World Academy of Science, Engineering and Technology
International Journal of Computer and Information Engineering
Vol:8, No:4, 2014

Automatic Intelligent Analysis of Malware Behaviour

H. Dornhackl, K. Kadletz, R. Luh, P. Tavolato

***Abstract*—**In this paper, we describe the use of formal methods to model malware behaviour. The modelling of harmful behaviour rests upon syntactic structures that represent malicious procedures inside malware. The malicious activities are modelled by a formal grammar, where API calls' components are the terminals and the set of API calls used in combination to achieve a goal are designated non-terminals. The combination of different non-terminals in various ways and tiers make up the attack vectors that are used by harmful software. Based on these syntactic structures a parser can be generated which takes execution traces as input for pattern recognition.

***Keywords*—**Malware behaviour, modelling, parsing, search, pattern matching.

I. Introduction

THE daily increase of new malware is a major challenge for antivirus companies. Malware writers use generating engines, which use encryption or metamorphic methods to produce myriads of variants of code – each version having a different binary appearance, while the procedures executed by the binary as well as the goal of the malware remains the same. Through these obfuscation techniques, the malware attempts to stay undetected by antivirus software and their signatures. This confronts malware analysts with an ever-increasing number of new samples of suspicious code demanding laborious inspection.

Malware analysis is a very time-consuming activity requiring a lot of domain-specific knowledge and intelligence. Therefore it is hard – probably impossible – to keep up with the growing number of new suspicious samples. Automating this procedure would be a great advantage in the battle against the dark sides of the information society. However, as the analysis process is a rather unstructured undertaking relying heavily on the experience and personal skills of the analyst, automating it – even only in parts – is a big challenge.

Automating behavioural analysis of suspicious code samples is a promising way of tackling this challenge. The way harmful programs modify the operating system and resources is called 'malware behaviour'. Malware manipulates an operating system by executing machine instructions, the most important of them those, which invoke application programming interface (API) functions. Such behaviour is independent of the actual representation or appearance of the code. In other words, behavioural analysis focuses on the semantics of a code sample, not on its syntactic guise.

H. Dornhackl, K. Kadletz, R. Luh, and P. Tavolato are with the Department of Information Security, University of Applied Sciences St. Pölten, Matthias-Corvinus-Straße 15, 3100 Austria (phone: +43/2742/313 228 - 634; e-mail: hermann.dornhackl@fhstp.ac.at, konstanin.kadletz@fhstp.ac.at, robert.luh@fhstp.ac.at, paul.tavolato@fhstp.ac.at).

The task of a malware analyst is not only to learn about the behaviour of a sample but also to estimate its maliciousness and ultimately decide whether it is malware or not. To automate this intrinsically intelligent task we have to solve the following problems:

1. Define elementary tasks that are constituents of malicious behaviour of code.
2. Define a mapping between these tasks and possible execution traces of code.
3. Having defined the code patterns a pattern-matching program can be created to locate malicious behaviour.

The definition of elementary tasks is described in [15]. This paper concentrates on step 2 and 3 of the above list.

Regarding step 2, we first have to decide on how we will represent the execution traces, the low-level behaviour, of the code. As mentioned above, the most significant activities executed are the system calls. Therefore, we restrict our analysis to system calls ignoring all other machine instructions in the trace. It is important to keep in mind that a single call must be considered safe, but certain combinations of calls can constitute one of the high-level elementary tasks defined in step 1 and therefore be dangerous for the user and the operating system.

We have to record all API calls executed by a code sample; these traces are the basis for subsequent pattern extraction. The patterns must be defined in a way so they can be processed automatically. One possibility is the use of formal languages to describe the structural layout of the patterns.

A parser based on the grammar describing malicious behaviour can perform pattern recognition (step 3). The execution trace of a suspicious code sample is used as input. The parser checks whether the input is generated by the rules of one of the behaviour patterns. If the parsing succeeds, this indicates malicious behaviour.

