RECOGNITION OF REGISTER REALIGNMENT BY SCATTERED CONTEXT GRAMMARS

Lukáš Ďurfina
Doctoral Degree Programme (1), FIT BUT
E-mail: idurfina@fit.vutbr.cz

Supervised by: Dušan Kolář
E-mail: kolar@fit.vutbr.cz

Abstract: The paper introduces a way how scattered context grammars can be build and used for recognizing a register realignment. The register realignment is a method of adjusting an executable file on a binary level. Scattered context grammar is based on human analysis of code. Such an analysis includes exploration of bytes affected by realignment and finding new valid values for them. The created grammar has LL property – an ability of parsing this type of grammar.

Keywords: scattered context grammar, register realignment, malware analysis

1 INTRODUCTION

Register realignment is basic technique of binary obfuscation and it is usually executed by a metamorphic engine. It is exploited by e.g. malware writers to fool and discard pattern matching, what is one of the main methods for searching the malware and it is used by majority of anti-virus software. This adjustment has to preserve a context, therefore a formal model, which is able to describe context languages, could be appropriate for describing it. LL scattered context grammars were chosen, because they can generate context languages and we can build parser for them [1].

Section 2 introduces primary knowledge about formal languages. In Section 3, we describe closely the register realignment. Its purpose and a reason why it is useful to recognize it is shown in Section 4. The idea for solving the problem is also part of this section. The following Section 5 presents an example and, finally, there is a conclusion.

2 PRELIMINARIES AND DEFINITIONS

It is expected that reader is familiar with formal models and with formal language theory [2].

Definition 1 (Context Free Grammars) A *context free grammar* (CFG) is a quadruple $G = (N, T, P, S)$, where N is a finite set of nonterminals, T is a finite set of terminals, $S \in N$ is the starting nonteminal, and P is a finite set of rules of the form $A \to w$, where $A \in N$ and $w \in (N \cup T)^*$.

Definition 2 (Scattered Context Grammars) A *scattered context grammar* (SCG) is a quadruple $G = (N, T, P, S)$, where N is a finite set of nonterminals, T is a finite set of terminals, $S \in N$ is the starting nonterminal, and P is a finite set of rules of the form $(A_1, A_2, \ldots, A_n) \to (w_1, w_2, \ldots, w_n)$, for some $n \geq 1$, where $A_i \in N$ and $w_i \in (N \cup T)^*$. If $(A_1, A_2, \ldots, A_n) \to (x_1, x_2, \ldots, x_n) \in P$, $u = u_1A_1u_2 \ldots u_nA_nu_{n+1}$, and $v = u_1x_1u_2 \ldots u_nx_nu_{n+1}$, where $u_i \in (N \cup T)^*$ for $1 \leq i \leq n$, then $u \Rightarrow v$ is a derivation step.

Definition 3 (LL SCG) Let $G_1 = (N, T, P, S)$ be a SCG. G_1 is LL SCG if $G_2 = (N, T, P[1], S)$ is LL CFG. (Definition of LL CFG is not listed here due to space restrictions and it could be found in [2])

Definition 4 (Basic Block) A *basic block* is a portion of the binary code that has one entry point, meaning no code within it is the destination of a jump instruction anywhere in the program, and it has one exit point, meaning only the last instruction can cause the program to begin executing code in a different basic block. Under these circumstances, whenever the first instruction in a basic block is executed, the rest of the instructions are necessarily executed exactly once, in order [4].

3 REGISTER REALIGNMENT

It is simple method, which is a part of every advanced metamorphic engine, Principle of the method is to exchange the registers in code without corrupting an original algorithm [5]. The result is that the instructions are still the same, but they work with other registers. In another words, a different admissible permutation of used registers is created. This update is reflected in binary code, where the parts of opcodes, which define the place of source or destination, is changed.

```
mov eax, ebx                         mov ecx, edx
xor ecx, ecx                         xor eax, eax
test eax, ecx                        test ecx, eax
```

The important note is that there are some restrictions. It definitely would not be a good idea to exchange the register `esp`, because it will cause corruption of stack and, most probably, a crash of application. The number of available registers for realignment is dependent on an architecture. On the architecture i386, there are 8 base registers, but we can effectively work only with 7 due to excluding the `esp` register. We know architectures with higher number of registers as ARM or MIPS, therefore there is much more available permutations and the register realignment can create much more combinations of the binary code.

