NAIST-IS-DD0961026

Doctoral Dissertation

Reversing Malicious Intents in Web Scripts: from Automating Deobfuscation to Assigning Concepts

Gregory Blanc

March 13, 2012

Department of Information Processing
Graduate School of Information Science
Nara Institute of Science and Technology

A Doctoral Dissertation
submitted to Graduate School of Information Science,
Nara Institute of Science and Technology
in partial fulfillment of the requirements for the degree of
Doctor of ENGINEERING

Gregory Blanc

Thesis Committee:

Professor Suguru Yamaguchi	(Supervisor)
Professor Hiroyuki Seki	(Co-supervisor)
Professor Minoru Ito	(Co-supervisor)
Professor Masakatsu Nishigaki	(Co-supervisor, Shizuoka University)
Associate Professor Youki Kadobayashi	(Co-supervisor)

Reversing Malicious Intents in Web Scripts: from Automating Deobfuscation to Assigning Concepts*

Gregory Blanc

Abstract

This dissertation presents program comprehension methods to infer malicious intents in web application scripts. Web 2.0 applications are plagued with web-based malware that leverage client-side scripting to exploit application and browser vulnerabilities. Moreover, scripting allows attackers to elaborate more complex attacks. On the other hand, they take advantage of distributed malware networks to reach to unsuspecting victims. Victims are often lured into accessing a vulnerable domain, and subsequently be redirected to an attack website. Web malware makes an intensive use of redirection, cloaking and obfuscation techniques to conceal its malicious intents.

My research motivation is to reveal these intentions in order to prevent browsers from getting exploited. Advanced countermeasures such as VM-based honeypots or browser emulators are vulnerable to some cloaking techniques, but are particularly lacking in usability since they rely on execution-based analysis. In fact, behavior-based detection has been very popular lately in the domain of malware analysis, overshadowing other alternatives. Indeed, scripts are interpreted, thus source code is readily available, offering the opportunity to apply static code analysis techniques.

In this dissertation, I propose a proxy-based solution to support realtime online analysis of staged web-based malware. I attempt to statically express the malicious intent of malware script by decomposing it in a limited number of concepts that can be represented by a UML sequence diagram. However static analysis alone cannot overcome the issue of deobfuscation, which is tackled using automated deduction. In particular, leveraging the properties of the Maude rewriting framework, I provide a sound and complete, yet terminating rewriting of obfuscated contents. This stage is actually dependent on the success of the previous tool which extracts obfuscated contents based on their syntactic patterns. Using a pushdown automaton, the system

*Doctoral Dissertation, Department of Information Processing, Graduate School of Information Science, Nara Institute of Science and Technology, NAIST-IS-DD0961026, March 13, 2012.

is able to match obfuscation patterns as subtrees of an abstract syntax tree representing the original script program to analyze.

The whole system can be seen as a set of 3 modules collaborating, but it can also be thought as independent tools, each contributing to further applications ranging from crawling to surveying to debugging.

Keywords:

web 2.0, JavaScript, malware, obfuscation, static analysis, automated deduction, term rewriting, program comprehension, concept assignment

Acknowledgements

As I reach the term of this long, tiresome, yet captivating work of research and study, I wish to express my deepest gratitude to all the people that supported me along the path. First, I must thank my supervisor, Professor Suguru Yamaguchi for his great support, his valuable comments and his challenging remarks that made me bounce back whenever my pace slowed down. I am indebted to him for these three wonderful years I spent under his patronage, thanks to his dedication to always make his students at home. I would also like to thank Professor Hiroyuki Seki for the insightful discussions and exchanges we had, that prompted the writing of my first journal paper. His wonderful pedagogy and patience allowed me to grasp complex knowledge that contributed greatly to this dissertation. I wish to thank Professor Minoru Ito and Professor Masakatsu Nishigaki from Shizuoka University for the honor they do me by serving on my thesis committee and the useful comments I received that greatly help to improve the overall quality of this dissertation. My special thanks go to Associate Professor Youki Kadobayashi for his endless support of my work and ideas, even when I was in doubt. His vision greatly impacted my work and his invaluable comments helped me to improve on my paper writing and presentation skills.

Collaborations on this dissertation were also very important towards its completion and I cannot miss the occasion to acknowledge Daisuke Miyamoto, now at the University of Tokyo, whose doctoral research work also inspired this dissertation. I really enjoyed our numerous discussions and invaluable comments I received, as well as his experience of research. I cannot dissociate from these discussions Mitsuaki Akiyama, at NTT Platform Laboratories, whose technical support fueled the accomplishments we made together. Ruo Ando, at the National Institute of Information and Communications Technology, provided superior skills and knowledge that impressed me a lot, and I hope I gained much from him through our collaborations. Other notable researchers with whom I had fruitful and inspiring discussions include Ahmad El Ghafari, Tomáš Flouri, Scott Martens, Yousuke Hasegawa, Daniel Reynaud-Plantey, Grégoire Jacob, Masaki Kamizono and Masata Nishida, as well as the friendly members from the WIDE Project with whom I interacted during SWAN BoF sessions or poster presentations. Some other well-known researchers kindly shared their datasets which is an essential resource in our area: Marco Cova, Blake Hartstein, Anton Chuvakin and Wei Lu.

The environment, where I dwelled during these years, was also very beneficial and pushed towards my goal, even when I was deprived from sleep or energy, the kind assistance of Assistant Professor Takeshi Okuda, Hiroaki Hazeyama and Shigeru Kashihara

never let me down. Especially, Hiroaki Hazeyama is to be credited for this day of 2006 when he, unlike others, would not ignore my email, and offered me the chance to experience with research in the domain of IP traceback. This wonderful experience was also possible thanks to the kindness of Professor Suguru Yamaguchi and the assistance of Associate Professor Youki Kadobayashi. I also wish to express my gratitude to Teruaki Yokoyama, who helped me polish my Japanese writing skills when I applied for scholarships. He is the life of the lab and his cheerfulness is always welcome. Also, I am grateful to have such interesting and joyful lab mates that made my stay never boring nor sad. On the contrary, they were great companions, in research but also in play, and I owe them the wonderful memories we share together, and that will never be forgotten. In particular, I wish to express my special thanks to the secretaries who always cheered me up as well as my mate Kazuya "Loppy" Okada. I am also grateful to Masatoshi Enomoto for his help as my tutor. I am also indebted to Doudou Fall, Sirikarn Pukkuwanna and Louie Zamora for their corrections. My 2 last years will be also greatly remembered thanks to the presence of Masato Jingu, Tomonori Ikuse, Kyohei Moriyama, Shinya Kanda.

Finally, I would like to express my love for my friends in Japan, France and all over the world, who never stopped cheering me up and offered me relief whenever I needed. No matter how much I rewrite it, or how well I rephrase it, it will never be good enough to express my deep feelings of true friendship for (in no particular order), Vincent T., Emeric, Vinh, Matt C., Armand, Vincent V., Antoine L., la bande des Frew, Trai, Renaud, Chibashu, Claire, Flora, Jue and my second family in Brittany (the Bordiec). I also should give a lot of credit for a very special person: Joséphine, who changed the course of my life in a way I could not imagine. My family also was always supportive, even though I kind of let them behind, and their love and support prevented me from ever giving up on my mission. I retained mostly the values taught by my parents, the pride of my brother and the love in the eyes of my grandma.

I gratefully acknowledge the financial support of the French Ministry of Foreign and European Affairs, through the Lavoisier scholarship, as well as, the NEC C&C Foundation, through the Grant for Non-Japanese Researchers.

Dedication

To the victims of the 2011 Tohoku Great Disaster.

Contents

Acknowledgements **iii**

Dedication **v**

1. Introduction **1**

1.1 Motivation 1
1.2 Protecting the User against Web-based Malware 2
1.3 Research Issues 4
1.4 Contributions 5
1.5 Organization 6

2. Web 2.0 Security **8**

2.1 Web-based Attacks 8
2.1.1 Server-side Attacks 8
2.1.2 Client-side Attacks 9
2.2 Web 2.0 Insecurity 10
2.2.1 Web 2.0 11
2.2.2 JavaScript and AJAX Security 12
2.2.3 The XMLHttpRequest Object and Related Threats 16
2.2.4 On JavaScript Malware and Related Threats 17
2.3 Attacks against Browsers 20
2.3.1 Drive-by Download Attacks 21
2.3.2 Web Cloaking 22
2.4 Countermeasures against Web Attacks 23
2.4.1 Validating User Inputs 23
2.4.2 Security Devices 24
2.4.3 XSS Defense 24
2.4.4 CSRF Defense 25
2.5 Countermeasures against Web Malware 26

3. Problem Analysis **28**

3.1 Methodology of Web Malware 28
3.1.1 Redirection and Cloaking 28
3.1.2 Deobfuscation 30
3.1.3 Environment Preparation 30

3.1.4 Exploitation . . . 31
3.2 Circumventing Current Web Malware Detection . . . 31
3.2.1 Virtual Machine Honeypots . . . 31
3.2.2 Browser Emulation . . . 33
3.2.3 Reputation-based Detection . . . 34
3.2.4 Signature-based Detection . . . 34
3.3 Issues on Redirection, Cloaking and Obfuscation . . . 35
3.4 Issues on Usability of Browser Protection . . . 36
3.5 Issues on Offline Analysis and Execution-based Side-effects . . . 37
3.6 Summary . . . 37

4. Overview of the Proposed System **39**
4.1 Requirements . . . 41
4.2 Intercepting HTTP Transactions . . . 42
4.3 Countermeasures against Redirection and Cloaking . . . 43
4.4 Detecting Obfuscation . . . 44
4.5 Reversing Obfuscation . . . 45
4.6 Analysis of Deobfuscated Script . . . 46
4.7 Recursive Characteristic of the System . . . 47
4.8 Summary . . . 47

5. The Concept of Intention **48**
5.1 Related Work . . . 49
5.2 Representation of the Intent of a Developer in a Program . . . 50
5.3 On Reverse Engineering . . . 51
5.4 Summary . . . 54

6. On Obfuscation **55**
6.1 Taxonomy of Obfuscation Techniques . . . 56
6.1.1 Measures . . . 56
6.1.2 Layout Transformations . . . 57
6.1.3 Control Transformations . . . 57
6.1.4 Data Transformations . . . 60
6.1.5 Primitives on Deobfuscation . . . 61
6.2 Obfuscation of Web-based Scripts . . . 62
6.2.1 Lexical Transformations . . . 63
6.2.2 String Manipulation . . . 64

6.2.3 Encoding Schemes . . . 65
6.3 Reversing/Cancelling JavaScript Obfuscation . . . 66
6.3.1 Manual Approach . . . 67
6.3.2 Automated Deobfuscation . . . 67
6.4 Anti-analysis Techniques . . . 69
6.4.1 Multi-partite Scripts . . . 70
6.4.2 Opaque Constructs and Cloaking . . . 71
6.5 Summary . . . 72

7. Identifying Hierarchical Structures Related to Obfuscation 74
7.1 Detecting Obfuscation . . . 75
7.2 Overview . . . 76
7.3 Obfuscating Transformations . . . 76
7.4 Abstract Syntax Trees in JavaScript . . . 78
7.5 Matching Subtrees to Detect Obfuscation . . . 85
7.5.1 AST Paths . . . 86
7.5.2 AST Fingerprinting . . . 86
7.5.3 Subtree Matching by Deterministic Pushdown Automaton . . . 87
7.6 Implementation . . . 90
7.7 Experiment . . . 91
7.7.1 Dataset Description . . . 92
7.7.2 Evaluation . . . 92
7.7.3 Performance . . . 101
7.8 Discussion . . . 102
7.8.1 Other Obfuscating Transformations . . . 102
7.8.2 Irrelevant Code . . . 102
7.8.3 Feature Selection . . . 106
7.9 Summary . . . 106

