Code synchronization by morphological analysis

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

Aurélien Thierry
Université de Lorraine
INRIA
athierry@loria.fr

Abstract

Reverse-engineering malware code is a difficult task, usually full of the traps put by the malware writers. Since the quality of defense softwares depends largely on the analysis of the malware, it becomes crucial to help the software investigators with automatic tools. We describe and present a tool which synchronizes two related binary programs. Our tool finds some common machine instructions between two programs and may display the correspondence instruction by instruction in IDA. Experiments were performed on many malware such as `stuxnet`*,* `duqu`*,* `sality` *or* `waledac`*. We have rediscovered some of the links between* `duqu` *and* `stuxnet`*, and we point out* `OpenSSL`*'s use within* `waledac`*.*

Let us consider a scenario in which some expert analyst has to understand the behavior of a suspicious program sample. In a first step, the programmer will typically run a disassembly software such as IDA. From that point, he will by-pass the packer's obfuscation to get the core code of the malware and, in a last step, he will delineate the key functions performed by the core code. Doing so, the expert will highlight some crucial part of the core code. The protection techniques—malware detection, malware removal and operating system restoration—will be derived from this analysis.

Writing a malware involves an important investment. It is then not surprising –as recent case studies have shown it– that one finds strong relationships between some malwares. The standard case consists of two variants of a malicious program, but such relationships may also extend to some apparently independent malware. For instance, `duqu` and `stuxnet` share a common infection scenario, but more importantly for us, they also share some binary code (see the report by Symantec [8, 1] or Szor's description of `duqu` [9]). An other point concerns libraries. Any inclusion of a library in the code will facilitate the investigation as long as the library has been recognized. We illustrate the point with `waledac` which endows some routines of `OpenSSL`.

This contribution presents a tool which establishes some correspondence between two binary programs. In the scenario described above, our tool can be used to recognize some parts of an already known malware—whose analysis would have been performed in the past— within the new program sample. The program expert can then refer to the previous analysis to save time on the current one.

Such instruction matchings of malware have already been considered in the past. It is the central point of the detection technique of Christodorescu et al [6]. The tool BINDIFF[1] which aims at making such correspondence has also been applied to malware analysis, see [7] for instance. Our technique differs from these contributions by the techniques involved to compare programs. First, we use a tree-automata technique to perform the sub-isomorphism, and second, we transform programs by a non semantics preserving procedure. This latter point could introduce false positive. However, in the context of detection, we have shown in [5] that there level remains very low, under 1%. Anyway, false positive is not as crucial as it is for detection. Indeed, suppose the system provides me a wrong matching. Bad, I will lose time to see it, but *actually* I won't miss a similarity.

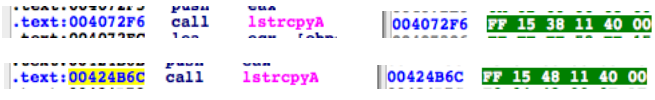
[1] `http://www.zynamics.com/bindiff.html`

Let us come back to our job. Given two program samples, due to Rice's Theorem, there is no general procedure to extract precisely their shared components. Actually, one even cannot expect to disassemble properly the two programs. However, in the scenario proposed above, coming after the manual analysis done by the expert programmer, we can make the hypothesis that we have actually access to the core code of the two malware. In that case, computing the correspondence becomes decidable. Nevertheless, issues do not wipe out at that point: indeed, there is still a complexity issue.

Given two disassembled executables, a naive procedure performing matching of a program `prg` of size n against a program `prg`$'$ of size m takes $O(n^2 \times m)$ steps. Applying it on some samples of size of the order 10^5 and 10^4 (e.g. `stuxnet` and `duqu`), one gets an amount of 10^{14}, that is almost unfeasible. Roughly speaking, to solve this issue, our tool first performs an abstraction of the input codes and only then runs a (clever) matching procedure.

As a by product of the abstraction procedure, our method is much more resistant to code obfuscations such as instruction replacements (e.g. `mov eax,0` turned into `xor eax,eax`), code reordering (by means of `jmp`), register swaps (`eax` substituted by `ebx`) and many other small tricks. It can even support some larger modification of the code whenever they do not extensively modify the shape of the Control Flow Graph (CFG). We will give an example on `sality`.

