Malware Detection: Hands on

Francesco Piccinno

July 20, 2012

1 What is malware?

Malware is a generic term to indicate a generic malicious software or malicious code. Several authors attempted to provide a specific definition for the term malware. Christodorescu and Jha [1] describe a malware as software program whose objective is malevolent. McGraw and Morrisett [7] instead classify a malware as any code added, changed or removed from a software system in order to intentionally subvert the intended functionality of the system itself. In this short report we consider the word "malware" a generic term that encompasses viruses, trojans, spywares and other intrusive code.

2 Malware Detector

A malware detection is a software component whose objective is to identify malwares. The detector attempts to help protect the system by detecting malicious behavior of a given piece of software. But how a automatic component can understand what is malicious and what is not? In order to simplify, we can see a malware detector as a black box system that takes two inputs: the knowledge of the malicious behavior and the input program under inspection. How this "knowledge base" is built and which information is encoded in it characterize the capabilities of the detection tool.

There are essentially three techniques that are employed for malware detection:

Signature based Signature-based detection exploits a previous knowledge of what is known to be malicious in order to decide the maliciousness of the program under inspection.

Anomaly based An anomaly-based detection technique does the contrary. By exploiting the knowledge of what constitutes normal behavior it tries to decide the maliciousness of a program by analyzing anomalies in its behavior.

Specification based Specification-based is a derivation from the anomaly-based approach. The technique by using some specification or rule set describing what is valid behavior is able to decide the maliciousness of a program by just checking if that piece of software is violating those rules or not.

Additionally, several sub-approaches are possible, irrespectively of the main technique being used. Namely there are static, dynamic and hybrid approaches. For example, using a static approach jointly with a signature-based detection would only leverage structural information of the file, such as specific sequence of bytes, in order to detect potentially harmful code. The dynamic approach will leverage runtime information, such as stack

analysis or sequence of system calls invoked, trying to detect a malicious behavioral pattern. Hybrid techniques combine the two approaches.

2.1 Signature based detection

Signature-based detection attempts to extract an unique feature or signature that matches a known malware for its future detection. Ideally a signature should be enough detailed to identify any piece of code or software exhibiting the malicious behavior encoded in the signature. A tool employing this technique must have access to a large repository of signatures and be able to detect all known malwares for which a signature is known. The main problem of this approach is the process of signature derivation mainly relies on human expertise, although some limited automatized solutions exists. Another big drawback is the fact that this malware detection solution cannot detect attacks for which a signature is not present (zero-day attacks).

2.2 Anomaly based detection

To realize a malware detector employing the anomaly-based technique two phases are required. The training phase is used to let the tool "learn" what is the *normal* behavior. The second phase, is the detection phase, in which the tool based on the knowledge acquired in the first phase, tries to detect potentially harmful programs. A key advantage of this detection technique is its ability to discover potential zero-day attacks or better attacks that are previously unknown to the malware detector. On the other hand there are two big limitations involving this procedure. The first one is is the complexity involved in determining which are the key features that should be extracted and processed in the learning phase. The second one is the high true-negative rate (a program classified as malicious though it is not). The problem is due to the approximation that the designer of the tool indirectly creates during the training phase of the tool. Indeed a condition that is never met during the training phase that instead shows up during the monitor phase would produce a false alarm.

2.3 Specification based detection

Specification-based detection was born as an attempt to address the high false alarm rate involved with anomaly-based detection techniques. In specification-based detection, the training phase can be thought as a derivation phase, whose aim is to extract a set of rules that describe all the valid behaviors any software component can exhibit. Of course a complete and rich derivation of all the rules is a tedious and complicated task. Indeed, for a quite large complex system, the complete and accurate specification of what is permitted and what it is not can be intractable.

3 Examples from the Literature

This section is dedicated to a brief overview of what we think are the most important and promising techniques for malware detection. The scope of this section is not to provide a complete review of all the possible solutions but just to highlight the common key concepts that have emerged from the study of this active research field.

3.1 Anomaly based detection - Dynamic solutions

In [12], Wang and Stolfo presented a very simple mechanism for anomaly detection targeted to the protection of several network services of a system. For each service port the expected payload is calculated and a byte frequency distribution is then evaluated. This distribution allowed the construction of a *centroid model* for each of the host's services. The authors then proposed to compare incoming payloads with the centroid model by measuring the Mahalanobis distance between the two. This distance takes into account the correlations of the data set and is scale-invariant therefore yielding a stronger statistical similarity measurement.