8. Automated Deduction 108
8.1 First-order Logic . . . 108
8.2 Equational Logic . . . 110
8.3 Rewriting Systems . . . 112
8.3.1 Properties . . . 113
8.3.2 Theorems . . . 113
8.4 Maude as a Rewriting Framework . . . 114

8.4.1 Equational Membership Logic . . . 114
8.4.2 Functional Modules . . . 116
8.4.3 Properties of Functional Modules . . . 118
8.4.4 The Reduction Command . . . 119
8.5 Summary . . . 119

9. Automated Deobfuscation of Script Contents using Rewriting-based Emulation 121
9.1 Deobfuscation . . . 122
9.2 Overview . . . 123
9.3 Automated Deduction using Maude . . . 124
9.3.1 Obfuscation as a Subset of JavaScript . . . 125
9.3.2 Axiomatizing JavaScript Packers in Maude . . . 125
9.3.3 Loop Conversion . . . 129
9.3.4 Postprocessing . . . 129
9.4 Example . . . 130
9.5 Discussion . . . 132
9.5.1 Performance . . . 132
9.5.2 Termination . . . 132
9.5.3 Other Obfuscation Techniques . . . 133
9.6 Summary . . . 133

10. On Program Comprehension 135
10.1 The Concept Assignment Problem . . . 135
10.2 Concept Recognition and Program Slicing . . . 136
10.3 Program Categorization . . . 138
10.4 Summary . . . 139

11. Knowledge-based Static Intentional Analysis of Unobfuscated JavaScript Code 140
11.1 Motivation . . . 140
11.2 Related Works on Web Script Analysis . . . 141
11.2.1 Machine Learning Based . . . 141
11.2.2 Emulation Based . . . 142
11.2.3 Abstract Syntax Based . . . 142
11.2.4 Symbolic Execution Based . . . 143
11.3 Inferring Intentions to Detect Web Malware . . . 144
11.4 Program Slicing . . . 145
11.5 Object Categorization . . . 147

11.6 Discussion . . . 148
11.6.1 Learning and Comparing Models . . . 148
11.6.2 Implementation and Performance . . . 150
11.7 Summary . . . 150

12. Conclusion **152**
12.1 Discussion and Summary of Contributions . . . 152
12.2 Avenues for Future Work . . . 156

References **160**

Appendix **173**

A. List of Publications **173**
A.1 Journal . . . 173
A.2 International Conference . . . 173
A.3 Technical Report . . . 174

B. Research Grants and Scholarships **175**

List of Figures

1 Simplified architecture of a Web 1.0 browser 12
2 Simplified architecture of a Web 2.0-capable browser featuring the AJAX framework 13
3 Workflow of the Browser-Rider botnet 20
4 Loop spraying the heap memory with a shellcode preceded by a NOP sledge 22
5 Anatomy of a drive-by download attack 29
6 Overview of the (sak_mis) system to counter Web malware 41
7 Aliasing of the eval() function 63
8 Concatenating strings to invoke the fromCharCode() method 64
9 Example of character substitution 64
10 The string test written in different encodings 65
11 Custom encoding based on a simple XOR 65
12 Example of a multi-partite scripts split into script blocks and remote sources 70
13 Context-dependent obfuscation using arguments.callee 71
14 Anti-emulation technique based on the time delay to trigger the decoding routine 72
15 Workflow of the ob_asti obfuscation extractor 77
16 Abstract syntax tree of a sample loop 79
17 Day 1 samples clustered by Ward's method based on their ASTF distribution 94
18 Bottom-up subtree of the example eval unfolding 104
19 Induced subtree of the example eval unfolding 105
20 Workflow of the u_adjet deobfuscator 123
21 Decomposition of a packed program and conversion of the decoding routine 126
22 Pseudo-code of a loop performing the function f() 129
23 Pseudo-code of a recursive function f() 129
24 Original HTML code 131
25 Maude functional module 131
26 Sample program clustered following slicing. 145
27 A resulting sequence diagram. 148

List of Tables

1 SOP check outcomes 14
2 Qualitative comparison of some web malware detectors 32
3 Common obfuscation techniques occurring in JS malware 77
4 Matched samples of Day 2 dataset clustered by obfuscation patterns . . 96

5 Matched samples of Day 3 dataset clustered by obfuscation patterns . . 98
6 Matched samples of the Alexa 25 dataset clustered by obfuscation patterns 100
7 Recapitulative table of the preliminary experiment 100

1. Introduction

> Imagination is more important than knowledge. For knowledge is limited to all we now know and understand, while imagination embraces the entire world, and all there ever will be to know and understand.
>
> Albert Einstein

1.1 Motivation

Internet has been one of the greatest technological achievements of computer and network scientists for half a century. Advancements of Internet technologies have rendered computing ubiquitous through the pervasive networking capabilities. Internet has made possible the old dream of "connecting people" all over the World. One meaningful instance of its applications is the Social Web. And as it connects more and more people, it provides everyone with a window to the whole World to express oneself, interact/share with others and or simply discover knowledge. Most popular applications of the Internet include web browsing, email services, Internet telephony, file sharing and media streaming. In particular, the Web paradigm has shifted at the beginning of the last decade, with a gradual change in both developers' and users' habits.

Web 2.0, as it is coined, is not an update of the Web itself, since it still relies (although partly now) on the HTTP protocol for transport and HTML-based technologies for the display of web pages. However, web contents themselves have evolved, both as a result of user's needs and as a result of technology advancements, feeding each other's development. One outstanding example of this paradigm shift is the Social Web where users share a web-based application platform on top of which they not only communicate but achieve most of the popular services offered by the Internet. The Social Web usually showcases recent advancements in Web research, but it also displays how deep-rooted are its vulnerabilities. In fact, the Web never really addressed application layer vulnerabilities as a whole and has often been left behind in the arms race against attackers, by constantly "patching" any new hole that was discovered. In particular, the Same-Origin Policy (SOP), which is at the heart of the Web security capabilities, has irreducible legitimate bypasses to maintain the possibility of cross-domain communications, which have never been as flamboyant as nowadays. The SOP stipulates

that a web page from a given domain (for example, example.com) is not allowed to access objects or properties from a web page from a different domain (notexample.com). A strict observance of such policy would obviously make impossible applications that thrive by communicating across domains or by mashing several web contents from different domains into a single web application.

In recent years, vulnerabilities such as cross-site scripting, which allows an attacker to inject malicious code that will be reflected in the user's browser, or cross-site request forgery, which allows an attacker to force an unsuspecting user into issuing an authorized request to a remote web page, have gained massive popularity among web criminals. In fact, not only exploiting these vulnerabilities allows the above-mentioned capabilities, worse, it allows an attacker to maintain a SOP bypass and build more sophisticated attacks on top of it. Web malware or JavaScript malware designate such malware-behaving programs that execute on top of a web page and exploits the user's browser local environment.

1.2 Protecting the User against Web-based Malware

The probability for a user to come across malicious contents has increased dramatically with the advent of the Web 2.0. There are several acceptable reasons for that:

- with the popularity of modern Web applications, the scope of a single attack has increased, and a user may not only be exposed directly to the source of an attack, but may also be indirectly through another victim;
- with the increasing security awareness of Web companies, attacking their assets directly has become very difficult, and attackers are more interested by quick profit, therefore turning against users or third-party content providers, such as advertisement providers;
- with the increasing number of novices and the lack of education, end-users have quickly been identified as profitable targets since it is easier to monetize stolen personal information.

Academic and industrial research has been active on client-side security issues with the production of various tools and browser plugins that ensure the user with a safe browsing experience:

- malware often exploits known vulnerabilities in browsers and their plugins to take advantage of the end-user's browser. Signature-matching provided by common

antiviruses should normally be able to detect such attacks, or at least inform the user that her browser is vulnerable;

- JavaScript (JS) is the de-facto standard scripting language of the Web, and is highly utilized in developing Web malware since any browser is shipped with a built-in JavaScript engine. With the increasing amount of JS code pushed to the client-side, plugins to disable whole or part of the scripts running on top of a web page are successful in preventing the execution of malicious JS programs. A popular example is NoScript [92];

- as it is unlikely that malicious web pages would participate in the effort of "securing" the Web, designing policies shared between legitimate web pages and browsers seems achievable to filter out both suspicious domains and suspicious requests. Such policy has been applied in RequestRodeo [80] to thwart forged HTTP requests and should be applied soon to regulate cross-domain communication;

- more lightweight approaches exist to prevent a user from an unfortunate visit to a suspicious web page. URL filtering usually occurs before a user browses a web page: the URL is checked against a blacklist. Blacklisting includes Google SafeBrowsing [60] or the Web of Trust browser plugin [143]. The process of adding a domain to a blacklist may vary from one provider to another but the decision if often made upon examination of the web page;

- setting a malicious attack network or a malicious web page has become quick and easy thanks to the industrialization of attack toolkits. Such toolkit provides an attacker with means to create an infected web page, register a domain and even spam users to recruit victims. A proactive way to detect such pages is provided by crawlers that browse thousands of web pages a day and analyze their contents to determine their nature. Analysis methods used to assess the malice of a web page are numerous and are further treated in Section 3. Additionally, there exist online analysis platforms that provide such service directly to the user: WEPAWET [31] and jsunpack [65] are the most well-known instances.

Current approaches described above offer satisfying solutions to some extent. In fact, concerns have been raised at several layers including the possible evasion of a system, the relevance of the deployment location, the lack of incentive for a user to deploy the system and the effectiveness of the system.

1.3 Research Issues

The ideas presented in the previous section raise a number of important research problems, which are outlined below.

Deployability Architecture is always a concern when dealing with security systems as it is transversal to other system properties. Since it can be a vulnerability, it is important to ensure that from the point of view of the architecture, the proposed solution is safe. In particular, it should not suffer possible hijacking at the client-side, or interception of information provided at the server-side. Actually, communication with the server-side is not encouraged as a user may deal with domains that do not offer such feature, let alone malicious ones. Also, client-side implication should be at the extent of preserving the user-experience.

Security The security of a security system is indeed critical. Protecting the user should not suffer any leakage of malicious contents to the user. Containment is a key feature of such system. Manipulating JavaScript is also not really recommended since JavaScript security is not guaranteed as developed in Section 2. Ideally, it is satisfying to prevent its execution in favor of static approaches, but this can be hindered by polymorphic capabilities of malicious JS code.

Reliability Related to the above property, the system should be difficult to evade, and should prevent further damage otherwise. Reliability is often shared with third-parties so we assume that reducing the number of stakeholders would be beneficial. Therefore, relying on the server-side is unlikely to increase the level of security and should be avoided. The proposed system should concentrate on weaknesses found in state-of-the-art solutions and eliminate these as much as possible. Such issues include obfuscation (the ability to render code unintelligible), cloaking (the ability to prevent detection/analysis), lack of completeness. They are further described in Section 3.

Usability Often overlooked, this property is yet crucial to the user. Failure to take it into account may lead to a malicious insider behavior where the end-user will deliberately choose to disable the system or ignore warnings from the system. Moreover, security decisions are difficult to make, and without proper education, it is often bound to failure. An acceptable solution would offer seamless enough protection in order not to bother the end user. Not collaborating with the end-user is a drastic choice but can be relaxed as a last resort, if the need arises.

