DIGITAL GENOME MAPPING – ADVANCED BINARY MALWARE ANALYSIS

Ero Carrera and Gergely Erdélyi
F-Secure Corporation Anti-Virus Research Team,
Tammasaarenkatu 7, 00181, Helsinki, Finland

Tel +358 9 2520 0700 • Fax +358 9 2520 5001•
Email ero.carrera@f-secure.com,
gergely.erdelyi@f-secure.com

ABSTRACT

Windows binary malware has come a long way. Today's average worm is often tens or hundreds of kilobytes of code exhibiting a level of complexity that surpasses even some operating systems. This degree of complexity, coupled with the overwhelming flow of new malware, calls for improvements to tools and techniques used in analysis.

Our paper elaborates on how to use graph theory to aid the analysis. Using graphs and extensions with the popular Interactive Disassembler Pro package, we hope to reduce the time needed to understand the structure of complex malware. These methods have proven to be helpful in finding similarities and differences between different malware variants and strains.

Focusing on the differences by keeping off already known code allows rapid analysis and classification of malware, while reducing redundant efforts.

1. INTRODUCTION

Nowadays, reverse engineers – especially those within the anti-virus and forensics industries – face the challenge of identifying and analysing a large number of binaries appearing at an incredible rate.

Some of the difficulties analysts face are:

- The sheer number of samples to check.
- The amount of code to analyse on each of them.
- Recognizing variants and versions of families of malware and other software.

Most of those problems can be solved with proper automation and algorithms. Ideas and results from prototypes will be presented in this paper. We are introducing a new platform based on Interactive Disassembler Pro and the Python programming language. The aim is to aid development of research prototypes and advanced analysis.

The resulting tools will be targeted to recognize similar features in the structure of executables. The theoretical base we rely on to achieve our objective will be graph theory, by looking at representations of code in the form of graphs. With them, the necessary level of abstraction is achieved and general features in the structure of a binary become distinguishable.

2. PROGRAMMING IDA PRO

People who perform analysis of binary code tend to collect and use a large arsenal of different tools. Apart from a good hex editor, probably the most commonly used tool is Interactive Disassembler Pro (IDA) [1]. IDA has earned its reputation by being extremely versatile, flexible and extensible. The already remarkable set of features IDA has can be further expanded with the built-in script language and compiled plugins. In the next section we will take a brief look at both extension methods.

2.1 IDC

The easy way to get started with IDA programming is IDC, the scripting language of IDA. The syntax of IDC is very similar to ANSI C, which makes it familiar for experienced C programmers. Unlike C, IDC is an interpreted language which saves us from the tedious write-compile-fix development cycle of C.

IDC comes with an extensive set of built-in functions that cover many aspects of the disassembly process as well as the structure and data of the program being disassembled. The semantics of all those functions are, not too surprisingly, close to regular C functions.

The following simple program is an example of IDC code. It is a trivial decryptor that uses XOR 0xFF to decrypt a zero-terminated ASCII string, starting from the current cursor position.

```
#include <idc.idc>

static main ()
{
  auto baseaddr;
  auto key;
  auto c;

  baseaddr = ScreenEA();
  key = 0xFF;

  while (1)
  {
    c = Byte(baseaddr);

    if (c == 0) { break; }

    PatchByte(baseaddr, c ^ key);
    baseaddr++;
  }
}
```

2.2 Plugins

In case the required code exceeds the capabilities of IDC we can resort to writing an IDA plugin. The SDK comes with IDA and defines an extensive API for plugins. Unlike IDC the API gives access even to the most intimate details of the IDA database and all other aspects of the disassembly. IDA plugins can be incredibly powerful with this level of control.

Plugins are developed in C++ and can be compiled with Borland, Microsoft, Watcom or GNU C++ compilers.

Since the code is compiled to native binary, the execution speed is similar to any other native application. The main disadvantage also comes from the need of compilation which makes the development tedious. Another problem is that a small bug can crash the whole IDA application, possibly corrupting the database. The following example shows the plugin implementation of the XOR decryptor with all the plugin administration code stripped.

```
void decode (void)
{
  ea_t baseaddr;
  uchar key;
  uchar c;
  baseaddr = get_screen_ea();
  key = 0xFF;
  while (1)
  {
   c = get_byte(baseaddr);
   if (c == 0) { break; }
   patch_byte(baseaddr, c ^ key);
   baseaddr++;
  }
}
```

The look of the code did not change much. The IDA API functions have somewhat longer names and more specific purpose than the IDC counterparts. To implement certain IDC functions, often several IDA API calls and data conversions have to be made.

3. NEED FOR CHANGE?

What is wrong with IDC and plugin writing? Generally speaking nothing. Either of them would be suitable to implement anything that can be implemented with the help of IDA. On the other hand there are other points to consider here. The undisputed dominance and popularity of C and C++ does not automatically mean that they are the optimum choice for every task. The lack of complex, abstract data structures make C unsuitable for rapid development and prototyping. Without trying to start a language debate we can also note that the extensions of C++ over C do not have the easiest to remember syntax either. C and C++ put efficiency over expressiveness, which certainly made sense when computing power was more expensive than programmers' time. However with computing horsepower getting cheaper and cheaper it does not sound so crucial any more to spare clock cycles at the expense of human productivity.

What would be a better tool then?

Before jumping to conclusions it is necessary to examine the most common practical applications of IDA extensions. As far as we can tell the two most common are:

- experimental reverse engineering tools
- quick tools for malware analysis.

Experimental tools evolve quickly, which clearly benefit from a development environment that supports frequent code and interface changes. The programming language and the tools must not limit the researcher's ability to significantly change the code whenever it is needed.