A completely different proposal is the one from Hofmeyr et al. [4], a work mainly targeted on the study of sequences of system call in order to detect malicious behaviors. First, a "normal" profile that represent the normal behavior of system's services is derived (n-gram based). An anomaly is then detected whenever a given system call sequence differs for more than a given threshold from the normal profile. Hamming distance is used as similarity measurement for comparing system call sequences.

Sekar et al. [9] proposed a similar approach based on a Finite State Automata. The change of state is driven by a syscall invocation. The training phase, which consists in the construction of the FSA is straightforward. Once a syscall is invoked a transition is added to the automaton. The resulting FSA represents the "normal" flow of the program. This knowledge is then exploited to track down possible abnormal sequences of syscall invocations. This method seems to have a lower false-positive rate than its counterpart based on n-grams.

In [8], Sato et al. proposed a more sophisticated approach based on statistical frequency analysis. The frequency of system calls is used to extract a profile of a given program. The more frequent a given system call is used the lower is its "ranking number" in this profile. Once the profile has been established, the distance or similarity between the profile and sampled data is evaluated by using the a DP matching algorithm. The distance given by the matching algorithm serves as an indicator of the maliciousness of the program.

3.2 Specification based detection - Dynamic approach

Sekar et al. [10] proposed a method to effectively conduct a live auditing of a given program. The key idea of the method is a translation process that takes in input the program and outputs an a representation of it in an custom language, called Auditing Specification Language. This ASL jointly with an infrastructure able to capture system calls being invoked, is then linked to the final program. The system call being intercepted are matched against the ASL code and if a mismatch is found the program is considered to be malicious.

A totally different focus is the proposal from Lee et al [6] whose aim is the protection from malwares. The authors exploited the LIFO nature of how calls are invoked to derive a quite effective protection mechanism. Indeed the tool they developed seems to be able to protect from most common stack smashing attacks. For this method to work correctly, the processor has to maintain its own stack, called a Secure Return Address Stack (SRAS). This structure is then used to detect possible control flow modification carried out by the malware. This is done by just comparing the return address with its safe version stored in the SRAS on each subroutine epilogue.

3.3 Specification based detection - Static approach

Bergeron et al. proposed a novel approach for detecting the maliciousness of an executable. By leveraging a static analysis and then by using a translation to an intermediate representation of the assembly code, a control flow graph of the executable is derived. Another derivation is then applied on the just constructed CFG graph. The output of this procedure is an API-graph where only API calls are considered. A set of security policies are then used to derive a critical API sub-graph, that is then evaluated for maliciousness.

3.4 Specification based detection - Hybrid approach

StackGuard [2] is the name of a product nowadays present in almost all the recent Linux distributions, that employs stack-canary techniques to protect the program again stack-smashing misuses. The product is a compiler extension that attempts to secure several routines valuated as unsafe (e.g. functions dealing with strings) by introducing a canary word before the return address. Almost any buffer overflow condition will result in an overwrite of the stack canary that if detected causes the premature abortion of the program. This modification is detected before the actual malicious control flow modification takes place.

3.5 Signature based detection - Static approach

Sung et al. [11] introduced a very interesting signature-based technique for detecting malicious programs in Windows environments. The signature is essentially a sequence of Windows API calls (n-grams). Each API call is a 32-bit number where the high 16-bits correspond to the module number, while the lower 16-bits are relative to the function being called. From a static inspection of the executable various sequences of API calls are extracted and compared with a database of known signatures. The similarity measurement in this case is the Euclidean distance.

4 Our contribution

As you will understand later, this work takes inspiration from different techniques presented in the previous section. Our contribution merely consists in the creation of a framework that let the analyzer experiment with the malware being inspected.

4.1 Obtaining a behavioral description with Ether

Initially, we thought on how to dynamically trace an unknown program in a isolated environment, in order to get a trace of its behavior. In order to maintain transparency during detection and at the same time to offer an isolated sandbox for analysis we focused our attention on virtualization technologies, mainly on Ether [3]. Ether is a tool that allows malware analysis via hardware virtualization extensions, by using as base the XEN hypervisor. The tool is a research level software targeting the 3.1 version of XEN and offers interesting features such as syscall tracing, instruction level tracing, memory write detection and also a simple unpack-execution detection that is able to detect various packers.

