POLITECNICO DI MILANO
Corso di Laurea magistrale in Ingegneria Informatica
Dipartimento di Elettronica, Informazione e Bioingegneria

Hybrid Malware Analysis Interface to Simplify Reverse Engineering and Crowd-Sourcing

Relatore: Prof. Stefano Zanero
Correlatore: Dott. Mario Polino

Tesi di Laurea di:
Emanuele Tironi, matr. 823335

Anno Accademico 2015-2016

Abstract

Malicious software, or malware, are one of the most serious threats related to computer systems. Every day the number of threats dramatically increase since malware authors are motivated by the financial gains. Being able to quickly analyze and understand the behavior of those threats is key to move forward in this adversarial battle. Analysts employ techniques mainly belonging to two general categories: static and dynamic analysis. Static methods analyze the code of a malware, they have full code coverage but they are slow and they can be easily rendered ineffective by obfuscation or packing techniques. Dynamic methods analyze the execution flow of a running program but some malware can avoid this kind of analysis by refusing to execute in a controlled environment. It is possible to combine these two methods, performing what is called hybrid analysis: a technique that exploits the advantages of static and dynamic analysis while, at the same time, mitigating their shortcomings. The complexity of the reverse engineering task, that requires a large skill set, and the large number of malicious samples compared to the number of analysts are major disadvantages toward the goal of finding signatures to detect malware. Hereby, we present a tool that allows to take advantage of the analyses results of Jackdaw, a tool that performs hybrid analysis, in a reverse engineering task on a malware, showing the behaviors and their semantic descriptions that are implemented in the sample, in order to help the analysts performing static analysis. Moreover, our tool renders Jackdaw partially crowd-sourced, allowing analysts to improve the behaviors by adding or editing their descriptions.

Estratto

Al giorno d'oggi i programmi malevoli, noti anche come *malware*, sono una delle principali minacce ai sistemi informatici e ai dati privati dei loro utenti. I malware sono lo strumento con cui i criminali informatici perpetrano un ampio spettro di attività illegali, dall'invio di e-mail di spam al denial of service, rivelandosi una profittevole fonte di guadagno per i loro sviluppatori, tanto che si può parlare dell'ascesa di un vero e proprio mercato nero, un esempio può essere quello dei ransomware, un tipo di malware che si è diffuso in modo estensivo negli ultimi anni e che cifra i file presenti su un computer, tenendo a tutti gli effetti come ostaggi i dati della vittima, finché essa non paga un riscatto. Non solo il numero di nuovi malware è in costante aumento, al ritmo di un milione di nuovi campioni ogni giorno, considerando anche che le vecchie versioni vengono costantemente aggiornate per poter evadere le nuove misure di sicurezza, questo significa che gli analisti hanno un elevato numero di malware da analizzare per poter trovare delle caratteristiche distintive, o firme, per poter poi fare una rilevazione corretta dei singoli campioni in futuro. Per poter trovare le firme gli analisti cercano un insieme di azioni che vengono eseguite dal malware e che definiscono il suo comportamento. Questo tipo di analisi viene fatta a mano ed è molto complessa, inoltre il grande numero di campioni da analizzare e le tecniche impiegate per contrastare gli sforzi degli analisti, cifrando e comprimendo il codice, fa si che siano necessari strumenti efficaci, veloci e precisi. In generale le tecniche per poter effettuare analisi su malware si dividono principalmente in due categorie, ognuna coi suoi vantaggi e svantaggi: analisi statica e analisi dinamica. L'analisi statica consiste nell'analizzare il codice macchina o sorgente di un malware, eseguendo un processo di ingegneria inversa utilizzando strumenti come i disassembler. Ha il vantaggio di poter analizzare completamente il codice del campione in esame, ma è un processo lungo e difficile anche con strumenti all'avanguardia, inoltre è reso ancora più complesso da tecniche che cifrano o comprimono il codice rendendolo impossibile da analizzare. L'analisi dinamica consiste nel lasciare eseguire un malware all'interno di un ambiente controllato, detto sandbox, e di analizzare le azioni che intraprende, in questo modo si superano le limitazioni dell'analisi statica ma non si ha la certezza che tutte le caratteristiche del campione siano state osservate, inoltre alcuni malware sono in grado di riconoscere quando eseguiti in una sandbox e nel caso si inibiscono.

Un buon compromesso tra analisi statica e analisi dinamica è quello di utilizzare strumenti, come Jackdaw, che effettuano quella che viene chiamata analisi ibrida. Jackdaw trova gruppi di chiamate alle funzioni di sistema (le WinAPI) connesse da dipendenze dati, fatte dal binario

sotto esame. Questi gruppi vengono chiamati *comportamenti*, in quanto definiscono cosa fa il binario una volta eseguito. Inizialmente Jackdaw esegue del clustering su dati, dei grafi di flusso di controllo, ottenuti da analisi statica, sfruttando anche delle dipendenze tra funzioni scoperte durante l'analisi dinamica del campione, quindi ,per ogni cluster, estrae dei gruppi significativi di chiamate agli API che saranno poi i comportamenti. Infine, Jackdaw assegna ad ogni comportamento una descrizione che definisce a livello semantico quello che il comportamento stesso fa in termini di funzioni. Per fare ciò cerca in Stackoverflow tutte le domande relative agli API che descrivono il comportamento ed estrae i loro tag. Alla fine di questo processo Jackdaw ha costruito un database di comportamenti e delle loro descrizioni. Con questo lavoro presentiamo un strumento in grado di fornire un aiuto agli analisti di malware che stanno effettuando un lavoro di ingegneria inversa su un file, semplificandone il processo, integrando i risultati dell'analisi ibrida di Jackdaw in strumenti che rappresentano lo stato dell'arte nell'ambito dell'analisi statica, il tutto senza che gli utenti debbano condividere i campioni che stanno analizzando. Inoltre, tramite questo lavoro, ci proponiamo di rendere Jackdaw uno strumento crowd-sourced, ovvero gli utenti che lo utilizzano contribuiscono al loro miglioramento aggiungendo descrizioni ai comportamenti o modificando quelle esistenti qualora non siano corrette. Vista l'importanza e l'aiuto che una buona visualizzazione può dare nell'analisi statica abbiamo adottato varie tecniche per migliorare l'esame dei malware: i comportamenti vengono visualizzati sotto forma di grafi nei quali i nodi rappresentano le chiamate di sistema, e sono arricchiti con informazioni riguardanti le chiamate stesse, e i collegamenti sono le dipendenze dati tra loro; per aiutare l'analista, il nostro strumento permette di evidenziare tutte le istruzioni relative a un comportamento selezionato con lo stesso colore che viene associato al comportamento stesso e che non cambia tra un'analisi e l'altra. Inoltre, quando una istruzione del malware viene selezionata il nostro strumento mostrerà in automatico a quali comportamenti appartiene. Infine sono state sviluppate funzioni aggiuntive che permettono di migliorare la parte di crowdsourcing, introducendo, oltre alla possibilità di modificare o aggiungere descrizioni per i comportamenti, anche il fatto di poter votare quelle che meglio li descrivono e la possibilità di inserire descrizioni private.

Contents

List of Figures

Chapter 1

Introduction

Malicious softwares, commonly known as *Malware*, are a widespread threat over computer systems and their users' information. Malware is the tool used by cybercriminals to perpetrate a wide range of malicious intents, from sending spam emails to denial of service, revealing to be a profitable source of money, to the point that we can talk about a rise of a proper underground economy, an example of which could be ransomwares, a kind of malware that became extremely common in the latest years: they encrypt files on a computer, holding the victim's data hostage until a ransom is paid. The number of new malware is constantly increasing, sources talk about one million of new threats emerging every day[1], and old samples are constantly updated in order to evade the existing security measures, this means that analysts have to deal with a large number of malicious programs to analyze in order to find distinctive features, or *signatures*, for an effective detection. To deal with this kind of problems it is necessary for the analysts to find a set of actions, a sequence of events, that the malware performs once running in the infected environment. This sequence of events is what is called a *behavior*. This analysis is accomplished by hand and the significant number of malicious programs relative to the number of analysts, and the obfuscation techniques, lead the malware analysts to the need to have faster, better and more precise tools available to them.

The common techniques employed in malware analysis are usually divided in two main categories, *dynamic* analysis and *static* analysis, and both of them have their own strengths and weaknesses. Static analysis is the analysis of the code of a malware by means of tools like the disassemblers, that show the machine code of a binary file. This kind of process has full code coverage

[1] http://blog.trendmicro.com/malware-1-million-new-threats-emerging-daily/

but it is a hard and slow, even with state of the art tools, and it is made difficult by the use of obfuscation or packing techniques that confuse the code or encrypt it. Dynamic analysis, on the other hand, is the observation of how a running program behaves in a safe, monitored environment, or *sandbox*. This kind of analysis overcomes eventual obfuscation techniques that their creator may have used to render difficult an eventual analysis of the program code, but it does not have full code coverage, that is that some of the function of the malware may not be executed and therefore some of its properties remain undiscovered. As we said, in both cases cybercriminals have many resources they can use in order to avoid the analysis process, such as packing and obfuscation, to make the code harder if not impossible to analyze in order to counteract static analysis, or anti-debugging and anti-sandboxing techniques, that make the malware sensible to the environment in which it is running in order to avoid dynamic analysis in controlled environments.

A good compromise between static and dynamic analysis is to use tools, such as Jackdaw, that exploit the advantages and mitigate the disadvantages of both the two methods, performing what is called *hybrid* analysis.

Jackdaw finds groups of data-related system calls (in particular WinAPI calls) invoked by the binary, these groups are called *behaviors*, as they defined what the binary file does once running. At first Jackdaw clusterizes data obtained by static analysis (some control flow graphs) exploiting taint dependencies discovered during dynamic analysis, then it extracts representative groups of API calls for each obtained cluster, these models are the final behaviors. Finally Jackdaw assigns to each behavior a semantic description, that is a tag that describes the behavior in natural language, generated by querying a database of Stackoverflow posts searching for questions related to the APIs that describe the behavior and extracting their tags.

At the end of this process Jackdaw has built a database filled with behaviors and their semantic descriptions.

The first goal of this thesis is to provide an aid to malware analysts performing a reverse engineering task on a file, simplifying the reverse engineering process, by means of integrating the results of Jackdaw's analysis into other popular tools used by analysts, all of this without having the users to share their samples, either for security reasons or because companies are reluctant to share new samples, but instead by computing, on the client machine, the list of fingerprints, that are sub-control flow graphs whose nodes are the calls to the WinAPIs, that appear in the binary file. The second goal of our work is to render Jackdaw a crowd-sourced tool, users can in fact improve Jackdaw

results, enhancing the descriptions associated to the behaviors.

The possibility of taking advantage of Jackdaw's hybrid analysis results takes the shape of a plug-in that integrates in an existing software and which provides the users with a list of the behaviors found in the analyzed malware and their semantic descriptions, and a server back-end, which handles the requests from the plug-in and queries the behaviors' database. The plug-in allows users to navigate the graphs of those behaviors that have been found in the binary file and to see their APIs, that are the nodes of such graphs, and their significant parameters. Also, given the significant advantage that a good visualization method has in the reverse engineering process, the plug-in improves the disassembly view by detecting if a selected instructions belongs to a behavior, and in that case showing its graph, or, when a graph is opened by the user, by coloring all those lines of assembly code that belong to that behavior. In this way it is much faster to recognize a behavior in a malicious software by means of static analysis with a disassembler: the analyst does not have to manually analyze an extensive number of lines of code to find the behaviors that characterize the binary, it is enough that those same behaviors had been found in other samples analyzed by the Jackdaw tool-chain and the plug-in will highlight them with different colors. Moreover the plug-in is meant to be a collaboration tool: analysts can enrich and improve the existing results, correcting or adding new semantic descriptions to those known behaviors that may be mistakenly tagged by the automatic process or for which a description had not been found and making the new modifications available to everyone or to closed teams.

Existing tools developed in our same line of works are described in State of the art (Chapter 2); the general approach to the problem of integrating Jackdaw's results in a disassembler, of behaviors visualization and of creating a collaborative tool for analysts, is described in Approach (Chapter 3); in Implementation (Chapter 4) we discuss the implementative details of the tool we developed; we performed some performance tests that are discussed in Chapter 5; in Chapter 6 we discuss the conclusions of our work, the limitations and the future works.

