GRAPHS, ENTROPY AND GRID COMPUTING: AUTOMATIC COMPARISON OF MALWARE

Ismael Briones and Aitor Gomez
Panda Security, Anti-Malware Research Team, Gran Vía 4, 48001, Bilbao, Spain

Tel +34 94 425 1100
Email {ismael.briones, aitor.gomez}@pandasecurity.com

ABSTRACT

Nowadays AV laboratories are saturated with huge collections of malware which are received daily. It's a fact that the industry needs better methods to automatically identify, analyse and classify these volumes of samples. AV laboratories cannot continue working as they did years ago (or even months ago).

In this paper we will describe an automated classification system to identify files with similar internal structures. We will use graph theory as a way to identify similar functions among malware samples. This system helps to minimize human error and false positive detection.

Previous research with graph theory has proven to be useful in finding similarities between malware variants [1], however these systems don't have good performance. To solve the performance problem we will discuss some methods that can be used for this purpose: an algorithm (based on entropy and a custom checksum in order to group similar files) and a grid computing system [2].

1. INTRODUCTION

Most of the samples received daily at AV laboratories are variants of previously seen samples. Malware writers develop huge numbers of variants as a way to bypass AV signatures and saturate the AV labs. These new samples share the same code with minor differences, so if we were able to analyse them focusing on their internal structure, identifying the shared code between them, we could group members of the same family together and therefore automatically identify and classify new samples received.

This paper describes a method for automatically analysing and comparing an unknown sample to the database of stored malware. This method provides a similarity metric that can show us how close two binaries are.

The method covers the whole process: first, how to analyse, extract and store the internal structure of the old samples; second, design several methods to discard and preselect the most accurate samples to compare against them; and finally the comparison algorithm, mainly based on graph theory. We also discuss how these methods could be applied in several malware analysis related tasks: automatic virus naming, sample clustering and migration analysis information among family members.

1.1 Related work

One piece of research that inspired this paper is the Digital Genome Mapping paper by Ero Carrera and Gergely Erdélyi [1]. Although that paper is a basic approach on how to use graph theory to help in the analysis and identification of samples with a similar internal structure, it became the seed that made me think about the viability of this research. The core of our system is the same as the Carrera & Erdélyi approach: programming *IDA Pro* with *IDA Python* [3] to analyse and store graph information in the database. The details of *IDA Pro* and *IDA Python* are outside the scope of this paper.

There are previous studies on graph-based binary analysis. Halvar Flake [4] has been using graphs and similar comparison metrics to find differences between different versions of a given binary [5–8]. His ideas and algorithms reveal some of the advantages in the automation of reverse engineering and code analysis. His work is fundamental to our research and it's an extension of the research presented in [8].

Marius Gheorghescu presented another interesting piece of research into automatic malware classification that claims to find related samples in a reasonable amount of time [9].

Anyway, since the beginning of the project, we have found several barriers that we needed to bypass (mainly relating to performance and speed of the comparison algorithm) before the project could be deployed in a production environment. We present the selected solutions for solving them.

2. GRAPH THEORY

'Graph theory is the study of graphs: mathematical structures used to model pairwise relations between objects from a certain collection. A "graph" in this context refers to a collection of vertices or "nodes" and a collection of edges that connect pairs of vertices.' (from *Wikipedia* definition [10])

Previous work to identify changes among similar pieces of code has been focused on byte- or instruction-level comparison [11]. Although this approach has worked in the past, nowadays it isn't useful because of aggressive compiler optimizations, instruction reordering, modifications in register allocation, branch inversion, obfuscation methods etc. A new approach, like the one explained in [8] and extended in [12], is needed.

Representing code as graphs on a global and functions level (Control Flow Graph [13]) provides an abstraction that allows us to identify pieces of code shared between binaries, or with minor changes, without having to match instructions as a sequence of lines and applying a sequence-comparison algorithm. The idea is to neglect as much as possible from the assembly code and to analyse only structural properties of the executable. In this way, minor changes to assembly instructions won't affect our graph structure.

2.1 Graphs

Executables could be represented as a graph (a.k.a. callgraph), as we previously said, which is a collection of nodes connected by a group of edges, where the nodes represent the disassembly of the functions and the edges represent the relations (function

calls) among these functions. It's a directed graph: an edge from f_1 to f_2 implies that f_1 contains a subfunction call to f_2 but not vice versa. Figure 1 is an example of a simple directed graph.

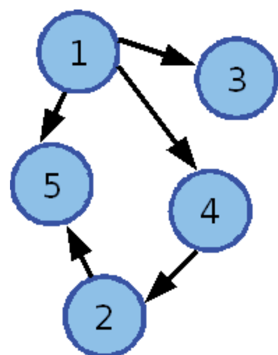

Figure 1: Directed graph.

As a graph represents the internal structure of an executable, we could suppose that similar samples of mutations derived from the same source code should generate similar graphs. Figure 2 is an example of graph isomorphism among several samples from the same malware family (Sasser).

Certain patterns can be appreciated among members of this family. If these patterns can be appreciated by seeing their graphs, could we use this as a way of determining how close they are. This is our next challenge.

PandaLabs has a huge number of samples, which is a treasure trove of information. We could use this information to automatically identify similar samples, improve our detection rate, do our technical work faster and more easily, perform better malware analysis and select malware nomenclature more accurately.

2.2 Adjacency matrix

Our main task will be to find a way to analyse the graphs. Graph isomorphism comparison is a time-consuming task. There isn't a quick algorithm that can be used to compare graphs (as images) to match similar structures, so a new approach is needed. We will use the mathematical representation of a graph: the adjacency matrix [14].

$$\begin{pmatrix} 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Figure 3: Adjacency matrix of a directed graph.

The adjacency matrix is the mathematical representation of a graph. It allows us to represent the interconnection among nodes. It's basically a square matrix where the columns and rows are the graphs' nodes and each element (I, J) of the matrix represents the edges (if there is a connection between node I and node J). Figure 3 represents the adjacency matrix of the previous graph (Figure 1).