Although the tool seems promising and complete in its feature set, the product is not using the current version of XEN code-base 4.1 and it is only supporting Windows XP SP2 guests (through hard-coded configuration parameters). Moreover the syscall tracing

capabilities in which we were interested in is limited, in the sense that a live inspection of the stack of the program and of other parameters passed during syscall invocations is not possible at the current state.

4.2 Dynamic binary instrumentation

Due to the limitations of Ether, we decided to invest a little time to create a simple and effective syscall tracer that exploits dynamic binary instrumentation techniques. Although the final tool is not transparent and can be detected by the malware, we decided to go for it instead of investing a larger amount of time and effort to introduce the same functionalities in Ether (also because we were not sure about the outcome of the experiment).

In order to produce a ready to go tool, we used PIN a dynamic binary instrumentation framework that allows to quickly create dynamic program analysis tools. Indeed, by just leveraging the `PIN_AddSyscallEntryFunction` we were able to detect all the syscall invocations in a given program. The functionality of the tracer was then improved in order to support a complete inspection of all parameters being passed to a given syscall invocation. To this end we have introduced a simple parser, that starting from Windows syscall definitions expressed in a series of `.h` header files was able to derive automatic code capable of logging the contents of the variables being passed on the stack.

The tool also provides to the user the possibility to hook into the code generation facility. This feature can be used to insert custom code in the middle of automatically generated code. This allows the creation of personalized functions that focus their attention on specific aspects of certain syscalls invocations. To better clarify, take in consideration Listing 1, a specific snippet of code that we used to control the `DeviceIoControlFile` syscall in order to log specific information related to TCP or UDP communications.

Listing 1: DeviceIoControlFile hooking

```
if (IoControlCode == IOCTL_AFD_CONNECT)
{
  W::PAFD_CONNECT_INFO info = (W::PAFD_CONNECT_INFO)InputBuffer;

  /* ... */

  if (info->RemoteAddress.sa_family == AF_INET)
  {
    W::PSOCKADDR_IN sin4 = (W::PSOCKADDR_IN)&info->RemoteAddress;
    LOGFN("\t\tSOCKADDR␣RemoteAddress␣=␣\"\"\"%hu.%hu.%hu.%hu:%hu\"\"\"\n",
        (short)((sin4->sin_addr.s_addr >> 24) & 0xff),
        (short)((sin4->sin_addr.s_addr >> 16) & 0xff),
        (short)((sin4->sin_addr.s_addr >> 8) & 0xff),
      (short)(sin4->sin_addr.s_addr & 0xff), __htons(sin4->sin_port)
    );
  }
}
```

From the higher level the syscall tracer tool simply inspects the target program and intercepts any syscall invocation. All the information are saved in a log file that can be later analyzed by the PyMal framework we developed.

4.3 PyMal Framework

The PyMal framework essentially consists of two basic component: the collector and the analyzer. The collector, as the name suggests, simply takes in input the log file produced

during the first phase of analysis and extracts all the possible information from it. The information being collected include: full dump of any in/out parameters of the syscall (also complex `structs` are supported), syscall number, syscall name, return code and the address of the caller (used for positional analysis).

The analyzer is instead a meta module that offers several functions, ranging from n-gram statistical analysis to cosine-similarity metrics. It can be used as is or in a interactive way, thus allowing the user to express novel analysis techniques quite easily from an interactive interpreter.

4.4 Playground

We used the tool to study several malwares with the final aim to find a common sequence of syscalls invocation that could be thought as malicious. Our idea is similar to the one proposed by Sung et al. [11]. The difference resides in two points. The first is that the analysis is focused only on syscalls instead of API calls. This is due to the fact that syscall analysis can be easily implemented at the VMM level (as Ether does) and also because it is also very hard to detect.

The second key difference in our approach is the final goal. We were looking for a common signature or better sequences of syscalls that were able to identify a given malware. We are not looking for modification of the control flow of a "sane" program due to exploitation condition. Instead we focused our attention on malware identification as is.