Chapter 2

State of the art

Several tools have been developed in order to perform and simplifying the reverse engineering task such as those using directed graphs to represent binary files, a technique that has been proven useful in analyzing malicious softwares [1][2] and diffing binary files [3]. Among these tools there are the open-source tools by Zynamics: BinDiff [4], which allows to discover differences in two binary files showing a graph-based view of the samples, and BinNavi [5] that allows to annotate the control flow graph of disassembled code and to share analyses results among a group of analysts. Moreover, BinNavi ships an IDA Pro plugin useful to import disassemblies generated in IDA. The analysis with this kind of tools is still, however, manual and can be time consuming, even if it is possible to involve debuggers such as GDB. Another tool, Jackdaw, has been implemented to overcome the limitations of static and dinamic analysis by performing hybrid analysis. Jackdaw can automatically extract the behaviors of binary files and it can assign them a semantic description.

2.1 BinDiff and BinNavi

BinDiff and BinNavi are both tools from Zynamics. BinDiff allows, by means of a graphic interface, to see the differences between two versions of a binary file, in order to quickly find similarities and differences in disassembled code, and to find vulnerabilities by comparing a patched version of a binary file and an un-patched one. The two versions are placed side by side in a symmetrical layout, as can be see in Figure 2.1. BinNavi, shown in Figure 2.2 is an IDE for binary analysis. It allows users to navigate the control flow graphs of binary files, to edit and annotate them and share analyses results within a group of

analysts. However, with BinNavi, the analysts must find the behaviors by hand by reverse engineering the code, and the annotations are specific of a binary file and are not used to enhance the behaviors that could be found again in future analyses on different files.

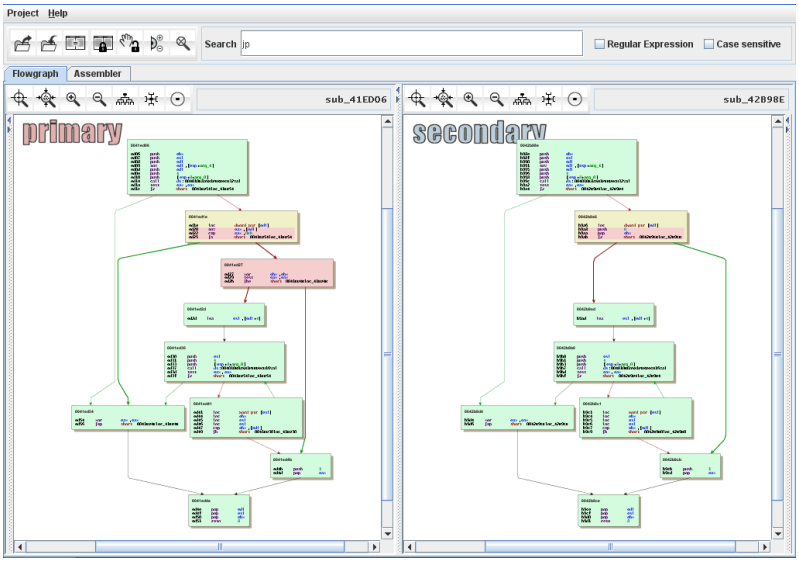


Figure 2.1: BinDiff: the graphs of two versions of a file are placed side by side.

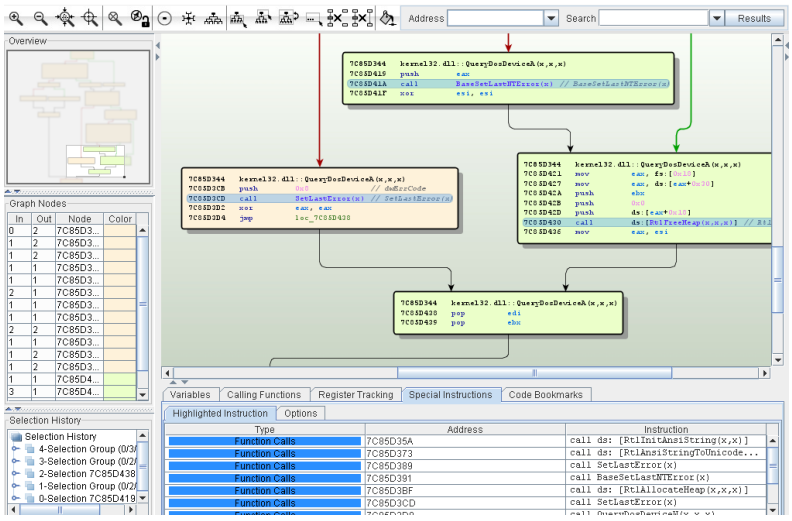


Figure 2.2: BinNavi: it allows to navigate, edit and annotate the CFG of a binary file.

2.2 IDA Pro

The *Interactive Disassembler* [6] is a software used to generate assembly language source code from machine-executable code and to explore binary programs. IDA Pro is considered the state of the art disassembler and it is one of the main tools used for the static analysis process. Also, IDA Pro can decompile a file, converting executable programs into a human readable C-like pseudo-code, however this function it is limited by many factors, besides being ineffective in case of obfuscated or packed code: in case of manually crafted code the results are not as good as for code generated by compilers, moreover it does not support exception-handling, it does not perform type recovery and global program analysis.

IDA Pro provides a graph view, shown in Figure 2.3, which displays the control flow graph of the malware, and in which functions are represented as a collection of nodes and edges. Nodes represent basic blocks and edges are the cross-references between them. A basic block is set of instructions that has a single entry (the first instruction of the block) and a single exit (the last instruction of the block), also every instruction transfer the control of execution to exactly one successor within the block.

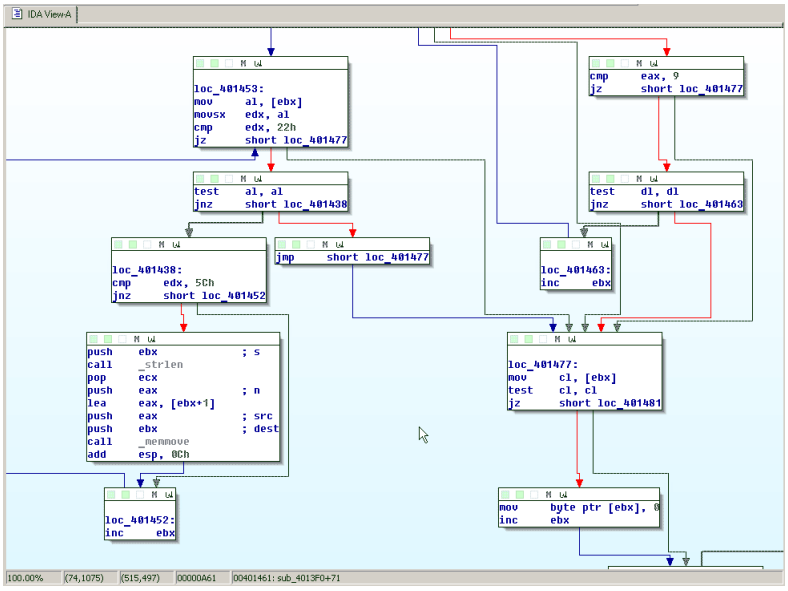


Figure 2.3: IDA Pro graph view

Moreover, if the analyzed binary is not obfuscated in some way, IDA Pro provides a list of API calls by looking at the Import Address Table. However,

is up to the analyst to figure out the dependencies between the calls and the relative behaviors. This is the point we try to improve: by taking advantage of the results obtained by Jackdaw, our plugin is capable to find the behaviors automatically, significantly simplifying the work of the analyst.

2.3 Jackdaw

There have been previous attempts to automate malware analysis, for instance one of the previously existing tools developed on this line of work is Reanimator [8]: it recognize different implementations of the same dynamic patterns, or *behaviors*, in a set of malware files, discovering functionalities that could have been remained hidden during a dynamic analysis, however it is not completely automated since it is necessary to manually specify those behaviors. Other analysis tools are based on the extraction of sub-Control Flow Graph for the static analysis and, as showed in [9], these sub-graphs are robust against polymorphism (different versions of the same algorithms), and this is a major advantage against malware authors' practice of changing their code to fool anti-malware static signatures. This evolution of code in malware families is the subject of Beagle [10], a tool that periodically downloads newer versions of a given malware and compares them with the older ones by means of static and dynamic analyses. In order to perform its analyses, Beagle needs a set of behaviors that, however, as in Reanimator, need to be manually defined.

Jackdaw is a hybrid analysis tool meant to automate the malware analysis process. Unlike Reanimator and Beagle, Jackdaw has the capability of automatically extracting the behaviors and they are enriched with information obtained analyzing more implementations of the same behavior. Moreover Jackdaw associates a semantic tag, that is a meaning, to each behavior, rendering them more "user friendly" and to the reach of less experienced analysts.

2.3.1 Sample analysis and fingerprints extraction

Jackdaw uses the open source sandbox Cuckoo [11] to perform the first part of its analysis: the samples are loaded and the system checks their entropy: if it is too high it could be a sign of obfuscation and the sample is analyzed by PINDemonium [12], an unpacker for windows executables which dumps the memory of the process and rebuilds the binary.

Cuckoo, with the Jackdaw modules, outputs the results of the taint analysis, with the API calls executed by the sample and the dependencies between them, and the fingerprints. The fingerprints are sub-Control Flow Graphs (CFG), described in [9], of the analyzed malware, indexed with their hashes. The main CFG is generated by data flow dependencies while the sub-CFGs are generated by recognizing them in different malware. Since only the semantics of the API calls are important to Jackdaw's analysis to create the behaviors, the names of the API calls can be normalized by stripping them from the suffixes that specify either the mode in which that same call is executed or a new version of the call ("A" for Ansi, "W" for Unicode and "Ex" for Extended). In addition to the names of the APIs, the names and the values of their parameters are also extracted, since some of them may be significant for the analysis.

2.3.2 Clusterization and behavior extraction

At this point only the taints that presents some fingerprints are significant since the next step is to group similar sequences of data-flow dependent API calls. The one-pass clustering algorithm is based on the ECM algorithm and it is applied to an input stream of data (Algorithm 1). In order to decide whether a sub-graph can be part of an existing cluster or if it has to be a new one, it is necessary to use some kind of similarity measure: for this purpose the similarity between the sub-graphs is measured using the Jaccard's Similarity (2.1).

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{2.1}$$

Jackdaw now has to extract the rules that characterize each cluster. For this, the first adopted heuristic is the frequency with which an API appears in a given cluster – always keeping in mind the sequences of data-flow dependent APIs – that is an API is considered representative of a cluster if it appears more frequently than a threshold f=0.75 in that cluster.

Once the most representative APIs are found, Jackdaw merges those clusters whose representative APIs has the same function names, obtaining in this way a graph showing also relationships between different behaviors.

Algorithm 1 Clustering algorithm based on ECM

```
1: Input : dataflow_set; clusterset = {c_1 ... c_l}
2: for all t ∈ dataflow_set do
3:     for i ∈ {1 ... l} do
4:         s_j ← J(t, c_i)
5:     end for
6:     s_i* ← argmax(s_i)
7:     if s_i* > u then
8:         c_i* ← c_i* ∪ {t}
9:     else
10:        new c_{l+1}
11:        c_{l+1} ← {t}
12:        clusterset ← clusterset ∪ c_{l+1}
13:        l ← l + 1
14:    end if
15: end for
```

2.3.3 Semantic tagging

Jackdaw must now tag each behavior with a semantic description, that is a set of keywords obtained by all the single APIs that compose the behavior. For this purpose, Jackdaw searches in a dump of Stackoverflow posts for the API names and their relevant parameters. A post in Stackoverflow has a title, a body and a set of tags, as shown in Figure 2.4; Jackdaw searches for the APIs, which describe the behavior, in these fields and then it assigns a score to each post retrieved based on the keywords that same post has been tagged with by the user who submitted it: a score of +1 is given to those tags that appears in a previously defined list of interesting tags; a score of -1 is given to those trifling tags that are not useful to define the behavior.

Only those posts whose score is positive are marked as relevant and then each tag is weighted with with the post score as:

$$Score(tag, post) = \frac{Score(post)}{N} Found(tag, post)$$

where $Score(post)$ is the overall score of the post, N is the total number of relevant posts and $Found(tag, post) \in 0, 1$ and it is 1 only if the *tag* is contained in the *post*.

Then a vote for each tag is computed as

$$Vote(tag) = \sum_{post \in allPosts} Score(post)$$

building in this way a ranked list of tags associated to the given behavior.

