Gorille sniffs code similarities, the case study of Qwerty versus Regin

Guillaume Bonfante, Jean-Yves Marion, Fabrice Sabatier

Gorille sniffs code similarities, the case study of Qwerty versus Regin

Guillaume Bonfante
Université de Lorraine
LORIA
bonfante@loria.fr

Jean-Yves Marion
Université de Lorraine
LORIA
marionjy@loria.fr

Fabrice Sabatier
INRIA
LORIA
fsabatie@loria.fr

Abstract

In the last decade, our group has developed a tool called `Gorille` which implements morphological analysis, roughly speaking control graph comparison of malware. Our first intention was to use it for malware detection, and this works quite well as already presented in [2]. However, morphological analysis outputs a more refine output than 'yes' or 'no'. In the current contribution, we show that it can be used in several ways for retro-engineering. First, we describe a rapid triggering process that enlighten code similarities. Second, we present a function identification mechanism which aim is to reveal some key code in a malware. Finally, we supply a procedure which separate different families of code given some samples. All these tasks are done (almost) automatically seen from a retro-engineering perspective.

1 Introduction

In november 2014, Symantec and Kaspersky reported the existence of a 'Highly sophisticated' malware called `Regin`. The announcement followed the reverse-engineering of the malware which is a difficult task. Since the quality of the defenders's answers depends largely on the analysis of the malware, it becomes crucial to help the software investigators with automatic tools. In this contribution, we report the experiments that lead us to the conclusion that `Qwerty` is a plugins of `Regin`. One of the major challenges was to perform such an analysis with as less human intervention as possible. Our analysis gives also evidence that there are three families of `Regin`'s loaders. Finally, our experiments reveal that `Qwerty` is a relative of `Regin`, and we make a precise correspondence between their shared code. One may argue that the facts we are presenting later were known in a large extent. That is right, but such analyses were essentially done manually. Here, we show how to use our automatic tool to quickly recover such informations. Second, from an epistemological point of view, we took a known case study to make comparisons with other analyses and thus assess the reliability of our method. Indeed, morphological analysis involves an abstract layer which could lead a priori to false positives, that is codes which are considered similar when they are not. The experiments show that this is not an issue.

In Section 2, we show how we treat breaking news about malware. This daily work is essentially malware collection. We apply automatically and blindly `Gorille` to catch interesting relationships between malware. Section 3 presents the way we react to some hot cases, in this contribution `Regin` and `Qwerty` as guest stars. We do a functional identification of common code. Finally, Section 4 and 5 show how morphological analysis can be used for deeper retro-engineering of malware.

1.1 Morphological Analysis

Let us present morphological analysis in a nutshell. For deeper explanations, we refer the reader to [2]. When we are given a malware, we use two forms of analysis. First, we apply a static analysis procedure, that is we disassemble (1, see Fig 1) the code without running it. We

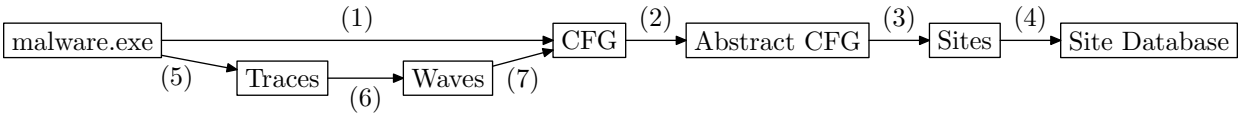


Figure 1: The Gorille data flow

extract from it its control flow graph, we abstract (2) it (we remove sequential instructions and some jumps), and finally, we cut it into small pieces (3), next called sites (graphs of size around 20). In a second step, we run the code it in a safe environment (5) which is made invisible to malware as much as possible. More precisely, we collect at any steps of the computation the value of registers (in particular the instruction pointer) and RAM cells. Then, we apply a self-modification analysis (6) of the runs which outputs RAM snapshots–next called waves–corresponding to key stones in the malware temporal evolution (see [4]). Then, for each wave, we build their control flow graphs (7) which we will abstract in a simpler form as explained above. In a third step, we reassemble static and dynamic analysis. We sum up the chain in Figure 1.

In the end, we keep for each sample the set of its sites. Technically, we build the map whose keys are hash of sites and values malware names. Executable are compared relatively to their sites.