4 IDEA

Take into account the following example: function `min`, which takes two integer values as arguments and returns the smaller value. Consider the function is also the basic block.

```
 0: 55                       push   %ebp
 1: 89 e5                    mov    %esp,%ebp
 3: 8b 45 08                 mov    0x8(%ebp),%eax
 6: 39 45 0c                 cmp    %eax,0xc(%ebp)
 9: 0f 4e 45 0c              cmovle 0xc(%ebp),%eax
 d: 89 ec                    mov    %ebp,%esp
 f: 5d                       pop    %ebp
10: c3                       ret
```

Figure 1: Original code

After running the metamorphic engine on the code from figure 1, we could obtain the code shown on figure 2.

The patterns for functions are (the different bytes are marked with bold font):

1. **55**89**e5**8b**45**0839**45**0c0f4e**45**0c89**ec5d**c3
2. **52**89**e2**8b**42**0839**42**0c0f4e**42**0c89**d45a**c3

```
 0: 52                    push   %edx
 1: 89 e2                 mov    %esp,%edx
 3: 8b 42 08              mov    0x8(%edx),%eax
 6: 39 42 0c              cmp    %eax,0xc(%edx)
 9: 0f 4e 42 0c           cmovle 0xc(%edx),%eax
 d: 89 d4                 mov    %edx,%esp
 f: 5a                    pop    %edx
10: c3                    ret
```

Figure 2: Obfuscated code

The anti-virus programs use regular expressions to solve that problem. The idea of that solution is to exchange the variant byte by a symbol, which means that this byte is not taken into account by a pattern matching algorithm. Then the pattern looks like this: `⋆89⋆8b⋆0839⋆0c0f4e⋆0c89⋆⋆c3`.

We see that this solution could give false positives, which means that it marks unrelated code as a correct result of the pattern matching. This is caused by a very free condition of the regular expression. The better results can be reached by studing the context dependency of changed bytes.

Scattered context grammars seem to be suitable formal model for describing this kind of problem. A challenge is to create this grammar automatically. In this paper, we will study a human creation of the grammar, which will be able to recognize the register realignment.

We assume that no other method for adjusting a binary code is used, so no instruction can be added or substituted. According to this assumption, the register realignment can be done only on a level of basic blocks. If we enabled the instruction substitution, the register realignment could be made with context on more blocks. We can demonstrate it on our code from figure 1. The result is returned in register `eax`. If we exchange `eax` and `ecx`, the result will be stored in the register `ecx`. In some cases, this could be solved by register realignment in a callee code, but this is not possible in every case and also the problem is a function call by register value (`call eax`), which cannot be generally solved by a static analysis of metamorphic engine.

Before building the grammar, we have to analyze the code and find the registers, which can be exchanged, these registers will be stored in the list A. The registers, which can take place instead of the original registers, will be put into the sets B_i, $i = 1,\dots,n$, where $n = length(A)$. An arbitrary register from set B_i can be used instead of register on the position i of the list A. If $i > 1$ then there is another problem. The selection of register from set B_i can influence the selection from set B_j $(i <> j)$, what has to be handled by the grammar.

In this paper we work with i386 architecture, therefore each set B contains a subset of `eax`, `ebx`, `ebp`, `ecx`, `edx`, `esi`, and `edi` registers, because every register which is used in specific basic block and is not in the list A is not included. Register `esp` is not included due to its special purpose. The definition of the list A is a bit more complicated. There is also rule that `esp` cannot be an item of the list. Moreover we have to be careful with `eax`, because it is used for returning a result value to callee. The instructions with implicit register as operand or as destination are the last trap. Good example is instruction `ADD` with opcode `05`, which takes immediate value as operand, adds it to value in `eax` or `ax` (according to size of operand) and finally, stores the new value in the same register.