3. BINARY ANALYSIS

In this section we describe the preprocessing steps that we take regarding our samples. All of them are based on analysis done through *IDA* and the *IDA Python* plug-in. This information is fundamental to the method explained here. *IDA* technologies, like FLIRT, are improved from version to version, so it's important to update *IDA* with the latest version, although it would imply the need to reanalyse the whole malware database. This task can be accomplished in a reasonable time with the distributed computing system discussed later.

3.1 General approach

The general approach of this method is to find as many fixed points as possible. Fixed points are elements from both binaries that can easily be determined to represent the same item in both executables. Once we have matched the maximum number of fixed points, we can match more elements (functions) based on call-tree functions to these initial fixed points.

The selection of good fixed points is essential for algorithm performance. The more fixed points, the fewer functions to match and therefore our analysis will be faster. Table 1 shows the selected fixed points used to construct the adjacency matrix.

Fixed points
Operating systems and library calls (APIs)
Functions' CRC32
Control flow graph (CFG)

Table 1: Fixed points used to construct the adjacency matrix.

3.2 Operating systems and library calls (APIs)

We need to define a basic set of nodes to start with. We use *IDA* as our disassembly engine. Fortunately, *IDA* is able to identify operating system and library calls. The FLIRT [15] technology is awesome at identifying library code. These will be our first initial nodes. These nodes don't perform calls to functions belonging to the binary and their names are common across different executables.

FLIRT signatures are improved with new *IDA* versions, so whenever possible it's important to maintain the system with the latest *IDA* package. Reanalysing the whole malware database is important if we want to improve the accuracy of our comparison system. However, an initial analysis of the new *IDA* features is recommended to determine whether it is an important update that justifies full reanalysis of the malware database.

3.3 Functions' CRC32

Nodes in the callgraph can be selected by common CRC32. In order to get more fixed points for the algorithm to quickly determine if two functions have the same instructions, the CRC32 checksums are generated with functions opcodes. They can be used directly to generate additional initial fixed points.

If two functions have the same CRC32 and this value is unique among all the functions, we can say that it's the same function in both binaries. Finding some evident correspondences among functions often leads to additional matches. In our tests, selecting nodes by common CRC32 has successfully increased the accuracy of our method, so it's a very good and easy filter to make fast discards of functions with the same code.

3.4 Control Flow Graph (CFG)

An executable can be seen as a group of nodes and edges, as we said before. However, every node can be seen as a directed

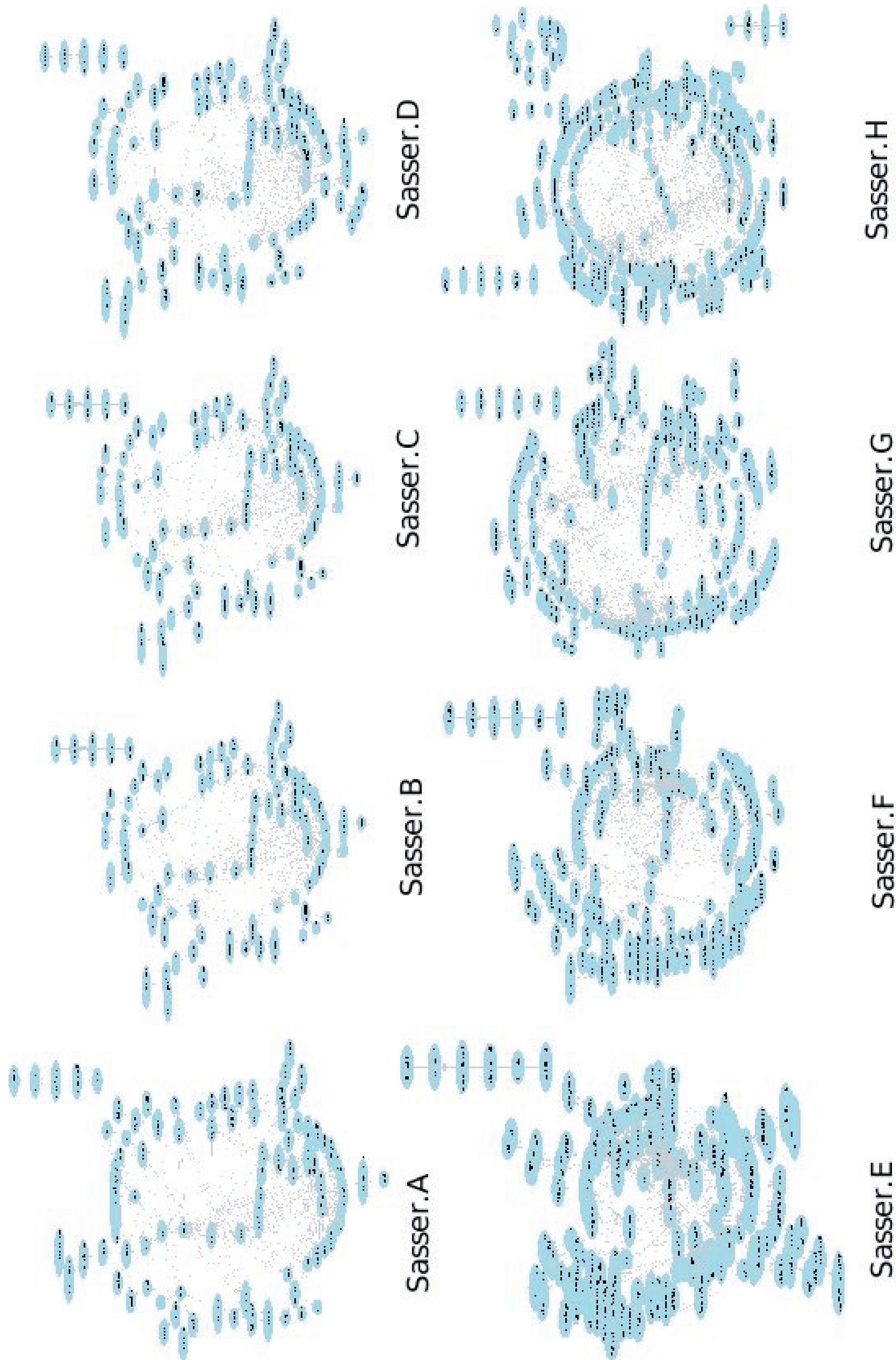

Figure 2: Graph isomorphism among Sasser family samples.