1.4 Contributions

This thesis discusses the design and prototype implementation of (sak_mis), a client-side system that protects against Web-based malware. It addresses the challenge of inferring the intents of a malicious web script by bridging the syntactic representation and the semantic representation of such script in order to associate the script constructs with human concepts that can be classified as being benign or malicious. Such process is often hindered by both the attacker's side that attempts to conceal malicious intents and the analyst's side who fails to capture the complete expression of a malicious script.

In this thesis, we identified the causes of such drawbacks and attempt to provide a satisfying solution, not only to these issues but also to the user who is the target of these attacks. Obfuscation and cloaking techniques prevent the analysis of malicious scripts. However, such techniques are not only used by attackers but also by legitimate web sites, making these difficult to decide per se. Dynamic approaches have often been applied to the analysis of malicious scripts with mixed results since execution involve undesirable side-effects that cannot be easily cancelled due to obfuscation and cloaking techniques. Also, we argue that end-users should be excluded from security decisions since they do not have enough expertise. Finally, we criticize former solutions on their lack of usability due to their offline analysis capability or their blocking features.

This dissertation subsequently covers 3 subjects besides the concept of intent that we attempt to clarify in Section 5 and the architecture of the system in Section 4:

- the reliable detection of obfuscation, in particular encoding and packing schemes, which are actually the most popular tools. Our approach differs from past string-based and statistical methods that only decide whether a web page is obfuscated. On the contrary, we are concerned with precisely detecting what part is obfuscated in order to extract obfuscated contents. Our contribution attempts to ultimately classify patterns of obfuscation based on the hierarchical structures exhibited by decoding routines. It deals with subtree matching by incorporating algorithms from automata theory. The prototype we developed is able to detect distinct obfuscation encoders with a relative accuracy and also offer clustering of samples by the patterns they express. A pattern is here a combination of obfuscation techniques. Additionally, the automaton performs in a reasonable time frame that is not incompatible with web browsing;

- the quick rewriting of obfuscated contents to a deobfuscated form. This is crucial to our approach since sound and complete analysis of a program is only

possible on an unobfuscated program. Concerned with JavaScript execution, we propose its emulation instead of emulating the browser as it has been previously done in past research. Our contribution attempts to provide sound and complete deduction of an obfuscated program, seen as a set of instructions, into an irreducible form, which is assumed to be the original program (or a semantically equivalent program). We make use of the Maude [25] language, which is a reflective language that supports both rewriting and equational logics. We associate Maude equational constructs with the subset of JavaScript used to write packers, and augment such code with additional equations that rewrite JS functions and constructs (such as loops) not present in Maude. Although the mapping is not complete, we envision further application to the resolution of other obfuscation techniques, such as opaque predicates. Opaque predicates are predicates of which value is only evaluated at runtime;

- the assignment of programmatic concepts to higher-level concepts. Finally, intents are revealed through the examination of the source code. We propose a representation called intention that associates a UML representation of the sliced decomposition of JS program with labels denoting the action carried out by the slice. Our contribution relies on the design of a forward decomposition slicing algorithm that can partition a program in a set of distinct objects, each purporting a distinct action. The action is inferred thanks to a knowledge base of the programming language of the analyzed script. The knowledge base classifies functions and objects to higher-order concepts that denote a specific action. An intention is therefore a combination of such actions. It is yet to decide if an intention is malicious or benign, but we may further classify intentions based on expert knowledge. Assigning concepts to decomposition slices has the additional contribution of accommodating polymorphism, that is, several distinct implementations are recognized as a same intention.

Additionally, the proposed system attempts as much as possible to provide a static analysis. This is however not possible when dealing with deobfuscation. It also stresses methods to offer a usable protection to the end-user.

1.5 Organization

The remainder of this dissertation presents the details of each proposal. After highlighting the particularity of Web 2.0 applications regarding security in Section 2, we

motivate the present work in Section 3 by describing related issues and countermeasures in the field of Web 2.0 security.

We outline the current proposal and present an overview of the proposed system, named (sak_mis), in Section 4, before elaborating on the concept of intention from which this research took inspiration (Section 5).

Sections 6 and 7 tackle the subject of obfuscation and how ob_asti is able to detect known obfuscation patterns and quickly match these within a larger script using pushdown automaton.

Then we detail the underlying logical background necessary to understand the Maude rewriting system in 8 and then, in Section 9, the actual emulation-based automated deobfuscation tool named u_adjet.

Finally, Section 10 details the rationale behind the concept assignment problem and Section 11 presents mi_oos, a static decomposition slicing algorithm to assign (malicious) intents to script slices.

Section 12 summarizes the contributions of this dissertation and further perspectives prompted by the results, as well as discussion of some open issues that were not covered in the present work.

2. Web 2.0 Security

> If you think technology can solve your security problems, then you don't understand the problems and you don't understand the technology.
>
> Bruce Schneier

In this section, the reader will be introduced to the realm of Web 2.0 security. We will cover past and current attacks, as well as, Rich Internet Application (RIA) vulnerabilities and subsequent exploits tackled by the present research work, as well as, the current countermeasures at the time of writing. For the sake of completeness, client-side attacks will be contrasted with server-side attacks, although they are often combined together: the exploitation of a server-side vulnerability leading to possible harm done to the end-user.

2.1 Web-based Attacks

Information and systems security has been intimately intertwined with the evolution of computer systems. First attacks were isolated, and directed at servers. But with the success of modern web applications, the target has shifted to users. With the advent of the Web 2.0, where applications push most of their code to the client-side, opportunities have increased for an attacker to directly harm users.

2.1.1 Server-side Attacks

Originally, Web services were restricted to publishing static pages and browsing. Resources were scarce and attacks would target only servers. Generally, a denial of service (DoS) attack originating from a single machine would be sufficient to stun a server. In 1999, Trinoo was a client program that helped an attacker perform distributed DoS (DDoS) attacks. Aside from the HTTP port (80), attackers would attempt to exploit other service vulnerabilities (on other open ports) in order to subvert the machine hosting the web server. In the year 2000s, many worms, such as Code Red and Nimda, were successful at "recruiting" nodes among vulnerable web servers.

Gradually, as the service offer evolved, attacks were distributed not only against web servers, but also against databases (SQLSlammer) and directory services (Sasser). Web sites then became web applications that not only publish static contents, but also provide services to users on these contents. First, web applications were used as

stepstones by attackers to reach their goal: the server. By spotting vulnerabilities or loopholes in the logic of the application, attackers were able to inject and execute arbitrary code on the server side. Common examples include:

- SQL injection (tampering with data, sensitive data disclosure or deletion, privilege escalation, etc.);
- command injection (sensitive data disclosure, privilege escalation, denial of service, etc.);
- remote file inclusion, also known as RFI (arbitrary code execution);
- buffer overflow and format string attacks (denial of service, arbitrary code execution, etc.);
- forced browsing and directory traversal (sensitive data disclosure).

Most of the attacks are submitted through web parameter tampering either in the URL, the payload or the HTTP headers.

2.1.2 Client-side Attacks

Later, attackers realized that injecting code could not only subvert server resources but could also be directed to other users of the application, at a large scale. While phishing spam may fail in attempting to lure a user into accessing a malicious website, it has proven more successful to have a user to first access a legitimate website and then be redirected to a malicious one. Past injection attacks were leveraged to either include code from malicious domains or silently redirect users to these domains. Popular attacks include:

- cross-site scripting, which comes in three different flavors: stored, reflected and DOM-based;
- cross frame scripting;
- HTTP request smuggling, as well as, its counterpart HTTP response splitting.

Above-mentioned techniques allow an attacker to, among other things, execute arbitrary code directly or by embedding code from a domain owned by the attacker; steal sensitive data; deface the contents of an application, visited by the user, by poisoning the cache or manipulating the Document Object Model (DOM).

Other classes of attacks attempt at impersonating another user either by stealing the user's credentials, hijacking the user's session or making the user performs actions that will benefit the attacker:

- cross-site tracing (XST) where credentials are read thanks to TRACE, a rarely used HTTP method;
- session sniffing where the attacker performs a man-in-the-middle attack by capturing network packets;
- session fixation where an attacker force a victim to use predefined credentials;
- cross-site request forgery, also known as CSRF or XSRF, is a scheme where an attacker gets a user to access a link or a page that will generate a request to an application the user is logged into, leveraging the fact that the browser automatically attaches authentication information to any request bound to an application that maintains a session.

The reader can refer to the websites institutions that specialize in web application security, such as the Open Web Application Security Project (OWASP) [132] or the Web Application Security Consortium (WASC) [134] for rather complete taxonomies of the attack landscape (the WASC Threat Classification is of particular interest). For readers not familiar with the HTTP protocol and web-based communication technologies, you may find relevant information, as well as, attack scenarios and other technical details in the literature [130].

2.2 Web 2.0 Insecurity

There is no strict difference between Web 1.0 and Web 2.0 but it is universally understood that Web 1.0 applications rely mainly on the HTTP protocol to download pages in a synchronous pattern. On the other hand, Web 2.0 applications do involve abundant processing on the client side through embedded scripts that will send and receive data to the server asynchronously, without the user experiencing delays. The richness of the browser-side processed information, and the amount of information exchanged without the user-explicit consent, have impacted the way attacks are carried out in Web 2.0, eventually reaching the browser. The attacks usually target a misuse in the application processing in order to manipulate the output by designing a malicious user input. The user, especially the user's private information, is now the target of a large

panel of browser-based attacks such as XSS, CSRF, phishing, etc., that were already present in earlier Web applications, but whose potential has increased.

2.2.1 Web 2.0

Web 2.0 does not rely on any particularly recent technology, but on technologies that have been spreading the Web since its early years for some of these. JavaScript and XML, at the origin of the coined word AJAX (Asynchronous JavaScript with XML) [57], were technologies designed in the mid-1990s. In fact, Web 2.0 applications did exist prior to 2001. However, the promotion of a framework of technologies (at that time, not coined as AJAX) has been boosted by the implementation of the `XMLHttpRequest` (XHR) DOM API in Mozilla 1.0 in 2002. It allowed the development of web applications that did not need to constantly reload the web page to update its content dynamically, leading to a more user-friendly, a more interactive and fluid experience of the Web.

Web 1.0 used to be a one-way information-providing Web in which users would browse the Web requesting contents they would download on their browser. Contents were mostly static and maintained only by their author. At that time, the browser architecture itself was very simple as shown in Figure 1.

On the other hand, Web 2.0 applications rely on rich Internet application frameworks and share many participative characteristics: they are dynamic, interactive and comparable to desktop applications, they facilitate communication, they allow collaboration between users, they rely on syndication to provide contents from many different origins and even allow web applications to interact in order to create new services, they also make use of Web standards and achieve scalability through cloud computing.

The XHR object is instrumental in rendering the fluidness of Web 2.0 applications, in comparison to plain HTML ones (the reader can observe the central place of the AJAX framework in Figure 2), but also in hiding a large amount of client-server communications from unsuspecting users. These communications exchange data that must be machine-readable and therefore the two mostly used formats are XML and JSON (JavaScript Object Notation) [34], a subset of the JavaScript programming language.

JavaScript is such an essential component of modern web applications that the security of browsers also partly depends on it, although other components or plug-ins should also be considered. But JavaScript has really been the "glue" between components in recent browser instantiations.