To make the correspondence between two programs, it is almost mandatory to work on the CFG of the programs rather than the binary code itself. Indeed, due to hard encoded addresses within the binary code, a character by character matching is irrelevant. For instance, consider the two addresses `0x424B6C` in `sality`.1 and `0x4072F6` in `sality`.2, seen byte per byte, the two strings are different even if they correspond to the same instruction, as shown by IDA.



Since CFG are the key structures of Morphological Analysis, a methodology developed by our group at the High Security Lab[2], it is natural to consider the problem of code synchronization in these views.

To sum up, our contribution: in a first step, we describe Morphological Analysis. In a second step we define more precisely the synchronization problem, and we show how Morphological Analysis can be used to solve it. Then, we will refine the analysis and we will point out that Morphological Analysis leads to approximate, robust synchronization. We end this paper by some experiments done at the High Security Lab.

1 Morphological Analysis

Morphological Analysis is a general methodology used to extract informations from binary programs. It can be used in various contexts such as virus detection [4] or libraries identification [3].In the present paper, we describe how morphological analysis can be used to resynchronize addresses in binary codes. We provide here only the main steps of morphological analysis. A longer description will be found in our former contribution [5].

Up to now, Morphological Analysis was used to perform detection, not binary synchronization. Though the formal objects used in both cases are always isomorphism identification on control flow graphs, the constraints are very different.

On the one hand malware detection needs to optimize the speed of the detection algorithm. In this case a false positive is easily defined as an innocuous program being detected as a malware and a false positive rate can be computed given a sample of malwares and harmless binaries. On the other hand comparing two (or more) binaries aims at helping an analyst, it doesn't need to be done in seconds. This allows the use of more complete algorithm on graph isomorphism. Besides, the criteria of false positive is not as clear: in some way, it is a matter of taste of the analyst who will evaluate if the two codes are sufficiently near to be considered equivalent.

1.1 MA-signatures

An *MA-signature* $\mathcal{M}$ is a set of *sites*, these are labeled graphs, each of them represents a semantic preserving abstraction of a piece of the control flow graph of some binary executable file `prg`. Roughly speaking,

[2] `http://lhs.loria.fr/`

the nodes of a site correspond to key control instructions of the program.

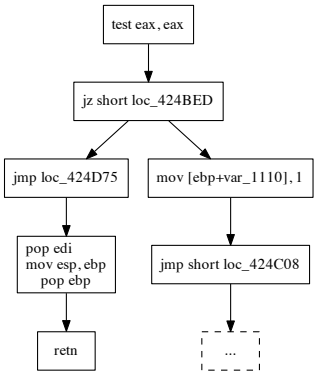
The nodes of sites are mapped to addresses in the program. There is an edge between node n_1 and node n_2 if, for some memory configuration, executing instruction n_1 leads to the execution of n_2. The labels of sites range in the set $\{\texttt{jmp}, \texttt{call}, \texttt{jcc}, \texttt{ret}, \texttt{inst}\}$. They are triggering (block of) binary instructions according to their behavior. The labels speak for themselves, `jmp` corresponds to instructions involving an unconditional redirection of the instruction pointer, `call` to function call, `ret` to the return instruction, `jcc` to binary conditional jumps and `inst` to instructions which do not modify the normal control flow.

1.2 From `exe`-files to MA-signatures

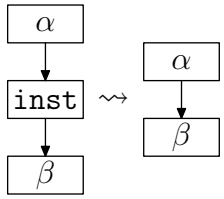
The (automatic) MA-signature extraction is done in three steps. First of all, our system scans *entirely* the binary code, disassembles it either statically or dynamically (in a secure environment). The procedure outputs (Step (2)) the control flow graph of the executable (see Figure 1).

For instance, the first instructions of the code extracted from `sality.1` at address `0x00424BE4` are given on the right. They are transformed into the following graph:

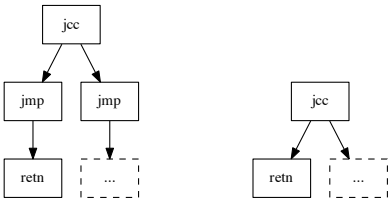
```
test eax, eax
jz short 424BED
jmp 424D75
mov [ebp+var_1110],1
jmp short 424C08
...
pop     edi
mov     esp, ebp
pop     ebp
retn
```



In a third step, the system applies some transformations rules to the CFG, closed to the one presented by Aho et al [2] for code optimization. We have defined three sets of rules. The first one–called O-transform–merges any instruction of type `inst` which is followed by an other instruction with that latter instruction. The process is run until fixpoint. On the right, the rule.