2 The timeline of experiments

So, let us come back to november, the 24th of 2014. Kaspersky [7] on one side and Symantec [11] on an other one showed up on their respective sites the existence of a new "highly sophisticated" malware named Regin. Its technical construction made the malware stealthy, and it is considered that the malware worked for years before being noticed.

That same day, the american magazine "The intercept" published on its site [5] some samples of Regin gathered by the software security community. The first question that came to our mind was to see if Regin could be related to a strain of known malware. In particular, we wanted to know whether the new malware belonged to the "tilde" platform that was put to light by Kaspersky, that is the one which was used to develop celebrities such as Stuxnet, Duqu, Flame or Gauss.

Coming back at the High Security Lab, we applied morphological analysis to the case, and without ambiguity, we could state that Regin is in no (technical) way related to Stuxnet and its family. For a more complete presentation, we refer the reader to our previous opus [4] that describes the relationship between Stuxnet and Duqu. But, let us illustrate the naive application of morphological analysis by the following experiment. We wanted to see the cross correlations matrix of the different samples of Regin loaders at our disposal. Typically, for the loader L-fd92fd, we get:

```
DIST: "/The\_intercept-regin/L-fd92fd":
100.00% (619/619): "L-fd92fd"
97.76% (612/626): "L-2c8b9d"
96.43% (594/616): "L-4b6b86"
82.88% (484/584): "L-26297d", "L-bfbe8c"
78.24% (604/772): "L-744c07"
52.04% (600/1153): "L-225e9596"
39.12% (602/1539): "L-47d0e8"
31.05% (154/496): "L-b26989", "L-b505d6", "L-ecd7de33"
29.88% (599/2005): "L-01c2f3"
27.18% (137/504): "L-a0e3c52a", "L-cca18507"
17.20% (274/1593): "L-20831e82", "L-7553d4a5"
15.72% (69/439): "L-1c024e", "L-5c81cf82", "L-ba7bb6"
9.44% (37/392): "L-d240f0"
9.25% (37/400): "L-6662c3"
9.18% (37/403): "L-b29ca4"
9.18% (37/403): "L-a6603f27"
8.51% (273/3207): "L-f89549fc"
8.35% (37/443): "L-ffb0b9"
6.44% (25/388): "L-187044"
6.31% (25/396): "L-06665b"
3.79% (51/1346): "L-d42300fe"
```

What to see in that output? Consider for instance the line

$$52.04\%(600/1153) : \text{"L} - \text{225e9596"}.$$

It says that the loader L-225e9596 contains 1153 sites and that on these 1153 sites, 600 are common with the ones of loader L-fd92fd. For the ones

who followed, the first line shows that 619 sites of L-fd92fd are in L-fd92fd! Let us go a little bit, we distinguish some families of loaders, some are very close such as L-fd92fd, L-2c8b9d and L-4b6b86, some have identical common shared code, (L-b26989, L-b505d6, L-ecd7de33) and some are clearly different. The alerting threshold is 5%, beyond we begin to look closely at the case.

2.1 From Regin to Qwerty and the other way around

The 17th of January 2015, the link between Regin and the "five eyes alliance" jumped out at us. Indeed, the new revelations on the Snowden's case are published by the german newspaper "Der Spiegel" [10]. It is stated that the alliance of the five make use of a keylogger called Qwerty composed of a driver and two DLLs. After extraction of the binary code, we applied once again morphological analysis. Against Stuxnet, Duqu, Flame or Gauss, we did not get any better results than what we had with Regin, but on Regin itself, the result was really interesting. Let us learn two samples of Regin named Orchestrator-e420d0cf and Orchestrators-41391495:

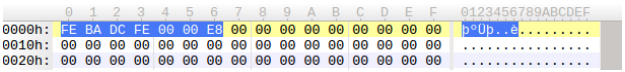
```
./sigtool --learn-sub -f raw -r alo
                  regin.db Orchestrator*
OK: Orchestrator-e420d0cf, 23622/49027
OK: Orchestrators-41391495, 23390/46489
```