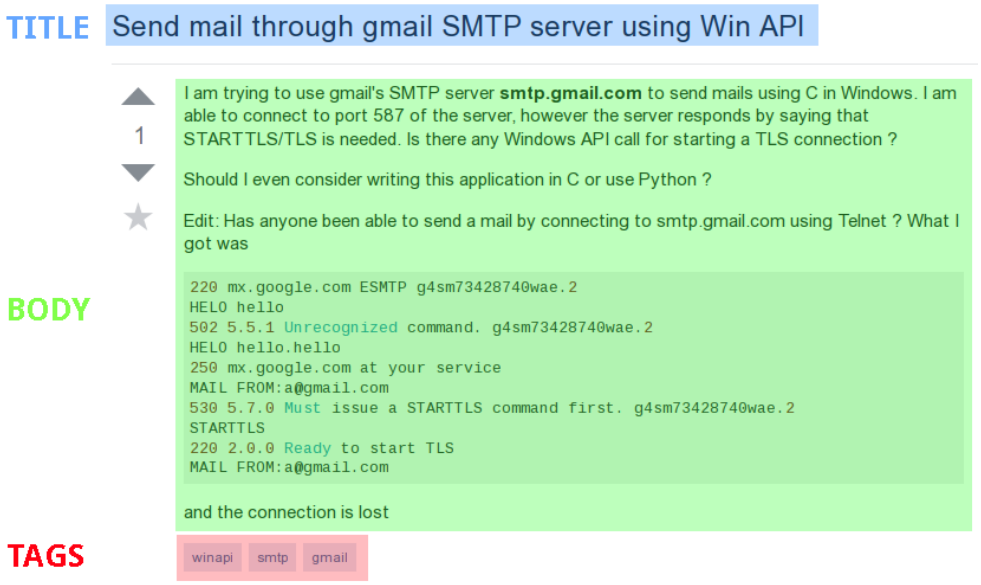


Figure 2.4: Example of a Stackoverflow post

At this point the behaviors and their semantic descriptions are ready to be used and this is the main goals of this thesis: making the behaviors available to analysts who are performing static analysis, enriching the features of IDA Pro, and eventually making Jackdaw in some way crowd-sourced, giving the possibility to the analysts to correct wrong behaviors and adding new ones.

Chapter 3

Approach

3.1 Overview

The general approach to the problem of using Jackdaw's hybrid analysis results into the process of reverse engineering has been to write a plugin for an existing software that is already used by analysts in the field, in this way users can still make use of tools they are used to, but with the advantage to have an additional aid in the analysis process, coming from our plugin.
The idea is that users can see all the behaviors and their descriptions of the binary they are analyzing, while looking at the assembly instructions. In this way it is possible to see which instructions originates which behaviors. Moreover, since it is possible that some of the descriptions may be inaccurate or wrong for some of the behaviors, the plugin gives the users the ability to modify or to add new descriptions, as it can be seen in Figure 3.7, improving the results provided by Jackdaw's automated analysis and, in this way, making Jackdaw a partially crowdsourced analysis tool.

Since the results that the plug-is is meant to show are provided by Jackdaw, that is running on a remote server, and the database of behaviors is on a remote location as well, the architecture used to implement our tool is client-server, in which the plug-in is the client and the server back-end we implemented answers to the plugin requests, acting as a middle layer between the plugin and the database, always checking the user status (registered, not-registered) in order to give the correct set of descriptions associated to a given behavior and to allow some advanced functions that are available only to registered users. Also the server has the task to return results that are easy for the client to handle, striping useless informations from the answers or re-arranging the information to be returned in a way that the client can easily parse and use.

An overview of the various components and the connections between them is represented in figure 3.1: on the client machine the sample is loaded in IDA Pro to be analyzed; our plugin wraps Disasm, that is executed on that same sample to retrieve its fingerprints, and communicates with the server backend over a secure TCP channel in order to send the fingerprints and retrieves the graphs; the backend queries the database for the behaviors that are generated by the Jackdaw's analysis and sends back the response to the plugin.

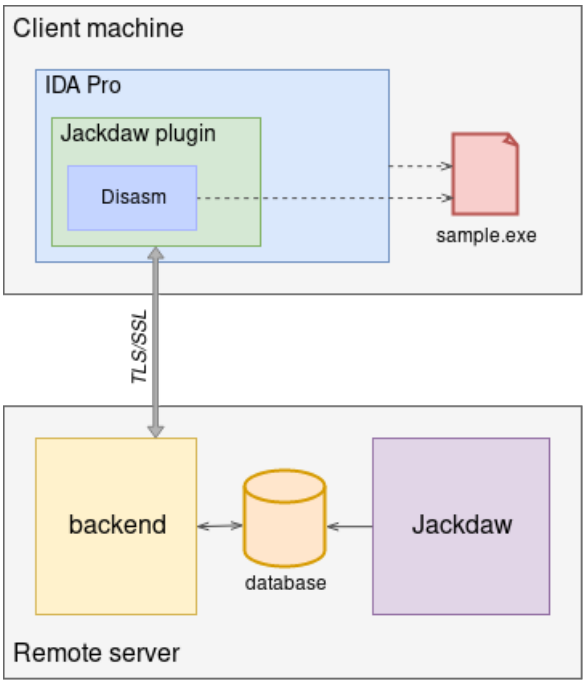


Figure 3.1: Scheme of the architecture

3.2 Disasming

When it is activated, the first step the plugin performs is the execution of Disasm [9], provided with the plugin, on the analyzed sample. Disasm produces a list of fingerprints, that are sub-control flow graph of size 10, containing also information about the instructions that implement them, each one associated to the corresponding offsets of the entry points of those functions that belong to the Data Flow Graph characterizing that fingerprint. Therefore the presence of Disasm on the analysis machine is a requirement for the proper functioning of the plugin. The alternative would have been to submit the

whole binary file to the server and extract there the fingerprints. We decided to execute the disasming process on the analysis machine, with the plugin, for two reasons: the main one is that some companies or analysts are reluctant to share new samples of malware they are analyzing, so with the method we have implemented these people can use our tool as well; the second reason is because submitting the fingerprints is much more convenient than submitting the whole binary to the server: in terms of performances it is faster to just execute Disasm on the client than uploading the binary file and extracting the fingerprints on the server; in terms of security the malicious sample does not leave the analysis machine, which is supposed to be a confined and isolated environment.

3.3 Querying for behaviors

As we described in Section 3.1, the next step is to send the fingerprints to the server in order to find the corresponding behaviors saved in the database.

For the database it has been decided to use MongoDb [13], a non-relational dbms that stores data into bson documents divided into collections. The main collections that have been identified are: the one containing the behaviors, in which each document must have 1) the path to the .gpickle file of the behavior and 2) the fingerprints associated to the behavior; the one containing the semantic descriptions of the behaviors, where each document represent a single semantic tag (we could have grouped all the descriptions relative to a single graph in a single document, but in this way the queries that the server would have had to make in order to add a new description or to edit an existing one, would have been more complex); and the one containing the users registered to our tool. During the implementative process other collections have been added to support the multiple functions that are available to the plugin: a collection to store descriptions voted by each user and one to store the open sessions of logged users. We decided not to put the votes directly in the *descriptions* collection to avoid to have complex documents in it, and in this way there is no useless data returned from the database when the server queries only for the descriptions. Also, in the collection for the users we decided to store groups of users, this because, as described in session 3.6, it is possible to make a description private by tagging users or groups, and in order to avoid tagging conflicts, it is not possible to have a user and a group with the same username/name, and having them in the same collection means to use a single simple query to check if a user or a group already exists when another

potential user is registering themselves, or when a group is being created.

The details of the structure of the database and its collections are discussed in details in the Implementation chapter (Chapter 4).

The server queries for the behaviors graphs and their descriptions (the semantic tags associated to those graphs), and then returns the parsed results to the client. The results are composed in a way that it should be more convenient for the client to parse it, by coupling each behaviors to its list of descriptions, instead of returning the two separated lists and leaving to the client the task to search and retrieve the correct descriptions for each behavior returned. Also, the server needs to know the status of the user, if they are logged in or not, in order to show the correct set of descriptions (some of them may be private and available only to certain users).

3.4 Behaviors visualization

As soon as the plugin receives the complete response from the server, it parses the results, opens a new view, side by side to the assembly code view, and it visualizes a list of all the behaviors found inside the binary under analysis. An example, in which it has been analyzed a binary file that performs some operations on the Windows' registry, is shown in Figure 3.2. The complete sequence of steps the plugin makes, and the interaction between the components of our system up to this point is showed in Figure 3.3: the user invokes the plug-in from the UI or from a keyboard shortcut; the plug-in executes Disasm on the binary file under analysis and it retrieves the output with the fingerprints that are then sent to the server; the server, upon receiving the request makes a query to the database of behaviors and returns the results to the client which displays them in a dedicated view.

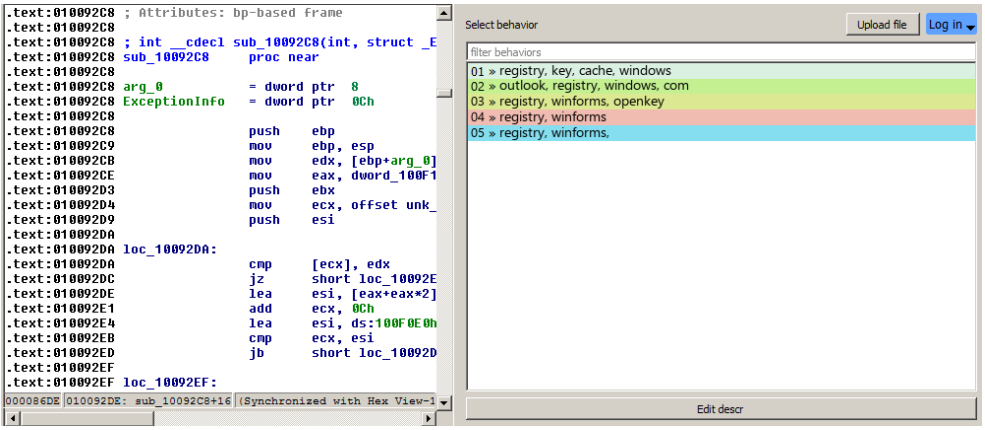


Figure 3.2: Behavior View on the right of the disassembled code

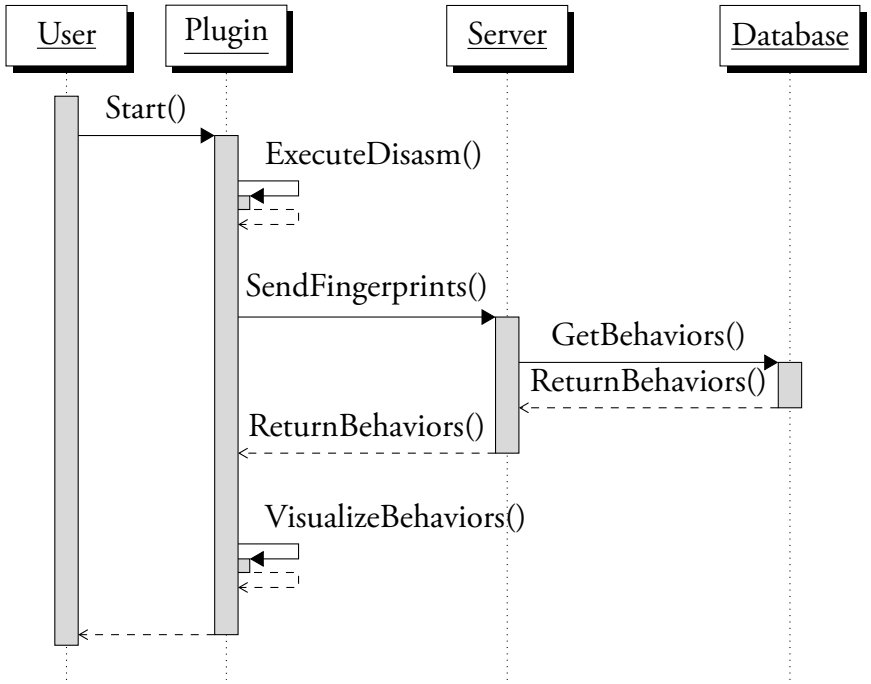


Figure 3.3: Interaction between plugin, server and database.

At this point the automatic tasks that the plugin performs since its activation are completed; the can now interact with the UI in order to fully take advantage of the advanced functions that our tool can perform: the integration with the asm view, described in Section 3.4, the possibility to see a more detailed list of descriptions for each behavior, described in Section 3.5, and other features, described in Section 3.6 that are available only for registered

users.