II. Related Work

The first attempt to formalize malware was proposed by Cohen [1] in 1987. He demonstrated in his work that there is no single algorithm that flawlessly detects all possible malware; Filiol [2] demanded a theoretical and formal treatment of malware. But there were only few contributions to this topic since. A paper written by Kramer and Bradfield [3] suggests a very abstract model for the definition of malware based on modal logics. Application of this model to real world problems is still far away (because of performance considerations among others). In [4], the problem of polymorphic malware is described and some analogies to techniques used in compiler construction are drawn.

The approach to model malware behaviour based on formal methods is rarely found in the scientific literature. In [5], attributed grammars are used to describe malware in general; in [6], a special variant of Lambda calculus is used. In [7], formal languages are used to describe traces generated by a dynamic analysis of code samples; these traces are then compared to predefined reference patterns. The authors do not provide any information about how these patterns were constructed.

Bayer did important work in the area of behavioural analysis of malware as did Moser, Kruegel and Kirda [8], [9]. Kirda and Kruegel [10] describe dynamic methods to analyze a large number of code samples. Interpretation of the output with respect to the behaviour of the samples is discussed in these papers and others: Christodorescu, Jha and Kruegel [11] specify a formalism and an algorithm to find malicious behaviour. All these methods, especially when they rely on statistical data, have problems with false-positives. Fukushima, Sakai, Hori and Sakurai [14] try to distinguish between malicious and benign code based on statistical analysis of the execution traces. Luh and Tavolato [13] use a dual approach for determining the potential danger of a sample: The decision is made based not only on malicious activities but also on benign activities of a sample's behaviour.

For reasons of practical importance, this paper concentrates on malware for Microsoft Windows systems. Due to the huge amount of possible different behaviour patterns, we focus on malware autostart capabilities in this work.

III. Mapping Tasks to Traces

As stated above, the ultimate goal is the definition of a mapping of elementary tasks to execution traces through the use of formal grammars. For every task, we define a grammar that generates all possible execution traces resulting in the accomplishment of this task. The alphabet of each of the grammars is a subset of the set of all API calls. First, when a binary file is started, all instructions along a certain execution path are executed. This path depends on the program's logic and does not necessarily imply that every instruction present in the code is actually executed. The operating system version and installed software influence the execution trace thus leading to differing results.

Binary files execute machine instructions. We focus on one type of instruction only: the API calls. These contain bundled functionality provided by external files or libraries and are responsible for an application's interaction with the operating system. Monitoring the execution yields in (very long) sequences of OS-level system calls. These sequences define various subtasks of the behaviour of the code samples under scrutiny. Consequently, they are the basic source for finding patterns that describe malicious activities.

To get a representative set of samples we executed and monitored a collection of about 1400 malware samples and collected their traces. Prerequisite for this step was a safe execution environment with the possibility to monitor all performed API calls.

First we had to identify sequences of API calls that modify the operating system with malicious intent. These sequences provide a detailed view on the different strategies malware uses to modify and infect the operating system. Because there is a huge number of different approaches to achieve the same malign goal, all possible strategies have to be part of the grammar. As every approach can be implemented in many different ways, profound knowledge of system internals and a good insight into the programming techniques used by malware writers is required. Therefore, an exhaustive enumeration of these sequences is not possible.

Defining the grammar can be accomplished either by analyzing traces of known malware and deriving patterns thereof or by defining the patterns manually. The benefit of the latter approach is that the modelling of malware behaviour is not influenced by outside factors apart from the analyst's knowledge of popular programming procedures. Using the monitored calls, on the other hand, covers a bulk of typical behaviour exhibited by malware in the wild. In our work we have applied a combined solution.

We used a virtual environment with the virtualization solution "Oracle Virtual Box" [19] and the monitoring Tool "Rohitab API Monitor v2" [16] installed on a Microsoft Windows 7 operating system. The captured calls are either Win32 API calls or Window native calls.