So, the learning process is OK, and respectively 23622 and 23390 sites where learnt. But, we can go further. First, static analysis and dynamic analysis output the same result. This is a witness that the code is not obfuscated. Paradoxically, this is good for the stealthiness of the malware: it is *very* normal code. Second, the high level (higher by some magnitude degree compared with what we had with loaders above is nice: it means that we should get some very precise conclusion. Let us now match the samples with Qwerty:

```
./sigtool --dist-sub -r alo regin.db
                              QwertyKM/qwerty/
DIST: "QwertyKM/qwerty/20120.dll":
    100.00% (118/118): "Orchestrator-e420d0cf"
    100.00% (118/118): "Orchestrators-41391495"
DIST: "QwertyKM/qwerty/20121.dll":
    100.00% (98/98): "Orchestrator-e420d0cf"
    100.00% (98/98): "Orchestrators-41391495"
```

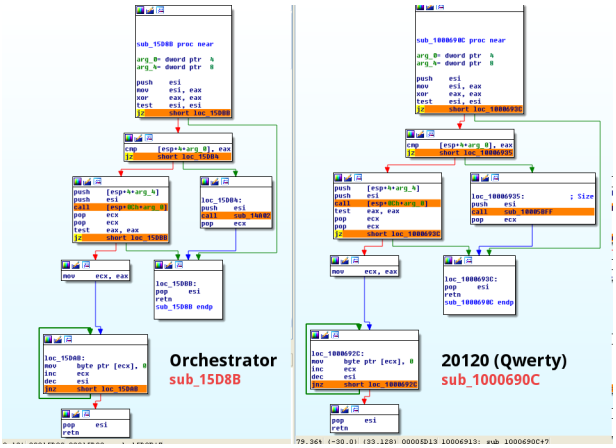
So, even if the Qwerty site amount is relatively small, they perfectly match with Regin's. We made a manual verification of shared sites which lead us directly to the conclusion that the code of Qwerty could be found in the dispatcher of Regin. Compared to Kaspersky's analysis as described in [6], it is not a correspondence with the module " 50251 Keyboard driver hooking" which is here established but a deeper correspondence at the core level of Regin (Stage 4 (32-bit) / 3 (64-bit) dispatcher module, disp.dll).

A short hint for those who would like to perform the experiment at home. The samples of the dispatcher (Ochestrator) are no executable files but memory dumps as underlined by "The Intercept": "The malware zeroes out its PE (Portable Executable, the Windows executable format) headers in memory, replacing "MZ" with its own magic marker 0xfedcbafe.".



3 Family links

At that point, we knew that there is a correspondence, we entered a new phase of work, the retro-engineering step. First, we wanted to establish a precise relationship between the two malware. Let us try to characterize the common functionalities. For that, we reemploy the method described in [3]. We extract maximal graphs extracted from common sites. Then, one recovers original instructions on both sides. On 20120.dll and Orchestrator-41391495, the result is the following excerpt from our IDA-pro plugins.



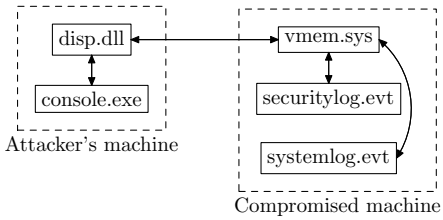
Underlined orange instructions are those that were identified as common on both sides. A careful reading shows that the correspondence is not perfect at the instruction level. Notice for instance that there is an instruction permutation in the third block from above. However, the behavior of both procedure is equivalent. Thanks to the abstraction step of morphological analysis, we catched this similarity.

Then, what about getting an analysis at a higher level than assembly code? But before we enter such a detailed inquiry, let us say few words on `Regin` and `Qwerty`'s technical details.

3.1 `Regin`, an implementation of a SOA

There are nice studies of `Regin`'s implementation. In particular, we mention the one by Tecamac [12] which gives a lot of details. The key features of its architecture are stealthiness (e.g. via CPU load spread), confidentiality (via cryptography), availability (via redundancy), scalability (via network structure) and reliability (via careful customization design).

Broadly speaking, `Regin` is build on a Service Oriented Architecture where modules (sometimes called plugins) implement functionalities. Modules communicate via Remote Procedure Calls (RPC) either locally or on a remote computer. In particular, modules can be operated by the attacker who may be anywhere on the net. The second main ingredient is that the file system is virtualized. Data (typically some configuration files) and some modules are contained in two files, one called `securitylog.evt`, the other `systemlog.evt`.