Code written for malware analysis purposes usually solves a single problem and, typically, it is not reused. Decryptors written for different malware's string scramblers are a good example of this group. According to our experience these tools cannot be fully reused too often as malware changes very rapidly, even in the same family.

In both cases the time required for development is more important than execution speed. The above observations lead us to the conclusion that we need to find a language which is very expressive, allows rapid prototyping and is embeddable into applications. Modern, very high-level scripting languages can easily fulfil these requirements. After the evaluation of several modern scripting languages our choice was Python.

3.1 What is Python?

Quoting from the Python website:

"*Python is an interpreted, interactive, object-oriented programming language. It is often compared to Tcl, Perl, Scheme or Java.*

"Python combines remarkable power with very clear syntax. It has modules, classes, exceptions, very high level dynamic data types, and dynamic typing. There are interfaces to many system calls and libraries, as well as to various windowing systems (X11, Motif, Tk, Mac, MFC). New built-in modules are easily written in C or C++. Python is also usable as an extension language for applications that need a programmable interface." [2]

Even though Python's history started over a decade ago it has gained mainstream recognition only quite recently. The language is continuously evolving yet Python developers have been able to maintain excellent backward compatibility with older source code.

One slogan of the Python movement is: "Batteries included". For most tasks all the required modules are included in the standard distribution. These modules range in functionality from simple data handling to server application components. A large number of third-party modules are also available, from simple utilities to scientific computing and number crunching.

3.2 Why Python?

Python has many properties that make it well suited for our needs. It is easy to learn – anyone with prior programming experience can pick it up really quickly. For newcomers an excellent tutorial is available on the Python website [3].

The language is object-oriented and supports different abstract code and data structures which make it easy to express complex algorithms. Python has built-in support for lists, dictionaries (hashes) and tuples. Using these, algorithms that would take several pages in C can be expressed in just a few lines. This provides a perfect playground for trying out ideas. If one of the ideas does not work out the code can be thrown away without the slightest feeling of regret. The same thing is much more

painful after the effort of chasing down all the pointers and off-by-one errors.

Of course, just like anything else, Python has some disadvantages too. Most notable is execution speed. The programs are interpreted and sometimes run 2–20 times slower than the C/C++ implementations. Where the execution speed is a primary factor Python might not be the optimal choice. On the other hand Python is extremely easy to extend with C/C++ code. In most cases careful separation of computationally intensive parts into C/C++ extension modules solves the performance problem. Another way of speeding up Python code is the use of Psyco [4], which is a Just-In-Time (JIT) compiler that can achieve significant speedup depending on the type of the code.

4. MEET IDAPYTHON

Our attempt to make the raw power of IDA more accessible with a user-friendly, very high-level language resulted in the birth of IDAPython. Implementation-wise IDAPython is an IDA plugin which wraps the IDA API and connects it to the Python environment as an extension module. Scripts run inside IDA have access to both the IDA API and all installed Python modules.

IDAPython consists of several modules, organized in three layers:

1. Low-level IDA API wrapper layer
2. IDA API user layer with class definitions
3. IDC-compatibility and utility modules

The following figure illustrates the structure of IDAPython:

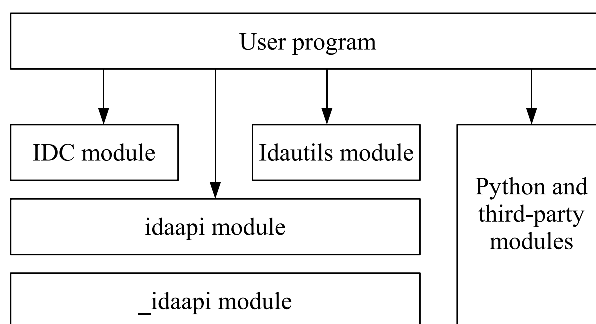

Figure 1: IDAPython organization.

4.1 Low-level wrapper layer

The low-level API layer is a direct Python wrapper around the IDA API. This module is statically linked into the IDAPython plugin and can be imported with the name '_idaapi'. The wrapper code is generated by Simplified Wrapper Interface Generator (SWIG) [5]. _idaapi directly reflects the C++ IDA API, functions have the same name, same arguments and return values. The IDA SDK documentation, with few exceptions, is directly applicable to the Python wrappers as well – although changes had to be made where the C++ semantics are significantly different from the Python. These differences from the official IDA API are documented in the plugin package. The differences mostly concern functions that return data in their arguments or use complex or unusual arguments. For example:

```
IDA C++ API version:
char *get_func_name(
              ea_t ea,
              char *buf,
              size_t bufsize);
```

The C++ version expects the function address and a pointer to the output buffer with bufsize size. It either returns the address of the output buffer or NULL if an error occurred. The idea of output buffer pointers is not used in Python so we have to change the definition that makes sense there as well.

```
Python _idaapi module version:
_idaapi.get_name(from, ea)
```

In the Python version the wrapper allocates memory for the output string and returns either a Python string or None if the function fails. Similar changes have to be made to other functions too.

This module should not be used directly, all the functions and data are available in the upper level module described in the next section.

4.2 User layer

The user layer module is called 'idaapi' which is a thin layer on the top of the '_idaapi' module. It contains all the functions and data from the module below and adds Python class definitions for the wrapped C++ classes, structures and unions. Using these classes complex data structures can be accessed with the usual class.member notation.

This is the module that most user programs that need IDA API access should use.