Fig. 1 shows the structure of a call beginning with the name of the library containing the called function followed by the name of the call itself. Every call may have a list of given parameters starting with a left parenthesis followed of parameters which are separated by comma and ending with a right parenthesis. An optional return value completes the line. The parameter's trailing asterisk symbolizes that it can occur zero or several times. The single parts of the input file surrounded by angle brackets are terminals in sense of the grammar.

```
<binary-file> <api-call-name> ( [<parameter>* [,]] )
[<return-value>]
```

Fig. 1 Structure for a monitored API call

A pattern of these calls with malicious intention is called an approach – a means of the malware to achieve its detrimental goal. Fig. 2 gives an example of such a call.

```
kernel32.dll RegOpenKeyExA (HKEY_CURRENT_USER,
"SOFTWARE\Microsoft\Windows\CurrentVersion\
Policies\Explorer\Run", 0,
KEY_ENUMERATE_SUB_KEYS |
KEY_QUERY_VALUE | READ_CONTROL, 0x0012fd34 )
ERROR_FILE_NOT_FOUND 2 = The system cannot find
the file specified.
ADVAPI32.dll NtOpenKey ( 0x0012fd34,
KEY_ENUMERATE_SUB_KEYS |
KEY_QUERY_VALUE | READ_CONTROL,  0x0012fb94 )
STATUS_OBJECT_NAME_NOT_FOUND
```

Fig. 2 Examples for an API trace

To illustrate the grammatical definition of a specific behaviour pattern we chose an activity often used by malware to start itself automatically at system boot (subsequently called "autostart behaviour" for brevity).

In general, autostart behaviour can be generated by specific changes in the file system, by altering configuration data, by using the system's autorun functionality, or by modifying the registry.

In order to make the example more accessible we restricted the exemplary modelling process to the most important of these possibilities, namely the modification of the registry. Bayer [18] showed that, out of more than 900,000 malware samples, 35.8% were changing the autostart entries in the registry to resume operation after a restart of the system.

Information about which registry keys and values are modified can only be found by looking at the parameters of the API calls. Therefore, detailed inspection of all relevant API calls able to modify autostart behaviour is necessary.

The Microsoft system documentation yields 84 different so-called "autostart locations" in the registry. Many of them exist only for compatibility reasons to older versions of the operating systems and programs. The locations are specific registry keys that are used by programs for automatic start. Autostart behaviour can be defined by the activity of starting the malware program either during system startup, upon user logon or triggered by specific user actions.

Next we had to examine which API calls can be used to access and manipulate registry entries. To implement the program's autostart functionality a new registry key and value must be added or an existing registry key and value must be manipulated. Before making changes to a registry key, one has to open or to create it. Executing a "create" API call for an existing registry key will lead to an implicit open of the key. Closing or flushing the entry is done in most cases but is not unconditionally necessary.

Some API calls have identical functionality and can therefore be substituted for each other since they will result in the same changes in the registry. Therefore, we had to consider which calls can be substituted by others and how their exact sequence looks like. Many of the calls exist in different versions for ANSI (suffix "A") and for Unicode (suffix "W") encoding of string parameters while others offer extended functionality (suffix "Ex"). Beside the calls from the Win32 API there exist, among others, so-called native calls (prefix "Nt") from the Native Kernel API. Win32 API calls are usually translated into native calls, so the corresponding native call usually closely follows the respective Win32 API call in the trace. Between these calls an arbitrary number of other calls (which do not manipulate the registry directly) may appear and must be neglected.