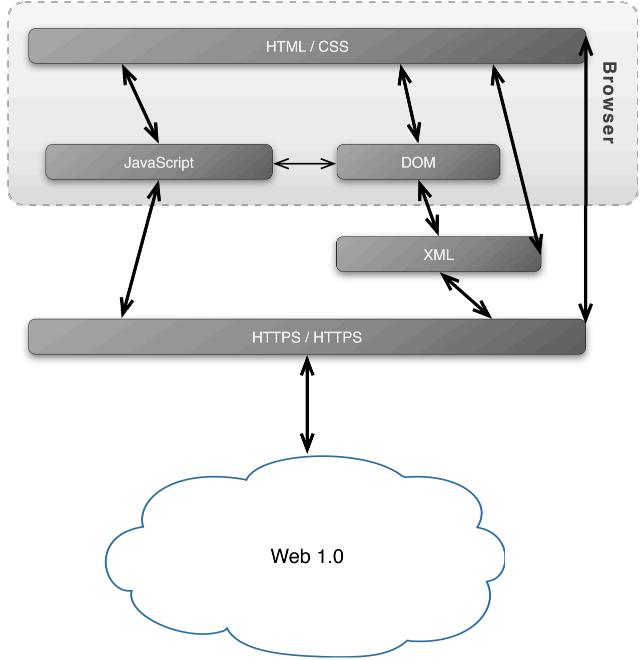

Figure 1. Simplified architecture of a Web 1.0 browser

2.2.2 JavaScript and AJAX Security

JavaScript (JS), which is the de facto standard in the AJAX implementation, is a dynamic, duck-typed, interpreted language that runs in browsers, and also locally in some applications, for example, PDF files [1]. It is a functional, event-driven, prototype-oriented (not really object-oriented) language that can modify a web page through its Document Object Model (DOM). Prototype-based languages feature a classless object-oriented paradigm where inheritance is performed via cloning existing objects (the so-called prototypes). JavaScript was created by Brendan Eich in the early days of the Web. *JavaScript* describes the Netscape's (now, Mozilla's) implementation of the language, of which standard is known as ECMAScript [41]. JavaScript features capabilities such as loading and executing remote files or scripts, which can be seen both as useful and harmful.

While widespread plug-in technologies such as Flash and Java can be subverted and do much more damage in terms of range, due to built-in networking or file-access functionalities, JavaScript is natively interpreted by most of the browsers and does not

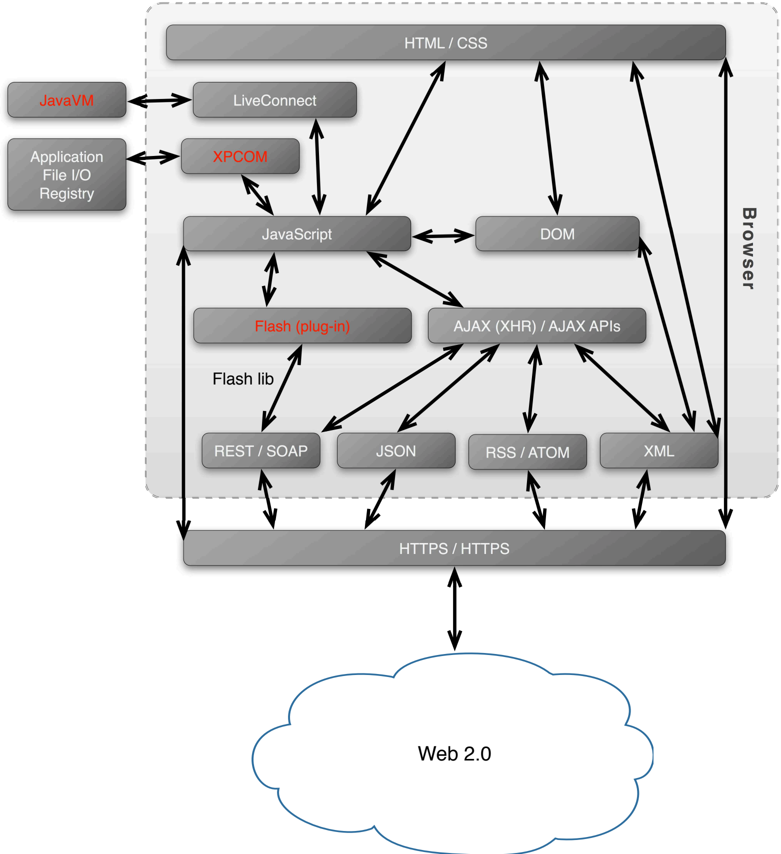

Figure 2. Simplified architecture of a Web 2.0-capable browser featuring the AJAX framework

need any prior installation. From its conception, the JavaScript language does provide some basic security mechanisms [51] such as:

- native sandboxing: client-side JavaScript does not implement utilities to have access to local files or to perform generic networking tasks. Therefore it is rather safe itself. Nonetheless, JavaScript language has the ability to manipulate other Web components such as ActiveX controls or Flash programs, as well as, Java applets, thereby relying on the security of these components. Although they may implement sandboxes to prevent communications with external URLs, the attack surface has now extended from the Web browser to browser plug-ins;

- same-origin policy (SOP): what is defined as origin is the combination of a pro-

Table 1. SOP check outcomes

URL	Outcome	Reason
`http://www.example.com/dir/page2.html`	Success	
`http://www.example.com/dir2/page.html`	Success	
`https://www.example.com/secure.html`	Failure	Different protocol
`http://www.example.com:81/dir/etc.html`	Failure	Different port
`http://test.example.com/dir/page.html`	Failure	Different host

tocol, a URL and a port (see Table 1 for some example checks against the page `http://www.example.com/dir/page.html`). Based on this triplet, an embedded script or a `XMLHttpRequest` object can not access and modify documents, frames or embedded contents that are not from the same origin. More precisely, what matters is the origin of the document in which the script is embedded and not the origin of the script itself. The SOP does apply to most of the Document properties but is implemented somehow differently from one vendor to another. The SOP prevents stealing proprietary information.

However, the SOP has been criticized for being too restrictive at times, especially in cases where a domain owns many subdomains. A trick to allow two web pages, from two different subdomains, to communicate is by setting the `domain` property of each page to a common value with the constraint that the new `domain` has to be a valid suffix of itself. An alternative method for relaxing the SOP is being standardized as the Cross-Origin Resource Sharing [140]. This technique extends the HTTP protocol with new headers: an `Origin` request header, which indicates where the cross-origin request originates from, an `Access-Control-Allow-Origin` response header, which indicates whether a resource can be shared.

A little-known feature of the JavaScript language lies in its functional and object characteristics: functions are objects and can be overridden. Moreover, JavaScript allows redefining functions after they have been declared, at runtime. This phenomenon, known as Prototype Hijacking [40], or more colloquially JavaScript Clobbering, enables a JavaScript program to intercept and modify automatically another application's source code [67], under the SOP. An attacker can abuse Prototype Hijacking to override the behavior of a function, even a native one, with a malicious version of the former while maintaining the original behavior in the eyes of the user. Clobbering functions is usually performed in 3 steps:

1. a reference is created to the original function

```
var oldFoobar = foobar;
```

2. a shim function is defined that performs operations to the attacker's benefit and subsequently calls the original function through the previously created reference. Finally calling the original function will lure the user into believing nothing special happened

```
function newFoobar(param) {
    out = "== Captured Information ==\n";
    out += param.contents;
    out += "\n==========================";
    oldFoobar(param)
}
```

3. the original function is clobbered to point to the new shim function

```
foobar = newFoobar;
```

Whenever, the clobbered function is called, the shim function is executed instead. In particular, the flow of *on-demand* AJAX applications, where the code is loaded "on-demand", can be hijacked on-the-fly to the attacker's profit.

Usually, CSRF is a *blind* attack in that an attacker cannot see the response of the forged request. But, leveraging the clobbering technique explained above and the fact that JSON is valid JavaScript, an attacker can apply a maliciously crafted constructor to some JSON data. This attack, called JSON hijacking (but also confusingly known as JavaScript hijacking [21]), is constructed using a CSRF request and a clobbered function. JSON [34] is designed for data interchange by serializing objects and transmitting these over a network connection. JSON objects are represented by simple data structures and associative arrays. Since JSON is a subset of JavaScript, JSON text can

be parsed into an object by invoking the `eval()` function, which results to be handy. However, this should never be done with untrusted data because of obvious security reasons. It is usually recommended to make use of a JSON parser that will accept only valid JSON. On the other hand, JSON data are prone to hijacking. Typically, an unsuspecting victim is lured into accessing a malicious web page containing two distinct scripts:

1. one script has its `src` property pointing to a web page, in a domain the user is logged into, returning a JSON array;
2. another script clobbers the `Array()` constructor in order to transmit the JSON array back to the attacker's domain.

JSON hijacking can also be carried out on JSON objects since the `Object()` constructor can also be clobbered. A JSON object literal is not valid JavaScript by itself though. Still, JSON object literal inside of parentheses are valid JavaScript [67].

2.2.3 The XMLHttpRequest Object and Related Threats

The `XMLHttpRequest` object was created by Mozilla as a JavaScript object interfacing with the nsIXMLHttpRequest built in the Gecko layout engine. This Gecko interface is an implementation of the IXMLHTTPRequest interface from Microsoft. The XHR has been the de facto standard in asynchronous HTTP transactions. It allows sending HTTP(S) requests to servers and loading responses into the scripting language. Exchange data format can be XML or plain text that can be formatted as JSON and later evaluated in JavaScript.

Attackers have been developing more elaborated classes of attacks since the XHR object loads the response back to its issuing script. The XHR objects does abide by the SOP though, but has been used in attacks where an XSS vulnerability was already present. At the time the XSS vulnerability is exploited, the attack becomes a local one making the XHR objects usable for asynchronous HTTP control [48]. That way, the XSS vulnerability is used as a proxy into the targeted domain as will be explained later in this section.

Recently, the World Wide Web Consortium (W3C) has proposed to enhance the `XMLHttpRequest` object with new features such as cross-origin requests [141].

2.2.4 On JavaScript Malware and Related Threats

Typical Web 2.0 applications feature SNS (Social Network Services), wikis, blogs, media sharing and broadcasting, webmails, online gaming, different types of mashup websites. The specificity of such applications is indeed crucial to the processing of new classes of attacks: the level of participation from users expands the number of injection points while the scale of communities allows for massive and fast propagation of malware. Additionally, cross-origin communication gives the opportunity to attackers to perform pivot attacks where, instead of directly attacking the targeted website, they exploit a more vulnerable domain, which performs cross-domain communication with the targeted website.

As we have seen, JavaScript, and when possible using the XHR object, has the potential to generate attacks more powerful than what we used to experience. Attacks are stealthier and more flexible since the attack is not a blind one anymore. Especially, the XHR object allows connections to persist and give the attacker access to the response contents. For that reason, malicious pieces of code written in JavaScript have become more elaborated, performing more tasks, targeting a wider scope, without disturbing the user's experience. Some security researchers have taken a closer look to what can be done to exploit an XSS vulnerability at its fullest [63, 66] and developed JS programs that resemble traditional malware, thus leading to the coined word *JavaScript malware*. And under the name of JS malware are understood various scripts that encompass reconnaissance, fingerprinting, authentication bruteforcing, proxying, etc. Leveraging the diversity of JavaScript malware, it is likely that an attacker can carry out a complete attack scenario from reconnaissance to botnet operation.

JS malware has been thoroughly surveyed by Martin Johns [79] so this section will not detail every single attack that he described, but shall rather concentrate on client-side JavaScript and especially XHR-based occurrences. Aside from displaying behaviors to common malware, JS malware characteristics can be summarized as:

- stealthiness: in order to thrive, it is necessary that the victim's browser, in particular the web page containing the malicious script, be open as long as possible (although not always true). JS malware deceives both the server and the user's vigilance through the use of various techniques to conceal its activity: hidden iframes, XHR requests (when possible), forged requests that resemble user-generated ones and need no user's explicit action to be authenticated;

- polymorphism: JS malware evades common signature-based detectors as well as

hinder the work of analysts by assuming polymorphic shapes. Common encoding techniques are sufficient to bypass filters on the server-side. Obfuscation techniques add a level of protection to malicious scripts by incorporating anti-analysis and cloaking techniques;

- scalability: thanks to the dynamic properties of JavaScript, JS malware can scale to constraints of the environment it is injected into. Techniques such as XSS channel and JS clobbering provides JS malware with the ability to maintain command channels to update its set of functionalities and overwrite its own functions to reduce its footprint. It is also able to rewrite application-defined functions to hijack their flow.