4.3 IDC compatibility module

Since most experienced IDA users are familiar with the IDC language a compatibility module has been developed. All the IDC definitions are automatically imported from the 'idc' module to provide a familiar environment for people with IDC experience. It also makes porting of IDC script to Python easier. The IDC module is developed with maximum compatibility in mind, however there are certain functions that have native implementation in the Python libraries and thus were removed. In the current state over half of the IDC functions have been implemented, the final goal is to implement all functions that are applicable in the Python environment.

4.4 Idautils module

To lift the burden of using low-level functions for everything a new module was created. Idautils is a module that contains assorted functions that wrap the most often used low-level code. Examples include functions that

provide a list representation of result that could previously be collected with *First() and *Next() type of calls and complex loop conditions. The development of 'idautils' is in progress and it will be extended with any functionality that qualifies to be included. All the functions are thoroughly documented with Python docstrings, so it is easy to experiment with them. A simple example follows in the next section to highlight the benefits of this module.

5. EXAMPLE

To demonstrate the advantages of the new approach we are going to implement a relatively short program. The purpose of the program is to enumerate all defined functions in a given segment and list the places they are called from. Typical output of the tool is something like this:

```
Function _start at 0x8048520
Function call_gmon_start at 0x8048544
  called from _start(0x8048543)
  called from .init_proc(0x804848e)
Function __do_global_dtors_aux at 0x8048570
  called from .term_proc(0x8048741)
Function frame_dummy at 0x80485b0
  called from .init_proc(0x8048493)
Function main at 0x80485e4
Function __libc_csu_init at 0x8048658
Function __libc_csu_fini at 0x80486a0
Function __i686.get_pc_thunk.bx at 0x80486f2
  called from __libc_csu_init(0x8048660)
  called from __libc_csu_fini(0x80486ad)
Function __do_global_ctors_aux at 0x8048700
  called from .init_proc(0x8048498)
```

This example is deliberately kept simple to focus on the implementation and not the algorithmic behaviour.

5.1 IDC version

First we implemented the small program in IDC:

```
#include <idc.idc>
static main()
{
   auto ea, func, ref;
   // Get current ea
   ea = ScreenEA();
   // Loop from start to end in the current segment
   for (func=SegStart(ea);
      func ! = BADADDR && func < SegEnd(ea);
      func=NextFunction(func))
   {
      // If the current address is function process it
      if (GetFunctionFlags(func) ! = -1)
      {
        Message('Function %s at 0x%x\n',
                GetFunctionName(func), func);
        // Find all code references to func
        for (ref=RfirstB(func);
             ref ! = BADADDR;
             ref=RnextB(func, ref))
             {
                Message(' called from %s(0x%x)\n',
                        GetFunctionName(ref), ref);
             }
        }
   }
}
```

The source code does exactly what we want it to do but it's not too easy on the eyes. It might need more than one glance to understand how it works, even after such short time as going for a lunch break. One of the problems is the C way of iterating through lists, which involves several functions and loops with complex conditions.

5.2 Python version

The second version was implemented in Python using only the idaapi module:

```
from idaapi import *
# Get current
ea ea = get_screen_ea()
# Get segment class
seg = getseg(ea)
# Loop from segment start to end
func = get_func(seg.startEA)
while func ! = None and func.startEA < seg.endEA:
    funcea = func.startEA
    print 'Function %s at 0x%x' % \
        (GetFunctionName(funcea), funcea)
    ref = get_first_cref_to(funcea)
    while ref ! = BADADDR:
        print '  called from %s(0x%x)' % \
           (get_func_name(ref), ref)
        ref = get_next_cref_to(funcea, ref)
    func = get_next_func(funcea)
```

The syntax has been converted to Python which uses indentation instead of braces for marking code blocks. This program serves as a good example that a good choice of the programming language itself does not warrant easy to read source code. This version of the routine is still not as clear and easy to understand as we would like it to be.

5.3 Python version with 'idautils'

The last version of the script has been rewritten from scratch to utilize both the IDC and idautils module.

```
from idautils import *
# Get current ea
ea = ScreenEA()
# Loop from start to end in the current segment
for funcea in Functions(SegStart(ea), SegEnd(ea)):
    print 'Function %s at 0x%x' % \
        (GetFunctionName(funcea), funcea)
    # Find all code references to funcea
    for ref in CodeRefsTo(funcea, 1):
        print ' called from %s(0x%x)' % \
           (GetFunctionName(ref), ref)
```

This version of the tool consists of six lines of effective code which took two to three times more with the other implementations. It is also much easier to understand than the others. Although we are not suggesting that all source will shrink to half its size, we are convinced that with the help of well-written, high-level wrappers over the most common functions code size and development time can be reduced. The main advantage of IDAPython is that any of the layers can be used even in the same code with certain precaution. One does not have to sacrifice the direct database access for IDC or the idautils package.

6. IDAPYTHON THE PLUGIN

6.1 Usage

Usage of the plugin is similar to IDC. Python code can be executed by either loading it from a file or entering it into a Python expression window. The latter is only suitable for experiments with short Python statements. Possibility to save the session could be added in the future as time permits. Future enhancements to the user interface might include features like a fully interactive Python session inside IDA. Unfortunately it is not possible to implement that with the current IDA API.

The following screenshot shows the Python expression window:

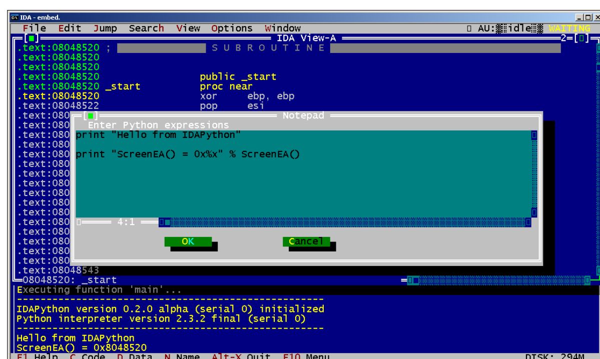

Figure 2: Example session.