Once the list of behaviors has been loaded, the user can start interacting with them and with the plugin, meaning that all the further functions implemented are executed only when the user requests them. However, after the behavior has been loaded, the plugin hooks itself on the IDA view showing the assembly instructions of the analyzed binary file, intercepting mouse clicks on the lines of code. If the highlighted instruction is in a function that implements one or more behaviors found in the file, the plugin filters the list of graphs showing only those behaviors, as shown in figure 3.4.

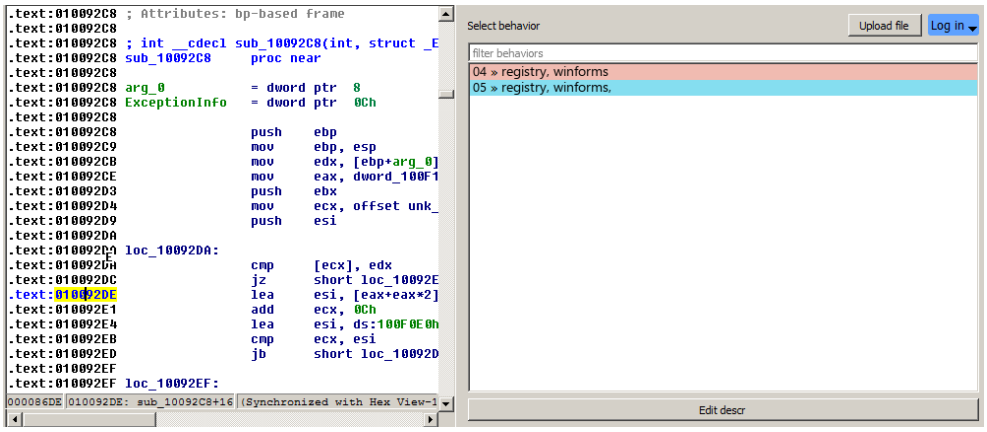


Figure 3.4: Clicking on the instruction at `.text:010092DE` the list of behaviors is filtered and it shows only those two graphs to whom that instruction belongs.

Users can see the graph of a behavior found in the analyzed binary file by double-clicking its entry in the Behavior View. In that case the plugin will open another view showing a visual representation of the graph. The graph is interactive, this means that by clicking its nodes it is possible to see every interesting attribute of such node; moreover, when a node is selected, the plugin highlights all the assembly lines in the main view that implement the behavior in question, as shown in figure 3.5.

The synchronization of the two views is useful in order to see in a more immediate way what some instructions are doing or which lines of code execute certain functions and it is supposed to be a major aid to the manual analysis process. Each behavior has a color associated that is generated using the hash of the behavior itself, in this way every behavior will have the same color every time it is encountered, not only on the same sample, but on every binary file that implements it.

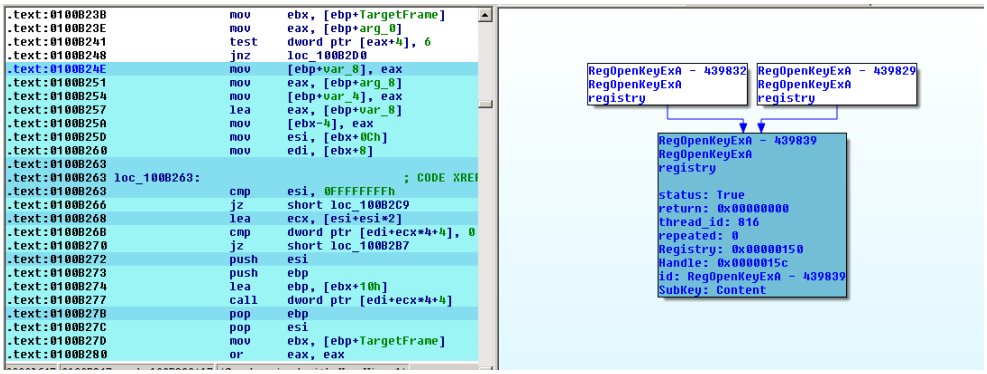


Figure 3.5: Clicking on a entry in the BehaviorsView opens the graph and highlights the instructions.

3.5 Behaviors Descriptions

In the Behaviors View (that can be seen in Figure 3.2) it is shown a list of colored entries (the colors are the same discussed in Section 3.4), one of each behavior, labeled with a number and with an overview of the behavior's most high-scored descriptions. To see the complete list of semantic descriptions the user can click on the "Description" button after having selected a behavior. At that point the plugin opens a new window, as shown in figure 3.6. We opted for a pop-up window instead of another view in order to avoid to over-load the user interface with too many opened tabs and sub-tabs.

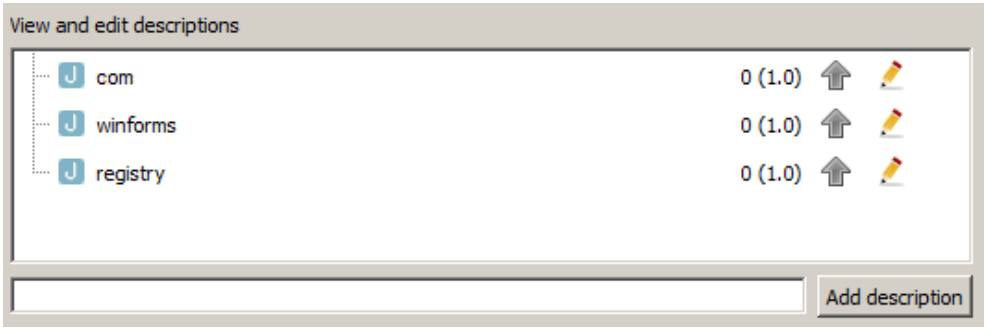


Figure 3.6: The Descriptions Window

In the Description Window it is possible to see all the relevant tags, and their edited versions, associated to the selected behavior. Also, every description has two scores: the first one it is calculated by the semantic tagging algorithm, and it is set to 0 in case a description has been manually inserted

by a user; the second one is the *user-score*, that is the number of users that consider such description fitting correctly the behavior (this feature is discussed in more details in Section 3.6). Moreover it is possible to see which are the original descriptions found for the behavior by Jackdaw and which are the ones added by human analysts, the firsts are in fact marked by the icon J. Also, registered users have available a wider set of functions, such as upvoting, editing and inserting new descriptions (see Section 3.6).

3.6 Functions for Registered Users

Since one of the goals of this thesis is to improve and correct the results obtained automatically by Jackdaw, more features have been implemented and are available to logged-in users from the Descriptions Window. If a user finds that the all the descriptions that are already available are not fitting the given behavior, they can add new descriptions themselves; or they can improve the existing descriptions editing them: in this case the new description does not substitute the old version but it is added to the list and the wrong one is marked as "edited", we adopted this choice in order to avoid user from mistakenly editing correct descriptions, deleting the right ones, and in order to maintain a history of edits. In Figure 3.7 it is possible to see how description can be edited or how a new one can be added.

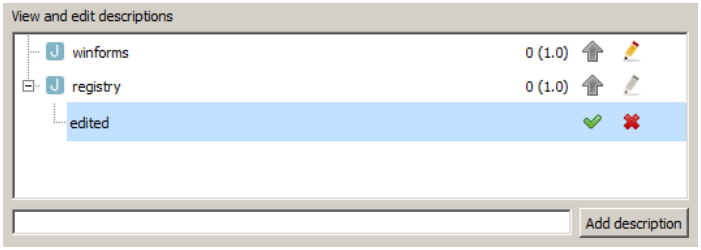


Figure 3.7: An example of how a tag can be edited. At the bottom of the form there is the inputbox from which it is possible to add completely new descriptions.

Moreover, logged-in users can upvote a tag. This means that if a user finds that a semantic description is particularly fitting the behavior, they can increase its user-score, as shown in Figure 3.8. If a user clicks on a vote button for a description they already voted for, that vote is canceled and the score is decreased by 1. This because we wanted to avoid multiple votes to a single

description from a single user and to give the users the possibility to eventually remove a vote they mistakenly gave.

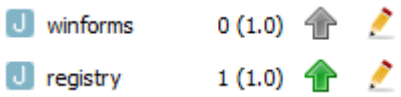


Figure 3.8: The tag "registry" is upvoted and its userscore is 1

Also, the new descriptions can be public or private. While public descriptions can be viewed by anyone, private descriptions can be seen only by those users who was tagged at the moment the description was being written and submitted. Users are tagged using the "commercial at" `@` followed by the username of the user that is meant to be tagged: `@username`. In order to simplify the tagging process, especially when the number of users is fairly large or when working in teams, we added the options of creating and managing groups of users. In this way, instead of tagging the individual users one by one, it is possible to tag the whole group at once.

Finally, it is possible that for a sample under analysis in IDA there could not be any behavior found, maybe because it presents completely new behaviors that Jackdaw haven't discovered yet in the past analyses. In this case it is possible to contribute to the behaviors database by submitting the sample to the analyses server. On the client side the user can select the file they want to upload from a simple file dialog, browsing through the file system. On the server side, once the sample is received, the backend automatically queue the analysis of the file simply using the command-line utility `submit.py`, provided with Cuckoo.

The server is responsible for checking on the correctness of the requests. If a user is trying to perform one of the previous actions without being registered, the server returns an error message. If the user is trying to insert a description that already exists for that behavior, the request is simply ignored. If the word following the "commercial at" does not correspond to a user or a group, the tag is ignored and the server considers it part of the description that is being inserted.

3.7 Login and Registration

From the Plugin Main View it is possible for the user to either Login or Register. The login allows users to access to the extra features described in

Section 3.6. The registration process consists in two steps: at first the user compiles the registration form — with username, password and email — and submit the data. When the server receives a registration request it validates the data checking whether the e-mail or the username are not already being used by other users (and in case of the username, if it is not already used as a Group Name, in order to avoid tagging conflicts), if this is the case or if the credentials inserted to login are wrong, an appropriate error message is returned, otherwise, if all data submitted is correct, it sends a confirmation email to the user who ends the process by visiting the confirmation url.

A more in-depth description of the implementative details of all the features available in our tool can be found in Implementation (Chapter 4).

Chapter 4

Implementation

In this chapter we will discuss the implementative details of our tool. In Section 4.1 we'll explain how the plugin client has been implemented using the IDA SDK and how it interfaces with IDA Pro (Subsection 4.1.1), how the server backend has been implemented and of what parts it comprises (4.1.2), how the database is structured (Subsection 4.1.3) and how all these three elements communicate with each other. Moreover, always in Section 4.1, we discuss in details how every function available to users is implemented. In Section 4.2 we explain how we improved the semantic tagging algorithm and finally, in Section 4.3 we discuss the external tools used in the implementation process.

4.1 Implementation details

4.1.1 Plugin setup

The IDA SDK offers two ways of loading the plugin on IDA startup. The first one is used returning the flag `idaapi.PLUGIN_KEEP` from the `init` method in the plugin class, it allows to keep the plugin in memory, but this is not necessary for our purpose, since we only need our tool to run when explicitly called by the user and since we do not need to hook onto the IDA processor module but only with the IDA database (of disassembled instructions). The second way of enabling an IDA plugin is the one we used: by returning the `idaapi.PLUGIN_OK` flag IDA unloads the plugin from memory after the startup, but it remembers that the plugin needs to work with the IDA database. Also, since IDA executes on startup everything in the `init` method, we could not insert the calls to the core functions of the plugin into the plugin main class,

so the method that starts the Disasming process and its following functions is bound to a keyboard shortcut and to a toolbar action.

This is how the plugin main file `Jackdaw.py` is structured:

```
#PLUGIN MAIN CLASS
class JackdawPlugin(idaapi.plugin_t):

        #MANDATORY METHODS
        def init(self):
                return idaapi.PLUGIN_OK

        def term(self):
                pass

        #THE MAIN LAUNCHER METHOD
        def launcher():
                execute_disasm()
                query_results()
                visualize_behaviors()

if __name__ == "__main__":
        create_actions()
        bind_actions_to_launcher_method()
```

In this way, only the user can activate the core functions of the plugin, by executing one of the actions, while the only thing that IDA executes automatically is the binding of such actions.

Note that the possibility to add an action to the IDA toolbar has been added only from IDA Pro 6.7[1], this means that the only way to activate the plugin in older version of the program is to use the entry from the Plugins menu or from a keyboard shortcut.

[1] http://www.hexblog.com/?p=886

The overall directory structure of the plugin is the following:

```
IDA Plugins folder/
        ├── Jackdaw.py
        └── jdaw/
                ├── data/
                ├── disasm_x86/
                ├── disasm_x64/
                ├── disasm.py
                ├── connector.py
                ├── view.py
                └── utils.py
```

`Jackdaw.py` is the file that IDA needs in order to load the plugin and to bind the actions on startup, while the folder `jdaw` contains the proper plugin core files, the `data` subfolder containing the icons used for the user interface, and the two versions (32bit and 64bit) of Disasm.