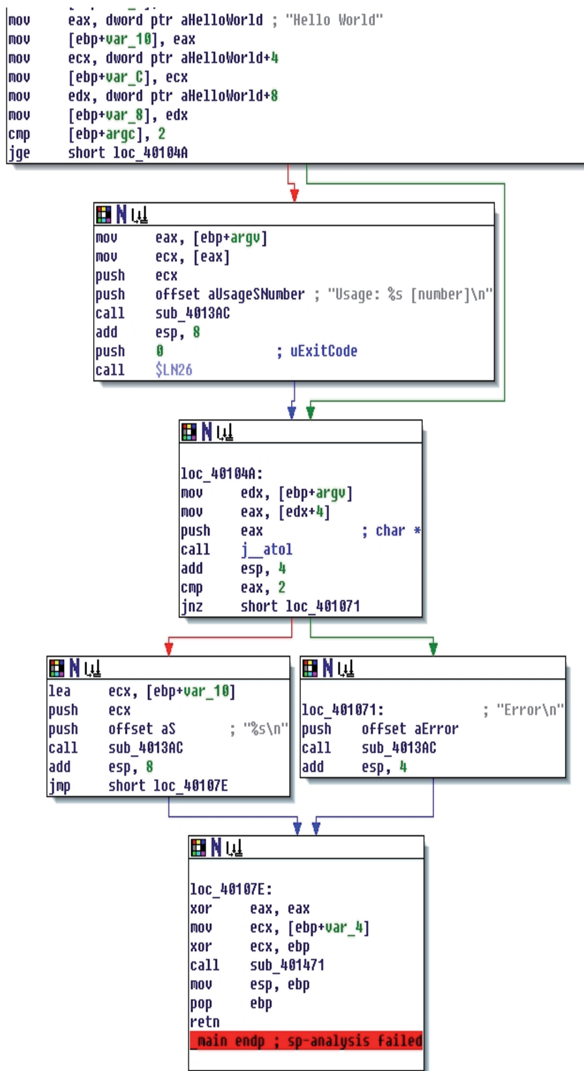

Figure 4: Control flow graph (CFG) of a function generated with IDA 5.

graph itself (Figure 4), with its nodes consisting of the individual basic blocks in the disassembly and the edges representing the branch relations. This graph is usually called a *control flow graph* (CFG) [13].

Every graph's node (called a basic block) represents assembly instructions. Each basic block ends when a jump is found or if the next instruction is the target of a jump. Basically, a basic block is a continuous sequence of instructions that contains no jumps or jump targets (jump targets start a block and jumps end a block). Edges are used to represent jumps in the control flow. There are two special blocks: the entry block (which is not linked to any other basic block) and the exit block (which does not link to any other basic block).

A comparison of these CFGs for the purpose of deciding whether two CFGs are isomorphic can be done by comparing the number of nodes and edges. The general idea is to generate a signature for every CFG in both binaries. We associate every CFG with a 3-tuple (a:b:c) (Figure 4):

a – Number of basic blocks.

b – Number of edges.

c – Number of subcalls.

The 3-tuple is interpreted as a vector in Euclidean space. The Euclidean distance between one element and the others is calculated to find a unique tuple with a minimal distance. Once a tuple with this property is found, we assume that we have found a new fixed point for our adjacency matrix.

Figure 5 shows the expression to calculate the distance for two 3D points in a three-dimensional Euclidean space.

$$P=(p_x, p_y, p_z)$$
$$Q=(q_x, q_y, q_z)$$

$$\sqrt{(p_x-q_x)^2+(p_y-q_y)^2+(p_z-q_z)^2}$$

Figure 5: Three-dimensional Euclidean distance.

4. COMPARISON ALGORITHM

In this section we will describe the core of the system: the comparison algorithm. Basically, as you'll see in the following section, the algorithm is the same as that described by Carrera & Erdélyi. Nevertheless, we will describe some optimizations that help us to reduce the number of samples for which the comparison process needs to be performed.

In the general approach section we explained that we needed a set of initial fixed points. With these initial nodes and applying the algorithm designed by Carrera & Erdélyi [1], we will get a function signature which we will use, as a first phase, to match similar functions between two samples. So given two executables A_1 and A_2 we have to find how to match as many functions (graph nodes) as possible between them.

Our first task will be to use *IDA* to disassemble our binary. Once the analysis has been completed, we'll execute a Python script to extract the callgraph. With the graph representing the binary under analysis, we will create its adjacency matrix. It will have a column for each fixed point and a row for each unidentified function in the binary.

Each element (I,J) of the matrix will be filled with a 1 if there is a call from function I to function J, and with a 0 there isn't. Each row will be a string of ones and zeros which will be considered as the call-tree signature of the function. If two binaries are derived from the same source code, most of their functions will be similar and will make similar calls to API and non-API functions. These adjacency matrixes can determine if the call-trees of the functions are the same.

4.1 Algorithm: step by step

The whole process is shown in the following tables and can be explained with the following steps:

- Given the graph of the binaries under analysis, we define:

- A_f = {Set of all functions from binary A}
- B_f = {Set of all functions from binary B}

- Find the common functions, that is:

$C = \{f_1 : f_n \in A_f \cap B_f\}$

- Set of functions which *IDA* identifies as operating system and library calls, which are shared between the two binaries.
- Set of functions, between binaries, which have the same unique CRC32.
- Set of functions, between binaries, which have a unique tuple (from CFG) with a minimal Euclidean distance.