MD5 Digest	Name
`0d8ff206e44059599747bc4ecf617240`	Allaple
`fda80b705a0ca0e493e0c9a1409a6abd`	Blackhole-zeus
`34a136aacdd2cc36fccd0debf930e4dc`	Dapato
`573276121a2df24f277bd54958e8c5fd`	FakeAV
`c059816a9e77113092f7c6adb2deeceb`	Gamaure
`607b2219fbcfbfe8e6ac9d7f3fb8d50e`	Ramnit
`bf53d17ace809cb3015eaed88a46d8aa`	SCKeyLog
`9fbf1209b37ea58fa85851673c368f89`	SpyEye
`dd9ebd92fc796eed8acba98d902933df`	Sub7
`d5c12fcfeebbe63f74026601cd7f39b2`	W32.Xpaj
`7b7890cc2d35c552e1fedef71960b49f`	Zeus

Table 1: Malware samples analyzed

Table 1 summarizes the malware we have analyzed in our experiment. During the experimentation phase, we analyzed each sample in an isolated Windows XP virtual machine by running each under our syscall tracer. The syscall invocation sequence was then extracted from each run and inserted in a SQL database. From this timeline, we extracted a sequence of k-grams, with k varying from 1 to 7. We then used these "k-timelines" to derive a PDF graph. The graphs are presented in Figure 1 and Figure 2. The first figure shows the PDF of all syscalls that were invoked (also including nested syscalls that is a syscall invoked during the service of another syscall) while the second figure just take in consideration the outermost syscalls.

The obtained results are not encouraging, in the sense that we were not able to find a commonality that could be used to track down malicious programs. This is somehow confirmed by the similarity measurements we obtained. Indeed by treating the k-gram