4.1.2 Server setup

We implemented the server using the `SocketServer` Python library. In particular we used the `TCPServer` class and, in order to serve multiple requests in asynchronous mode, the `ForkingMixIn` class; also, the TCP socked is wrapped with ssl protocol. The server listens on a predefined port (the default port is 5000, it is possible to define an alternative port on server startup) for requests from the plugin clients.

When a new user registers to the service, the server sends them a confirmation email: this means that the server needs access to a smtp server. It is possible to specify url and port of the smtp server and the name and password of the mailbox used to send the confirmation emails, on server startup.

The server needs also access to a second port used to show a html page of registration confirmation when a new user clicks on the confirmation links in the confirmation email.

4.1.3 Database

As DBMS we used a MongoDB [13] instance with 5 collections. The first one is the *behaviors* collections, this is its scheme:

```
{
  _id,
  graph,
  fingerprints,
}
```

in which `_id` is the unique identifier assigned from MongoDB to the document, `graph` is the path to the `.gpickle` file on disk in which the graph is stored, and `fingerprints` is a list of all the fingerprints associated to the behavior. This collection has also an index on the fingerprints that is necessary to speed up the queries.

The descriptions are stored in a separate collection and they have the following structure:

```
{
  _id,
  tag,
  author,
  score,
  user_score,
  graph_id,
  edit,
  (private,)
}
```

in which `tag` is the description itself, `author` is the username of the user who inserted the description or "Jackdaw" if the description was found using the semantic algorithm, the `score` field is the score computed by the semantic algorithm (if the description has been inserted by a user the value of this field is 0), `user_score` is the total upvotes that the description has from the users, `graph_id` is the `_id` of the graph in the behavior collection to which the description in object is associated, the `edit` field indicates whether a description is an edit to another one or not: in case it is, the field contains the description that has been edited. Finally some of the descriptions may have the field `private` that is a list of usernames of those users that are the only ones who can see such descriptions. In order to improve performances on the queries, this collection has also an index on the `graph_id` field.

A separate collection has been used to keep track of the votes of the single users. This was necessary in order to avoid that users could vote multiple times on a single description, and to eventually give the users the possibility to remove the vote they had given to a description. The *votes* collection is empty at the beginning and a new document is created for a user the first

time such user upvotes a description.

The structure of the documents in this collection is the following:

```
{
  _id,
  user,
  graph_id: [descriptions]
}
```

This document contains `user` that is the username of the user, and a list of `graph_ids` with the descriptions that has been upvoted by the user for that graph. Every time a user votes for a description for a new behavior (a behavior that is not already in the votes document), a new `graph_id: [description]` pair is added to the document, otherwise, if the graph_id is already present, only its list of descriptions is updated.

Another collection used in our database is necessary to store the registered users and also, as we discussed in Approach (Chapter 3), the groups of users. This is the structure of the documents in this collection:

```
{
  _id,
  username,
  type,
  (pwd,
  email,)
  (users,
  admin,)
}
```

where `username` is the user's username or the name of the group, `type` indicates whether it is a group or a user; the fields `pwd` and `email` are present only if the object is a user and they contains the salted and hashed password and the user email; the fields `users` and `admin` are present only if the object is a group and they contain the list of users in the group and the group admin.

Finally another collection has been used in order to keep track of the open sessions, that is the users that are logged in the application. This is the following structure of the documents of such collection:

```
{
  _id,
  session-id,
  username,
  date,
  ip,
}
```

`session-id` is a random generated string encoded in base64 that is created at the moment the user logs in (see Session 4.1.16), the `date` field contains the time the user logged in with `YYYY:MM:DD hh:mm:ss` format, and `ip` is the ip address from which the user has logged in.

4.1.4 Communication protocol

As we already stated in Section 4.1.2, the main communication protocol used between server and client is TCP. However we developed a communication protocol that works over the TCP channel in order to parse and send messages between client and server. Messages are packed as format strings, using the `struct` Python library. Every message has an header that contains the full message length and the message code. The header is the first thing that gets unpacked once a new message is received, either by the client or by the server, and it is used as an aid for the parsing process and for an additional integrity check. The message code identifies the kind of query (if it has been sent by a plugin client) or the kind of response (if it has been sent by the server to a client), and it determines how the rest of the message needs to be unpacked.

4.1.5 Disasm and fingerprints extraction

The first step that get executed when the plugin is activated is the extraction of the fingerprints from the analyzed binary file, using Disasm.

Before executing Disasm, however, the plugin needs to retrieve from IDA the path to the binary file that is currently being analyzed, using the `idc.GetInputFilePath()` method from the IDAPython `idc` library.

Disasm is executed by the plugin using the `subprocess` library and its output is read and parsed into two dictionaries: the first one has the fingerprints as keys and the offsets as values; the second dictionary is the opposite: it has the offsets as keys and the fingerprints that have those offsets as values.

Disasm outputs a string with the following format:

```
fingerprint1,offsetA,offsetB,offsetC,...
fingerprint2,offsetA,offsetB,offsetE,...
⋮
```

While the two dictionaries have the following structure:

```
dic1 = {
        fingerprint1: [offsetA, offsetB, offsetC, ...],
        fingerprint2: [offsetA, offsetD, offsetE, ...],
```

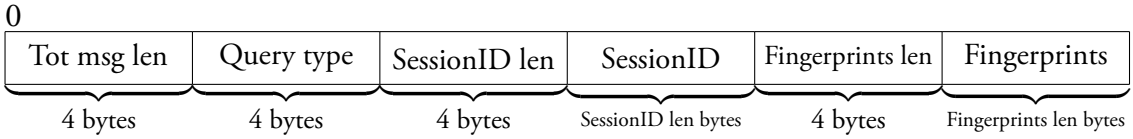
```
        ⋮
        }

dic2 = {
        offsetA: [fingerprint1, fingerprint2, ...],
        ⋮
        }
```

The choice of using this couple of dictionaries has been made in order to improve the performances in other functions that the plugin has to perform later on.

4.1.6 Sending the fingerprints to the server and behavior retrieval

The next step is to send the fingerprints to the server and retrieve the associated behaviors. The plugin organizes the fingerprints generated by Disasm into a comma separated list, and that is the main part of the message that is going to be sent. Since some of the description associated to the behaviors may be private, it is necessary to include in the message the session-id of the user, if they are logged in, or `None` if they are not. In this way the server will know the status of the user, and by using the session-id, if it is not `None`, it can retrieve the user's username and select the correct descriptions that have to be returned to the client. The final message can then be sent to the server. This is the structure of the message:



The message is then sent over the TCP channel and as soon as the server receives it it starts unpacking it. The server reads and unpacks the first 8 bytes of the message, the first 4 bytes are the length of the total message and the server uses them for an integrity check, the second 4 bytes are the query code: they encode the type of the message that is being received. The code for the query for retrieving the behaviors and their descriptions is `01`.

Now the server knows the query code requested by the client and it can execute the respective function. The first step at this point is to unpack the rest of the message, if the structure of the message is not what expected given

the query code, the server returns an error message. The elements expected here are the session-id length, the session-id, the length of the comma separated sequence of fingerprints and the sequence itself. The two lengths are necessary in order to unpack the session-id and the fingerprints, because it is not possible to know their lengths a priori.

Once the session-id and the fingerprints are unpacked, the server parses the sequence of fingerprints, that is in string format, in a Python list, and then it queries MongoDB for the graphs in the *behaviors* collection using the index on the fingerprints:

```
fingerprints_list = fingerprints_seq.split(",")
graphs = behaviors_collection.find(
        {"fingerprints": {"$elemMatch":
                        {"$in": fingerprints_list}
                        }}).hint(["fingerprints", 1])
```

Once all the graphs are retrieved, the server collects all their `_id`s in a list `graph_ids` and another function `query_for_descriptions()` uses it for querying the respective descriptions. This function is wrapped in a Python decorator that checks whether the user is logged in or not, and if they are it retrieves the username using the session-id from the *sessions* collection in MongoDB; this because some of the descriptions associated to some of the behaviors may be private, so the MongoDB query must include this information: the server queries all those descriptions that are not private or that are private but the user's username is in the field `private` (see 4.1.3):

```
descriptions = descriptions_collection.find(
        {"$and":[
                {"graph_id": {"$in": graph_ids}},
                {"$or": [
                        {"private": {"$exists": False}},
                        {"private": username}
                        ]
                }
                ]}).hint(["graph_id",1])
```

Now the server has all the documents containing the descriptions, this documents need now to be parsed to remove useless information (in order to reduce the size of the message that the server has to send back to the client) and it needs to include a flag `upvoted` on those descriptions that has been upvoted by the user. To do so the server queries for the user's document in the `votes` collection, and then it iterates on all the description, removing the field `"author"` if it is not "Jackdaw" or the user themselves, adding the field `voted`

and grouping those description belonging to the same graph. At this point the server has a Python list containing all the documents of the behaviors graphs and a dictionary in which the keys are the `graph_id`s and the values are the descriptions, now it has to load the graphs from their files on disk and then it has to associate the behaviors to their respective descriptions. Note that an alternative approach would have been to query the descriptions for every graph one by one, in order to have them already associated in couples (graph, descriptions), but this would have significantly increased the number of accesses to the database. The server iterates on the list of graphs, loads the single graph from disk opening its .gpickle file, retrieves its descriptions from the dictionary and parse them in json format. While the server iterates on the graphs it packs both the behaviors and their descriptions, and it starts to build the string (the message) that has to be returned to the client.

```
1: message ← ""
2: procedure GenerateReturnMessage(graphs, descriptions)
3:     for all graph in graphs do:
4:         g ← LoadGraphFromDisk(graph)
5:         d ← Descriptions(graph.graph_id)        ▷ Get description from descriptions dictionary
6:         packed_graph ← Pack(g)
7:         packed_description ← Pack(d)
8:         packed_couple ← Concatenate(packed_graph, packed_description)
9:         message ← Concatenate(message, packed_couple)
10:    end for
11: end procedure
12: return message
```

At this point the server adds the header to the main message. The header contains the length of the total message, the code of the response (in this way the client knows how to unpack it), and, only for this kind of response, the total number *n* of couples graph-descriptions that are present in the message (this makes the unpacking process on the client much easier). After the header is added, the message is sent on the TSL/SSL-wrapped TCP channel, back to the plugin client that is waiting for the response.

4.1.7 Behaviors visualization

The unpacking process on the clients works exactly in the same way as on the server, as described in Section 4.1.6: the plugin unpacks the header first,

reading the total length of the message and performing an integrity check, the response code, and, with the proper response code, the number of couples graph-descriptions.

If the response code is an error code the plugin shows the relative error message, otherwise it starts unpacking the rest of the message. Knowing in advance how many *n* pairs graph-descriptions there are makes the unpacking process a simple matter of iterating *n* times on the remaining part of the received message, and unpacking one pair at each cycle. In this passage the plugins builds also a list of all the fingerprints found in the returned behaviors, to have this list is useful because before opening the new *Behavior View* the plugin generates a list containing the intersections of the fingerprints found by Disasm and those present in the behaviors returned from the server:

```
intersection_list = [fingerprint for fingerprint in
    behaviors_fingerprints if fingerprint in disasm_fingerprints]
```

After all the pairs graph-descriptions are unpacked, the plugin opens a new IDA view. This new view extends the class `idaapi.PluginForm` made available by IDAPython/IDA SDK. The use of this class extension is necessary in order to fully integrate the results into IDA. The behavior view is opened side by side to the IDA Main View (called "IDA View-A"), this is done with the `idaapi.set_doc_pos` method and passing it the correct flag accordingly to the placement we want:

```
idaapi.set_doc_pos("Behaviors view", "IDA View-A",
        idaapi.DP_RIGHT)
```

The *Behaviors View* created by the plugin contains the *user button*, a search box, a *Qt* `ListWidget`, and a button that visualizes the descriptions of a selected behavior. The *user button* is used to login or to register, or, if the user is logged in, it shows the user username and it opens a menu from which it possible to access to the *Groups Management* view or to log out. The search box it is useful to filter behaviors by name, by api, by descriptions or other attributes.

The *Qt* `ListWidget` is populated with all the behaviors found for the analyzed binary file: each behavior is labeled by a number and by a list of its most high-scored descriptions, in order for the user to have an overview of the behaviors descriptions without opening the *descriptions* window.