- Compose two adjacency matrixes, one for each binary. These matrixes will have the initial fixed points (API, CRC32, CFG minimal Euclidean distance) as columns and the rest of the unidentified functions of the binary as rows, that is:

$A_r = A_f - C$

$B_r = B_f - C$

Matrix A

	C1	C2	C3	C4
f1	0	1	0	1
f2	0	0	0	0
f3	1	0	0	0
f4	0	0	1	0

Matrix B

	C1	C2	C3	C4
f1’	0	1	0	1
f2’	0	0	0	0
f3’	1	0	0	0
f4’	1	0	0	0

Remember that element in position (I,J) indicates if a function in row I performs a call to the function in column J.

- Once we have these adjacency matrixes, we start the comparison algorithm.

- Function f_1, from adjacency matrix *A*, will be identical to function f'_1, from adjacency matrix *B*, if the call-tree is identical and unique:

$f_1 \equiv f_1' \Leftrightarrow f_1 \neq \{f_2' \ldots f_n'\}$

- If a match is found (and is unique), we will use these new functions (f_1 and f'_1) as new column elements in each matrix which will allow us to match new functions not relying only on common functions (initial fixed points). In this case:

f1=f1’ and f1 != {f2’,f3’,f4’}

Matrix A

	C1	C2	C3	C4	f1
f2	0	0	0	0	1
f3	1	0	0	0	1
f4	0	0	1	0	0

Matrix B

	C1	C2	C3	C4	f1’
f2’	0	0	0	0	1
f3’	1	0	0	0	1
f4’	1	0	0	0	0

- After a match is found we will start the process again from the beginning (from the first row):

f2=f2’ and f2 != {f3’,f4’}

Matrix A

	C1	C2	C3	C4	f1	f2
f3	1	0	0	0	1	1
f4	0	0	1	0	0	1

Matrix B

	C1	C2	C3	C4	f1’	f2’
f3’	1	0	0	0	1	0
f4’	1	0	0	0	0	1

- This process ends when there are no more matches. After that, we will have a matrix with all the identified functions as columns (initial fixed points and non initial fixed points) and all the unmatched elements as rows. These unmatched functions will be of two types:

- The ones which can’t be identified and can’t be matched.
- Elements which don’t have a unique match.

f3 != {f3’, f4’} and f4 != {f3’, f4’}

Matrix A

	C1	C2	C3	C4	f1	f2
f3	1	0	0	0	1	1
f4	0	0	1	0	0	1

Matrix B

	C1	C2	C3	C4	f1’	f2’
f3’	1	0	0	0	1	0
f4’	1	0	0	0	0	1

- Once we have exhausted the call-tree signatures, we could try to match more unidentified functions with CFG again. Some of the non-unique tuples (from CFG) could now be unique, because lots of the previously unidentified functions are now identified and have been moved to matrix columns.

4.2 The index of similarity

Once the comparison algorithm has finished, we will know how many functions are shared between both binaries. Now it’s time to measure how close the two binaries are. We will call this the metric index of similarity. First we will define some terms:

A_e = *{set of equivalent functions in binary A}*

B_e = *{set of equivalent functions in binary B}*

Both of them include the API functions which are common in the two binaries. Carrera & Erdélyi define this index as follows:

$$\frac{|A_e||B_e|}{|A_f||B_f|} = \frac{|matchedfunction|^2}{|A_f||B_f|}$$

$A_e == B_e$ because they contain the same elements (matched functions).

We propose another expression:

$$\frac{|A_e|}{2|A_f|}+\frac{|B_e|}{2|B_f|}$$

which behaves more linearly. Figure 6 shows the behaviour for an increasing number of matched functions.

Both indices of similarity give values between 0 and 1. Values close to 0 indicate a low degree of resemblance, and values near 1 mean that they share almost all of their code.

5. OPTIMIZATIONS

5.1 Programming language

Two algorithm implementations were produced during the development of the project. We carried out earlier studies of the algorithm viability to verify the accuracy of the comparison. We decided to implement it with Python because it suited our needs well. Python is easy to learn, object-oriented and supports complex data structures which are an essential part of the method. When the system speed became the bottleneck, we decided to migrate the entire algorithm to C/C++.

5.1.1 Python version

Our first version was developed using Python. Python allows fast development and helped us to verify the viability of the system. Python has built-in support for a wide range of data structures: lists, dictionaries (hashes) and tuples. Using Python, the algorithm can be written in a few lines. On the other hand, Python has some disadvantages: execution speed is one of them. Python was perfect for the test version. Nevertheless, it became very slow for our purposes. We tested other ways of speeding up Python code: Psyco [16], a Just-In-Time (JIT) compiler and extending our Python code with C/C++. Finally, after some performance tests, we concluded that we needed to develop the comparison algorithm in C/C++.

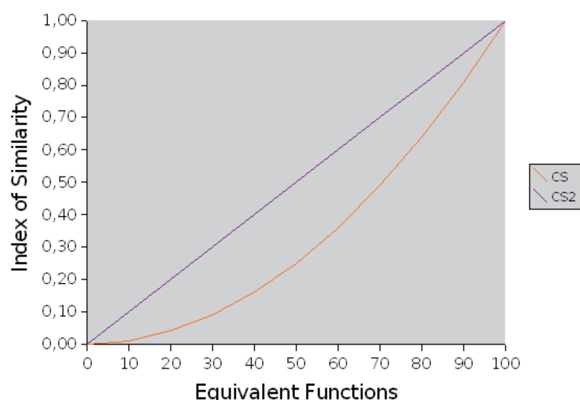

Figure 6: Behaviour of both indices of similarity for an increasing number of matched functions.

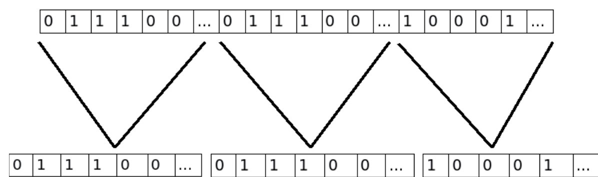

Figure 7: Matrix rows as blocks of 32 bits.

5.1.2 C/C++ version

Although Python (via *IDA Python*) is awesome at extracting and exporting malware data, Python programs run slower than C/C++ implementations and were very inefficient for our comparison algorithm, which needed to be as fast as possible. After trying several approaches with Python, implementing the algorithm in C/C++ was considered the best solution.