6.2 Status and availability

IDAPython is under heavy development, yet in an advanced state already. The plugin is used in-house by the *F-Secure* Antivirus Research Team on a regular basis. More improvements and testing are in the works. By the time this paper is published the first feature complete version of the IDAPython plugin will be released at http://www.d-dome.net/idapython/. The package will include the full source code and can be used for any purpose, free of charge. The reader is invited to try it out and provide much valued feedback. Improvement ideas, code and other contributions are most welcome.

7. INTRODUCING GRAPHS

7.1 Previous work

Far from being a seminal work, this paper attempts to raise the importance of clear and visual methods of analysis. Halvar Flake is a distinguished figure in the binary reverse engineering world, he has been using graphs and similar comparison metrics as used here in specific steps of our algorithm, aiming at tools which find differences between different versions of a given binary. Flake's work [6] elaborates on the use of graphs for malware analysis and exposes some of the advantages that we will reintroduce in this paper.

7.2 The proposed approach

Graphs have always provided a clear way of looking at problems. Complex hierarchies instantly become easier to understand once they are represented in a way humans find easy to assimilate. Graphs are one such way of displaying data, instead of more cumbersome representations like endless lists of numbers and references to code and data as provided in a traditional disassembly.

Additionally, a rich literature already exists on graph theory with algorithms to treat and analyse them. Once we are consistently representing code as graphs, a door is open to know mathematical methods for analysis and further topics such as clustering.

In this paper we will be representing code as graphs on global and function levels.

Graphs provide abstraction, and that is exactly what enhances the ability to build simple algorithms to work on top of them.

7.3 Preparing the data for analysis

7.3.1 Exporting the data

A tool was created to export IDA databases to a metalanguage representation, after a given binary has been loaded into IDA. This tool does not export the whole set of information supported in IDA's database format, but a sufficient subset including:

- Every defined function.
- Every instruction. The full binary representation, mnemonic and operands as presented by IDA to the user.
- All the defined referenced strings and names.
- All code references.
- Code flags: whether the code is flow (useful to coherently extract the basic blocks of a function).
- Miscellaneous flags: whether a function belongs to a known library or is a thunk.

All this information has been found to be enough to provide a fertile playground for our research. Future additions will include all comments in order to add as much high-level readable information as possible.

The name *REML* was coined for the resulting metalanguage file, standing for *Reverse Engineering Meta-Language*, or *RevEngML*.

7.3.2 Interfacing the data

The next step was generating an object-oriented framework, focusing on usability and ease of use, while giving lower priority to performance issues.

A *Python* module was created to work on the *REML* files. The module implements a top BINARY class, which

provides methods to access the generic binary information such as the entry point to the code and the list of its functions. It is also possible to query functions by their names or addresses in their body.

Subsequently, an extensive *Function* class was implemented. Among others, it provides the following:

- Access to all the instructions.
- Ingoing and outgoing code and data references.
- Generation of the list of basic blocks from which to generate a *CFG* (*Control Flow Graph*).

A third, smaller class, was added under the rather unoriginal name *Instruction* in order to access the instructions and their operands.

7.3.3 Processing the data

Our research and prototypes lie on top of the interface we created in order to access the data. It gave us, among other possibilities, that of quickly generating graphs from a binary. In a fairly small number of lines (as compared with *IDA*'s *IDC* language) we could iterate over the whole set of functions, gather their references and create a graph object [7].

We took the approach that every single function in a binary is represented by a vertex in the graph and *calls* between them translate to the graph's edges.

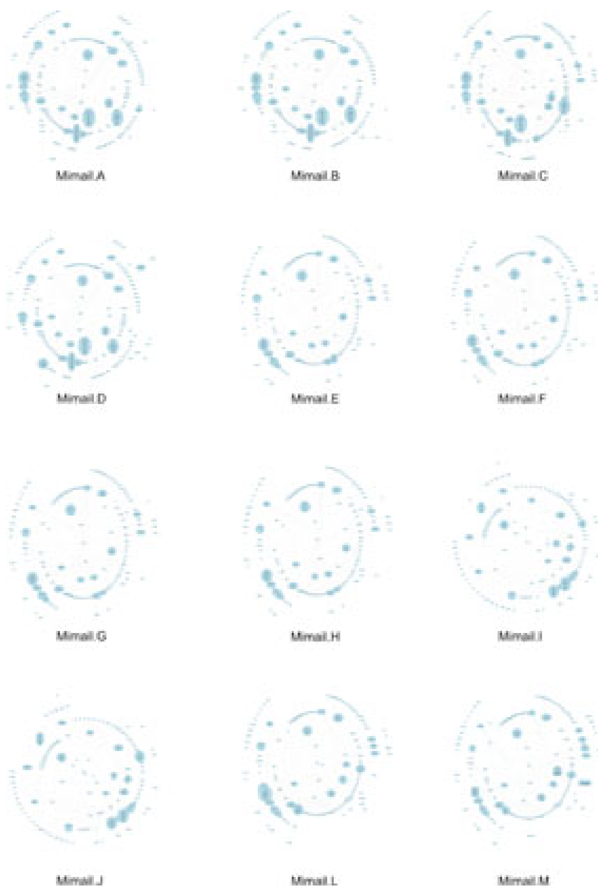

Figure 3: Mimail family graphs. Certain patterns can be appreciated between members of this family.

The plotting algorithms and style were fine tuned, over some initial testing period, until we achieved understandable representation of rather complex and extensive binaries.