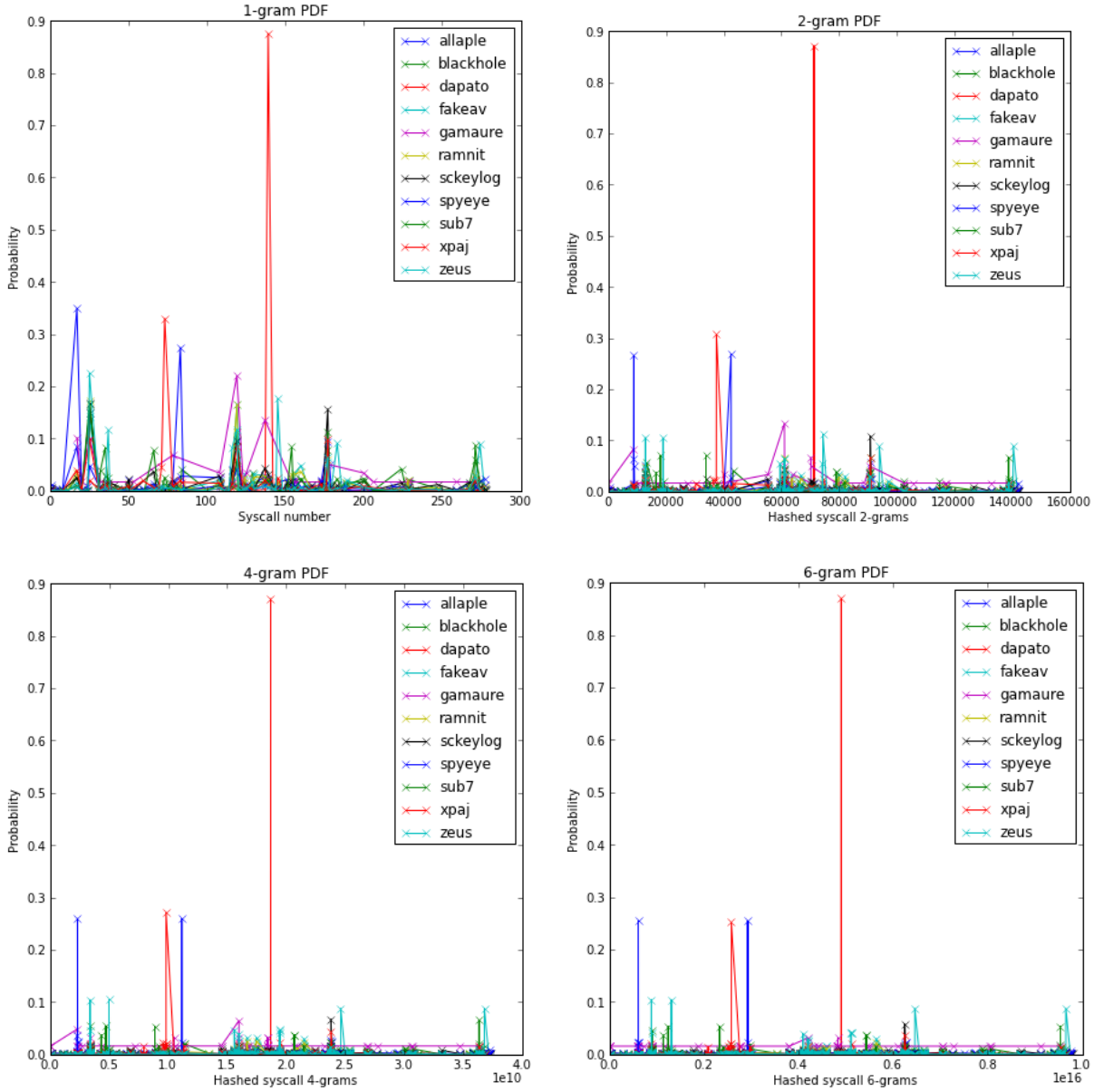


Figure 1: PDF with nested syscalls

sequence of syscalls as a multidimensional vector and evaluating the cosine similarity between various samples, as defined by the relationship 1 we got almost exact matches with similarity above 95% for almost all the samples.

$$\text{similarity} = \cos(\theta) = \frac{A \cdot B}{\|A\|\|B\|} = \frac{\sum_{i=1}^{n} A_i \times B_i}{\sqrt{\sum_{i=1}^{n} (A_i)^2} \times \sqrt{\sum_{i=1}^{n} (B_i)^2}} \tag{1}$$

Although we were not able to derive an automatized approach for malware identification, the tools we developed can be used for a preliminary analysis of an unknown program, leaving to the expertise of a human analyzer a deeper study of the sample.

To this end we briefly mention that the log can be easily parsed, in order to understand which files or pipes are used, created or read by a program, which modules are being loaded, which DNS queries are executed or which packets are being exchanged between TCP or UDP endpoints. For example Listing 2 shows some files being read by the zeus