Each module implements some self-contained functional operation such as cryptography (whose module ID is `0x000f`), networking (`0x0009`), compression (`0x000d`), virtual file system implementation (`0x0007`) and so on. On our sample, we counted 13 modules. Each module provides some internal methods (or routine). For instance, the second routine of the cryptographic module `0x000f` decrypts a buffer given its key (for a variant of the RC5 protocol).

Communications are organized through RPC as follows. First, the client builds a data-structure, a container which is initially empty. Then, it fills the attributes using functions provided within `disp.dll`. Among attributes, we mention the network identifier of the machine on which the function is computed (typically `127.0.0.1` for a local call), the module identifier, the routine identifier, two buffers, one for the inputs, one for the outputs and their corresponding size, and an attribute which differentiate asynchronous calls from synchronous ones. The data are serialized within the buffer by means of routines such as `Serialize_byte_plugin`, `Serialize_qword_plugin`, that is depending on their type.

Once the structure is built, a call is performed to `disp.dll`which itself dispatches the task. The task is received by the target machine, run locally following the same procedure, but in the reverse direction, e.g. input data are deserialized. Once the result are set and properly serialized, they are sent back to the caller.

Let us present now an other nice feature of the malware. Plugins are associated to threads (which will justify the separation of synchronous and asynchronous calls). This is done by a routine of `Regin` which we call `CreatePluginThread`. Actually, threads are called via a hook of the `kernel32.dll`'s export function `OpenProcess` which has been preliminary done by `Regin`:

```
.text:7C81E079   pushf
.text:7C81E07A   clc
.text:7C81E07B   jnb  near ptr 0D80004h
```

Then, each time OpenProcess is called, it verifies if one of the registers eax, ebx, ecx or edx contains the authentification value 0x3e23271e in which case, it is a Regin's thread call, otherwise, it jumps on the standard procedure.

This mechanism ensures the stealthiness of the malware, all threads being seen as part of kernel32.dll. It ensures also the reliability of the malware and its scalability as we will see in next subsection. However, before we continue the presentation of Regin, we want to point out that we saw the hook of the OpenProcess with morphological analysis. Indeed, the hook turns out to be self-modifying code. Recall that our first task when we compute execution trace is to split them into waves. For Regin, we observed two waves, the one before the hook, the one after it. Then, given that we get the address of the self-modification, it is easy to see that it corresponds to some kernel32.dll code address! Again, we emphasize that this is routine for us, anything but the conclusion being done automatically.

3.2 Extending Regin

Regin's architecture is made to be opened to new modules. In other words, it is possible to augment the number of modules on the fly. To join Regin's SOA architecture, each module must create in a first step its call structure and register its routines. Registration of the plugin is done by a call to PLUGIN_FUNCTIONS.Alloc_plugin at address 0xc919 with inputs being the network address, a pointer on the framework interface, the plugin identifier and a memory address offset.

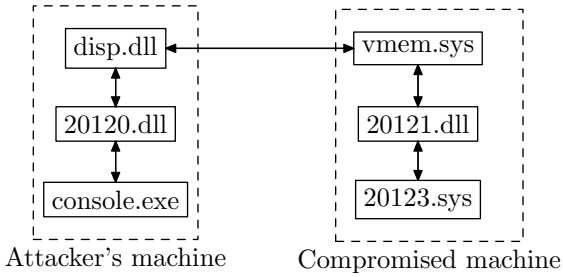
Then, routine registration is done by a call to PLUGIN_FUNCTIONS.register_plugin at address 0xc760 by giving a pointer on the structure, an identifier and the address of the method.

To conclude on Regin's architecture, seen from a software engineering point of view, what a smart code! No doubt that Regin was created by well educated coders whom skill is not to be proven.

4 Qwerty, a Regin's module

As far as we know, there is no such detailed report on the link between Regin and Qwerty. All what we present here is extracted from experiments we did at the High Security Lab.