Two main modes of visualization were found to provide good results. A hierarchical representation provides good results for binaries up to certain number of functions; this layout is provided by the *GraphViz*'s *dot*.

A second, radial layout (Figure 3), was obtained through *GraphViz*'s *twopi*, which proved to be useful to visualize larger cases.

After the initial visual advantages of graphs, exploiting their mathematical properties became the target of our efforts. This is the topic of the next section.

8. ALGORITHMS BASED ON GRAPH REPRESENTATION OF BINARIES

8.1 Binary comparison, graph similarities

In this section we will address the process of finding how close two binaries are to each other. The result of our algorithms will be a value indicating how much of their call flow graphs overlap or relate to each other.

Once we have such values, we are in the position of attempting to perform classification of malware by their similarities. Borrowing from the field of *Phylogenetics* [8], we will apply *clustering analysis* algorithms in order to obtain a taxonomy of the malware at hand. The results from this approach are encouraging, indicating an area of research which should be explored further.

8.1.1 The comparison algorithm

Given the graph representing the binary under analysis, we will create its adjacency matrix. This matrix will have a row and column for each of the functions present in the binary, and its element in position (I, J) will indicate whether a function in row I performs a call to the function in column J.

The objective we pursue is to be able to relate functions from two executables based on their position in the graph resulting from the connections among all the functions. A function which calls and is called by a given set of functions, is matched against another with a similar set of called and caller functions.

Finding graph isomorphisms is known to be a fairly complex task, no polynomial time algorithm is known to solve this problem. Fortunately, the problem at hand is much simpler as we have more information to put into our model.

To define a basic set of nodes from which to start, we choose operating system and library calls. In other words, any function which does not perform a further call to code belonging to the binary and which will have a common name across different executables. With this approach we automatically rule out binaries which do not call any such

functions, although it would be still applicable if a researcher would manually find a basic set of common functions between two binaries and name them accordingly.

Once we have a set of commonly named functions on two different binaries, we compose an adjacency matrix and suppose as identical the functions with identical unique rows in this matrix.

We incorporate the information gained in each pass and rerun the matching, effectively progressing through the graph, until a maximum of functions is matched.

Additional optimizations have been put in place when ambiguities arise, as function matching according to metrics calculated from their internal structure.

After the process in finished, we have a set of functions common to both binaries.

8.1.2 Definition of the algorithm

- We have two executables' graphs which we want to compare. A source *S* and a target one *T*. We define the following:
  - S_f = {set of all functions composing S}
  - T_f = *{set of all functions composing T}*
- Find the common set of *atomic functions* [9]. That is: $C = \{f : f \in S_f \cap T_f\}$
- We define $S_r = S_f - C$ and $T_r = T_f - C$ as the sets of remaining functions to be matched.
- Now, for $f \in S_r$ we create a *call-tree signature* [10] for *f* from *C* and consider it identical to $f' \in T_r$ if the signature of f' generated from *C* is identical and unique.
  - If such match is found we augment the *C* matrix with the signature of *f*. which will allow us to match higher level functions not necessarily relying on *API* functions.
- Once this algorithm does not yield any more matches, we will have a matrix *C* containing all the functions occupying the same positions in the graphs of both *S* and *T*. At the end of the process we will have unmatched functions of two types.
  1. The ones which are different and can not be matched.
  2. Functions which do not have unique signatures.

In a further development of the algorithm we enhanced its matching capabilities by using a new kind of signature, namely the list of edges connecting basic blocks from the functions CFG (*CFG signatures*). Functions with non-unique *call-tree signatures* can be matched if unique *CFG signatures* exist.

We implemented this enhancement to be run every time we exhaust the *call-tree signatures*. Finding some correspondences between functions will often lead to additional matches in further passes of the algorithm. This has successfully increased the accuracy of our method.

8.1.3 The index of similarity

Once we have a set of supposedly equivalent functions between both binaries, the index of similarity is obtained as follows.

Let's call the set of equivalent functions in the source binary *S*, S_e = *{equivalent functions in S}* and similarly define T_e = *{equivalent functions in T}*, note that both sets contain the *API* calls found to exist in both binaries. Let $\sigma'(S_e, T_e)$ denote the similarity index function and be defined as:

$$\sigma\prime(A,B) = \frac{|\, A \,|}{|\, A \bigcup B \,|}$$

which leads to the non-symmetric similarity indexes (or distances):

$$\sigma\prime(S_e,T_e) = \frac{|\, S_e \,|}{|\, S_e \bigcup T_e \,|} \quad and \quad \sigma\prime(T_e,S_e) = \frac{|\, T_e \,|}{|\, T_e \bigcup S_e \,|}$$

we define a final combined similarity as:

$$\sigma(A,B) = \frac{|\, A \,| \cdot |\, B \,|}{|\, A \bigcup B \,|^2}$$

This function, $\sigma(A,B)$ will always yield a value between $0 \le \sigma(A,B) \le 1$. With values close to 0 indicating a low degree of resemblance, and values near 1 for pairs of malware which share almost all of their code.

8.1.4 Case studies

We will now give the distance matrices for different families of malware, the values are percentages indicating how close the malware was found to be to each other. Here we will follow the procedure introduced in the previous section.