The second transformation–called L-transform–removes unconditional jumps. The A-transform removes some conditional jumps and empty calls. Applied to our running example, after O and O+L-transform, we get:



In a fourth step, the system extracts relevant sites from the CFG. In particular, the system removes any too small site and any sites from a *white list*. This white list is composed from sites which are considered to be irrelevant. For instance, libraries of the Microsoft Windows distribution, the standard library `libc` and a certain number of degenerate cases. Then, the MA-signature is compiled to a tree-automaton to ensure the efficiency of the matching procedure. Finally, a wrapper to python is produced for use in IDA.

The procedure is summed up in Figure 1.

1.3 Back to `exe`-file

As mentioned above, the node of a site contains the address of its corresponding instruction. We have developed a plugin–which will be soon freely available–for IDA which can be used to get a visual presentation of sites. Technically speaking, MA-signatures can be presented as a folder containing a bundle of sites, each site being presented as a graph in the standard `*.dot` format. Each node of a site must be presented according to the format:

n [label="instruction type"]

where n refers to the address of the instruction within the program and "instruction type" is in `INST, JMP, JCC, RET, CALL`.

The following figure gives a partial view of a site extracted from `duqu`. The instructions corresponding

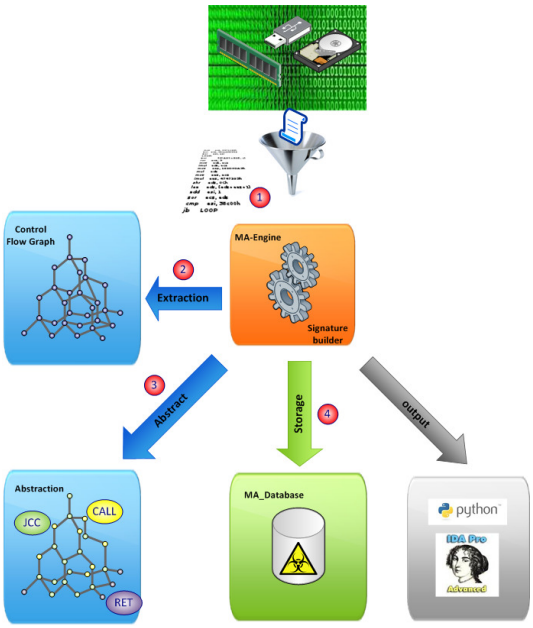
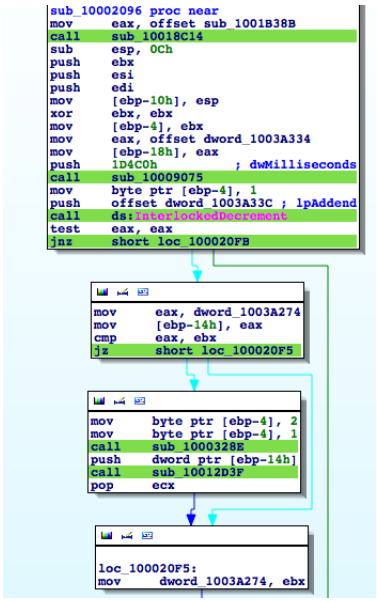


Figure 1. Learning MA-signatures

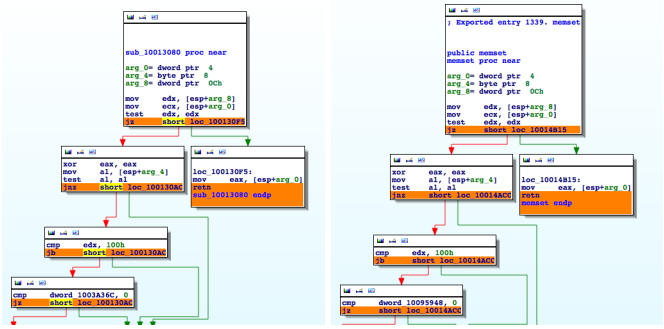
to nodes of the site (here after O-transform) are highlighted in green.