So, first, Qwerty consists in two DLLs 20120, 20121 and a driver 20123 together with configuration files. In the present case, the architecture is updated as follows:



The DLL 20120 contains three exported functions, the first one serves for plugin registration, the second for registration release and the last to get a pointer on its own structure. This was obtained by manual observation of the code:

```
; Exported entry   1.


public _20121_1
_20121_1 proc near

arg1_framework_interface= dword ptr  4
arg_4= dword ptr  8

mov     eax, [esp+arg1_framework_interface]
mov     ecx, [eax+FRAMEWORK_INTERFACE.plugin_functions]
push    ebx             ; Number ID
push    eax             ; Framework interface
push    20121           ; Plugin ID
push    offset var_g_plugin_20121_plugin_interface
xor     bl, bl
call    [ecx+PLUGIN_FUNCTIONS.Alloc_plugin_interface]
add     esp, 0Ch
test    al, al
jz      short loc_3910E3
```

This is a witness that Qwerty is a plugin for Regin. Then, as expected, we see plugin registration:

```
39379F  mov   eax, var_plugin_interface
3937A4  mov   ecx, [eax+framework_interface]
3937A7  mov   ecx, [ecx+plugin_functions]
3937AA  push  ebx
3937AB  push  offset 20120_Method_1
3937B0  push  1
3937B2  push  eax
3937B3  xor   bl, bl
3937B5  call  [ecx+register_plugin_method]
```

The DLL 20120.dll contains 20 routines, with identifiers 1—9, 14, 15, 20, 21, 24, 25, 32—35, 233. Actually,

20120.dll is distributed with xml description file detailing command use. Here is an excerpt of the commands:

```
<command id="1">
<name>loggingStatus</name>
<description> This command will return the
current status of the Keyboard Logger (Whether
it is  currently turned "ON" or "OFF").
</description>
<examples> <example>qwstatus</example>
</examples>
<alias> <aliasName>qwstatus</aliasName>
<aliasDef>20120 1</aliasDef> </alias>
</command>
<command id="2">
<name>TurnLoggingOn</name>
<description> This command will switch ON
Logging of keys. All keys taht are entered to
a active window with a title on the target list
will be captured </description>
<examples> <example>qwstart</example>
</examples>
<alias> <aliasName>qwstart</aliasName>
<aliasDef>20120 2</aliasDef> </alias>
</command>
```

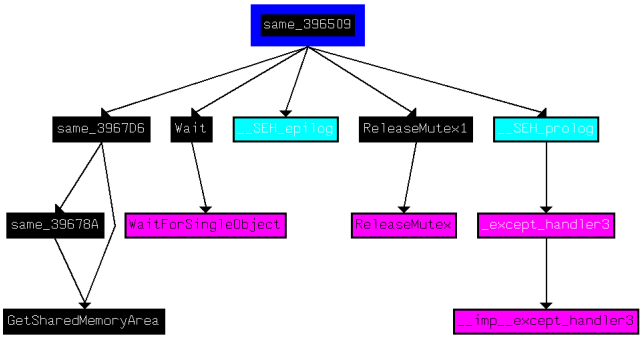
After that setup phase and the one of DLL 20123 and 20121, the plugin is operational. At that point, one may run some commands in the console. Typically, a call to qwlist which outputs the list of currently listened window gives:

```
0130FD08  <.?.x.m.l. .v.e.r.s.i.o.n.=.".1.
0130FD28  ..0.". .e.n.c.o.d.i.n.g.=.".U.T.
0130FD48  F.-.8.".?.>...<.r.e.s.p.o.n.s.e.
0130FD68  x.m.l.n.s.:.x.s.i.=.".h.t.t.p.
0130FD88  :././.w.w.w...w.3...o.r.g./.2.0.
0130FDA8  0.1./.X.M.L.S.c.h.e.m.a.-.i.n.s.
0130FDC8  t.a.n.c.e. ....
```

Everything is fine! But, by the way, why did we found a link between Regin and Qwerty? Let us come back to morphological analysis. We got a code correspondence between instructions within Regin and Qwerty. The correspondence can be lifted up to the function level. Doing so, we got (partially cut):

disp.dll	20120.dll
sub_1003CE47	same_39379F
Substract	same_394F97
zfreeWithSize	same_395BD9
GetDaysSinceStartOfYear	same_395F03
FreeStreamBuffer	same_39609C
sub_1000609C	same_396509
ReleaseMutex	same_3964D7
sub_1000B316	same_3967D6
sub_1000B2CA	same_39678A
GetSharedMemoryArea	same_39664
Wait	same_3964B0
...	...