Although the C/C++ algorithm implementation seemed to be, at first sight, as fast as we wanted, we noticed that the comparison between samples with thousands of functions wasn't very efficient and required an excessive amount of time to complete. In the following section we will discuss some of the improvements we made.

5.2 Algorithm optimizations

5.2.1 Matrix rows as bits

Comparison of adjacency matrix rows (generally from a hundred to thousands of elements) is a time-consuming task. They could be compared as strings. However, string comparison is slow and would decrease the performance of our algorithm. We needed a better approach.

Rows are combinations of ones and zeros, so we could treat these as groups of bits, as shown in Figure 7. Comparison of bits is faster than string comparison. Basically our approach is:

- Split each row into blocks of 32 elements (each column represents a bit), padding the last block with zeros if needed. Each block represents 32 bits.
- Compare rows as groups of 32 bits.
- Once a block isn't equal, discard the comparison and start with a new row.

5.2.2 Streaming SIMD Extensions (SSE)

'Streaming SIMD Extensions (SSE) is a SIMD (Single Instruction, Multiple Data) instruction set extension to the x86 architecture, designed by Intel and introduced in 1999 in their Pentium III series processors as a reply to AMD's 3DNow! (which had debuted a year earlier). SSE contains 70 new instructions.' (from *Wikipedia* definition [17])

Even better, why compare 32 bits each time if we can compare 4x32 bits at once? SSE added eight new 128-bit registers: XMM0 through XMM7. Each register packs together 32-bit single precision floating point numbers. So we could use them to compare 4x32 bits at once. Instead of comparing 32 bits in 32 bits, now we are comparing 4x32 bits in 4x32 bits, which is much faster.

6. SAMPLE PROCESSING FLOW

Our expert system should gather enough information from already known malicious code to identify new variants. The whole process should be totally automated. Figure 8 covers the flow of the sample through our system:

- Unpack sample.
- Place the unpacked sample in the grid queue:
  - XML-RPC server.
- Once a computer, belonging to the grid, requests a new task (via XML-RPC), it will do the following:
  - Receive the unpacked sample.
  - Analyse, extract and store the internal structure of the sample in the DB (graph, CFG, functions' CRC32).
  - Select the most accurate samples to compare against.
  - Compare it against the selected samples.
  - Send comparison result.

6.1 Unpack system

Nowadays samples are often packed or encrypted. This system only works with unpacked samples; therefore unpacking them is essential for analysis and comparison with this system. This is one of the system's deficiencies. However, our unpacker system is able to unpack most of the received samples, so we assume that our samples are processed and converted to a plain, unencrypted and unpacked form.

In cases where the system isn't able to unpack them, they will be assigned to a technician who will unpack them manually and place them in the grid queue.

The unpacker system is out of the scope of this paper and so won't be explained here.

6.2 Grid system

'"Distributed" or "grid" computing in general is a special type of parallel computing which relies on complete computers (with onboard CPU, storage, power supply, network interface, etc.) connected to a network (private, public or the Internet) by a conventional network interface, such as Ethernet.' (from *Wikipedia* definition [2])

Nowadays we receive thousands of new samples daily. To cope with all the samples we needed a fast and multi-core computer. Even so, it wasn't enough and our beta system wasn't fast enough to analyse all the samples in time. A new approach was needed.

A multi-core computer wasn't able to handle, in a reasonable time, all the received samples. So what about several computers working together? Distributed computing appeared to be the best solution. After some research and evaluation of public grid systems BOINC [18], GLOBUS [19] and Advanced Resource Connector (ARC) [20], we decided to design and develop our own grid system which could help in our analysis.

The primary advantage of distributed computing is that combining enough nodes can produce similar computing resources to a multiprocessor supercomputer, but at lower cost. Other advantages are:

- Computers with different hardware or software characteristics could be joined to the grid (multiplatform clients).
- This system could easily be scaled, adding more nodes (computers) to the grid whenever necessary.

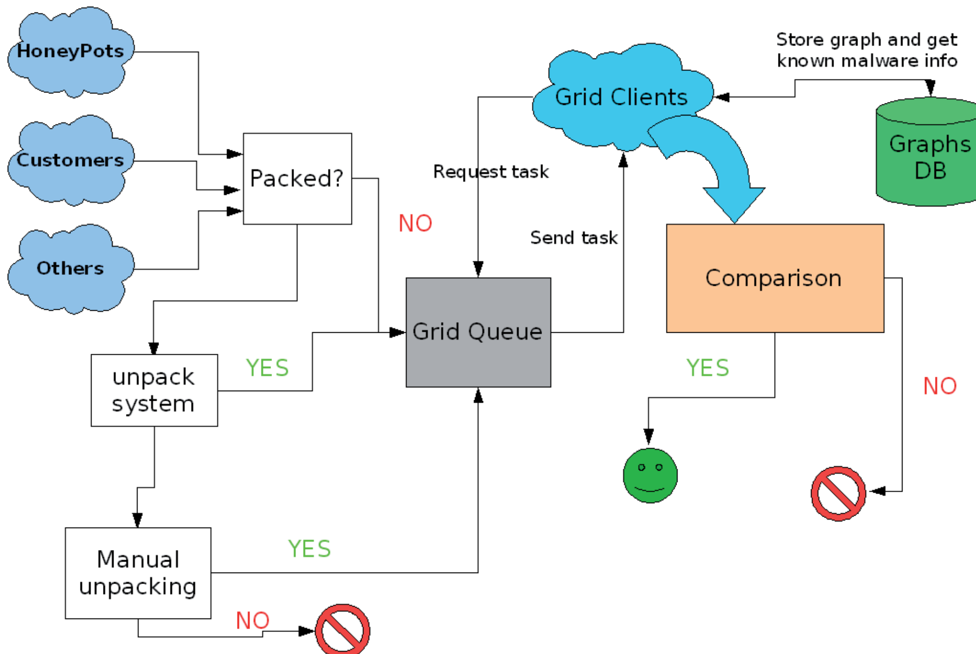

Figure 8: Sample processing flow.