2 Synchronizing code

Let us formalize the synchronization problem. We suppose given two binary code `A.exe` and `B.exe`. The output of the synchronization will be a correspondence between some instructions in `A.exe` and some instructions in `B.exe`. The correspondence is intended to render some shared internal structure of the two executables. As a matter of fact,we mean that the two programs share locally the same instructions. Thus, more technically, this amounts to find a subgraph A' within `A.exe`'s CFG and a subgraph B' within `B.exe`'s CFG containing respectively the addresses i and j such that A' and B' are isomorphic and *this* isomorphism maps i to j. We will use the word *correspondence* to speak about the mapping from instructions to instructions and the word *matching* to speak about the common subgraph of the two CFG. A correspondence may involve many (disjoint) matchings.

For instance, there is a partial correspondence between `duqu` on the left and the `libc` on the right, which, in other words, means that `duqu` contains some procedures of the `libc` (not surprising!).



When is a matching interesting? First point, the larger is the common behavior, the more characteristic is the correspondence. Indeed, if the two subgraphs are too small, some correspondence may be inappropriate: some common subgraphs between two binary codes may emerge from randomness, or simply because they are patterns of compilers. To illustrate the point, in the extreme case, think of a sub-graph of size 1, that is one instruction, say `ret`. There is no reasons to match a `ret` on one side to any other `ret` on the other side. Concerning compiler patterns, consider the typical sequence on the right. It corresponds to a standard closing of a function definition. Again, there is no reasons to make the correspondence between such a sequence with an other one.

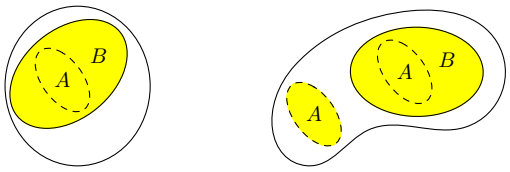
```
mov esp,ebp
pop ebp
ret
```

Second point, consider the case when the expert wants to make the link between two malwares. Then,

knowing that the two malwares share some routine from a standard library is not particularly interesting[3] in general. In the malware analysis, one puts the focus on the cores of the two codes. Thus, it is reasonable to remove some irrelevant matchings. Such a consideration is taken into account by the white list filter.

Third point, to make sure a correspondence is meaningful, let us come back to the earlier experimental evaluations in [4]. We were showing that taking graphs–outside the white list–above 15 nodes leads to a good separation between malware and safe programs. More quantitatively, the ratio of false positive is 0.7% for common subgraphs of 13 nodes and 0.1% for 15 nodes. In the present settings, we took 24 nodes as a basis, so, clearly beyond 15. At that level, there are very few chances any correspondence to be fortuitous.

Finally, we will give priority to the largest matchings. What does that mean? For instance, consider two graphs G_1 and G_2 such that G_1 contains a graph B containing itself some subgraph A and G_2 contains two copies of A but one is surrounded by B. Then, the correspondence will map nodes in A in G_1 to the nodes A contained in B in G_2.



3 Speed up!

As mentioned in the introduction, the main issue to synchronize codes of two binary executable is its computational cost. Indeed, in general, synchronization takes $O(n^2 \times m)$ for codes of respective size n and m. Since the size of data is usually relatively high, for instance, `stuxnet` is a 1.2Mb file, `duqu`, 376 Kb, `sality`, 250Kb and `waledac`, 1.5Mb, a naive algorithm would simply fail. One could argue that the right measure is the number of instructions, not the size of the files, but even doing so, after disassembly, the number of instructions in the text section are respectively 11×10^4, 3.8×10^4, 2.4×10^4 and 20×10^4.

[3] Don't miss the point: knowing that a malware involves some library may be of interest, but the fact that two malwares both call say "strcpy" does not give insights on their distinctive behavior.

To get around the issue of efficiency, we propose a) to consider to an abstract analysis, that is to make use of the O+L+A-transforms and b) to use a bottom-up/top-down technique in the style of Ullmann (see [10]). As a matter of facts, after a finer analysis of the procedure, we will show that we actually provide a much more robust tool against modifications of programs.