Mimail

The Mimail family exhibits high indexes of resemblance, as can be seen in the following distance matrices.

	mimail.a	mimail.b	mimail.c	mimail.d	mimail.e	mimail.f
mimail.a	0	90.8	85.4	87.4	75.0	75.0
mimail.b	90.8	0	84.7	88.0	74.3	74.3
mimail.c	85.4	84.7	0	81.5	81.3	81.3
mimail.d	87.4	88.0	81.5	0	72.3	72.3
mimail.e	75.0	74.3	81.3	72.3	0	95.4
mimail.f	75.0	74.3	81.3	72.3	95.4	0

	mimail.h	mimail.i	mimail.j	mimail.l	mimail.m	mimail.q
mimail.h	0	81.7	83.2	90.9	88.8	41.6
mimail.i	81.7	0	95.0	81.7	79.9	46.9
mimail.j	83.2	95.0	0	83.2	81.3	47.8
mimail.l	90.9	81.7	83.2	0	90.3	42.4
mimail.m	88.8	79.9	81.3	90.3	0	40.6
mimail.q	41.6	46.9	47.8	42.4	40.6	0

Klez

The *Klez* family comparison also yields good results. The classification splits the samples in two branches, one containing *Klez* variants *A*, *B*, *C* and *D*, and a second one with *E*, *F*, *G*, *H*, *I* and *J*, see Figure 4.

	klez.a	klez.b	klez.c	klez.d	klez.e
klez.a	0	79.6	70.3	70.3	49.3
klez.b	79.6	0	73.4	73.4	49.3
klez.c	70.3	73.4	0	88.2	49.6
klez.d	70.3	73.4	88.2	0	49.6
klez.e	49.3	49.3	49.6	49.6	0

	klez.f	klez.g	klez.h	klez.i	klez.j
klez.f	0	87.0	80.7	80.7	87.0
klez.g	87.0	0	80.7	80.7	87.0
klez.h	80.7	80.7	0	89.6	80.7
klez.i	80.7	80.7	89.6	0	80.7
klez.j	87.0	87.0	80.7	80.7	0

Netsky

The large family of Netsky appears clustered together as well. A feature worth mentioning is the closeness to the Sasser family, which was created by the same author. Such resemblance yields high indexes of similarity with our algorithm, given the fact that the author used common code in both families. See Figure 4.

	netsky.aa	netsky.ab	netsky.ac	netsky.l	netsky.n
netsky.aa	0	75.8	39.9	56.1	49.0
netsky.ab	75.8	0	40.2	64.3	50.6
netsky.ac	39.9	40.2	0	31.2	26.6
netsky.l	56.1	64.3	31.2	0	43.1
netsky.n	49.0	50.6	26.6	43.1	0

	netsky.s	netsky.u	netsky.v	netsky.w	netsky.x
netsky.s	0	82.5	69.5	52.6	70.6
netsky.u	82.5	0	74.7	50.8	69.6
netsky.v	69.5	74.7	0	46.5	60.6
netsky.w	52.6	50.8	46.5	0	48.5
netsky.x	70.6	69.6	60.6	48.5	0

Roron

The Roron family is tightly clustered together given the values in distance matrix generated from the graphs. See Figure 4.

	roron.ac	roron.ad	roron.ae	roron.k	roron.p	roron.q	roron.r
roron.ac	0	85.0	83.7	63.2	85.1	85.1	84.9
roron.ad	85.0	0	84.6	62.9	83.5	83.9	84.1
roron.ae	83.7	84.6	0	62.2	82.3	82.7	82.9
roron.k	63.2	62.9	62.2	0	64.1	63.3	63.2
roron.p	85.1	83.5	82.3	64.1	0	86.2	86.0
roron.q	85.1	83.9	82.7	63.3	86.2	0	87.7
roron.r	84.9	84.1	82.9	63.2	86.0	87.7	0

	roron.s	roron.t	roron.u	roron.v	roron.w	roron.x
roron.s	0	84.8	85.8	85.0	84.8	86.2
roron.t	84.8	0	85.2	84.8	89.3	85.2
roron.u	85.8	85.2	0	85.9	85.6	84.6
roron.v	85.0	84.8	85.9	0	85.2	84.6
roron.w	84.8	89.3	85.6	85.2	0	85.2
roron.x	86.2	85.2	84.6	84.6	85.2	0

Bagle

The Bagle worms provide an example where the similarity between certain variants is low. The reason for such results is that some of the variants were just dropping a backdoor and did lack their own replication system, actually rendering them quite different form each other. The algorithm successfully indicates such fact.

	bagle.ac	bagle.i	bagle.j	bagle.k	bagle.m	bagle.v	bagle.w	bagle.x
bagle.ac	0	8.0	1.6	1.6	7.5	1.2	8.3	8.3
bagle.i	8.0	0	4.3	4.3	23.4	3.2	23.5	23.5
bagle.j	1.6	4.3	0	81.3	4.8	37.7	4.3	4.3
bagle.k	1.6	4.3	81.3	0	4.8	37.7	4.3	4.3
bagle.m	7.5	23.4	4.8	4.8	0	3.6	22.8	22.8
bagle.v	1.2	3.2	37.7	37.7	3.6	0	3.2	3.2
bagle.w	8.3	23.5	4.3	4.3	22.8	3.2	0	23.8
bagle.x	8.3	23.5	4.3	4.3	22.8	3.2	23.8	0

Sasser

Sasser is another sample of highly similar malware.

	sasser.a	sasser.b	sasser.c	sasser.d	sasser.e
sasser.a	0	90.6	88.0	74.7	77.1
sasser.b	90.6	0	94.6	79.3	80.5
sasser.c	88.0	94.6	0	79.3	78.1
sasser.d	74.7	79.3	79.3	0	79.2
sasser.e	77.1	80.5	78.1	79.2	0

Random selections

We will now show some distance matrices of randomly picked samples. We have previously shown families of malware, which do not demonstrate well the case where samples are different from each other. The following data shows that unrelated samples yield low similarity indexes.