6.2.1 XML-RPC

'XML-RPC is a remote procedure call protocol which uses XML to encode its calls and HTTP as a transport mechanism.' (from *Wikipedia* definition [21])

Distributing computing is essential for our system. We should be able to add, in an easy way, new nodes if needed. More nodes means more samples analysed. Our grid system should be able to manage client requests coming from a wide variety of environments (*Linux*, *BSD*, *Windows*) and even be able to manage different types of tasks. We decided to use XML-RPC as a communication protocol because of its simplicity, minimalism and ease of use.

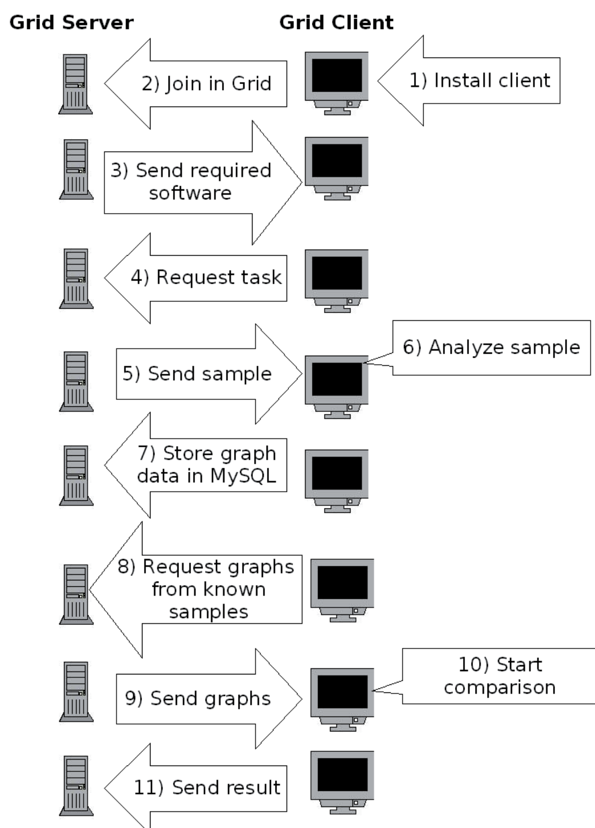

Figure 9: Communication between grid server and clients.

It is designed to be as simple as possible, while allowing complex data structures to be transmitted, processed and returned.

The XML-RPC server should handle:

- Client requests for joining into the grid.
- Client requests for updates.
- Client requests for new tasks (for example: samples to analyse).
- Management of the sample's queue which is waiting to be processed.
- Monitoring client status.
- Enabling/disabling clients.

Figure 9 shows the normal communication between the grid server and clients.

6.3 Analysing, extracting and exporting the data

6.3.1 Binary analysis

IDA is the 'disassembler' and we will exploit its awesome features. *IDA* with *IDA Python* [3] is the perfect weapon to accomplish our first task: analysing and exporting information from samples to database. Most of the time required to analyse a sample is spent on this phase. Future improvements could be made by replacing *IDA* with a faster disassembler, although it's not easy to find a disassembler as versatile as *IDA*. There are faster programs, nevertheless they don't do as many things as *IDA* does: call-tree graph, API recognition, extended functionality with plug-ins, scripting languages (IDC, *IDA Python*), multiplatform (*Windows*, *Linux*).

6.3.2 Python scripts

Several scripts have been written to extract and export the sample information: call-tree graph, functions' CRC32 (only non-API func.), functions' CFG, and to store it all in our database. These scripts are distributed, with the rest of the required software, by the grid server when a new client joins the system.

Python has modules for several database engines and hence our scripts, executed inside *IDA*, can submit any queries to our graphs database.

6.3.3 Database

Our samples will be compared against a battery of previously seen samples. These known samples and their internal structure will be stored in a database. We have chosen MySQL and InnoDB as table engines. This provides a fast, transaction-safe, disk-based storage with row locking and it allows foreign keys.

7. SAMPLE FILTERING

The filtering process to reduce the number of samples for which the comparison needs to be computed is really important. Currently our graphs malware database has approximately 1,100,000 preprocessed samples. This huge number of samples is used to compare against new binaries. Although the algorithm optimizations have increased the performance of the comparison algorithm, it is impossible to compare one sample with the entire database in a reasonable amount of time.

Several filters are applied to select the 'best' samples to compare with. This filtering process is really important, due to the speed requirement. The range of applied filters goes from simple methods (file size, compiler type, number of API and unidentified functions etc.) to more complex ones such as sections entropy and custom checksum.

Table 2 shows some of the filters used in our system. Later, we will discuss some of them.

7.1 Entropy & checksum

Two executables could share data or very similar code zones, even between goodware and malware. Some of the reasons for this similarity are:

- Same compiler or programming language
- Packed with the same packer
- Embedded in an installer
- Derived from the same source code
- Infected files

Malware samples that are derived from the same source code or even which are the same program, but compiled with different options, could generate binaries with few binary modifications.

Selection filters	
File size	Select samples with similar file size. **Advantages:** A bigger file implies more functions, and hence more functions which probably couldn't be matched. Avoiding comparison with huge files improves the system performance. **Disadvantages:** Some samples, when unpacked, keep the packed code shared with the unpacked (increasing the file size). This could result in discarding samples whose unpacked code could be equivalent.
Compiler type	Select samples which have been compiled with the same compiler. **Advantages:** Avoids comparison of malware programmed in different languages. (Comparing a sample compiled in Delphi with a sample compiled in C++ is a waste of time.) **Disadvantages:** There isn't a general method to detect compiler types. PEiD signatures are a good approach. However, it's not 100% efficient.
Number of API functions	If the number of API functions between compared samples is not very similar, it probably means that the samples are not related.
Number of custom functions	The same as number of APIs. If the number of functions belonging to the samples aren't close enough, the index of similarity will be very low. The idea is to get the index of similarity supposing that all functions match. If it's less than a given value, discard the sample.
Entropy & checksum	This is the best filter to select samples that we've added. It usually gives samples with a very close similarity, so the comparison algorithm is faster and more accurate. The number of selected samples usually varies from several hundred up to two thousand.