But first of all, one may benefit from the asymmetry of the problem: in the scenario presented in introduction, the expert programmer performs the matching of one graph against a known one. Then, we can take profit from Malware Analysis: as an offline process, our system collects sites (that is potential matchings) in a tree automaton. After this learning step, the complexity of the recognition of a matching in a program is $O(n \times M)$ where n is the size of the program and M is the size of the matching.

However, learning all sites of all sizes would lead to a gigantic tree-automaton, which practically cannot be done. Indeed, in the general case, the number of subgraphs of some graph is exponential in the size of the graph. Thus we only learn sites of some fixed size. In terms of Constraints Programming, these matchings are sharp constraints on the set of the solutions.

3.1 Abstract analysis

The key idea of abstract analysis is to perform synchronization on transformed CFG (by one of the O, L, A-transforms or some of their composition) and not on the original graphs. After this step, the correspondence found on transformed CFG can be raised back to the original code as we have seen in Section 1.3. Two good points must be noticed about abstract analysis.

First, the transformations respects the graph structure of CFG, that is if two programs share a subgraph, their transforms will share the transformed subgraph. In other words, the algorithm is complete: we cannot miss a solution.

The second good point is that the three code transformations (O-transform, L-transform, A-transform) output programs representations which are smaller than their inputs. The following tabular gives for our running examples the size of each of the graphs after transformation:

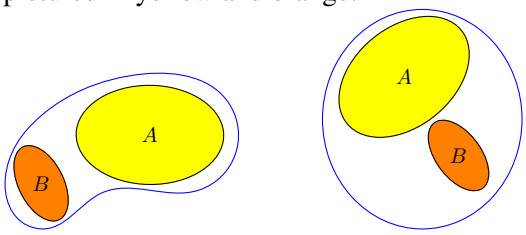
Exec	original	O	O+L	O+L+A
`stuxnet`	11×10^4	19508	17629	15751
`duqu`	3.8×10^4	8152	7095	5863
`sality`	2.4×10^4	675	511	471
`waledac`	20×10^4	18155	16000	14626

More or less, we obtain a size compression factor around 8 for the (O+L+A)-transform. Then, mechanically, this will influence the complexity of the recognition of matchings, themselves being smaller. Since the size of both graphs, the source graph and the target graph are lowered by a factor 8, the transform will speed up the synchronization by $8^3 = 512$.

To conclude, the stronger is the transformation, the faster is the algorithm. However, the algorithm is not safe: we will show that some solutions found on the abstract forms are not solutions of the initial problem. We will come back to this problem in the next section.

3.2 Bottom-up/top-down technique

The principle of the technique is to orient search according to the size of matchings. Recall that the larger are the matchings, the higher is the relevance of the correspondence. At the same time, the larger are the matchings, the more costly they are to find. Thus, it is crucial to catch these large matchings in the shortest time, this is the aim of the bottom-up step. However, one should not miss any relevant matching which may be smaller than the largest of them. Think of a situation with two programs sharing two matchings A and B pictured in yellow and orange:



The B matching, though not as large as the A one, remains interesting. The second step (top-down) rakes up smaller matchings.

For the bottom-up phase, we begin to search for sites of size 24, then for sites of size 48, then 96 and so on until there is no more matching. Then, by dichotomy, we find the largest of the matchings, say an isomorphism $\phi : A \to A'$ with A in G_1 and A' in G_2. The variable `map` gathers the correspondence.

```
foreach i in A
    map.put(i,phi(i))
```

For the top-down phase, we begin by the largest of the matching, and we decrease 1 by 1 the size of the searched matchings until we reach the minimal size, in the present case 24. Each time a matching $\phi : A \to A'$ is found, we apply:

```
foreach i in A
    if (i not in map.keys())
     then map.put(i,phi(i))
```

thus collecting all the matchings. At the end of the top-down phase, the dictionary `map` contains the correspondence. Again, there are two good points: since sites have size smaller than the original code, finding them in some program goes faster. Second, the procedure is complete: we cannot miss any correct correspondence.