	roron.q	netsky.s	sober.f	netsky.ac	netsky.l	klez.i
roron.q	0	14.8	0.0	19.5	8.0	20.8
netsky.s	14.8	0	0.0	62.5	19.3	14.9
sober.f	0.0	0.0	0	0.0	0.0	0.0
netsky.ac	19.5	62.5	0.0	0	31.2	21.0
netsky.l	8.0	19.3	0.0	31.2	0	9.9
klez.i	20.8	14.9	0.0	21.0	9.9	0

	netsky.aa	mimail.h	sasser.e	sober.d	mydoom.g	sober.b
netsky.aa	0	0.0	22.7	0.0	0.1	0.0
mimail.h	0.0	0	3.5	0.0	4.9	0.0
sasser.e	22.7	3.5	0	0.0	4.9	0.0
sober.d	0.0	0.0	0.0	0	0.0	11.9
mydoom.g	0.1	4.9	4.9	0.0	0	0.0
sober.b	0.0	0.0	0.0	11.9	0.0	0

	klez.i	bugbear.e	klez.b	roron.w	bagle.i	mydoom.k
klez.i	0	2.8	48.2	20.5	1.0	14.5
bugbear.e	2.8	0	2.9	2.0	0.5	1.8
klez.b	48.2	2.9	0	22.0	0.8	17.6
roron.w	20.5	2.0	22.0	0	0.6	13.8
bagle.i	1.0	0.5	0.8	0.6	0	0.6
mydoom.k	14.5	1.8	17.6	13.8	0.6	0

	roron.ad	klez.i	bugbear.g	mimail.b	netsky.u	mimail.f
roron.ad	0	20.4	2.2	1.9	13.5	2.1
klez.i	20.4	0	2.9	2.4	13.9	2.4
bugbear.g	2.2	2.9	0	4.9	3.1	5.5
mimail.b	1.9	2.4	4.9	0	3.8	74.3
netsky.u	13.5	13.9	3.1	3.8	0	4.9
mimail.f	2.1	2.4	5.5	74.3	4.9	0

8.2 Possibilities, exploiting the acquired knowledge

We will now discuss several techniques which benefit from the foundations laid in section 8.1 of the paper. We will briefly present several areas whose overall quality improves by having some metric defining the degree of similarity between malware. This also relates to which parts of two specific binaries correlate, proving similar or the same functionality. Some of the techniques discussed in this section have already been implemented and preliminary results are available; others, although briefly tested, lack a solid prototype.

8.2.1 Clustering and classification

Once we created a reliable method to find approximate distances, or similarity indexes between binaries, we can move into such area as classification. Philippe Biondi demonstrated to one of the authors the results of applying clustering algorithms, such as X-tree, to a previously generated *distance matrix* of malware. Those encouraging results fostered further research of the topic.

Recently Stephanie Wehner has done some work on classification using techniques based on *Kolmogorov Complexity*. Her preliminary results [11] show classifications of worms into families without any underlying analysis performed on the samples. This approach is highly interesting when a fast classification is needed.

Our approach is based on *phylogenetics*. An extensive literature exists on that area on classification of *taxa* and *cluster analysis*. A series of algorithms exist which can be applied to distance matrices, that is, a square matrix providing numbers expressing how similar two *OTU* [12] or malware are to each other.

Algorithms such as *Neighbor Joining* [13] applied to such matrices yield a hierarchical classification, which opens the door to taxonomies of malware. New samples can immediately be classified if a sample fulfilling certain requirements is provided [14].

In the future, given a common algorithm agreed beforehand, this procedure may aid in the classification task and diminish the confusion caused by randomly named malware. The existing approach in biological sciences might prove of value, where popular taxa have colloquial names, besides the purely scientific classification.

Previous work [15] has already explored ways the phylogenetic approach may help in computer virus classification, although the metric used in that case was byte sequences present in the malware file.

We believe that approach is limited, as even simple polymorphic malware may result in no possibility of application of the technique, let alone packed/encrypted code.

While our approach currently faces similar restrictions if aiming for completely automated analysis, we already laid a base which yields acceptable results with a strong metric such as the one provided by graph isomorphisms, see Figure 4.

8.2.2 Helping quick analysis

The knowledge of which functions in two different variants of malware are the same allows us to mine this information, aiding in rapid analysis of threats of the same family. Starting from an already analysed binary, the possibility exists (although it is not implemented yet) to mark certain areas on it, i.e. string references, registry keys, port numbers; and track whether or not they have been modified from one variant to the next. If such modification took place, we could obtain the new values and automatically generate the corresponding information in form of a description for the general public.

8.2.3 Migrating analysis information between code versions

Porting existing knowledge from the analysis of a variant to a recently discovered one has proved to be a practical and valuable tool. The knowledge of which functions map from a binary *A* to a binary *B* allows us to rename all the known code in the, yet to be analysed, new variant. In some cases it allows us to start an analysis with 80–90 per cent of the defined functions already meaningfully named.

This reduces the time needed for an initial overview and enables us to focus on more important parts of the code and algorithms and not just reanalysing previously seen code.