Altogether there are more than hundred system calls that are related to accessing the registry; for the sake of simplicity we restrict our example to the following 17 calls which are used to open, create, or set registry keys.

a) Open a registry key
   RegOpenKeyA
   RegOpenKeyW
   RegOpenKeyExA
   RegOpenKeyExW
   NtOpenKey
b) Create a registry key
   RegCreateKeyA
   RegCreateKeyW
   RegCreateKeyExA
   RegCreateKeyExW
   NtCreateKey
c) Set registry value
   RegSetValueA
   RegSetValueW
   RegSetValueExA
   RegSetValueExW
   NtSetValueKey
d) Close a registry key
   RegCloseKey
   NtClose

There are further calls that can be used to manipulate entries in the registry. One scenario could be to remove the autostart of antivirus software with "delete" API calls.

Bayer [18] showed that the majority of all malware uses the registry key

"HKEY_LOCAL_MACHINE\Software\Microsoft\Windos\CurrentVersion\Run"

to gain autostart functionality. The reason for is that this key is also available in most of the older versions of Windows. All contained subkeys are sequentially processed and executed; but they are not executed if the operating system is started in "safe mode" unless the subkey has an asterisk (*) prefix. There is an equivalent registry key in the user-dependent section of the registry

"HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run"

with the difference that programs registered here are only started for the user logged in.

The dual approach of manual construction and monitoring malware behaviour resulted in 7 general patterns. The patterns are partly overlapping in their functionality. Examples for this overlapping are the interdependent calls for "open" and "set". The manipulation with "set" depends on an opened registry key. Such coherences can be modelled once and reused in the grammar. Another example is the recurring sequences of Win32 API and native API calls that belong together. Such repeating micro-patterns can be abstracted to "non-terminals" in terms of the formal grammar.

The non-terminal "create" in Fig. 3 is used in other non-terminals in the grammar. This helps to describe complex behaviour patterns of malware in a concise way (Fig. 4).

To create a simple and clear grammar we have excluded the "comment problem" here: The real trace contains a lot of additional system calls which are not relevant with respect to autostart behaviour. It would be straightforward to augment the grammar with rules to skip these calls.

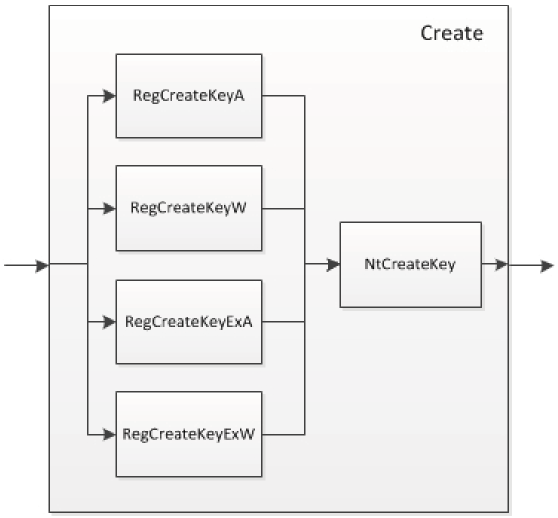

Fig. 3 Win32 API call followed by native API call to create a registry key

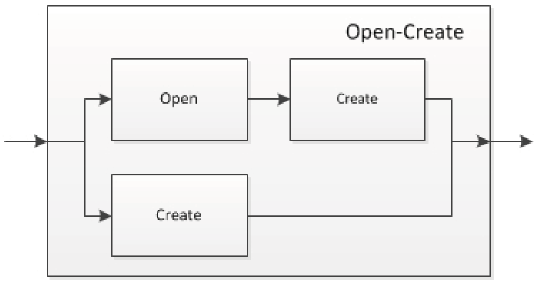

Fig. 4 Simple grammar rule

The context-free grammar defining simplified autostart behaviour can be given as follows (assuming task T stands for autostart behaviour):

$$G_t = (SC_t, V_t, P_t, T)$$

where SC_t, a subset of SC (the set of all system calls), is the set of terminal symbols, V_t is the set of non-terminals, P_t is the set of production rules and T is the start symbol.