Also, each behavior is colored with a unique color generated before the view is opened. The colors are generated by hashing the graph and by taking 3 bytes of the obtained hash, that is 6 hex digits or the encoding of a color in

hex format. The color is then brightened if it is too dark, this step is necessary in order to keep everything readable. Remembering that a color in hex format is a triplet RGB in which every component is a number ranging from 0 to ff, or 255 in decimal, we used the Equation 4.1 to determine if a color was too dark to maintain readability of information computing the *Luma (Y')*, a value that represents the perceived brightness of a color[2]. For our purposes, we found that considering colors as "bright" when their Luma is higher than a threshold=125 leads generally to good readability. In case of a "dark" color its values of R, G and B are increased until its Luma value is equal or higher than our threshold.

$$Y' = 0.299 * R + 0.587 * G + 0.114 * B \tag{4.1}$$

Upon creating the *Behaviors* view, the plugin hooks itself onto the IDA-Main View, this is done by extending the IDAPython class `idaapi.IDAViewWrapper`. This hooking is necessary to have a more complete integration between the Jackdaw results and the code of the binary the user is analyzing: the plugin intercepts all the mouse clicks on the IDA Main View, in particular on the lines of assembly code, and, if the selected lines belongs to some of the behaviors, the plugin updates the list of behaviors with the interesting ones. In order to do this we exploit the fact that IDA generates the flow chart of the analyzed binary.

In details, when the user clicks on a line of code, the plugin retrieves the flow chart of the binary from IDA, then it searches in the graph for the block in which there is the selected line of code and once it finds it, the plugin retrieves the entry point of the block. The entry point is the instruction whose offset is one of those to which Disasm may have associated some fingerprints, so the plugin retrieves the fingerprints associated to that entry point from the dictionary `{offset: list_of_fingerprints}` described in Section 4.1.5. However not all the fingerprints found by Disasm may be associated to a behavior returned to the server, so in order to select only the interesting fingerprints, the plugins checks for every one of them that it is also in the `intersection_list` that has been generated previously.

4.1.8 Graph visualization

Since each behavior is a graph whose nodes are the called APIs characterizing such behavior, it is useful to see such graph. Every behavior entry in the

[2]http://www.poynton.com/ColorFAQ.html

`QListWidget` on the *Behaviors* view is linked to a *double click action*: if the user double clicks on a selected behavior the plugin will open another view showing the generated graph. This *Graph View* is stacked in a new tab over the Behaviors view tab, and it is created as an extension of the IDAPython `GraphViewer` class, this makes the integration into IDA and the generation of the graph much easier, however it does not allow much control over the graph itself.

In the generated graph every node represents a called API, and the text in every node shows the name of the API and its type (registry, file, etc.). To see more attributes of an API call it is possible to click on its node: the selected node is updated with all the attributes belonging to its API call. Also, at this point, the plugin makes use of the hooking on the IDA Main View once more: when the user clicks on a node the plugins highlights every line of assembly code in the IDA Main View belonging to that behavior. The lines that have to be colored are found using the IDA flow graph. For every single one of the fingerprints in the selected behavior, the plugin retrieve its offsets from the {`fingerprint: list_of_offsets`} dictionary described in Section 4.1.5, ignoring those fingerprints that are not in the dictionary, then, for every offset, it searches in the flow graph for the basic block containing it, and every line in that block is colored.

4.1.9 Descriptions visualization

To see all the descriptions for a given behavior and to interact with them, we implemented a *Descriptions Window* that can be accessed by a button from the *Behaviors View* or by left-clicking on the behavior graph and selecting the option in the contextual menu.

The Descriptions Window presents a *Qt* `QTreeWidget` containing all the descriptions, and an input box used to insert new ones. The main items in the TreeWidget are all the actual descriptions, whether they are original or edits to other ones. The children to the main items are their edits. The tree never exceeds this two levels: when a description that is an edit to another one is edited in its turn, the new modification is inserted as a main item in the TreeWidget, and as child to the main item of the edited description. We adopted this choice to avoid a deep level of nesting of edits. Every item in the tree widget is a custom `QTreeWidgetItem`: besides containing the text of the description it also has an icon that indicates if the description has been found by the tagging algorithm, two buttons to edit the description and to

upvote it, and an icon that indicates if the description is private or not.

Pressing the edit button adds a temporary child `QTreeWidgetItem` to the selected description. This custom item contains an inputbox, in which the user can insert their edit, and two buttons to confirm or discard the edit. If the user confirms the edit is sent to the server and, when the server confirms the insertion of the corrected description, the temporary child is removed and the correction is inserted as explained before.

4.1.10 Description addition

From the *Descriptions Window* it is possible to insert a new description for a given behavior. The text of the new description must be typed in the `QLineEdit` widget and such description is inserted when the user press the insert button. At this point the plugin starts building the message to send to the server: the message includes the session-id of the user, the id of the behavior, and the text of the description. The structure of the message is exactly the same as the one described before: the header contains the total length and the message code (in this case `02`), and every other information is preceded by its length, in order for the server to correctly unpack the message.

Once the server receives the message from the client over the TCP channel, it starts unpacking it as already described: it first reads the header, with the length and the message code, and then, based on the code, calls the `ins_description` method, which unpacks the rest of the message, reading the session-id, the id of the behavior and the text of the new description.

The session-id is used to check that the user is logged in. Since the possibility to add a new description is given only to those users who are registered into the application, if the session-id is empty or if it is wrong (it does not match to any of the sessions that are actually in the *sessions* collection in MongoDB) the server returns an error message to the client. After this first check, the server parses the message text in order to read some eventual tagged user. Since the tagging is done by typing the *commercial at* followed by a username or a groupname, we used a regular expression to retrieve all the instances of this kind of construct and to build a list of them:

```
possible_users = [user for user in re.findall(r"[@]\S*",
        description_text)]
```

Now the server has a list of possible usernames or groupnames. To verify if they are actual users or groups the server needs to query the *users* collection in MongoDB:

```
actual_users_or_groups = users_collection.find(
        {"username": {"$in": possible_users}})
```

In this way, of all the strings preceded by a @ sign we now only have those who are registered users or groups (remember that the users and the groups share the same collection in the database).

At this point the server creates the new object to add to the database as a new description by setting the its text and the id of the behavior, and initializing both the score and the user_score to 0. Also, if `actual_users_or_groups` is not empty, the server marks the new description as private, adding the tagged users to the `private` field and the tagged groups to the `groups` field. The description is then added to the database by means of a query to the *descriptions* collection. If the addition is successfully completed, a confirmation message is packed and returned to the client that updates the list of descriptions it has for the behavior, and adds the new description to the *QTreeWidget* in the *Descriptions Window*.

4.1.11 Description editing

The mechanism for the editing of an existing description is mostly like the one described in Section 4.1.10 for the insertion of a new description, but with minor differences.
In this case the editing is performed by clicking on the *edit button* of the description the user want to modify. A new temporary custom child `QTreeWidgetItem` is added to the description, and the user can insert the text of the modification in the `QLineEdit` that the child includes. The modification is confirmed by clicking on the *confirm button* on that same custom child item, otherwise the user can discard the modification pressing the *cancel button*.
If the edit is confirmed, the plugin creates the message to send to the server. This message includes the session-id, the old description and the new one, and then the plugin adds the header with the total length and the query code (in this case `03`).

Upon receiving the message, the server unpacks it as previously described and, since the description editing is a function available only to registered users, if the session-id is `None` or if it does not exists in the *sessions collection*, an error message is returned to the plugin client. Regarding the *private* attribute, the edits to an existing description inherit the same private status of their parent description, that is that the edits to a private description are private and only the users tagged in such description can see it; moreover, the edits

to a public description must be public too and eventual tags are ignored and considered part of the description. This is done by querying the existing description and retrieving its *private* status before inserting its edited version. At this point the server can add the edit as if it was a new description, with the scores set to 0, the id of the behavior and the author of the edit, but with the difference that the `edit` field contains the old description.

If the query used to insert the edited description into the *descriptions* collection is successful, the server returns the confirmation to the plugin client, which updates the descriptions it has with the new one, and it shows it in the *Descriptions Window* as a parent item and a child item of the edited description.

4.1.12 Description voting

In order to for the users to help colleagues in the reverse engineer process, it is possible for them to vote those descriptions that better fit a particular behavior. The *upvote* is done by clicking on the respective vote button that every parent item in the *QTreeWidget* in the *Descriptions Window* has.

When a user votes for a description, the plugin sends the vote to the server in the form a message with the code `07` and in which the plugin packs the session-id, the description that is being upvoted and the id of the behavior that description belongs to.

As usual the server unpacks the message and makes sure that the session-id is not `None` and that it exists in the *sessions* collection. At this point, if the user is correctly logged in, the server updates the *user-score* of the upvoted description.

First, the server retrieves the votes of the user from the *votes* collection, then it makes sure that the description that has been upvoted is not already in the retrieved votes; if this is the case the server consider the action as an "undo vote": the vote is canceled and the *user-score* of the description is decreased by 1. Otherwise, if the user has never upvoted the description before, the *user-score* is increased by 1.

If the upvote process is successful, the server returns a message to the client with the updated value of the user-score; the clients then updates the user-score in the *Descriptions Window* and in the data it has in memory.
As usual, if the upvote process is not successful, the server returns an error message that the client shows to the user with a `idc.Warning`.

4.1.13 File Upload

As we discussed in Section 3.6 in Approach, it is possible to submit a file to be analyzed by Jackdaw. Clicking on the Upload Button in the main view of our plugin opens new window that presents the option to open a file dialog, in which it is possible to select a file on the file system, and an upload button. After having selected the file to submit with the file dialog, the user can press the upload button and at this point the plugin opens a session to the remote xmlrpclib server and sends the file. Once the server receives the sample it simply stores it and submits it to Cuckoo's analysis queue using `submit.py`.

4.1.14 Groups Management

Pressing on the user button on the main plugin view, the *Behaviors View*, shows a menu with the *Groups Management* entry, selecting it opens a new *Groups Window* from which it is possible to create new groups, and to add or remove users from existing groups.

Every group management-related query that is sent to the server has two query codes. The first one is `08`, and it indicates that the request is relative to a group; the second one is the query sub-code, and indicates which action must be performed on the group: `01` gets all groups of a user; `02` adds a user to a group; `03` removes a user from a group; `04` creates a new group. When the server receives a message with code `08` it calls the `manage_groups_requests` method, a handler of the groups-related queries, which parses the rest of the message, reads the query sub-code, and calls the respective function.

When the *Groups Management* windows is opened, the plugin asks the server for the groups in which the user is in, so it sends on the TCP channel a message with the session-id, the sub-code `01`, and, in the header, the total length and the query code `08`.

The server unpacks the message and uses the session-id to check if the user is logged-in, and, if that is the case, to query the *users* collection and get all the groups of which the user is a member.
Once all the groups are retrieved, the server parses them in a dictionary with the following structure:

```
{group_name: {"users": [list_of_users], "admin": group_admin
              }, }
```

Then, the server dumps the dictionary in .json format and packs it with the Python `struct` library, it adds the header and sends the message back

to the plugin, that will parse the response and load the .json string into a dictionary.

The *Groups Management* window is an extension of the `idaapi.PluginForm` class and it is composed by an inputbox from which it is possible to add new groups, and a *Qt* `QTreeWidget` that contains custom items. The main items in the tree widget are the groups of which the user is a member, while the children items of each of the groups are the other members of such groups.

The *group* items have a button from which it is possible to add new users, while the *user* items have a button from which it is possible to remove the respective user from the group. These two operations are available only to the admin user, that is the user who created the group in the first place.

When the users creates a new group by typing the name in the input box in the *Groups Management* window and pressing the *New Group* button, a message is sent to the server with the sub-code `04`, the session-id and the group name. After the server receives it, and after it has checked that the session-id exists in the *sessions* collection, it must check that the group name is not already used by another user or group with a simple query to the *users* collection. If the name is available the server creates the group by adding a new document in the *users* collection with the following structure:

```
{
        "username": name_of_the_group,
        "type": "group",
        "users: [user_that_is_creating_the_group],
        "admin": user_that_is_creating_the_group,
}
```

The `users` contains only one user (the admin that is creating the group) and it is up the admin to add other users to the group.

If the group creation is successful, an acknowledgement is returned to the plugin that shows the new group into the *Group Management* window.