Actually, within the table, we played the game fairly: those functions with names are the one that we analyzed in the past. Function prefix 'same' indicate that the function has been recognized identical to an other one. Thus, any retro-engineer would immediately recognize previous enquiry. The correspondence is extracted from the output of Gorille on the computation of the similarity of Qwerty given the site database corresponding to Regin. The result is here displayed by an IDA-plugin that is freely available. In that same plugin, we propose the function call graph:



Black nodes are functions recognized similar between two malware. Cyan nodes correspond to functions of the standard library which are inserted as malware code and purple ones to function call to the standard library. Again, the coloring is an application of morphological analysis.

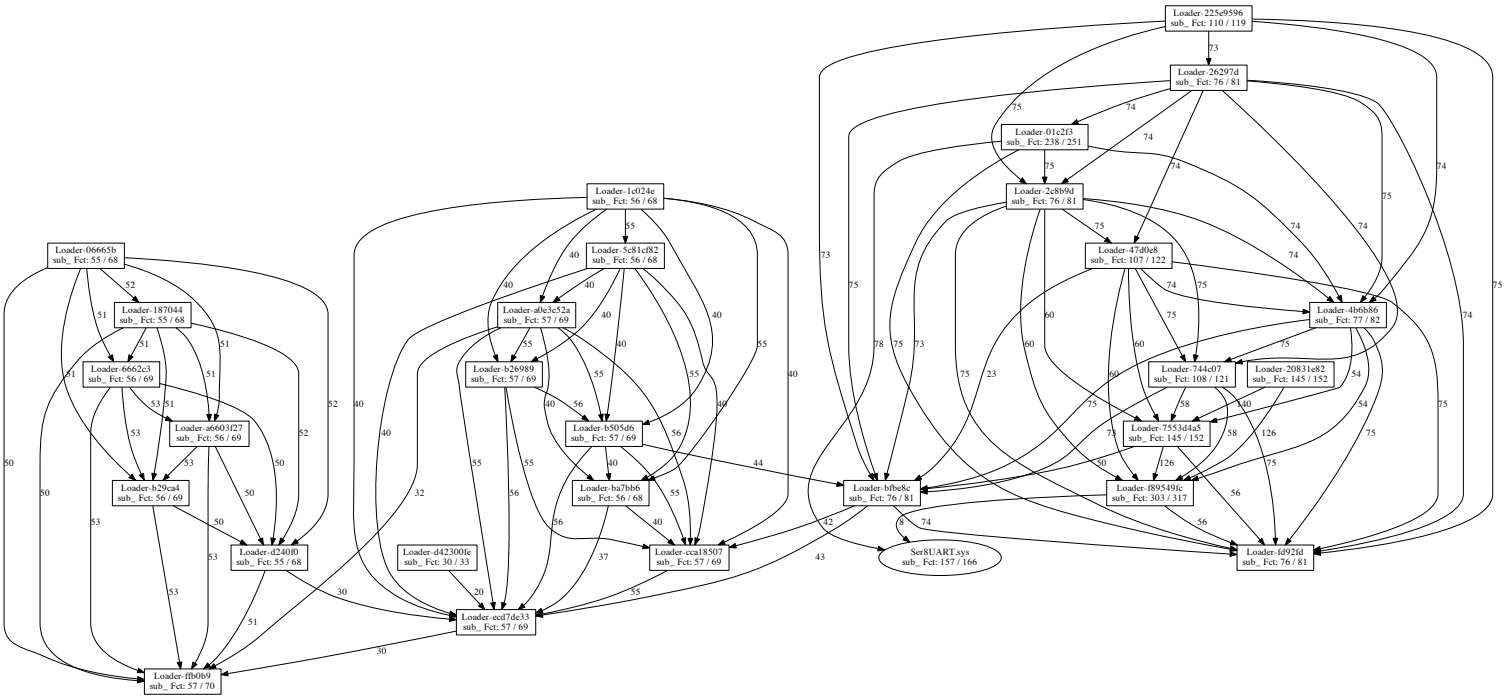


Figure 2: The three families of `Regin`'s loaders.