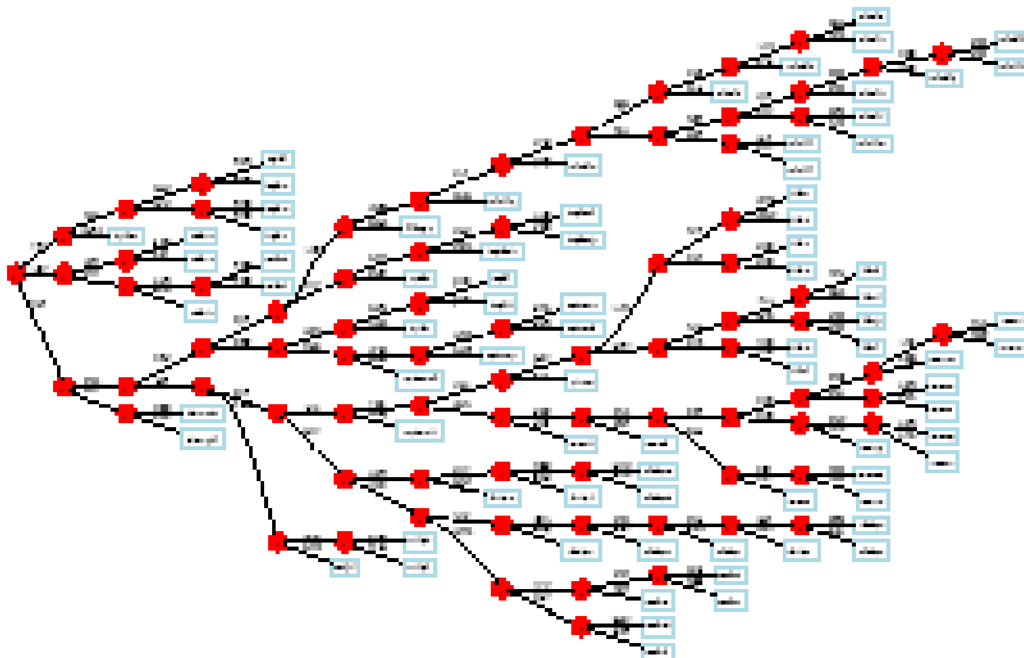

Figure 4: Phylogenetic classification of a group malware.

8.3 Other aspects of code visualisation using graphs

Recently we created a tool which allows us to browse a disassembly by just looking, scrolling and zooming a graph.

The main interface shows the whole defined call tree of the binary, every function node containing a list of the strings referenced from within it. Clicking in any of the functions opens a new window, which shows its *CFG* (*Control Flow Graph*) and in each basic block node, a list of function and string references. Placing the mouse over any of the basic blocks shows the corresponding disassembled code.

More features have been added, i.e.:

- Rename functions
- Search functions in the graph
- Change visualizations modes (hierarchical or radial)
- Compare different versions of binaries (common code is represented in red)
- Export the renamed functions as an IDC script to rename them in the IDA DB.

The tool has allowed us, so far, to perform quick general analysis of a binary. The holistic perspective provided is really convenient when handling binaries with several layers of abstraction, where the *API* calls may lay some levels away from the investigated function. In a disassembler such as IDA, perceiving the logical position and purpose of the function within the binary might be nearly impossible without jumping to all the called code. In a graph, that becomes trivial, as all those relations are patent in one sight.

9 CONCLUSIONS AND FUTURE DIRECTIONS

9.1 Limitations

Several limitations exist to a fully automated graph generation, comparison and classification approach. Malware is commonly packed and/or encrypted. That represents one major inconvenience which must be overcome before achieving our final goal of full automation.

The graph creation and comparison algorithms are architecture-independent. The only requisite is the existence of such concept as calls (branching is acceptable as long as *IDA* can recognize the functions).

The binaries must use *API* services provided by the operating system, and have a significant quantity of code.

9.2 Future directions

Further research in the classification of malware promises to yield rewarding results. Some of the basic concepts discussed in this paper should be extended.

9.2.1 Graph database

The creation of a graph database has not yet been explored. In such a repository, the graph representation and properties of a binary would be stored. Upon arrival of newly discovered malware, the database would be queried for certain patterns. If the search turns out positive results, comparisons and posterior classification would be performed automatically.

REFERENCES & NOTES

[1] Datarescue IDA page, http://www.datarescue.com/idabase/index.htm.

[2] Python website, http://www.python.org/.

[3] Guido van Rossum and Fred L. Drake, Jr.: *Python Tutorial,* http://docs.python.org/tut/tut.html.

[4] Psyco website, http://psyco.sourceforge.net/.

[5] SWIG website, http://www.swig.org/.

[6] Halvar Flake, ‘Graph-Based Binary Analysis’, *Blackhat Briefings* 2002.

[7] We also created a python module, namely pydot, to quickly generate and interact with appealing visual layouts of graphs through *AT&T*’s renowned GraphViz software. The python module is publicly available at one of the author’s personal website, http://dkbza.org/pydot.html.

[8] *Phylogenetics:* “Phylogenetics is the taxonomical classification of organisms based on how closely they are related in terms of evolutionary differences.”

[9] By ‘atomic function’ we mean a function which performs no further calls and has a common name across different executables. In such way that it can be considered to be a basic building block. In graph theory they are the leaf vertices with indegree 1 and outdegree 0.

[10] A signature for a function f is created by inspecting the list of atomic functions, L_a, and creating a binary string, containing a 1 in position n if f calls the atomic function in the index n of L_a and 0 otherwise.

[11] S. Wehner, ‘Analyzing Worms using Compression’, http://homepages.cwi.nl/~wehner/worms/

[12] In phylogenetics, the term OTU stands for Operational Taxonomical Units and refers to members of different species to be classified given their evolutionary differences.

[13] N. Saitou and M. Nei, ‘The neighbor-joining method: a new method for reconstructing phylogenetic trees’, *Mol. Biol. Evol. 4*, pp406–425.

[14] The sample must be unpacked/unencrypted and have a properly reconstructed import table.

[15] L.A. Goldberg, P.W. Goldberg, C.A. Phillips, G. Sorkin, ‘Constructing Computer Virus

Phylogenies', *Journal of Algorithms*, Vol 26(1) (Jan. 98) pp.188–208.