imperative Representation

This section is devoted to the presentation of the different transformations, mainly based on flow analysis, required for the translation of the binary code into a more abstract representation.

Before to be able to apply flow analysis, the binary code must be disassembled. In this work, we rely on commercial of the shelf disassemblers since there are excellent ones that are available on the market. Once disassembled, we still need to transform it into a more analyzable form. Many transformations are required:

- Identification of the so-called idioms. An idiom is a sequence of instructions that has a logical meaning which cannot be derived from individual instructions. Actually, in the compilation process, each high-level instruction is transformed into a sequence of low-level (assembly) instructions. The aim of this transformation is not to decompile the assembly code but to decrease the complexity of the detection phase. Moreover, without performing such transformation, unnecessary and cumbersome dependencies between statements will arise during the flow analysis process.

- By applying data-flow analysis we can improve the analysability of the assembly code. For example, we can apply the following transformations to the code:

  – Stackless code: The main disadvantage of analyzing stack-based code is the implicit uses and definitions of stack locations. The elimination of assembly instructions `push` and `pop` from the code allows each statement to refer explicitly to the variables it uses. The elimination is possible by considering the stack as a set of temporary variables. Hence, each implicit reference to the stack is transformed to an explicit reference to a temporary variable. The following example illustrates such transformation:

  ```
  push eax              mov tmp1,eax
  push esi              mov tmp2, esi
  call strcat    ⟶     call strcat
  pop ecx               mov ecx, tmp2
  pop ecx               mov ecx, tmp1
  ```

  This method works as long as the depth of the stack is fixed for every program point throughout the execution of the program. This is guaranteed during the compilation process. The situation could be different if the binary code is obtained directly from an assembly source code. In this case, the depth of the stack could be different at some program point depending on the execution of the program. Figure 2 illustrates an example of stack elimination when the depth is not fixed. The solution is to transfer blocks in order to avoid multiple branches in the control flow graph. In this figure, block $B3$ is transferred to both $B1$ and $B2$. By doing so recursively, we can ensure that the depth is fixed at each point in the control flow graph.

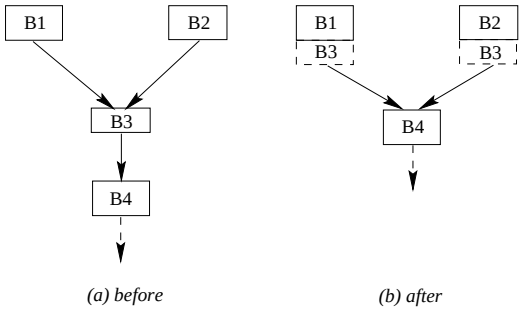

Figure 2. Stack Elimination Example.

– Parameters and return values: in this case, the purpose is to compute actual parameters and return values of different subroutine calls. For example, by applying data-flow analysis on registers, we can identify actual parameters and the return value of a subroutine call in the program:

mov `eax`, **call** `subA13(ebx,eax,0)`

- With APIs and library subroutine prototypes, we can compute actual parameters and return values for the different API and library subroutine calls involved in the program. For example, we can transform the instructions presented below into the following one:

mov `eax`, **call** `strcat(tmp2,tmp1)`

- When a jump or a call on a register is met, it is not possible to determine statically the target addresses. Calls on register are used in high-level languages to implement pointers to function invocation. Also, different compilers implement the high-level instruction `case` by using an indexed table that holds different target label addresses. In both cases, jump on register are used. To solve this problem, intraprocedural slicing can be performed on the register variable to obtain the set of instructions the register depends on [4]. The analysis of these instructions reveals the different addresses that are involved.

As we mentioned previously, the rationale underlying these transformations is to get an imperative high level representation that is more suitable for flow analysis.