SC_t = {RegOpenKeyA, RegOpenKeyW, RegOpenKeyExA, RegOpenKeyExW, NtOpenKey, RegCreateKeyA, RegCreateKeyW, RegCreateKeyExA, RegCreateKeyExW, NtCreateKey, RegCloseKey, NtClose, RegSetValueA, RegSetValueW, RegSetValueExA, RegSetValueExW, NtSetValueKey}

V_t = {T, OPEN-CREATE-CLOSE, OPEN-CREATE, OPEN-SET-CLOSE, OPEN, CREATE, CLOSE, SET}

P_t = {
T => OPEN-CREATE-CLOSE OPEN-SET-CLOSE

OPEN-CREATE-CLOSE => OPEN-CREATE CLOSE | OPEN-CREATE

OPEN-CREATE => OPEN CREATE | CREATE

OPEN-SET-CLOSE => OPEN SET CLOSE | OPEN SET

OPEN => RegOpenKeyA NtOpenKey | RegOpenKeyW NtOpenKey | RegOpenKeyExA NtOpenKey | RegOpenKeyExW NtOpenKey

CREATE => RegCreateKeyExA NtCreateKey | RegCreateKeyExW NtCreateKey | RegCreateKeyA NtCreateKey | RegCreateKeyW NtCreateKey

CLOSE => RegCloseKey NtClose

SET => RegSetValueExA NtSetValueKey | RegSetValueExW NtSetValueKey | RegSetValueA NtSetValueKey | RegSetValueW NtSetValueKey
}

This grammar describes specific syntactic patterns in a trace. Implementing a parser for this grammar will provide an efficient way of automatically matching such patterns. There is one major drawback with this solution: Since API calls are used as terminal symbols an evaluation of parameters for semantic validation is not possible. One solution to this problem is a further breakdown of the terminals to a lower abstraction level. For a call all elements of a call shown in Fig. 1 (library, API call, brackets, commas, parameters, and return value) must be used as terminal symbols of the grammar. This results in a detailed evaluation of all existing parameters and possible data types of every used API call. This would improve the semantic validity of the grammar considerably. Such a detailed breakdown is described in [12].

IV. Parsing

Having defined the grammar mapping malicious behaviour to execution traces it is easy to develop a parser to match such behaviour in arbitrary execution traces. The parser was build using the tools lex for lexical analysis and yacc for syntactic analysis. Additionally, we implemented the semantic evaluation of call parameters in yacc using yacc actions. There are two possible solutions to solve the mentioned “comment problem”: One possibility would be the prefiltering of input data by excluding all superfluous calls. This method is simple but inefficient and was therefore rejected. We implemented a more sophisticated solution by using the error handling features in yacc.

For lex regular expressions describing the input tokens were developed. See Fig. 5 as an example.

```
[A-Za-z0-9]+[.]{1}[A-Za-z0-9]{3}
yylval.string=strdup(yytext); return DLL;
```

Fig. 5 Example for the token DLL

Before applying the grammar rules described above some low-level grammar rules for yacc based on the given lex tokens are built representing a single line of the input. All basic yacc rules for the registry API calls look similar to the "open" API call in Fig. 6.

```
DLL OPENKEYWINAPI '(' param_registry_handle ','
param_registry_key ',' ADDRESS ')' return_value new_line
```

Fig. 6 Example for yacc a rule

The uppercase strings in Fig. 6 represent the tokens delivered by lex and the lowercase strings are non-terminal symbols from the context-free grammar. The low-level rules also introduce a more abstract tier to distinguish between standard API calls and the extended versions of them which is important as they usually have different numbers of parameters.

Based on these low-level rules the implementation of the high level grammar rules representing the 7 general patterns of autostart behaviour is straightforward.

V. Conclusion and Further Work

Tests with the parser proved the viability of this approach to behavioural analysis of suspicious code. For a specific behaviour often found in malware (autostart registration) a context-free grammar was developed. The parser for this grammar was implemented using common parser generation tools. With this parser execution traces of programs at the level of API and system calls can be parsed to automatically detect behaviour patterns.