Table 2: Some of the selection filters used in the system.

Although they are the same program they are different. These small binary differences make it impossible to use hash algorithms like MD5, SHA1, etc. to find a correlation among samples.

However, other methods could be used to measure these small binary changes, for example generating a checksum [22].

In this section, we will describe an alternative method aimed at classifying equivalent files. The checksums, generated from significant zones of an executable, and taking into account the entropy of these zones will be the core of the algorithm.

7.1.1 Checksum

'A checksum is a form of redundancy check, a simple way to protect the integrity of data by detecting errors in data [...] It works by adding up the basic components of a message, typically the assorted bits [in our case each byte]*, and storing the resulting value.'* (from *Wikipedia* definition [22])

Some interesting checksum properties are:

- Two or more blocks which are very similar at binary level have very close checksum values. Figure 10 shows two blocks, with minimal differences, whose checksums are respectively 0x260A and 0x276D. The checksum difference is only 0x163.
- In order to generate a checksum from our block of data, we have to bear two factors in mind:
    - Size of the data block.
    - Base, related to the basic component selected (bits, bytes etc.).

Table 3 shows checksum sizes depending on base and block sizes:

Block size (bytes)	Base	Maximum checksum size
1024	256 (1 byte)	255 x 1024 = 0x03FC00 (3 bytes)
	65,536 (2 bytes)	65,535 x 512 = 0x01FFFE00 (4 bytes)
	4,294,967,296 (4 bytes)	4,294,967,295 x 256 = 0xFFFFFFFF00 (5 bytes)
256	256 (1 byte)	255 x 256 = 0xFF00 (2 bytes)
	65,536 (2 bytes)	65,535 x 128 = 0x7FFF80 (3 bytes)
	4,294,967,296 (4 bytes)	4294967295 x 64 = 0x3FFFFFFFC0 (5 bytes)

Table 3: Checksum sizes depending on base and block sizes.

If we increase the base size, the checksum size increases for the same data block. This increase has an advantage:

a) Checksum: 0x260A

b) Checksum: 0x276D

Figure 10: Checksums from two blocks with minimal modifications don't differ so much.

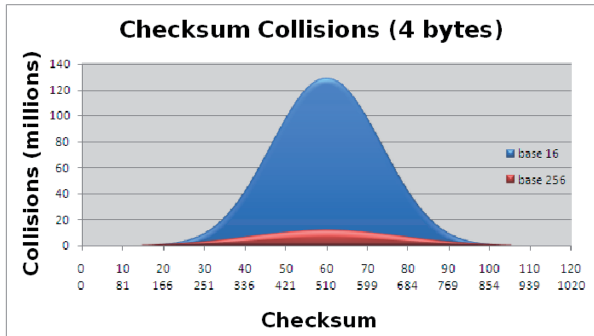

Figure 11: Number of checksum collisions vs base size, for the same data block.

- It decreases the probability of collision (same checksum), as we show in Figure 11.

And of course it has some disadvantages:

- It uses more bytes to represent the same data block.
- Similar data blocks could get significantly different checksum values (Figure 12).

For this reason, as code is represented as bytes, our approach will use a base of 256.

7.1.2 Checksum disadvantages

The simplest form of checksum, which simply adds up the assorted bytes in the data, cannot detect a number of types of error. Such a checksum, for example, is not changed by:

- Reordering of the bytes in the data.

$$Base\,256(BYTE)$$

$$(AA)+(BB)+(CC)+(DD)\rightarrow Checksum=0x30E$$
$$(BA)+(BB)+(CC)+(DD)\rightarrow Checksum=\underbrace{0x31E}_{(diff=0x10)}$$

$$Base\,65536(WORD)$$

$$(AABB)+(CCDD)\rightarrow Checksum=0x017798$$
$$(BABB)+(CCDD)\rightarrow Checksum=\underbrace{0x018798}_{(diff=0x1000)}$$

Figure 12: Bigger base, furthest checksum difference for similar data blocks.

- Inserting or deleting zero-valued bytes.

The number of collisions in different data blocks is evident because different data distributions could lead to the same checksums. However, we've identified some characteristics that aid our purpose. Analysing the checksums obtained from all the possible combinations of small data blocks and selecting a small base, we have observed that we get a representation of the normal or Gaussian distribution [23], as shown in Figure 11.

The X and Y axes represent respectively the checksum value and the number of collisions of the checksum in different data blocks. From this chart we could infer some checksum properties:

- Checksums with minimal and maximal values are safer (fewer collisions). However, the entropy [24] of these zones is very low so they aren't meaningful.
- Data blocks with high entropy always give central checksums (Figure 13).
- Data blocks with medium-low entropy could be in any zone. It depends on the data distribution.
- Denial of a block automatically gives a symmetrical checksum.

7.1.3 Checksum and entropy approaches

We will use these calculated entropies to classify equivalent files. Although we've seen some problems with checksums, as described in the previous section, we have some solutions to bypass them.

7.1.4 Penalize checksums from high-entropy zones

Data blocks with high entropy usually belong to encrypted or packed data. For this reason, two blocks with high entropy shouldn't be really similar and shouldn't be used. To avoid these data blocks with high entropy interfering with checksums from data blocks with low entropy, we designed a function to move the checksum value further away. So minimal checksum changes in high-entropy blocks will generate distant checksum values.

Basically, if entropy is equal to or higher than 7 (really high entropy), the checksum will be multiplied by a value to move it

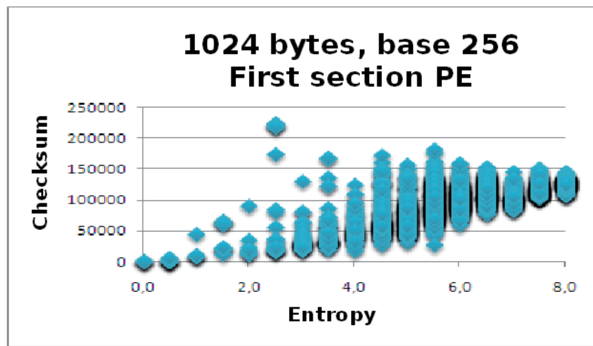

Figure 13: Checksum distribution.