XSS/CSRF Amplification. The XSS attack is no longer constrained to only a single task such as stealing cookies, private information or user's inputs but can also make requests for resources and read the contents of the response, in a seamless manner, with the browser automatically adding authentication information. From then, a user initiating a single request can actually produce several requests in a sequential way, or in parallel, and wait for the responses to trigger one or several subsequent stages.

In recent days, malware has been witnessed running upon social networks in a way that resembles well-known Internet worms. Finding an XSS or a CSRF vulnerability on top of a social network is sufficient indeed to launch a large-scale attack. And the fact that XHR is constrained by the SOP is no more a problem if the whole JS malware payload is injected within the target website. However, web-based malware only persist on top of a browser, more precisely a user's session and ends once the user closes the web page. On the contrary, XSS worms mimic the typical autonomous worm behavior of reproduction and propagation.

John Jean demonstrated the power of CSRF and XSS worms respectively on top of the Facebook website [78]. By leveraging a couple of XSS vulnerabilities on top of the Touch and mobile versions of the Facebook website, the author designed a false Facebook application (to social-engineer users into accessing it), which is actually a CSRF worm. The CSRF worm takes advantage of a CSRF vulnerability of the mobile version of the *Like* button to obtain a user's ID. The author also demonstrated a second worm, an XSS worm leveraging an XSS vulnerability present on both Touch and mobile versions of the Facebook website, affecting the functionality to access external pages. These two social network worms illustrate very well the capabilities of JS malware: performing several commands silently and automatically.

Reconnaissance Malware. As Martin Johns defines it [79], *JavaScript malware describes attacks that abuse the browser's capabilities to execute malicious script-code within the victim's local execution context*. As stated earlier, the SOP prevents a domain from arbitrarily including contents from another domain and accessing data and properties. For the sake of allowing cross-domain communications, Johns remarks that the SOP is relaxed for some tags, namely the `<iframe>`, `<img>` and `<script>` tags, with capabilities conferred to the user that differ from one browser to another. This loophole allows, to a certain extent, an attacker to intercept events, at the time of the inclusion of the external domain element, as well as, the properties of the element, after inclusion. A JS script can thereby make requests to remote contents but still, there is no way to see the response. Grossman [63], then Hoffman [66] demonstrated what was later denominated as the Basic Reconnaissance Attack (BRA) [81, 79]: by leveraging the event model of JavaScript, it is possible to conduct scanning or fingerprinting tasks at the application level, like an attacker would do at the network level.

Grossman and Niedzialkowski showed that we can push the envelope to fingerprint the detected Web servers by requesting images that were typical of some vendors' platforms.

Attack Proxy. As stated above, the range of attacks that can be performed on top of a browser is varied. Once hijacked, a browser can be forced to maintain a connection with a remote domain, and subsequently proxy more attacks. Typically, mashup applications, which aggregate the contents of one or several applications from different domains, provide malicious developers with the opportunity to attack other domains through the mashup, provided there is no sufficient isolation. Attackers can also seek to hijack external services featured in popular websites, in so-called pivot attacks, to inject a malicious code that will be served to countless of users. Hoffman demonstrated a type of attack where the Google Translate service was used as a proxy to allow contents from two different domains to communicate under Google's domain [66].

More generally, Anton Rager [116] showed that chaining two XSS vulnerabilities was sufficient to obtain an *XSS channel*, which communicates with a remote controller loaded through the second vulnerability. Manipulating the concept of iframe remote scripting [30], Rager builds a command channel that awaits commands from his XSS proxy tool, which handles script inclusions and command communications through 3 iframes.

Eventually, a tool allowing to build a complete *Command & Control* (C&C) system was developed by Benjamin Mossé [101]. Browser-Rider is a client/server web appli-

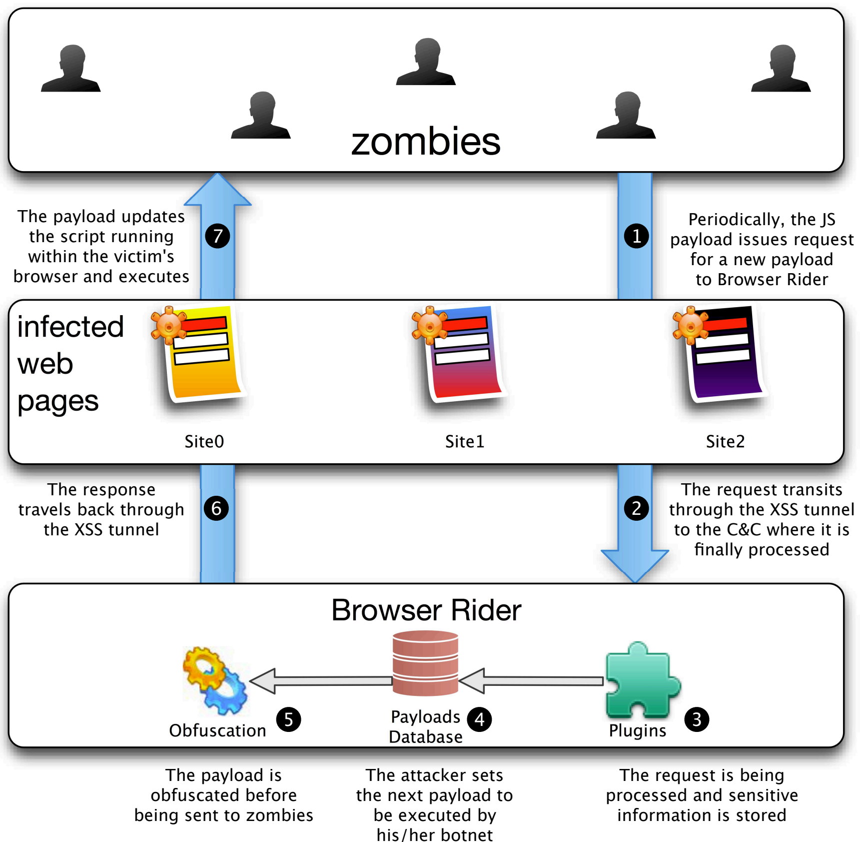

Figure 3. Workflow of the Browser-Rider botnet

cation that acts as a C&C for what has been dubbed an XSS botnet. Browser-Rider is constituted of a PHP server that handles the requests issued by remote zombies, manages payloads to be sent back, as well as the obfuscation and compression of these payloads. Browser-Rider also features a web-based user interface that allows an attacker to monitor her botnet. The connection with Browser-Rider is maintained thanks to the first payload injected to the vulnerable website, which is a simple reloader.

2.3 Attacks against Browsers

Some client-side attacks are not targeting applications but rather the browser and its associated plugins. Depending on the browser configuration, a malicious web page can tweak the attack vector to launch itself or spare browsers not targeted by the

specific attack, in order to prevent detection. Such behavior has been more frequently witnessed and the term *cloaking* generally refers to a set of techniques deployed at different layers to prevent disclosing malicious contents to no other than the targeted configurations. Once a malicious web page has made sure the client it runs on is vulnerable, it may target vulnerable assets, either the browser or any installed plugin to infect the underlying system.

2.3.1 Drive-by Download Attacks

Many of the attacks presented in this section usually feature the download and execution of arbitrary code. Contrary to attacks described above, drive-by download attacks utilize JavaScript capabilities to interact with APIs or plugins in order to download a binary malware. Drive-by download attacks leverage two major strategies [44]:

- abusing application programming interfaces (APIs): some APIs provide the possibility to download files from external URLs and to write the downloaded contents to a location of choice in the filesystem;

- exploiting vulnerabilities in browsers and plugins: by exploiting a vulnerability of the browser or in a plugin (that shares the same address space), a malicious script can jump into a shellcode previously loaded. The shellcode is then responsible for the download and execution of additional malware from an external source.

These strategies heavily rely on a scripting language that can be JavaScript or VBScript.

Heap-spray The biggest challenge is to predict the location of the shellcode in memory. A common solution is to prepend the shellcode with a NOP sledge, which is a series of no-operation instructions. Therefore, the attacker does not need to precisely indicate the return address but only estimate a possible address within the NOP sledge. Eventually, the execution will sled down to the shellcode. To better increase the probability of landing into the NOP sledge, attackers have been developing a technique to literally spray the heap. *Heap spraying* has been widely used since 2005 in web browser attacks: a typical script will assign copies of a string through a loop to fill large portions of the browser's heap memory as depicted in Figure 4. The first loop is responsible for constructing the NOP sledge and shellcode. The second actually sprays the heap 200 times by allocating the NOP sledge and shellcode to an array. Some techniques have been also designed to manipulate the heap to ensure the control flow will be redirected to the sprayed heap [127].

```
var nop=unescape("%u9090%u9090");

while(nop.length<=0x100000/2) {
            nop+=nop;
}

nop=nop.substring(0,0x100000/2-32/2-4/2-shellcode.length-2/2);

var x=new Array();

for(var i=0;i<200;i++) {
    x[i]=nop+shellcode;
}
```

Figure 4. Loop spraying the heap memory with a shellcode preceded by a NOP sledge

Heap spray is itself not an attack and does not exploit any vulnerability.

2.3.2 Web Cloaking

The term "cloaking" was originally employed by search engine optimization (SEO) and applies to the process of serving "optimized" contents to the search engines, which differ from those served to users. As stated previously, malware usually thrives by preventing detection and going unnoticed. It is therefore necessary that a malicious web page refrains from returning malicious contents that may get trapped by some detectors or prompt an error on an inappropriate browser configuration. Cloaking is therefore a set of techniques to distinguish between real browsers and automated user-agents, as well as, between vulnerable and non-vulnerable browser configurations. A recent survey [142] on cloaking distinguishes 4 main types of cloaking:

- repeat cloaking: distinguishes between first-time visitors and nth-time visitors based on states stored either on the client or the server. First-timers are presented with potentially malicious contents while user-agents that revisit the page, being prone to be crawlers, are given a benign page;
- user-agent cloaking: `User-Agent` strings can provide as much information as the browser name, version, build and rendering engine, the platform and its operating

system, and the locale. This allows not only to easily detect search engine bots and crawlers that publicly advertise themselves, but also to filter user-agents more finely and adapt the contents, in particular the language. Cloaking can be further refined to detect the presence or absence of a particular plugin and its configuration;

- referrer cloaking: based on the `Referer` HTTP header, this allows to filter out requests that did not went through a given redirection or a targeted search engine. Also known as *click-through cloaking*, this technique, when combined with repeat cloaking and redirections, can create one-time-use URLs to thwart security researchers;
- IP cloaking: by far the simplest technique. This shuns blacklisted IPs, the ones mapped to search engine or security organizations, in order to serve them benign contents.

It is now a completely integrated feature of any web-based malware.

2.4 Countermeasures against Web Attacks

2.4.1 Validating User Inputs

Web attacks are usually prompted by arbitrary contents that have been injected to the server side through vulnerable web pages. Therefore, a straightforward solution is to perform validation against user-generated inputs. Validation should take place for both request and response processing. HTML tags and undesired characters are usually filtered out. This approach is often termed "blacklisting" and focuses on potential threats. An attacker can however try to bypass any filters by encoding whole or part of the payload [64].