4. Slicing Algorithm

One of the part of the detection process level in the architecture presented in Figure 1 consists to slice the abstract representation of the program to retain only the relevant instructions, especially the API calls, that influence the value of some registers. Program slicing algorithm uses both control and data-flow graphs. Given a pair $C \equiv (s, V)$, called the slicing criterion, where s is a node in the control-flow graph and V a subset of the program's variables, it produces a set S_C of program instructions that are relevant for the computation of the variables in V. The set S_C is called a slice.

By doing so, we focus our analysis only on some interesting statements rather than the entire program. Moreover, program slicing keeps only those statements that are relevant to the suspicious one i.e. the one specified in the slicing criterion.

In the presence of subroutines, pointers and unconditional jumps (`JMP`), the standard backward slicing algorithms must be adapted. For this purpose, we attempt to combine and accommodate different techniques to deal with each of these cases.

The algorithm presented in this section uses the system dependence graph (SDG) of a program which consists on a collection of procedure dependence graphs (PDGs), one for each subroutine in the program, connected by interprocedural control and data dependence edges. Each PDGs contains vertices, which represent the different instructions of a subroutine, and edges which represent data and control dependencies between these vertices. The data dependencies part of this graph corresponds to the data dependence graph (DDG) while the control dependencies part corresponds to the control dependence graph (CDG).

The presence of pointers requires the DDG to be augmented. Intraprocedural aliasing is computed by applying an iterative technique [3]. This computation requires three components:

- The aliases holding at entry node of the SEG (Sparse Evaluation Graph) which is interprocedurally computed.
- The aliases holding immediately after the call-site which is interprocedurally computed.
- The transfer function for each node that contains a pointer assignment.

The transfer function describes the data flow effect of nodes in the CFG.

To compute interprocedural aliases, we use the interprocedural algorithm proposed in [3]. The method to compute aliases information is based on the procedure call graph (PCG). The alias information at the entry node of a procedure P is computed by merging the intraprocedural aliases at the call sites which invoke P and then propagating the resulting set into the entry node of P. The interprocedural algorithm traverse the PCG in a topological order until each aliasing set converges.

Once the aliases sets are computed, we can improve the data dependence graph of each subroutines in the program

by taking into account the may-aliases relations. This can be done as follows:

A-DDG(S_i,S_j) $\Leftrightarrow$ S_i defines a variable X,
S_j uses a variable X',
X and X' are may-aliases.

The following example illustrates the utility of an alias analysis:

```
1: mov tmp, 2
2: lea eax, tmp
3: mov ebx, eax
4: mov [eax], 4
5: call putchar( [ebx] )
```

Without alias analysis, the instruction 5 and the instruction 3 are connected in the DDG. But after an alias analysis of the code, it appear that register `ebx` and register `eax` are aliases. Thus, the instruction 5 is connected to the instruction 4 in the DDG as the later change the content of both registers.

The presence of unconditional jumps requires the CDG to be augmented. More precisely, the CDG is constructed from an augmented flow graph of the program in which new edges are added from the nodes representing jumps statements to the nodes that immediately lexically succeed them in the program [1]. By doing so, unconditional jumps could be correctly included by the conventional slicing algorithm.

Once the augmented CDG and DDG are correctly computed, the PDGs of each subroutines of the program is generated by merging its two corresponding graphs. In addition, each call statement in a subroutine is represented in its corresponding PDGs by a call vertex and two kinds of vertices, actual-in and actual-out, which are control dependent on it. Similarly, each subroutine entry is represented using an entry vertex and two kinds of vertices, formal-in and formal-out, which are control dependent on it. Actual-in and formal-in vertices, and formal-in and formal-out vertices are included respectively for every parameter that may be used or modified and for every parameter that may be modified by the called subroutine.

The SDG is obtained by connecting the different PDGs of a program. More precisely, each subroutine call in a PDGs is connected with the corresponding PDGs of the called subroutine. For this purpose, three kinds of interprocedural edges are used: (1) call edges which are used to connect each call vertex to the corresponding subroutine entry vertex; (2) parameter-in edges which are used to connect each actual-in vertex at the call site to the corresponding formal-in vertex in the called subroutine; (3) and parameter-out edges which are used to connect each formal-out vertex in the called subroutine to the corresponding actual-out vertex in the call site.