Pressing the *Add User* button adds a temporary child item, of the same type of the one described in Session 4.1.11, to the group item in the tree view. The admin user types the username of the user they want to add to the group, and then they press the confirm button. The plugin packs a message with the `08` code in the header, the `02` sub-code, the session-id, the username of the new member and the group name. As usual every string part of the message is preceded by its length in order for the server to correctly unpack the message. The server makes sure of the correctness of the session-id, and then it checks first that the group and the new member in the message exist as group and

user respectively in the *users* collection, and then if the session-id belongs to the admin of the group. If this series of checks is successful the new member is added to the group by a simple update query to the *users* collection:

```
users_collection.update(
        {"$and": [ {"username": group}, {"type": "group"}]},
        {"$push":{"users": username}})
```

At this point the server must updates all the descriptions in which the group has been tagged by adding the new member to the list of users of the `private` field.

```
descriptions_collection.update(
        {"groups": group},
        {"$push":{"private": username}}
)
```

If the updates are successful, the server returns a success message and the plugin removes the temporary child from the group item in the tree widget and adds a new user item.

The removal of a user from a group works exactly in the same way. The only differences are that the query sub-code included in the message is `03` and that the update queries to the *users* and *descriptions* collections have the command `$pull` instead of `$push`.

4.1.15 Registration

The registration process is not mandatory for a user, but it is necessary in order to use all the functions available in the plugin. The registration form is accessible from the main plugin view, the *Behaviors View*, from the *user button*, that will show the text "Log in". To register, the user must insert the desired username, the email, the password and the password confirmation. The username and the email fields are validated by two masks on the respective *QLineEdit*:

```
rx_user = QtCore.QRegExp("^[A-Za-z0-9]+(?:[_-][A-Za-z0
    -9]+)*$")
rx_mail = QtCore.QRegExp("^[a-zA-Z0-9_.+-]+@[a-zA-Z0
    -9-]+\.[a-zA-Z0-9-.]+$")
```

This means that the username must only contain alphanumeric characters, the underscore and the minus characters. The regex on the mail field makes sure that the email address has the correct format `local-part@domain-part`. If all the fields are correctly validated, the plugin packs and sends a message to

the server with the code `06`. The message contains the username, the password encoded in base64, and the email address. The password is not hashed client-side, but the message is sent on the SSL-wrapped TCP channel.

When the server receives the message, it unpacks it, it decodes the base64 version of the password, and it performs the same validations the plugin did on the username and the email. Moreover it checks if the username is already taken or not, or if the email address is already associated to another user and, if these are the cases, an appropriate error message is returned to the plugin and showed to the user.

If all the validations checks succeed, the password is salted and hashed with

```
hashed_password = bcrypt.hashpw(password, bcrypt.gensalt(14))
```

Then, the server generates a token of `N_BYTES`=128 with the `M2Crypto` library or, if *M2Crypto* is not available, with the `OpenSSL` library:

```
try:
        from M2Crypto.m2 import rand_bytes as randbytes
except:
        from OpenSSL.rand import bytes as randbytes

token = base64.b64encode(randbytes(N_BYTES))
```

The user then is added to the database in the *users* collection and its document has a `pending` field containing the token.

The token is used to confirm the registration: an email is sent to the address the user specified at the registration step, the servers uses an external *smtp* server for this step. The address of the smtp in `ip:port` format defaults to the Google mails servers, `smtp.gmail.com:587`, while the name of the mailbox and its password must be specified on the server startup.

The email contains a link with the format `domain.com/token=randomstring` that the user can use to confirm the registration. The server listens on a https port with a `ConfirmationHandler` class that we built by extending the `BaseHTTPServer.BaseHTTPRequestHandler`. The handler parses the parameters in the url, getting the `token`. Every other parameter is ignored, and if the token is not in the parameter list or if the token does not match any pending users, an error is showed in the html page.

Once the server has the token it queries the *users* collection to retrieve the respective user and to updates it removing the `pending` field from it.

Every user with a `pending` field is considered not registered and they cannot login or use the advanced functions of the plugin.

4.1.16 Log in

The login form, like the registration form, is accessible by clicking on the *user button* on the *Behaviors View*.

To login, the user must insert the username and the password. The password is encoded in base64 and then the plugin starts preparing the message to send to the server. The message has the code `04` and contains the username and the password. The server unpacks the message and it checks if the decoded `inserted_password` and the password in the *users* collection (`hashed_password`) match for the given username. The comparison is made with:

```
hmac.compare_digest(bcrypt.hashpw(inserted_password,
    hashed_password), hashed_password)
```

If the comparison is successful the server generates a random 128 bytes string, in the same way as described in 4.1.15. This string is encoded in base64 and used as a `session-id`. The session is added to the *sessions* collection and the server generates a cookie that is returned to the client. The cookie contains the username, the session-id, and the login time.

Once the plugin client receives the message from the server and unpacks it, it saves the cookie in a `Jackdaw` sub-folders in `Applications Data` directory, and it is used to retrieve the session-id every time the plugin makes a request to the server.

Once the user has logged in, the *user button* on the *Behaviors View* shows the username, and the login/registration function in the menu is replaced by the *Log Out* action.
The Log Out function sends a message to the server with the code `05` and the *session-id*. The server unpacks the message as usual and checks if the session-id matches a logged in user in the *sessions* collection, and if it does, the server deletes the correspondent document. If this step is successful an acknowledge message is sent back to the plugin, which, at this point, deletes the cookie that has been saved on login.

4.2 Semantic tagging

The semantic tagging algorithm has been revisited and slightly improved.
The first version of the algorithm searched Stackoverflow and then it scored the posts it found based on three sets: *interesting tags*, *trifling tags* and a *blacklist set*.

The score of a posts is increased for everyone of its tags that is in the Interesting tags set, and it is decreased for everyone of its tags that is in the trifling tags set. Finally the blacklist set is used to remove all those tags that do not describe the behavior.
The downside of this method is that given the large number of Stackoverflow posts to search in, the number of posts returned from the initial query can be large, leading to a large number of tags remaining after the blacklisting.
The first obvious step to improve the algorithm was to improve the three lists by adding more trifling and blacklist tags. Second, since the stackoverflow dump is loaded in a MongoDB collection we thought of taking advantage of the queries capabilities of MongoDB in order to reduce the number of documents found from the search query. We introduced a second blacklist that is used in the initial search query to MongoDB: with a single query we search for all those documents that contains the data from the behavior we need to describe (API names, registry keys, parameters, etc) but that do not contains any of the words in the new blacklist. In this way the number of documents, on which is applied the scoring function, is way smaller.

4.3 External Tools

4.3.1 Disasm

Disasm [9] is the utility used to extract the fingerprints from binary files.
Since Disasm was meant to be compiled, and used, on Linux, while the IDA plugin has been developed on Windows, despite the fact that the plugin itself should work on every operating system that supports IDA, we needed to recompile Disasm using Cygwin, a Linux-like environment. Disasm has been compiled on both the architectures available of Windows 7, resulting in two version of Disasm: 64bit and 32bit. The plugin recognize the architecture of the OS in which it is being executed and it automatically uses the corresponding version of Disasm.

4.3.2 QT and Pyside

IDA User Interface is built using the cross-platform QT framework [15], and plugins built for IDA can take advantage of this by using the same framework, whose *dll*s are shipped with IDA itself. Since the Jackdaw plugin is written in Python it was necessary to use a binder library to use *Qt* and the one used by IDAPython and the one used to develop the user interface of the plugin is

Pyside [16]. However, from IDA 6.9 the binding library used by IDAPython is PyQt, that is most updated and compatible with the latest version of the QT framework. This means that in order to use the plugin with the most recent versions of IDA it is necessary to add the new library PyQt to its imports.

4.3.3 MongoDB

The DBMS used is MongoDB. MongoDB stores json-like objects, called bson, with dynamic schemas in collections; since it does not use a traditional table-based structure it is considered a NoSQL database. MongoDB offers a python driver that is necessary to interface our plugin with the database itself. Every query to MongoDB returns a *cursor* object, that is defined by the MongoDB documentation as a pointer to a result set of a query. However, most of the times the cursor is converted by the server in other data structures, for instance, a convenient way is to convert the cursor into a python list: `list_results = list(mongo_cursor)`.

Chapter 5

Performance tests

We performed some tests in order to measure the times taken by our tool for visualizing the behaviors of a file in IDA Pro. We expect that for larger files, which are more likely to implement more behaviors, the time taken to visualize the results is longer than for smaller samples, measured from the instant in which the user clicks on the the toolbar button to start the process.

5.1 Testbed

The performances are measured using IDA Pro 6.7 installed in a VirtualBox[1] virtual machine to which we dedicated 2 cores and 1600MB of ram. We analysed a total of 194 files with various sizes, ranging from about 40kB to about 11MB. The samples have been downloaded from VirusTotal[2] and we selected only Windows PE (Portable Executable) files detected as malicious by more than 15 source. We took three timestamps along the whole automatic workflow, as shown in Figure 5.1: the first one at the end of the disasming process, as soon as the fingerprints are sent to the server; the second time is measured on the server side, starting from when a request with the fingerprints is received to when the results with the behaviors found in the database are sent back to the client; the third interval measures the time taken by the plugin to parse the results received from the server and to visualize them in the dedicated view in IDA Pro.

Moreover we recorded the number of behaviors retrieved for every binary file, its size in term of nodes and in term of attributes. Also, since we needed to measure the only performances of our tool, we made sure to wait for the

[1] https://www.virtualbox.org/
[2] https://www.virustotal.com/

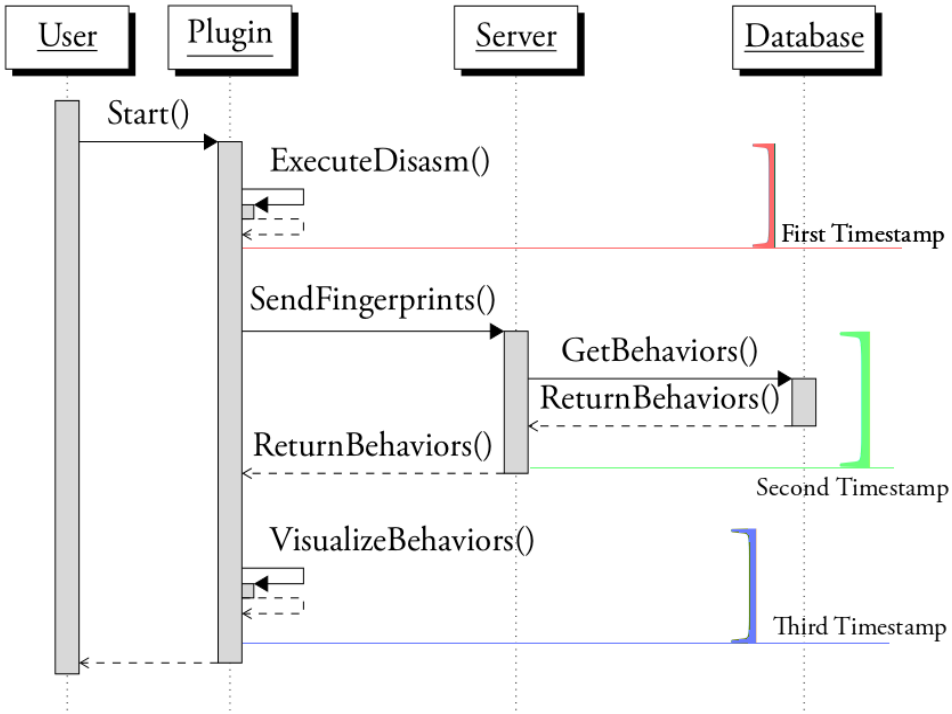


Figure 5.1: Timestamps taken for the tests

IDA's initial auto-analysis to be completed before clicking the button to start the plugin, in this way it was not slowed down and the timings we took were not influenced by other IDA functions.