On the other hand, developers can choose to allow a subset of characters, or enforce other properties such as the length of a given input, according to the characteristics of the expected input. This approach is termed "whitelisting", and tends to be closer to the application's behavior, but also suffers from a higher false positive rate. Whitelisting focuses on potential vulnerabilities and is therefore more complex to build. Blacklisting and whitelisting are both inflexible in that they need constant update (whitelisting in a lesser extent). But combining them often results more efficient.

With Web 2.0 applications, the task is even more complex since it is not rare to deal with rich user input. Some applications may actually need to accept scripts or at least XML-like tags as legitimate values. Proper validations include, among others [68]:

- markup language validation: compliance with a scheme or a protocol as for the structure and type of inputs. Contents may also be subject to whitelisting;
- binary file validation: similar to other type of inputs, binary file structure, size and type of the embedded data need to be validated, possibly using whitelisting, although unrecognized structures happen to be discarded;
- JS validation: it is not trivial to tell whether some JS code is malicious or not since a same function can be used both in malicious and benign contexts [104];
- JSON: to address the issue of JSON hijacking presented in an earlier section, it has been suggested that prepending an infinite `for(;;)` loop before the JSON data can efficiently evade a hijacking script.

A framework to validate inputs in web applications has also been proposed in [13].

2.4.2 Security Devices

While secure development lifecycle (SDL) processes cannot be deployed, security may rely on external audit. An alternative and a more proactive approach features the deployment of security devices. However, since firewalls and intrusion detection/prevention systems do not usually operate at the application layer, they might not be appropriate to tackle web application security issues. On the other hand, web application firewalls (WAFs) have been built from the grounds of the HTTP protocol. WAFs are often seen as a temporary solution to buy time for developers to fix potential vulnerabilities discovered in an application. This process is called Just-in-Time Patching (or also Virtual Patching) [119].

Overall, these countermeasures suffer common downsides, such as their inability to provide protection against client-side attacks. Especially in the case of a DOM-based XSS where the malicious payload does not reach the server-side, making server-side protections are helpless.

2.4.3 XSS Defense

Its very high prevalence and the ease to detect vulnerable injection points earned cross-site scripting spot number 2 in the 2010 edition of the OWASP Top 10 [133]. It was already a high-profile attack back in 2007 when Jeremiah Grossman declared [62]:

> XSS is the New Buffer Overflow, JavaScript Malware is the new shell code.

For years, XSS has gained a lot of attention from the academic community prompting several research projects to detect XSS attacks or prevent XSS vulnerabilities from being exploited. We can classify these contributions in four main categories:

- client-side: *these solutions usually analyze HTTP responses or block suspicious HTTP requests.*
- server-side: *these solutions sanitize web documents to be included in HTTP responses.*
- policy-based: *these solutions specify application-specific policies that are enforced on the client-side.*
- vulnerability assessment: *these solutions allow developers to audit their applications.*

All these approaches often demonstrate good results in their evaluation, but it is always surprising that they do not get much deployed in real-life. Aside from the obvious impossibility of securing all the web applications in the world, client-side solutions often suffer from certain drawbacks such as time overhead that harms the user experience. Also, some approaches are obviously vulnerable to prototype hijacking.

2.4.4 CSRF Defense

Cross-site request forgery is another popular web attack that got particular attention in recent years, and especially with the advent of social networks, where it is leveraged to harvest privacy information or propagate XSS worms. This is why it deserves spot number 5 on the OWASP Top 10 [133]. Since it is an undeniably different vulnerability from XSS, distinct countermeasures have also been proposed.

CSRF is not as widely spread as XSS but it has some added value: it allows an attacker to impersonate the victim without having to steal the victim's credentials. The attack has gained significant attention when, in September 2007, Petko Petkov demonstrated a CSRF vulnerability in Gmail [111], Google webmail service, where an attacker can obtain from the victim to set a specific filter to forward all past and future mail transactions to an arbitrary email address. This attack durably installed CSRF as a critical vulnerability in web applications as it decorrelates the will of the user from the action of the browser. Subsequently, many proposals sought to get the user's agreement or infer her intention to filter out unintended requests, likely forged by a third-party.

We can distinguish between server-side approaches that try to harden authentication by distrusting the browser automatic credential binding, and client-side approaches that apply heuristics to filter out between intended and unintended requests.

Traditional server-side CSRF protection is the recommended CSRF token, that is, an additional secret that is shared between the application and the user, which is user-specific and attached to forms and links [124]. Most server-side protections follow more or less this trend. An early solution was NoForge [82] that was implemented as a proxy intercepting authenticated requests, that is, requests bearing the session ID and then checking whether that session ID has any token associated to it. In fact, for any live session, the proxy generates and attaches an extra token to links and forms that will be sent back if the request is not forged.

On the other hand, client-side and browser-based solutions, being application-agnostic, necessarily form decisions on the user's intention. Johns and Winter [80] were the first to propose a solution that differs from the common extra token solution in 2006. Their local proxy solution would try to distinguish suspicious requests from *entitled requests*, that is, requests initiated because of the user's interaction with the web page. They designed a basic policy where suspicious requests get stripped off from authentication credentials.

Aside from technical considerations, some researchers have looked into hardening protocols to ensure safe communications. Proposals were made to implement intended request checking based on the `Referer` HTTP header, an optional header that features the domain of origin of the request. In particular, it was one of the defense measures proposed against login-CSRF [10], a special flavor of CSRF attack where the forged request is actually made in the name of the attacker instead of the user.

Measures against XSS and CSRF are often specific and may fail in some particular cases. Obviously, defending against web malware implies that previous protections fail or that the user is in presence of a malicious website, rendering protections against XSS or CSRF ineffective.

2.5 Countermeasures against Web Malware

The Web is made of a countless number of domains (and even domain-less servers) that provide applications, vulnerable or not, as well as many malicious domains. There is therefore some probability that a user will encounter, in the course of its everyday business on the Web, malicious contents. And only visiting deemed benign websites will not prevent a user from being tricked into phishing events or pivot attacks [61]

where an injection attack takes place in some domain's contents that are included into the targeted web page. Thus, attacks targeting the user or the browser definitely need a more specific response.

Since many threats are incumbent to the JavaScript language, a radical measure is to disable JavaScript in web pages. But disabling JavaScript highly affects the user experience as many web applications rely on JavaScript, so this is not a usable solution. Browser plugins or extensions, such as the popular NoScript [92], allow a user to enable JavaScript only on trusted web pages specified by the user. Once more, the security of the user relies on the user's decision to trust a web page or not.

Security decisions are critical and complex, especially for a non-expert user that may not understand the consequences her actions may have for herself and potentially for others. In recent years, many researchers from academia and industry have proposed and implemented countermeasures on the client-side to prevent malicious contents from hijacking the control flow of an application and relieve the user from making any security decisions. To the best of our knowledge, no browser extension offers full analysis of JavaScript programs. Indeed, most of these are execution-based and cannot be deployed in the browser itself. Analysis platforms such as WEPAWET [31] or jsunpack [65] offer offline processing of files or URLs provided by the user. The analysis usually takes from seconds to minutes and features a detailed report of the found vulnerabilities. However, this seems not usable as a safebrowsing solution and blacklisting services are privileged since they are transparent to the user. Recent advances in client-side analysis of web malware are not covered here but detailed later in this dissertation, in Section 11.

3. Problem Analysis

> Human instinct lags in most of the places where cyberspace is swelling and ramifying.
>
> Ari Juels

Despite a late but growing interest in web malware issues, their mechanisms have not been studied and addressed equally. Although web malware has been demonstrated to perform like common malware [32], common countermeasures are not always considering the different stages of an attack, eventually failing to detect or prevent hazard [117].

In this section, will be covered the consecutive stages unfolded during a web malware attack as well as the techniques used to evade detection. In particular, detectors often struggle with redirection pages when not directly thwarted by cloaking techniques. On the other hand, deobfuscation is still underestimated and many approaches cannot be applied to the user because of their lack of timeliness. As a matter of fact, the present proposal seeks to address these limitations.

3.1 Methodology of Web Malware

Web malware attacks usually carry out the following pattern:

1. after determining the fingerprint of the browser, an appropriate malicious script is downloaded to the browser;
2. the script needs to be first deobfuscated prior to be executed;
3. once the original code has been recovered, its execution will first yield to some preliminary stage in order to "land" the exploit safely;
4. upon exploitation of one or several vulnerabilities, the attack is completed.

Although, the above steps are presented sequentially, it should be understood that these steps may be repeated several times, leading to an intricate interleaving.

3.1.1 Redirection and Cloaking

Cloaking, as explained in Section 2.3.2, attempts to fingerprint the browser's *personality*, that is, the user-agent information and other installed plugins. Cloaking satisfies two requirements of modern malware:

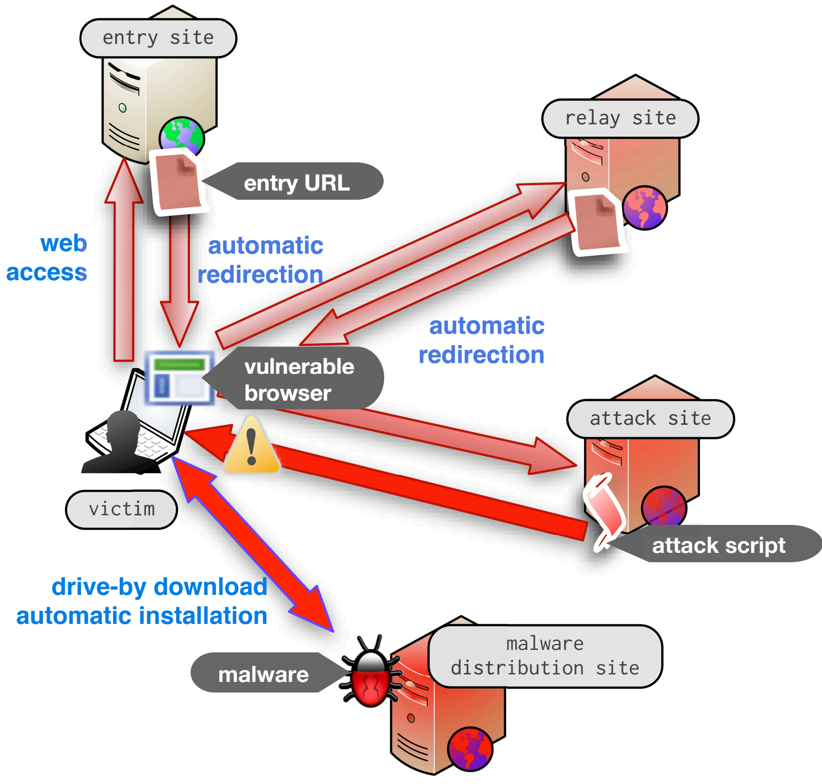

Figure 5. Anatomy of a drive-by download attack

- it reduces the target range to the vulnerable subset of browser personalities, evading possible disclosure of the ongoing attack;
- it allows tweaking the attack to the target's environment.

Redirection is complementary to cloaking: it improves stealthiness by distributing the malicious code across multiple web pages or origins, making a single piece of code look innocuous. Additionally, redirection is often performed silently as a user is made to believe she is browsing a single web page while several iframes download contents from different domains. The victim's browser would be consecutively led from a trap page to a vulnerable web page, which is used as a proxy for attack and malware distribution websites in a typical drive-by scenario (see Figure 5).