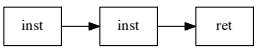
Moreover, contrary to the abstract technique, the bottom-up/top-down technique is safe: indeed, in the top-down phase, we try matchings of all size, thus covering all the solutions.

4 Precision or robustness?

We have mentioned above that the abstraction technique may raise incorrect matchings. The simplest case is instruction clash. For instance, the two sequences of instructions below would not be considered to be isomorphic:

```
mov esp,ebp          xor eax,eax
pop ebp              xor ebx,ebx
ret                  ret
```

Seen in our setting, that is as sites, they both look like:



Actually, the problem is even deeper, the structure of the graphs may be involved. For instance, the two following graphs do not share any sub-graph of size 2



but their O-transforms do:



However, experimentally, we have observed that if two programs share a sufficiently large transformed graph, with a very high expectation, the correspondence is correct, or–at least–interesting.

Anyway, from the mapping obtained on the transformed graph, one may derive a mapping on the initial code. The correspondence can then be checked at the code level. The procedure takes linear time with respect to the size of the initial program, thus more or less negligible. So, why don't we do that?

4.1 Approximate correspondence

If we refer to the initial scenario, the expert programmer tries to analyze the program sample by reference to older programs. He doesn't necessarily care whether the program has been processed with say a new version of some compiler, or, if it has been slightly modified, say by the insertion of a new variable. What he tries to do is to make some links with older programs, possibly approximated links. Thus, the notion of exact correspondence is not the right concept. There is room for some approximated correspondence.

Actually, this is precisely what (O,L,A)-transform do! Let us take benefit from it. And, moreover, since the transforms respect the semantics of programs, you have the best of both world: you cannot miss exact solutions and due to the transform, you will gather some approximated ones. We have observed this phenomenon on two versions of `sality`. Both modify the key register of the machine:

```
push    offset a914     ; "914"
lea     ecx, [ebp+String]
push    ecx             ; lpString1
call    lstrcatA
mov     edx, [ebp+var_1110]
and     edx, 0FFh
mov     eax, [ebp+var_1110]
imul    eax, edx
push    eax
push    offset aD       ; "\\%d"
lea     ecx, [ebp+String]
push    ecx             ; lpString
call    lstrlenA
lea     edx, [ebp+eax+String]
push    edx             ; LPSTR
call    wsprintfA
add     esp, 0Ch
lea     eax, [ebp+hKey]
push    eax             ; phkResult
push    0F003Fh         ; samDesired
push    0               ; ulOptions
lea     ecx, [ebp+String]
push    ecx             ; lpSubKey
push    80000001h       ; hKey
call    RegOpenKeyExA
test    eax, eax
```

and

```
push    offset String2  ; lpString2
lea     eax, [ebp+String]
push    eax             ; lpString1
call    lstrcpyA
xor     ecx, ecx
mov     cl, byte ptr ds:dword_4394E0
mov     edx, ds:dword_4394E0
imul    edx, ecx
push    edx
push    offset aD       ; "\\%d"
lea     eax, [ebp+String]
push    eax             ; lpString
call    lstrlenA
lea     ecx, [ebp+eax+String]
push    ecx             ; LPSTR
call    wsprintfA
add     esp, 0Ch
lea     edx, [ebp+hKey]
push    edx             ; phkResult
push    0F003Fh         ; samDesired
push    0               ; ulOptions
lea     eax, [ebp+String]
push    eax             ; lpSubKey
push    80000001h       ; hKey
call    RegOpenKeyExA
test    eax, eax
```

We have observed three other occurrences of such variations on the two versions of `sality`.

5 Experimental data

Waledac vs OpenSSL We present some work we have done on `waledac` and `OpenSSL`. We wanted to verify that `waledac` contains some of `OpenSSL`'s function. To make the comparison, we used the version `OpenSSL 0.9.8e` (Feb. 2007). After we have learned `OpenSSL`, we scan `waledac`:

```
sigtool --dist -r ola ssl.db Waledac48.int
DIST: "Waledac48.int":
50.0% (363/726/17873): libeay32.dll
```

showing 363 matchings between `OpenSSL` and `waledac`. Let us go a little bit into details.