It is important to say that every occurrence of register from list A has to be exchanged by the same register from assigned set B. This condition dictates minimal length of rule in the grammar. If we have only one item in the list A, the minimal length is equal to the maximal length, because we do not need to handle anything except the exchange of one register in all occurrences. If we have more items in list A, we have to handle conflicts, which can be raised up by exchanging registers. This special handling causes that the rules are longer.

We have only one condition on grammar. It has to be in LL form, this condition is given by need of possibility to parse the code [1]. According to the definition we have to satisfy LL condition for the first part of each SCG rule.

Now, we can make next step and create grammar for the code from figure 1. In the code, the registers `esp`, `ebp` and `eax` are used. Now, we have to create list A from these registers, we exclude `esp` due to facts presented in the previous section, also the register `eax` is used for returning the result, so it is excluded from this set. Only `ebp` is the item in the list A.

The next step is creation of set B_1 for the item from list A. We start from base `eax`, `ebx`, `ebp`, `ecx`, `edx`, `esi`, and `edi`. We exclude `eax`, because this register is used in this basic block and `eax`$\notin A$. $B_1 = \{\texttt{ebx},\texttt{ebp},\texttt{ecx},\texttt{edx},\texttt{esi},\texttt{edi}\}$. The grammar will have 6+1 rules (1 starting rule and one rule for one exchange of register `ebp` from list A by one register from set B_1). There are 7 variant bytes, therefore the exchange is performed on 7 positions, what specifies the length of each rule to 7 (except starting rule).

5 EXAMPLE

The grammar is working directly on the binary code. For each variant byte we have to find a code, which will be on that position for specific register exchange. This code is obtained by examining the affected instruction and finding out new opcode for the instruction with changed register.

We will show the finding opcodes for one realignment. We take register `edx` and find new opcodes for the instructions. First affected instruction is `push %ebp` with opcode `55`, after adjustment to `push %edx` the opcode is `52`, we gain the first variant byte with value `52` for `edx`, we can mark this variant byte as X_1. The second obfuscated instruction is `mov %esp,%ebp` with opcode `89 e5`, after realignment to `mov %esp,%edx` the opcode is `89 e2`. We have the second variant byte X_2 with value `e2`. The next updated instruction is `mov 0x8(%ebp),%eax`, the opcode is changed from `8b 45 08` to `8b 42 08`, it gives us X_3 with value `42`. By the same way we get the variant byte X_4 from instruction `cmp %eax,0xc(%edx)` with value `42`, and variant bytes X_5, X_6, and X_7 from instructions `cmovle 0xc(%edx),%eax`, `mov %edx,%esp`, and `pop %edx`.

We create the patterns for each possible combination, which can be established by the metamorphic engine. The register, which is used for the given pattern, is listed on the end of pattern in brackets. Finally, there is pattern, where we substitute the variant bytes by nonterminal symbol. This pattern will be used for starting rule of the grammar.

- **55**89**e5**8b**45**0839**45**0c0f4e**45**0c89**ec5d**c3 (ebp)
- **53**89**e3**8b**43**0839**43**0c0f4e**43**0c89**dc5b**c3 (ebx)
- **51**89**e1**8b**41**0839**41**0c0f4e**41**0c89**cc59**c3 (ecx)
- **52**89**e2**8b**42**0839**42**0c0f4e**42**0c89**d45a**c3 (edx)
- **56**89**e6**8b**46**0839**46**0c0f4e**46**0c89**f45e**c3 (esi)
- **57**89**e7**8b**47**0839**47**0c0f4e**47**0c89**fc5f**c3 (edi)