First Section	1Kb
	1Kb
	1Kb
	1kb
	...
Second Section	1Kb
	1Kb
	1Kb
	1kb
	...
More Sections	...
Last Section	1Kb
	1Kb
	1Kb
	1kb
	...

Figure 14: We calculate the checksums of the first, second and last section.

further away, so zones with high entropy but similar checksums won't be treated as similar. This penalized checksum will be calculated as:

$if\ (Entropy \geq 7)$

$\Downarrow$

$PenalizedChecksum = \log2((8 - Entropy)) \times (-60) \times Checksum$

7.1.5 Several checksums are better than one

Reliability is an important factor during our classification system, so we need to use several data block checksums to maintain the trustworthiness of our system. Actually we use data blocks of 1 KB until we cover 4 KB of the selected zone.

We are classifying PE files, so the selected zones we will use are (Figure 14):

- 4 KB, in blocks of 1 KB, from the beginning of the first section.
- 4 KB, in blocks of 1 KB, from the beginning of the second section.
- 4 KB, in blocks of 1 KB, from the beginning of the last section.

7.1.6 The algorithm

The main purpose of this algorithm is to classify files depending on the proximity of their sections. First we need a database of knowledge; in this case, it's a database of pre-calculated checksum/entropy of first, second and last sections of our previously seen malware. This will be the database against which to compare our sample.

So, given a sample 'A' and a database of pre-calculated entropy/checksums, we want to find the samples which carry out the next condition:

$\forall \varphi \in \{1...4\}\ SampleA \approx SampleB \leftrightarrow$

$ABS(SampleA.IniSec\varphi_1K - SampleB.IniSec\varphi_1K) \leq diffChecksum$

AND

$ABS(SampleA.IniSec\varphi_2K - SampleB.IniSec\varphi_2K) \leq diffChecksum$

AND

$ABS(SampleA.IniSec\varphi_3K - SampleB.IniSec\varphi_3K) \leq diffChecksum$

AND

$ABS(SampleA.IniSec\varphi_3K - SampleB.IniSec\varphi_3K) \leq diffChecksum$

AND

$ABS(SampleA.IniSec\varphi_4K - SampleB.IniSec\varphi_4K) \leq diffChecksum$

Resuming, these will be the steps:

- First, calculate and store the checksum or penalized checksum (depending on the entropy) from the first, second and last sections from all the samples you want to compare. As we said before, the checksum is calculated in blocks of 1 KB from the beginning of the sections, up to a total amount of 4 KB ($IniSec\varphi_\{1...4\}K$).
- Select a maximum checksum difference (*diffChecksum*).
- Select our sample, calculate checksum/entropy from sections (as we did before) and compare against the entire database, selecting only those which meet the outlined conditions.

8. CONCLUSIONS

Automatically classifying malware based on their internal structure is a very time-consuming task and its efficiency isn't 100%. However, our project has reduced the time required to verify how similar a piece of malware is to previously seen samples. Although we have proposed some methods to solve some of the significant restrictions which exist there are still some which need to be solved:

- The method only works with unpacked or non-packed samples.
- It is hard bound to *IDA*. Sometimes it requires too much time to analyse a sample (sometimes more than five minutes).
- Big database storage is required.
- Delphi and VB samples don't give reliable results.

9. POSSIBILITIES

The creation of a graph repository could be used to perform quick malware analysis. For example:

- Porting existing information from previous analysis of a malware variant to a newly discovered one, can help achieve very fast analysis.
- Shared code identified among samples can help to develop better generic signatures.
- Associating malware functions with their behaviour (sockets, registry, file system, rootkit techniques etc.) could be used to generate malware information automatically.

REFERENCES

[1] Carrera E.; Erdélyi, G. Digital genome mapping – advanced binary malware analysis. Proceedings of the 2004 Virus Bulletin Conference, pp.187–197, 2004.

[2] http://en.wikipedia.org/wiki/Grid_computing.

[3] IDA Python. http://d-dome.net/idapython/.

[4] http://www.zynamics.com/.

[5] Flake, H. Graph-Based Binary Analysis. Black Hat Briefings 2002.

[6] Flake, H. More fun with Graphs. Black Hat Federal 2003.

[7] Flake, H. Automated Reverse Engineering. Black Hat Windows 2004.

[8] Flake, H. Structural comparison of executable objects. DIMVA, pp.161–173, 2004.

[9] Gheorghescu, M. An automated virus classification system. Proceedings of the 2005 Virus Bulletin Conference, pp.294–300, 2005.

[10] http://en.wikipedia.org/wiki/Graph_theory.

[11] Sabin, T. Comparing binaries with graph isomorphisms.

[12] Dullien, T.; Rolles, R. Graph-based comparison of Executable Objects.

[13] http://en.wikipedia.org/wiki/Control_flow_graph.

[14] http://en.wikipedia.org/wiki/Modified_adjacency_matrix.

[15] DataRescue. http://www.datarescue.com/.

[16] http://psyco.sourceforge.net/.

[17] SSE. http://en.wikipedia.org/wiki/Streaming_SIMD_Extensions.

[18] BOINC. http://boinc.berkeley.edu/.

[19] GLOBUS Toolkit. http://www.globus.org/toolkit/.

[20] Advanced Resource Connector (ARC). http://www.nordugrid.org/middleware/.

[21] XML-RPC. http://en.wikipedia.org/wiki/XML-RPC.

[22] http://en.wikipedia.org/wiki/Checksum.

[23] Normal or Gaussian distribution, http://en.wikipedia.org/wiki/Normal_distribution.

[24] Entropy, http://en.wikipedia.org/wiki/Entropy.