As a matter of fact, the window inside the browser does not roam from domain to domain but rather downloads contents pointed by iframe or script inclusions. Most of

the scripts are highly obfuscated and may not be readily executable by the browser's JavaScript engine.

3.1.2 Deobfuscation

Obfuscation is used to evade detection by string-matching filters and can be implemented in various ways ranging from simple encoding to complex full-blown encryption schemes (more details are provided in Section 6). As observed by Cova et al. [32], deobfuscation techniques often resort to dynamic code generation and execution. A single script can usually be obfuscated through several layers of obfuscation and redirection, that is, the script is being divided in multiple pieces, each one being independently obfuscated. Although an obfuscated script will eventually get deobfuscated prior to being executed, one should not expect that a simple execution would yield the cleartext. In fact, even though encrypted contents are to be decoded through a routine (possibly relying on a key), deobfuscation is further complicated through the combination of obfuscation schemes with redirection and cloaking. Encrypted contents, the decoding routine and the key may be available in different domains and only to specific browser personalities.

3.1.3 Environment Preparation

This stage is specific to drive-by download attacks where the browser's memory is manipulated in order to hijack the browser's execution and execute arbitrary code. In other web malware where binary malware is not "planted", environment preparation is minimal or even non-existent. In cases where drive-by download attacks target memory corruption vulnerabilities, arbitrary code (also known as *shellcode*) is injected into the browser's memory, as a first step, through legitimate string initialization operations. Then, a vulnerability is exploited in the browser or one of its plugins to hijack the flow of execution to the shellcode.

Other attacks that abuse API functionalities generate some objects that will be used during exploitation such as an HTTP connection handler, a vulnerable object that allows untrusted data to be written to the disk, and an object that has execution privileges.

Environment preparation can be either available with the exploit code and obfuscated together, or the output of a distinct deobfuscation.

3.1.4 Exploitation

The attack is complete when a vulnerability has been exploited to the benefit of the attacker. Drive-by download attacks end when the browser is finally forced to download malicious contents from a malware distribution site and install it to the victim's computer filesystem, or even execute it within the browser environment. On the other hand, elaborated XSS-based malware will "plant" several scripts within the browser's scripting environment of the trap page and exploit the browser's capabilities to mount staged attacks. Exploit scripts are as varied as the attacker's intents and even more varied in shapes as they are polymorphic.

Web malware do not thoroughly follow all these steps but may rather recursively repeat these steps through several layers of redirection, cloaking, obfuscation and execution. These combinations impose several constraints on detectors that, if failed to be dealt with, will allow malware to either evade detection or even reach the client-side.

3.2 Circumventing Current Web Malware Detection

According to a technical report [117] published in July 2011, four of the most prevalent solutions in terms of web malware detection present weaknesses that lead to partial or total circumvention of these. Obviously, their prevalence does not presume of their efficiency. The four solutions considered in this survey encompass a various range of approaches and technologies as presented in Table 2. Despite the obvious downsides of some of the solutions, they seemed to be widely used. What follows is partly a synthesis of important results and conclusions of this survey.

3.2.1 Virtual Machine Honeypots

Virtual machine based detectors provide complete virtualization of a system that monitors changes to the operating system as a whole. Such system runs as a blackbox and can detect attacks against unknown vulnerabilities. Honeypots are however limited in their ability to precisely identify the resource that triggered an exploit or the vulnerability that has been exploited, especially when the targeted vulnerability is not present.

Honeypots are passive per se and attackers have taken advantage of that fact to design attacks that actively require the user's action to be performed. The above-mentioned report has witnessed an increase, even though small, in the number of social engineering attacks, that is, attacks in which an unsuspecting user is enticed into doing

Table 2. Qualitative comparison of some web malware detectors

Criteria	VM-based	Emulation	Reputation	Antivirus
type	passive	active	passive	passive
analysis	dynamic	hybrid	static	hybrid
accuracy	medium	high	low	low
level of details	low	high	low	medium
scalability	low	good	good	medium
overhead	CPU / time	time	n/a	CPU / time
vulnerabilities				
– update	No	Partly	Yes	Yes
– redirection	No	No	Yes	No
– cloaking	Yes	Partly	No	No
– obfuscation	No	Partly	No	Yes
availability	offline	offline	online	online
usability	low	low	good	medium

a sequence of actions. This method is a type of cloaking that thwart automated VM-based detectors as it only reveals the malicious content after the user's interaction. Aside from that, it is important to notice that a honeypot only instantiates a given OS with a given browser personality, which means that one should set up several virtual machines to monitor distinct environments. Doing so further complicates the administrator's task as managing multiple VMs containing different combinations of exploitable software is a tedious task.

SpyProxy [100] was one such proxy that intercepted the HTTP flow in order to analyze the output of executing a web page. The proxy solution would sit between a user's browser and the Internet and execute web pages to detect any exploitation. If the virtual machine detects an attack, the user is shown a warning message and the web page is blocked. On the contrary, the web page is forwarded but a second execution on the user's browser will fetch again contents, in particular scripts. For that matter, authors observed that their system were vulnerable to non-deterministic execution where a displayed web page could feature only benign contents at one time, and malicious contents another time. This obviously implies some time overhead that can harm the user experience.

3.2.2 Browser Emulation

Originally proposed to address the shortcoming of VM-based honeypots, browser emulators take a more active look on web malware detection since they are designed to contain and analyze their execution. Browser emulators usually provide emulation of all basic browser functionalities, as well as, the script engine and can even instantiate fake ActiveX objects, for example. During execution of a web page in the emulated browser, features are extracted to detect any abnormal behavior that might indicate a possible exploitation. This allows for precise identification of the vulnerability and even the recovery of the chain of causality that led to the exploit, that is, the sequence of HTTP transactions between the emulated browser and the attack sites.

However, in the case the personality of the browser does not fit the exploit's target, or if the targeted vulnerability is absent from the emulated environment, the malicious web page will not be detected. Additionally, attackers also design their malicious code using heavy obfuscation that may fail to decode itself in an emulated environment, or leverage small quirks between browser behaviors in order to evade detection. Browser emulators also need continuous update to provide exploitable environments.

JSAND [32], integrated to the malware analysis platform website WEPAWET [31], is based on the HtmlUnit [15] browser emulator, a Java-based "GUI-less browser" that embeds the Rhino [16] JavaScript engine and can impersonate both Internet Explorer and Firefox browsers. According to the authors, the motivation for using HmtlUnit is three-fold:

- it can simulate multiple browser personalities and rub out discrepancies between ECMAScript implementations (JavaScript and JScript namely);
- it can simulate an arbitrary system environment and configuration by accepting any call to ActiveX control or plugin, and loading a logging facility that will keep track of any deeds of the given control, ultimately allowing to detect unknown vulnerability;
- it allows to implement anti-cloaking mechanisms: more precisely, it maximizes the code coverage by forcing the invocation of any defined function that were not called during execution.

WEPAWET is a popular and recognized website but still has failed to detect some obfuscating transformations in the past or has been impeded to deobfuscate some payloads. As useful as it can be for security researchers and analysts, it may fail to analyze

malware when not providing the accurate environment expected by the attack web page. Plus, it also implies some time overhead due to trace analysis and multiple executions.

3.2.3 Reputation-based Detection

Reputation-based detectors are leveraging public blacklists of known malicious pages to prevent unsuspecting users to access these. The maintenance is pretty low as it is basically restricted to blacklisting new domains. Some novel researches have also highlighted interested results in predicting new malicious domains by using DNS.

Since reputation is content-agnostic, it should be coupled to other methods in order to collect domains to blacklist. Attackers have therefore relied to the mass registration of domain names in order to circumvent blacklists. To thrive longer, attackers also set many redirectors and it is not rare that a victim be redirected through several intermediary sites. Overall, domain rotation and redirection can ensure a longer exploitation spree for attackers.

There exist many blacklisting websites and popular ones include the PhishTank [108] initiative, the Badware Busters community [129] (sponsored by some popular Internet stakeholders) and of course, the Google Safe Browsing [60] initiative and the Web of Trust [143].

3.2.4 Signature-based Detection

Signature-based detectors, mostly represented by antivirus, have been available for a long time and it is no wonder that they have also been applied to web malware detection. The main functionality of antiviruses is to scan payloads and look for signs of malice but their task has been thwarted by packing and obfuscation techniques. This has often prompted antivirus vendors to flag as malicious any content that was found packed or (heavily) obfuscated.

By definition, an antivirus needs to be kept up-to-date in order to be efficient. The report [117] shows that attackers tend to integrate quickly newly public vulnerabilities to their exploit kit and that obfuscation can be cited as a criterion to distinguish malicious and benign pages. However, it does not compare malicious obfuscated and benign obfuscated scripts, but rather points out the sophistication of recent obfuscation schemes.

Nowadays, mainstreams computers are shipped with an antivirus as vendors anticipate that users may connect to the Internet and eventually be infected online. Unfor-

tunately, it sometimes happens that a user, frustrated with the load imposed on her machine by the antivirus software, will simply disable the antivirus. Failure to properly update the virus signatures is also another cause of infection, if not to be blamed on the vendor side for not providing signature updates in time.

As evidenced by this synthesis and summarized in Table 2 above, each detector has its field of expertise and can contribute to detection. Therefore, the Google report concludes that combining these 4 approaches would significantly increase the detection rate.

3.3 Issues on Redirection, Cloaking and Obfuscation

As an aspect of the continuous arms race that takes place in the realm of web application security, attackers always try to shift security vendors' expectations while the latter attempt to anticipate next attack trends. Obfuscation has been for long one of the many techniques in the attacker's arsenal since packing has obviously paved the way to nowadays' obfuscating transformations. Yet, security researchers have disregarded obfuscation as a threat though Provos et al. [115] have demonstrated that obfuscation is not an indicator of malice. It is true that, in most cases, obfuscation will be easily cancelled to reveal the script in cleartext but trends showed a growing sophistication coupled with redirection and cloaking to circumvent state-of-the-art detectors. Obfuscation is not as trivial as it used to be and even simple encoding schemes that rely on browser or plugin tricks have proven able to evade detection by even advanced detectors such as WEPAWET. For example, a known anti-analysis trick features the `arguments.callee` object, which returns the body of the function in where it is called. Using this peculiar object, an attacker can encrypt a malicious script to obfuscate it and prevent its deobfuscation whenever the contents of the function are tampered, which usually happen when instrumented for analysis.

In general, modern obfuscation techniques can be seen as a subset of cloaking techniques since they allow to filter out detectors and since decryption keys are often made up from the concatenation of pieces of information relevant to the targeted system. For example, some fingerprinting scripts dynamically generate a string by concatenating version numbers of available plugins and send a GET request to the malware distribution site with such string as a parameter. The malware distribution site will return an exploit corresponding to the fingerprinted environment if available. While cloaking reduces the attack space of attackers, its obvious goal is to ensure an attacker's success by evading detection. Therefore, even though some users may be spared from being

infected, security analysts do not have tools to pursue their study and risk being left behind in the arms race. Could this be a lesser evil?

Finally, among the three main evasion techniques discussed in this section, redirection remains the oldest and simplest one. Redirection's original purpose, as stated previously, is to thwart blacklisting and some redirectors can be vulnerable benign sites that are used as proxies. But with constant threats on DNS [37] and recent SSL certificate hijacking [105], the notion of trust on the web is growing weaker.

Nonetheless, cloaking represents the greatest challenge now, as highlighted by the Google report and future efforts should be directed to tackle this issue.

3.4 Issues on Usability of Browser Protection

On a related note, it is also questionable how usable a security solution is, furthermore when it is intended to be deployed on the client-side. As stated previously, it is not always desirable to make users bear the burden of making security decisions. Yet, users might not want that security policies be imposed on them. It has often be the case with users uninstalling antivirus software because its was interfering with their user experience. A usable solution can be defined as one that has a minimal footprint on user experience.