To consider calling context of a called subroutine, the SDG is augmented with a particular kind of edges, summary edges, which represent transitive dependencies between actual-in and actual-out vertices due to the effects of subroutine calls.

Table 1 presents the slicing algorithm of a program, which corresponds to the algorithm proposed in [6]. This algorithm is computed using two passes over the SDG. In the first pass, the algorithm starts from some specific vertices, and goes backwards along flow edges, control edges, summary edges and parameter-in edges, but not along parameter-out edges. Summary edges permit to move across a call site without having to descend into the called subroutine. In the second pass, the algorithm starts from all the vertex reached in the first pass, and goes backward along flow edges, control edges, summary edges and parameter-out, but not along call or parameter-in edges. The result of the algorithm is the sets of vertices reached during both passes.

The following example illustrates how the slicing is useful to extract potentially malicious fragment of code. In Table 2, we propose a fragment of a high-level representation of a disassembled code. Table 3 presents the slice obtaining by slicing the fragment of code from location 15 and the variable `esi`. As shown in this table, and more precisely in the figure, the result of the analysis reveals that the information sent on the Net comes from a specific file named `''security.txt''`. Through slicing techniques, we drew the conclusion that the program has a fragment of code where some sensitive[1] information is transmitted over the network.

5. Related Work

Very few has been published about the static slicing of binary executables. In [4], the authors propose an intraprocedural static slicing approach to be applied to binary executables for the purposes of determining the instructions that affect an indexed jump or an indirect call on a register. They identify the different changes required to apply conventional slicing techniques on binary code. Data dependencies are determined using use-definition chains on registers and condition codes. However, no alias analysis are performed.

As related work, we can also cite all what have been done in reverse engineering of software systems and program understanding. More specifically, we distinguish the decompilation research whose aim is to translate an executable program into an equivalent high-level language program [5]. Our work is close to this research. In fact, many tools used

[1] We suppose that all the information contained in the file `''security.txt''` are security critical.

`RetReachingNodes(`$G, l, kinds$`):`
return a set of nodes from G that can reach a given set of nodes l along certain $kinds$ of edges.
`SelectAndRemove(`l`):` select and remove an element from l.
`Add(`n, l`):` add node n to the list l.
`RetDepNotKindAndNotReaching(`$G, n, kinds, l$`):`
return the nodes in G that can reach node n and that are not in l and are not of kind $kinds$.

```
Procedure Slicing ( P, s )
begin
   sdg = SDG( P )
   l = RetReachingNodes( sdg, {s}, {param-out} )
   l' = RetReachingNodes( sdg, l, {param-in,call} )
   return ( l' )
end

Procedure RetReachingNodes ( G, l, kinds )
begin
   wlist = l
   rlist = ∅
   while (wlist ≠ ∅) do
      n = SelectAndRemove( wlist )
      Add(n,rlist)
      dlist = RetDepNotKindAndNotReaching( G, n, kinds, rlist )
      wlist = wlist ∪ dlist
   od
   return ( rlist )
end
```

Table 1. Slicing Algorithm.

```
 0: lea var3, var2
 1: mov tmp1, 7
 2: mov tmp2, 4
 3: lea edx, tmp2
 4: lea ebx, tmp1
 5: mov ebx, edx
 6: mov tmp1, [ebx]
 7: mov tmp3, var3
 8: mov eax, ebx
 9: call subA(tmp3)
10: test tmp3, tmp3
11: jz 14
12: mov tmp4, edx
13: jmp 15
14: mov esi, var3
15: mov eax, call Send (var1, esi)
```

```
subA proc near
  varx dword ptr 4
20: mov eax, Call OpenFile ("security.txt")
21: mov ebx, eax
22: mov tmp1, ecx
23: mov eax, Call ReadFile (ebx, varx)
24: mov ecx, eax
25: mov eax, Call CloseFile (ebx)
26: mov ecx, eax
27: ret
```