Extending the grammar to comprise most of the patterns for malicious behaviour will certainly be a huge and time-consuming effort. We are therefore working on algorithms to partially automate this process. In [17], an algorithm is sketched that deduces regular and context-free grammars from a set of valid words of a language. Being heuristic in nature these algorithms can only produce approximate solutions to the problem as their quality is largely dependent on the selection of example words. Hence it is still necessary to analyze a larger number of sample traces in order to be able to choose the ones that are most suitable as input for the algorithm.

References

[1] Cohen, F.: Computer Viruses: Theory and Experiments. In: Computer and Security 6/1, 1987, pp. 22-35.
[2] Filiol, E.; Helenius, M; Zanero, S.: Open Problems in Computer Virology. In: Journal of Computer Virology 1(3-4), 2006, pp. 55-66.
[3] Kramer, S.; Bradfield, J.C.: A General Definition of Malware. In: Journal in Computer Virology, 6/2, 2010, pp. 105-114.
[4] Jacob, G.; Debar, H; Filiol, E.: Functional polymorphic engines: formalisation, implementation and use cases. In: Journal in Computer Virology, 5/3, 2009, pp. 247-261.
[5] Jacob, G.; Debar, H; Filiol, E.: Malware Behavioural Detection by Attribute-Automata Using Abstraction from Platform and Language. In: Lecture Notes in Computer Science 2009, Vol. 5758/2009, pp. 81-100.
[6] Jacob, G.; Debar, H; Filiol, E.: Formalization of Malware through Process Calculi. In: Journal in Computer Virology, 5/3, 2009, pp. 247-261.
[7] Beaucamps, P.; Gnaedig, I.; Marion, J.: Behaviour Abstraction in Malware Analysis. In Lecture Notes in Computer Science 2010, Vol. 6418/2010, pp. 168-182.
[8] Bayer, U.; Kirda, E.; Kruegel, C.: Improving the Efficiency of Dynamic Malware Analysis. 25th Symposium On Applied Computing, Lausanne, 2010.
[9] Bayer, U.; Moser, A.; Kruegel, C.; Kirda, E.: Dynamic Analysis of Malicious Code. Journal in Computer Virology 2/1, Springer, 2007.
[10] Kirda, E.; Kruegel, C.: Large-Scale Dynamic Malware Analysis. PhD Dissertation, Technical University of Vienna, 2009.
[11] Christodorescu, M.; Jha, S.; Kruegel, C.: Mining Specifications of Malicious Behaviour. ESEC/FSE'07, September 3–7, 2007, Cavtat near Dubrovnik, Croatia, 2007.
[12] Dornhackl, H.: Syntaktische Musterdefinition von ausgewählten Malwareverhalten und Implementierung eines Parsers. Master thesis, FH St. Pölten, 2013 (in German).
[13] Luh, R.; Tavolato, P.: Behaviour-based Malware Recognition. Forschungsforum der österreichischen Fachhochschulen, 2012.
[14] Fukushima, Y.; Sakai, A.; Hori, Y.; Sakurai, K.: A Behaviour Based Malware Detection Scheme for Avoiding False Positive. Secure Network Protocols (NPSec), 2010 6th IEEE Workshop on Secure Network Protocols, 2010.
[15] Dornhackl H., Kadletz K., Luh R., Tavolato P.: Using Formal Methods for Malware Behaviour Modelling, to be published.
[16] Batra, R.: API Monitor. retrieved from http://www.rohitab.com/apimonitor, last accessed 2013-10-14.
[17] Gonzalez, C.; Thomason, M.: Syntactic Pattern Recognition. Addison-Wesley, 1978.
[18] Bayer, U.: Large-Scale Dynamic Malware Analysis. PhD thesis, Technische Universität Wien, 2009.
[19] Oracle Corporation, Oracle Virtual Box retrieved from https://www.virtualbox.org, last accessed 2013-10-14.