5 A family analysis

Let us come back for a while to `Regin`'s loaders. We could observe that they are all a little bit different one an other. This gives us the possibility to show an other application of morphological analysis. Given similarity evaluations, we extract distance notions (there are many possibilities here: number of common site, Hamming distance, largest common graph, and so on) from which we build a weighted graph. It is then possible to make a small number of sub-families emerge. Doing so, the retro-engineer may apply his analysis to central nodes which are representing their families.

In the following example, we provide the graph which has been obtained by measuring the number of common functions. Doing so, we provide a form of function classification which is highly valuable when it is time not to spear some. Here is the example with the loaders mentioned at the beginning:

On this graph, there are many side informations. Consider for instance the loader `L-06665b` upper left. It contains 68 functions, 55 of them are not OS ones. The edge to `L-187044` shows that 52 functions are common to these two loaders. In other words, almost 95% of common functions! In other words, you can forget about one.

Naturally, this classification opens new questions. For instance, we could not relate the families to the infection mechanism or to targets (in terms of geography, sociology, etc). We let that idea for future works.

6 Conclusion

The case study shows that morphological analysis is a nice way to help retro-engineering. We show that it can be used to connect identical pieces of code and even better, almost identical pieces of code. We show that it serves to make function identification and finally, that it can be reinterpreted to build some classifying families.

We are not alone to use graph to compare binary executables. There are nice approaches on the topic. We want

to mention the typical work by the Louisiana group [8] which–compared to us–relies partially on some semantics consideration. An other example is given by [1] which focuses on function identification in binary malware. Our tool involves this abstraction step which gives it a little bit different flavor.

The notion of wave that we use is a little bit different to the one introduced by Dalla Preda, Giacobazzi and Debray [9] (called phase). The main advantage of our approach is that waves can be computed on the fly. In particular, as the experiments showed it, we could recognize the hooking of the `OpenProcess` kernel function at run time.

Finally, the question is not addressed here, but we aware any reader who would like to reproduce the experiment that it is necessary to follow all thread corresponding to the DLL and `kernel32`. Samples that were used in the contribution are available on demand.

References

[1] S. Alrabaee, P. Shirani, L. Wang, and M. Debbabi. Sigma: A semantic integrated graph matching approach for identifying reused functions in binary code. *International Journal of Digital Investigation*, December 2014. Accepted for Publication (To Appear).

[2] Guillaume Bonfante, Matthieu Kaczmarek, and Jean-Yves Marion. Architecture of a Morphological Malware Detector. *Journal in Computer Virology*, 5(3):263–270, 2009.

[3] Guillaume Bonfante, Jean-Yves Marion, Fabrice Sabatier, and Aurélien Thierry. Recognition of binary patterns by morphological analysis. In *RECON 2012*, 2012.

[4] Guillaume Bonfante, Jean-Yves Marion, Fabrice Sabatier, and Aurélien Thierry. Analysis and Diversion of Duqu's Driver. In *Malware 2013 - 8th International Conference on Malicious and Unwanted Software*, Fajardo, Puerto Rico, October 2013. IEEE.

[5] The intercept, 2014. https://firstlook.org/theintercept/2014/11/24/secret-regin-malware-belgacom-nsa-gchq/.

[6] Kaspersky. http://securelist.com/blog/research/68525/comparing-the-regin-module-50251-and-the-qwerty-keylogger/.

[7] Kaspersky, 2014. http://25zbkz3k00wn2tp5092n6di7b5k.wpengine.netdna-cdn.com/files/2014/11/Kaspersky_Lab_whitepaper_Regin_platform_eng.pdf.

[8] Charles LeDoux, Arun Lakhotia, Craig Miles, Vivek Notani, and Avi Pfeffer. Functracker: Discovering shared code to aid malware forensics. In *Presented as part of the 6th USENIX Workshop on Large-Scale Exploits and Emergent Threats*, Berkeley, CA, 2013. USENIX.

[9] Mila Dalla Preda, Roberto Giacobazzi, and Saumya Debray. Unveiling metamorphism by abstract interpretation of code properties. *Theoretical Computer Science*, 577(0):74 – 97, 2015.

[10] Der Spiegel, 2015. http://www.spiegel.de/media/media-35668.pdf.

[11] Symantec, 2014. http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/regin-analysis.pdf.

[12] tecamac@gmail.com. Malware instrumentation application to regin analysis. http://artemonsecurity.com/regin_analysis.pdf, 2015.