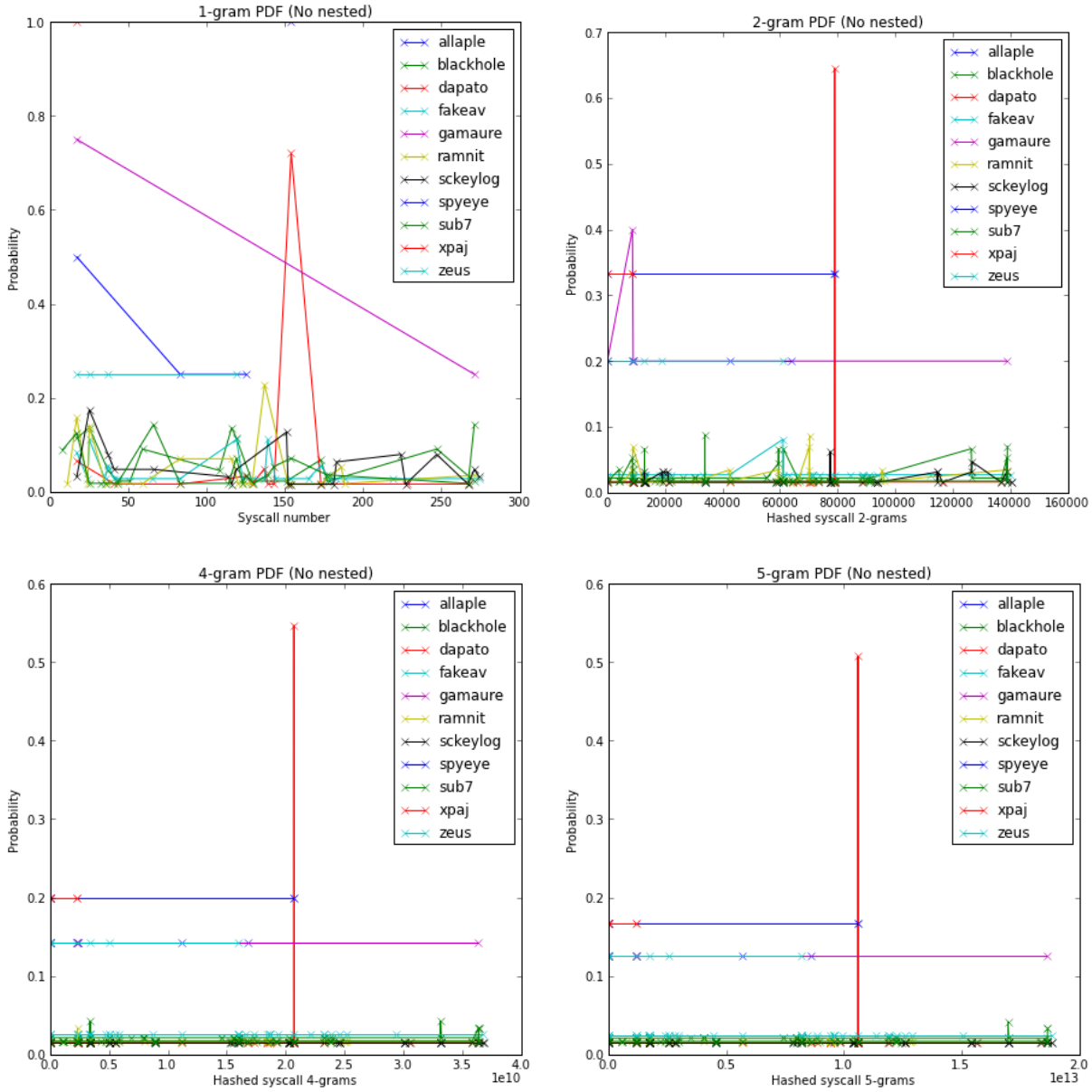
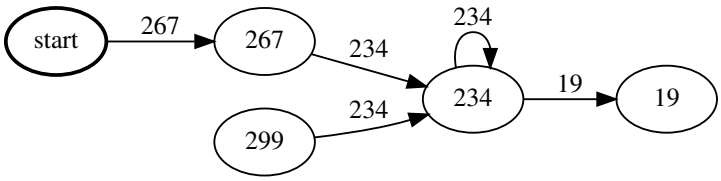


Figure 2: PDF without nested syscalls

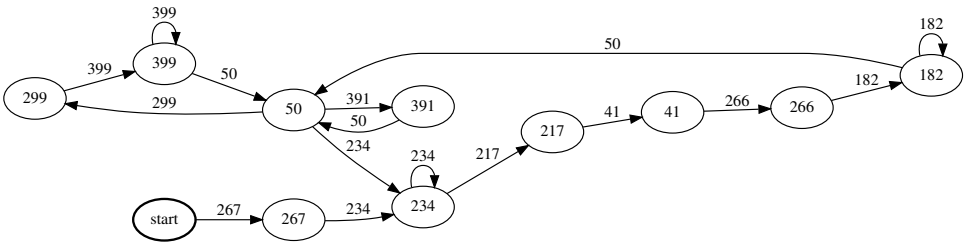
sample. In this case the malware is trying to extract passwords stored in Google Chrome web browser.

Listing 2: Simple string analysis

```
$ grep "CreateFile):␣0" -A 6 logs/* | grep ObjectName | \
  sed -e 's/logs\/\([a-z0-9]*\).txt-.*"""\(.*\)"""/\1: \2/' | \
  sort | uniq -c | grep -i "c\:"
  ...
  1 zeus: \??\C:\%APPDATA%\Google\Chrome\User Data\Default\Login Data
  1 zeus: \??\C:\%APPDATA%\Google\Chrome\User Data\Default\Login Data-journal
  1 zeus: \??\C:\%APPDATA%\Google\Chrome\User Data\Default\Web Data
  1 zeus: \??\C:\%APPDATA%\Google\Chrome\User Data\Default\Web Data-journal
  1 zeus: \??\C:\%APPDATA%\SharedSettings.ccs
  1 zeus: \??\C:\%APPDATA%\SharedSettings.sqlite
  1 zeus: \??\C:\%APPDATA%\SharedSettings_1_0_5.ccs
  1 zeus: \??\C:\%APPDATA%\SharedSettings_1_0_5.sqlite
  ...
```



(a) Original version



(b) Infected version

Another possibility consists in deriving the FSM describing the syscall invocations carried out by the program under inspection, as Sekar et al. proposed. The FSM can be then used to check for malicious variation of the behavior of the original program. To this end, you can take in consideration Figure 4.4 that depicts the FSM derivation of a toy program (Listing 3). The FSM on the left represents the safe execution while the second path represent the same program being abused to spawn a command interpreter.