5.2 Results

We can start analyzing the results by giving an overview of the total times taken by the whole process in function of the size of the analyses files, the graph obtained is shown in figure 5.2

As we can see the general trend confirms what we expected: the larger the files the longer the process would take. However we can see that there is a lot of noise, especially for larger files. This can be explained by making a consideration first: as we can see in figure 5.3 the step that takes the longer to complete is the parsing and visualization of data on the plugin. On very small files (less then 100kB) it takes on average about the 35% of the whole process (and, considering that for such small files the total time taken is short, we can consider the parsing time as negligible as well), while on larger files the parsing

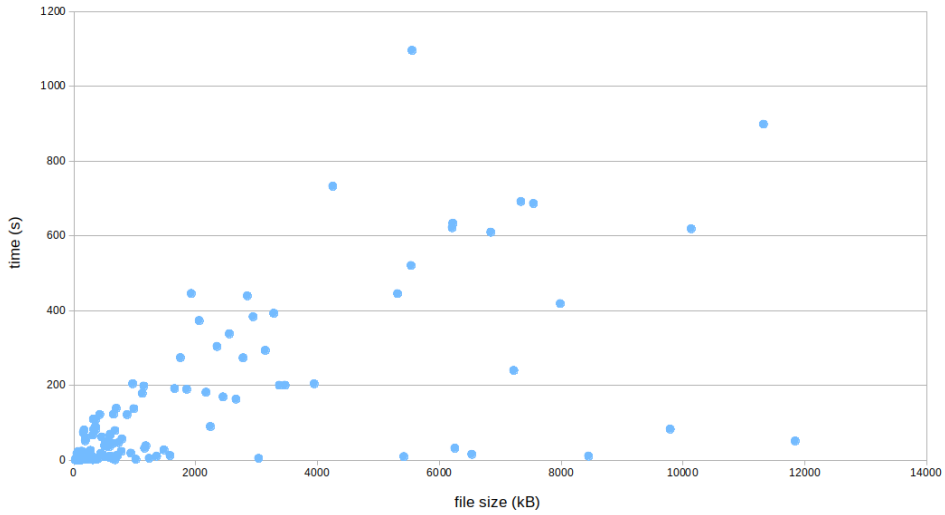


Figure 5.2: Total analyses times in function of the files size

of the behaviors on the client could reach the 97% of the total time, and in this case we are talking about several minutes. This means that the parsing and the visualization step has a great impact on the whole performance of our tool. After this consideration we can analyze the noise in the data and we can trace it to two main factors: there could be larger files for which few behaviors are found, in this case the parsing process on the client is quicker than for some of the smaller files in which more behaviors are recognized; the second factor is the complexity of the behaviors, and then the size of their graphs: some of the files have more complex behaviors than others that have their same size or that are even bigger, this means that the parsing process takes longer, increasing the total duration of the analysis, as we can see in figure 5.4, where we plotted the time taken for the whole process in function of the mean graph size, computed as described in 5.1.

$$MeanGrapSize = \frac{N_{attrs}}{N_{behaviors}} \tag{5.1}$$

where N_{attrs} is the total number of attributes from all the graphs found in the binary file, and $N_{behaviors}$ is the total number of behaviors found in the binary file.

We can observe the same phenomenon in figure 5.5: generally the time taken for the complete process increases with the number of behaviors found in the sample, however we can see a lot of noise and also some outliers, es-

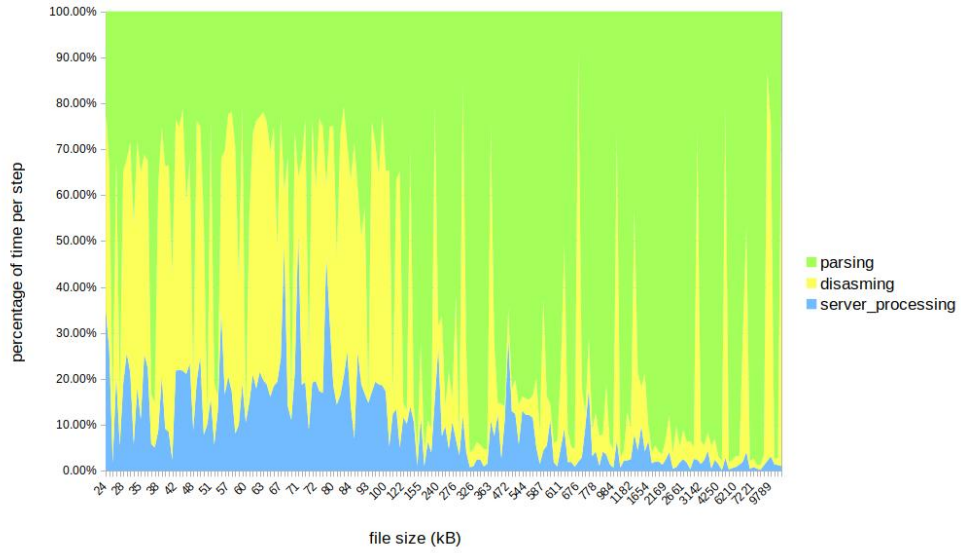


Figure 5.3: Percentage of time taken by each step of the process

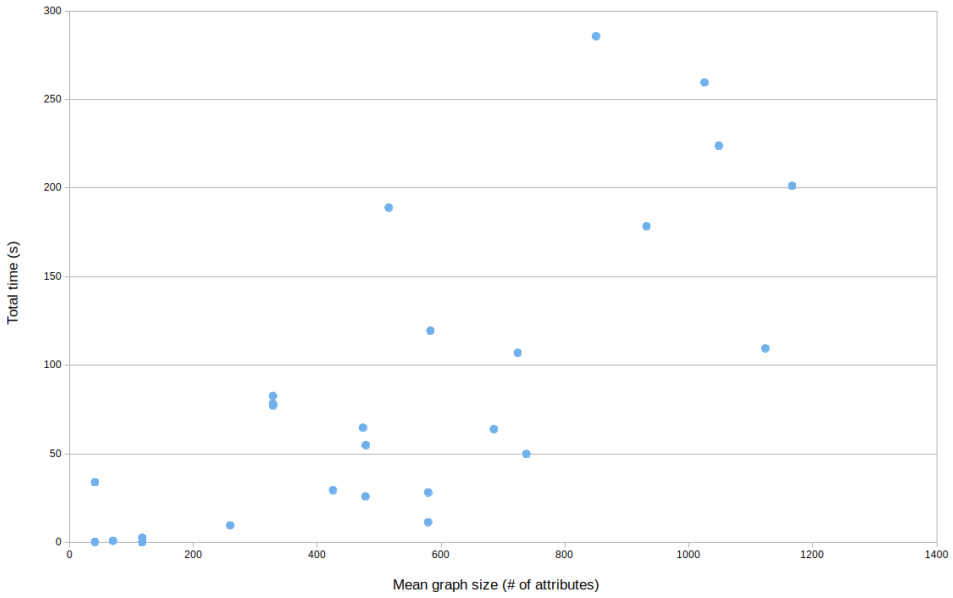


Figure 5.4: Total times in function of the size of the graphs

pecially when the number of graph retrieved is large (more than 200): in these cases the parsing is much faster because the graph are simpler, so the visualization happens before despite the large number of graphs listed in the Behaviors View.

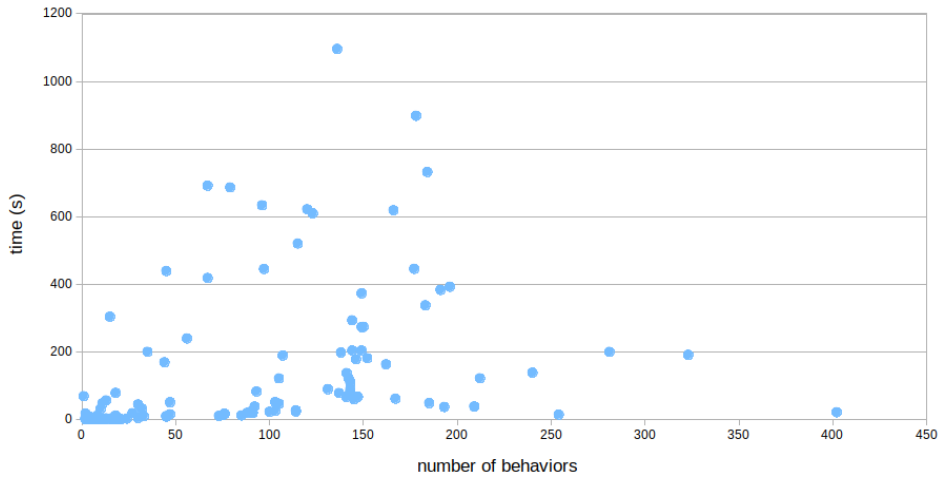


Figure 5.5: Total times in function of the number of behaviors

We can also compute the average time needed by our tool to retrieve and parse each behavior by dividing the total time taken for the whole process to complete by the number of behaviors found in the analyzed binary file. For very small files (less then 150kB) the mean time for behavior is about 0.184s, for files between 150kB and 1MB the mean time for behavior rises to 0.884s, for files larger than 1MB the mean time for behavior is about 2.480s. These times takes in consideration the disasming process, for which it is more likely to take more in larger files.

Chapter 6

Conclusions, limitations and future works

6.1 Limitations and future works

The main limits of this work are due to its integration with IDA Pro. If on one side this is a major advantage, because users can still use IDA as their main analysis tool, on the other end, on an implementative point of view, we have to made the most by using the current IDA SDK, undergoing the limitations about manipulating the user interface, especially regarding the asm view (the IDA main view) and the functions available to draw and visualize custom graphs. Maybe future versions of the SDK will be enriched with more API in order to expose more functionalities and at that point both the integration with the IDA Main View and the visualization of the graphs of the behaviors can be improved. Another limitation comes from the fact that IDA Pro is single threaded, with very basic support for the development of multi-threaded plugins, this means that when our plugin executes some functions, the IDA UI freezes until the task is completed. Finally there could be added more functionalities for the registered users, in order to improve even more the collaborative side of our tool.

6.2 Conclusions

The greatest contribution of this work to Jackdaw is the development and implementation of a tool able to visualize the behaviors, that are the results of the analysis of the tool-chain, of a binary file in a way that can be useful to human analysts who are statically analyzing malware disassembling them.

The aid provided by our solution could, in the first place, lead to a significant decrease of the time needed for an analysis, and then to a reduction of the skill set required to perform this kind of task, increasing the number of people able to do this kind of analyses. The solution we proposed in this thesis is designed to enhance an existing tool, IDA Pro, in this way analysts can simply keep using a powerful software they are already used to, while taking advantage of the additional benefits that Jackdaw and our plugin introduce.

The second contribution to the Jackdaw project is that of making it partially crowd-sourced and, in a way, a collaboration tool for analysts. The semantic descriptions can be improved by users and shared among groups of people, while introducing a voting system that is necessary in order to keep some sort of reliability of the descriptions. In this way Jackdaw can still be used to provide the ground information (the behaviors extracted by its automated analyses and tagged with the semantic algorithm) that is then improved by those analysts who are referring to it for the static analysis.

Bibliography

[1] F-Secure weblog. *A different look at Bagle*. 23 September 2015. URL: www.f-secure.com/weblog/archives/00000662.html.

[2] F-Secure weblog. *Graphing Malware*. 25 October 2005. URL: www.f-secure.com/weblog/archives/00000324.html.

[3] T. Dullien, R. Rolles. "Graph-based comparison of Executable Objects." In: *Symposium Sur La Securite Des Technologies De L'Information Et Des Communications (SSTIC)*. 2005.

[4] Zynamics. *BinDiff*. 2005. URL: www.zynamics.com/bindiff.html.

[5] Zynamics. *BinNavi*. 2011. URL: www.zynamics.com/binnavi.html.

[6] Hex-Rays. *IDA Pro*. 2015. URL: www.hex-rays.com/products/ida.

[7] Mario Polino, Andrea Scorti, Federico Maggi, and Stefano Zanero. "Jackdaw: Towards Automatic Reverse Engineering of Large Datasets of Binaries." In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer. 2015, pp. 121–143.

[8] Paolo Milani Comparetti, Guido Salvaneschi, Engin Kirda, Clemens Kolbitsch, Christopher Kruegel, and Stefano Zanero. "Identifying dormant functionality in malware programs." In: *2010 IEEE Symposium on Security and Privacy*. IEEE. 2010, pp. 61–76.

[9] Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna. "Polymorphic worm detection using structural information of executables." In: *International Workshop on Recent Advances in Intrusion Detection*. Springer. 2005, pp. 207–226.

[10] Martina Lindorfer, Alessandro Di Federico, Federico Maggi, Paolo Milani Comparetti, and Stefano Zanero. "Lines of malicious code: insights into the malicious software industry." In: *Proceedings of the 28th Annual Computer Security Applications Conference*. ACM. 2012, pp. 349–358.

[11] *Cuckoo Sandbox*. 2014. URL: www.cuckoosandbox.org.

[12] STEFANO D'ALESSIO and SEBASTIANO MARIANI. "PinDemonium: a DBI-based generic unpacker for Windows executables." In: (2016).

[13] *MongoDB*. 2009. URL: https://www.mongodb.com/.

[14] *Python*. 2016. URL: https://www.python.org/.

[15] *QT*. 2014. URL: https://www.qt.io/.

[16] *Pyside*. 2016. URL: www.pyside.org.