▷ $X_1$89$X_2$8b$X_3$0839$X_4$0c0f4e$X_5$0c89X_6X_7c3

Now we can create LL SCG G, $G = (N,T,P,S)$, $N = \{S,X_1,X_2,X_3,X_4,X_5,X_6,X_7\}$, $T = \{00,01,02,\dots,$ $fd,fe,ff\}$, $P = \{$

p_1 : $S \rightarrow X_1$`89`X_2`8b`X_3`0839`X_4`0c0f4e`X_5`40c894`X_6X_7`c3`
p_2 : $(X_1,X_2,X_3,X_4,X_5,X_6,X_7) \rightarrow$ `( 55, e5, 45, 45, 45, ec, 5d)`
p_3 : $(X_1,X_2,X_3,X_4,X_5,X_6,X_7) \rightarrow$ `( 53, e3, 43, 43, 43, dc, 5b)`
p_4 : $(X_1,X_2,X_3,X_4,X_5,X_6,X_7) \rightarrow$ `( 51, e1, 41, 41, 41, cc, 59)`
p_5 : $(X_1,X_2,X_3,X_4,X_5,X_6,X_7) \rightarrow$ `( 52, e2, 42, 42, 42, d4, 5a)`
p_6 : $(X_1,X_2,X_3,X_4,X_5,X_6,X_7) \rightarrow$ `( 56, e6, 46, 46, 46, f4, 5e)`
p_7 : $(X_1,X_2,X_3,X_4,X_5,X_6,X_7) \rightarrow$ `( 57, e7, 47, 47, 47, fc, 5f)` $\}$.

The prove that grammar is LL is trivial and it is left to reader. We can easily outline a process of parsing. Firstly, the starting rule is applied, we gain the string with nonterminal on the start and this nonterminal is X_1. According to input one rule from $\{p_2,\ldots,p_7\}$ is chosen, and this rule determines the values that have to be on next positions of all variant bytes.

6 CONCLUSION AND FUTURE WORK

We have successfully created LL(1) SCG, which is able to parse binary code and recognize the register realignment. The disadvantage of this solution is requirement of human interaction during the code analysis and also during creation of SCG. If we want to apply this process in malware analysis, the whole operation has to be automatic.

The future work should cover creation of methods and algorithms for completely algorithmic solution. This task requires a separation of code to basic blocks, what is a quite well explored area. In the second step, the definition of list A and sets B_i is to be performed. The last challenge is building of SCG, what is composition of two problems. The correct opcodes have to be found for each part of each rule. The conflicts between multiple register exchanges have to be handled. This is a problem only if the list A has more than 1 item.

Binary obfuscation knows much more techniques than register realignment. The recognition of these other methods will be another milestone after finishing the actually raised problem.

ACKNOWLEDGEMENT

This work has been supported by the research funding TAČR TA01010667, and by the Research Plan No. MSM 0021630528.

REFERENCES

[1] Kolář, D.: Scattered Context Grammars Parsers. In Proceedings of the 14th International Congress of Cybernetics and Systems of WOCS. Wroclaw: Wroclaw University of Technology, 2008. s. 491-500. ISBN: 978-83-7493-400-8.

[2] Meduna A.: Automata and Languages: Theory and Applications. Spring-Verlag. London 2000.

[3] Meduna A., Techet J.: Scattered Context Grammars and their Applications. WIT Press, 2010. ISBN: 978-1-84564-426-0.

[4] Cocke J.: Global common subexpression elimination. In Proceedings of a symposium on Compiler optimization. New York, 1970.

[5] Karnik A., Goswami S. and Guha R.: Detecting Obfuscated Viruses Using Cosine Similarity Analysis. Modelling Simulation, 2007. AMS '07. March 2007.

[6] Intel: Intel® 64 and IA-32 Architectures Software Developer's Manuals. Online 20th Feb 2011. `http://developer.intel.com/products/processor/manuals/index.htm`