First, learning and scanning phases take relatively short time (2.53GHz, Intel Core i5):

Operation	Files	Time (s)
Learn	OpenSSL (28313 nodes)	12s
Scan	Waledac (14626 nodes)	2.0s

Second, the result depends on the compiler options chosen for `OpenSSL`, not on the version of the library. As shown on the following tabular:

OpenSSL version	Comment	Common matchings
0.9.8x	Released in May 2012	53
0.9.8e	Compiled for performance (/Ox /O2)	53
0.9.8e	Compiled for file size (/O1)	1264

So that we know that `waledac` uses a version of `OpenSSL` optimized for its code size, not performance. If one wants a more functional vision, our plugins output subroutines in the IDA framework:

libeay32-098e.dll	subroutine	Waledac48.int	subroutine
10002CDE	CRYPTO_new_ex_data	00455B5C	sub_455B55
10002CE0	CRYPTO_new_ex_data	00455B5E	sub_455B55
10002D0A	CRYPTO_free_ex_data	00455B72	sub_455B6B
10002D0C	CRYPTO_free_ex_data	00455B74	sub_455B6B
10021F6D	AES_set_encrypt_key	00452C1E	sub_452C1B
10021F7C	AES_set_encrypt_key	00452C2D	sub_452C1B
10021F88	AES_set_encrypt_key	00452C39	sub_452C1B
10021F99	AES_set_encrypt_key	00452C4A	sub_452C1B
10021FA0	AES_set_encrypt_key	00452C51	sub_452C1B
10021FA7	AES_set_encrypt_key	00452C58	sub_452C1B
10021FB3	AES_set_encrypt_key	00452C64	sub_452C1B
10021FD9	AES_set_encrypt_key	00452C8A	sub_452C1B
10021FEF	AES_set_encrypt_key	00452CA0	sub_452C1B
100224D9	AES_set_encrypt_key	0045317D	sub_452C1B
100224E0	AES_set_decrypt_key	00453184	sub_45317E
100224F0	AES_set_decrypt_key	00453194	sub_45317E
100224FA	AES_set_decrypt_key	0045319E	sub_45317E

which lets us say that `waledac` implements AES for symmetric encryption, X.509 (certificate) handling, RSA and/or DSA algorithm and primality tests.

Stuxnet vs Duqu Following the same procedure, we have compared `duqu` vs `stuxnet`. It appears that 26.5% of `duqu`'s subgraph are common with `stuxnet`, these 26.5% of subgraphs actually correspond to 60.3% of the code. So that we can conclude that both malware are closely connected. We let a complete study of these links for further research.

6 Conclusion

As recent news have shown it, malware may infect any kind of systems, anywhere. We think that there is a strong need to help defenders in their sisyphean task. Understanding some code cannot be automatized in general, this remains a job for analysts. Our contribution shows one way to help him.

As our experiments have revealed it, synchronizing code still demands heavy computational resources. However, at the same time, we have shown that this is accessible even at a real scale. We have discussed the tradeoff between quality and robustness of the correspondence. First, there is a need of formalization of code approximation. Approximation may concern either data or algorithm, or both. It is clear that they involve different views corresponding to different needs. That should be studied more deeply.

References

[1] W32.duqu: The precursor to the next stuxnet, October 2011.

[2] A. Aho, M. Lam, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley 2nd edition, 2007.

[3] G. Bonfante, J. Calvet, J.-Y. Marion, F. Sabatier, and A. Thierry. Recognition of binary patterns by morphological analysis. In *Recon*, Montreal, Canada, 2012.

[4] G. Bonfante, M. Kaczmarek, and J.-Y. Marion. Morphological Detection of Malware. In *Malware '08*, 2008.

[5] G. Bonfante, M. Kaczmarek, and J.-Y. Marion. Archtecture of a morphological malware detector. *Journal in Computer Virology*, 5(3):263–270, 2009.

[6] M. Christodorescu, S. Jha, S.A. Seshia, D. Song, and R.E. Bryant. Semantics-aware malware detection. *IEEE Symposium on Security and Privacy*, 2005.

[7] Sabre-security.com. Using sabre bindiff v1.6 for malware analysis.

[8] Symantec. W32.stuxnet dossier, Feb 2011.

[9] Peter Szor. Duqu– threat research and analysis. Technical report, McAfee, October 2011.

[10] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, January 1976.