Table 2. Fragment of a high-level representation of a disassembled code.

```
 0: lea var3, var2
 7: mov tmp3, var3
 9: call subA(tmp3)
10: test tmp3, tmp3
11: jz 14
13: jmp 15
14: mov esi, var3
15: mov eax, call Send (var1, esi)

subA proc near
 varx dword ptr 4
20: mov eax, Call OpenFile ("security.txt")
21: mov ebx, eax
23: mov eax, Call ReadFile (ebx, varx)
27: ret
```

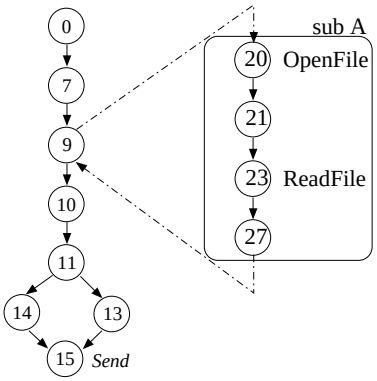

Table 3. Slicing Result with its CFG representation.

by them are also required in our work. For example, loaders, disassemblers, signature generator, etc. are tools that are necessary for the analysis of binary code and its translation to a more abstract representation.

Finally, in [7], the authors propose a method for statically detecting malicious code in C programs. Their method is based on the so-called tell-tale signs which are program properties that allow one to distinguish between malicious code and benign programs. The authors combine the tell-tale sign approach with program slicing in order to produce small fragments of large programs that could be easily analyzed.

6. Conclusion

This work is part of our research on the malicious code detection in COTS components. In this research, we have reported a methodology that is structured in three major steps. The first step consists in disassembling the binary code, which yields an assembly version of the code. The intent of the second step is twofold: First, we need a suite of program transformation that aim to get a high level imperative representation of the code. The intention is to significantly improve the analysability while preserving the original semantics. Second, we aim to decrease the complexity of the detection problem. This is achieved by applying slicing techniques. Moreover, slicing techniques allow us to extract those code fragments that are critical from the security standpoint. Finally, these fragments are subjected to verification against behavioral specifications to statically decide whether they exhibit malicious behaviors or not.

As future work, we plan to put the emphasis on the elaboration of techniques to achieve efficient and practical program checking of potentially malicious slices against behavioral specifications. Furthermore, we hope to come up with practical tools that address the automatic detection of malicious code in COTS.

References

[1] T. Ball and S. 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.

[2] J. Bergeron, M. Debbabi, J. Desharnais, B. Ktari, M. Salois, and N. Tawbi. Detection of Malicious Code in COTS Software: A Short Survey. In *First International Software Assurance Certification Conference (ISACC'99)*, Washington D.C, Mar. 1999.

[3] J.-D. Choi, M. Burke, and P. Carini. Efficient Flow-sensitive Interprocedural Computation of Pointer-Induced Aliases and Side Effects. In *Conference Record of the 20th ACM Symposium on Principles of Programming Languages*, pages 232–245, 1993.

[4] C. Cifuentes and A. Fraboulet. Intraprocedural Static Slicing of Binary Executables. In I.-C. Press, editor, *Proceedings of the International Conference on Software Maintenance*, pages 188–195, Bari, Italy, Oct. 1997.

[5] C. Cifuentes and K. J. Gough. Decompilation of Binary Programs. *Software - Practice and Experience*, 25(7):811–829, July 1995.

[6] S. Horwitz, T. Reps, and D. Binkley. Interprocedural Slicing using Dependence Graphs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 23, pages 25–46, Atlanta, Georgia, June 1988.

[7] R. W. Lo, K. N. Levitt, and R. A. Olsson. MCF: A Malicious Code Filter. *Computers and Security*, 14(6):541–566, 1995.

[8] F. Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.

[9] M. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.