Listing 3: Toy program example used to demonstrate the FSM derivation

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  if (argc == 1)
    printf("Nothing␣harmful");
  else
    system("cmd");

  return 0;
}
```

As you can imagine we can use the FSM to track down possible misuses of the program quite easily. Indeed the framework allow us to do so with just two lines of code.

5 Conclusion and future works

Although we were not able to bring any sort of novel contribution, we are still confident in the versatility of the tool we developed. Here we just briefly mention future possibilities and functionalities that can be implemented by with very little effort.

First of all we remind that the huge level of information provided by the logging capability of the tracer can be better utilized. To this end one might think to introduce some sort of coalescing and grouping capability to the parser, in order to put in relation several syscalls that share a given goal. For examples syscalls such as `NtCreateFile`, `NtWriteFile`, `NtCloseFile` are usually called on a given handle. So grouping this high-level information could in some way be useful in order to see how the program operates on certain resources.

Another possibility includes exploiting the functionalities offered by PIN framework and our tool in order to create an AppArmor like tool. AppArmor is essentially an enforcement tool that by loading a configuration file describing all the resources and operations a given program has access to is capable of denying operations that are not listed in this profile.

The last possibility would be the introduction of some sort of taint analysis in the syscall tracer. This mechanism would allow us to discover potential compromisable paths in the control flow of the program under inspection quite easily. The information gathered can be used to complement a blackbox fuzzing analysis.

References

[1] Mihai Christodorescu, Somesh Jha, Sanjit A. Seshia, Dawn Song, and Randal E. Bryant. Semantics-aware malware detection. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, SP '05, pages 32–46, Washington, DC, USA, 2005. IEEE Computer Society.

[2] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th conference on USENIX Security Symposium - Volume 7*, SSYM'98, pages 5–5, Berkeley, CA, USA, 1998. USENIX Association.

[3] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: Malware analysis via hardware virtualization extensions. In *In Proceedings of the 15th ACM Conference on Computer and Communications Security*, pages 51–62, 2008.

[4] Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji. Intrusion detection using sequences of system calls. *J. Comput. Secur.*, 6(3):151–180, August 1998.

[5] Mathur Idika. A survey of malware detection techniques. 2 2007.

[6] Ruby B. Lee, David K. Karig, John Patrick McGregor, and Zhijie Shi. Enlisting hardware architecture to thwart malicious code injection. In Dieter Hutter, Gunter Muller, Werner Stephan, and Markus Ullmann, editors, *SPC*, volume 2802 of *Lecture Notes in Computer Science*, pages 237–252. Springer, 2003.

[7] Raymond W. Lo, Karl N. Levitt, and Ronald A. Olsson. Mcf: a malicious code filter. 14(6):541–566, 1995.

[8] Yoshinori Okazaki, Izuru Sato, and Shigeki Goto. A new intrusion detection method based on process profiling. In *Proceedings of the 2002 Symposium on Applications and the Internet*, SAINT '02, pages 82–91, Washington, DC, USA, 2002. IEEE Computer Society.

[9] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, SP '01, pages 144–, Washington, DC, USA, 2001. IEEE Computer Society.

[10] R. Sekar, T. Bowen, and M. Segal. On preventing intrusions by process behavior monitoring. In *Proceedings of the 1st conference on Workshop on Intrusion Detection*

and Network Monitoring - Volume 1, ID'99, pages 4–4, Berkeley, CA, USA, 1999. USENIX Association.

[11] A. H. Sung, J. Xu, P. Chavez, and S. Mukkamala. Static analyzer of vicious executables (save). In *Proceedings of the 20th Annual Computer Security Applications Conference*, ACSAC '04, pages 326–334, Washington, DC, USA, 2004. IEEE Computer Society.

[12] Ke Wang and Salvatore J. Stolfo. Anomalous payload-based network intrusion detection. 2004.