Based on qualitative criteria in Table 2, the usability of web malware detectors cited by the Google report has been evaluated. In particular, based on the performance overheads they impose on the user and their availability or unavailability at hand, a general comment can be made as to their possible integration in the user's secure browsing experience. While emulation seems by far the best performing solution able to detect even complex attacks, the fact that it is disconnected from the user's control flow is a huge disadvantage. Indeed, it is not reasonable to have a user check pages in an emulator each time she doubts the contents of a web page, let alone several ones in a given domain. Similarly, VM-based solutions suffer from the same drawback as their successor. Moreover, both methods often need several passes to decide on the malice of a given web page, which delays the user browsing experience.

On the other hand, while reputation- and signature-based methods are not as efficient, they offer almost seamless integration in the user browsing experience.

Therefore, it seems important to thoroughly consider usability issues in order to design a solution able to integrate within the end-user browsing environment, without harming her experience. This translates into requirements for a realtime, on-the-wire solution that does not impose any burden to the browser. The present solution attempts

to enhance these particular points, starting from its design.

3.5 Issues on Offline Analysis and Execution-based Side-effects

To conclude with the description of the problem statement, the present proposal wishes to stress out the opposition between static and dynamic analysis. Dynamic program analysis accounts for most of the proposed and implemented systems presented in Section 2. Although static code analysis is sometimes used as a complementary approach, it is quite surprising that it has not gotten more attention given that dynamic analysis reveals to be useful after several executions. Moreover, static analysis seems to be more fit when dealing with source code, which is the case with JavaScript programs. One may argue that dynamic analysis takes advantage of machine learning methods to reduce the range of inputs to be tested but it still incurs side-effects inherent to execution. In addition, since cloaking and obfuscation seem to be the trend, it is to be feared that execution might disrupt and divert the program's control flow. On the contrary, static code analysis provides arbitrary path selection and analysis.

Nonetheless, cloaking remains a problem for both approaches, especially in the case of the remote browser personality cloaking technique mentioned above. In fact, in such case, a static tool will have no chance but to provide a given personality to the attack website and depending on whether this personality is targeted, the response will contain malicious contents or not. Finally, another drawback that is a consequence of execution side-effects is the necessary containment of the said execution. This requires that the analysis be done afterwards and then is incompatible with online processing.

This shall prevent that execution-based solutions be integrated in front of a user's browser for realtime protection.

3.6 Summary

State-of-the-art web attacks do not only exploit vulnerabilities but also strive to conceal their malicious intents as much as possible. Recent surveys have shown that web malware has been successful in thwarting current web malware detectors. In particular, reputation-based detectors are abused by the massive number of domains and redirectors that characterize attack networks. While signature-based detectors are easily circumvented by the polymorphic characteristic of obfuscated malware, cloaking techniques prevent emulation-based detectors from triggering malicious behavior. As for VM-based detectors, they are far from being adequate since they do not detect accurately web attacks but rather exploits at the OS level.

An important observation is the lack of actual deployment of web malware detectors in the browser due to the computing intensive nature of actual approaches. VM-based and emulation-based actually incur delays because of the multiple executions they require. Moreover, they are often deployed as offline detectors, which degrades their usability in the scope of client-side protection.

These obstacles have prompted us to consider an alternative solution to ensure a safe browsing experience to end users. The next section describes this solution to offer an online, mostly execution-less and safe detector of client-side web malware.

4. Overview of the Proposed System

> Those who cannot remember the past are condemned to repeat it.
>
> George Santayana

The proposed system is a client-side proxy solution that sits in front of the user and is responsible for intercepting HTTP communications. Its goal is to prevent the exploitation of the user's browser while accessing the World Wide Web. The proxy performs three main operations, as detailed in Figure 6: prefetching of scripts and related contents (for example, iframes, suspicious links and images), automated deobfuscation of obfuscated scripts, and reverse engineering of the intents of a deobfuscated script. Since the system ultimately model the intents as an object sequence diagram of the script, the system has been named **(sak_mis)**, which stands for Static and Automated Knowledge-based Modeling of (malicious) Intents in web Scripts.

Before detailing (sak_mis) internals, we wish to form a few assumptions on which we are basing our work:

- Web 2.0 applications provide client-side dynamic contents and often rely on complex cross-domain dataflow;
- Web malware are occasionally injected into Web 2.0 applications through an unpatched cross-site vulnerability;
- Unsuspecting users fall victim to phishing attacks or other social-engineering based attacks on top of social networks;
- Web malware often follow four common steps (redirection and cloaking, deobfuscation, environment preparation, and exploitation) but not necessarily in that order. It is important to note that the steps are not always consecutive and necessarily found in that order. Additionally, there may be several iterations of such steps;
- Redirection and cloaking techniques are performed using links present in the original page or generated during deobfuscation;
- Obfuscated contents are necessarily going to be deobfuscated. However, deobfuscation may be coupled to prior or later attack steps, thwarting the hooking of *critical sinks*. Critical sinks are points in the source code where data is used with

some privilege, usually parsed and consequently executed. A common example in the JS language is the `eval()` function, which evaluates strings into executable JS code;

- Obfuscating transformations, or at least obfuscation toolkits, constitute a finite set;
- An unobfuscated or a completely deobfuscated script does not feature any obfuscated contents or contents to be embedded and is directly executable as is;
- There is a finite number of commonly used APIs, although each browser vendor may feature different ones. We also assume that there is a correspondence between implementations of JavaScript and JScript. In particular, we can form an assumption of polymorphism where a function can be implemented in several ways using different libraries or APIs.

The workflow of (sak_mis) is depicted in Figure 6 as follows:

1. (sak_mis) is a proxy that intercepts HTTP requests issued by the end user and subsequent responses returned by the server;
2. upon reception of an HTTP response, the proxy commences the *prefetching* stage:
3. the requested web page is parsed to detect script inclusions, links, and potential malicious locations (iframes, images, etc.). Script contents are then retrieved (prefetching) and inlined into the original web page;
4. if newly downloaded contents also contain links or inclusions to contents of interest, prefetching is also performed on these contents. This scenario also applies when new inclusions are uncovered after deobfuscation;
5. once prefetching is completed, the aggregated script page is sent to an external application server;
6. the application server is responsible for automating *deobfuscation*: obfuscated contents and decoding routines are extracted first. The deobfuscation stage of the attack scenario is emulated by the server. The process is repeated in case the deobfuscation generates new obfuscated contents;
7. once scripting contents can be directly interpreted by the machine, the *decision* module applies static analysis to extract a model of the script's intents. This

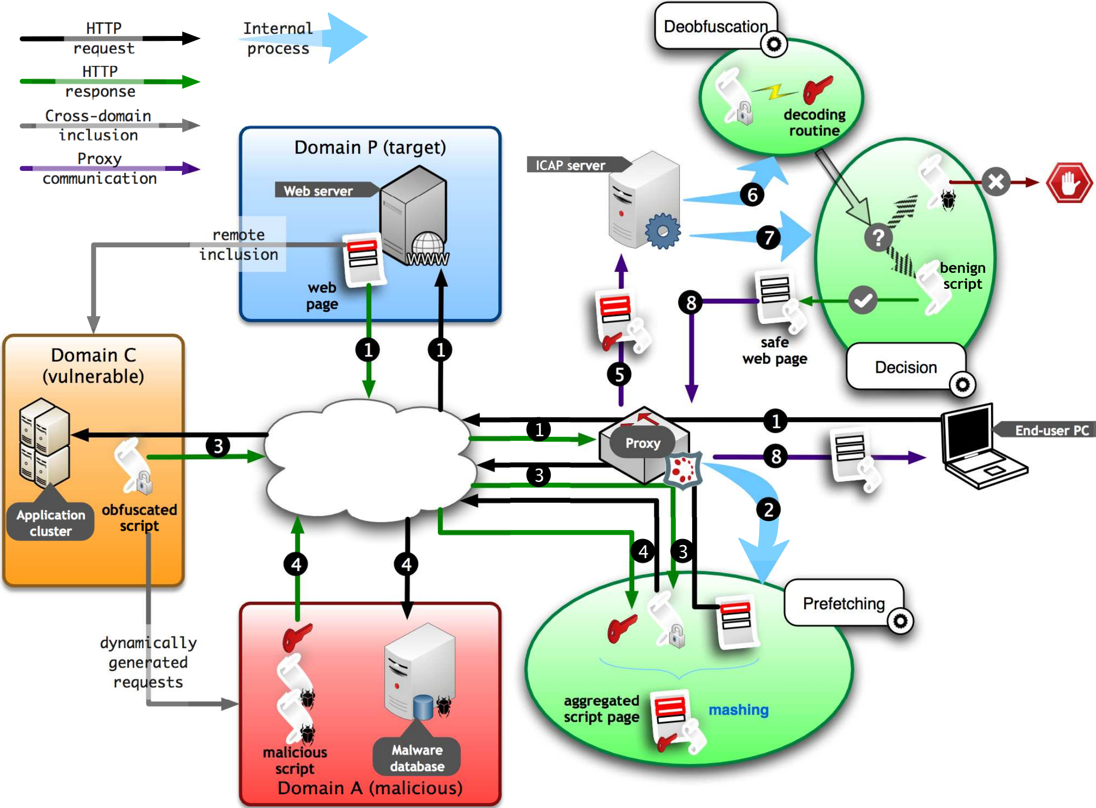

Figure 6. Overview of the (sak_mis) system to counter Web malware

model is then looked up in a knowledge base in order to infer whether it is a model of malicious intents or not;

8. in case the script is benign, the deobfuscated script is injected back into the original web page and served to the end user.

4.1 Requirements

The present proposal stems from the different observations made in the previous section and attempts to satisfy the following general-purpose requirements:

1. provide realtime online analysis of JavaScript contents;

2. avoid executing JavaScript contents, as much as possible;

3. arbitrarily fetch, as much as possible, JavaScript contents;

4. provide a safe automation of JavaScript deobfuscation;

5. isolate the user client from potentially malicious contents;

6. reduce, as much as possible, any disruption to the user experience.

The first requirement is paramount to this proposal that seeks to protect the user while she is browsing the Web. The second requirement is also another specificity of this proposal that seeks to reduce the footprint of JavaScript execution. The third and fourth requirements result from the previous one: the JavaScript execution control flow is bypassed and arbitrary actions, therefore, can and should be taken to perform analysis. The fifth and sixth requirements ensure the usability and safety of the proposed system.

4.2 Intercepting HTTP Transactions

As a proxy, (sak_mis) intercepts the network traffic. Since the focus is on the web application layer, (sak_mis) interposes between the Web and the end-user's browser, parsing HTTP responses and generating arbitrary HTTP requests. The proxy may be limited against HTTPS transactions since HTTP payloads are encrypted. It is, however, possible to delegate HTTPS processing to the proxy, leading to (sak_mis) accommodating HTTPS communication as well. Privacy should not be a concern since no data are preserved even if the proxy has effective access to any sensitive data contained in HTTP payloads. It should be noted that some attacks such as JSON hijacking can actually corrupt the operation of the JavaScript engine on the client-side. Therefore, (sak_mis) having access to sensitive data contained in JavaScript (JSON included) or to-be-embedded DOM contents is a lesser evil.

The choice of a client-side proxy solution is not insignificant and actually satisfies two requirements of the proposal:

- protecting the user cannot be done exclusively on the server-side since intending so would mean that every server should be made safe. It is necessary to deploy a solution at the client-side to ensure the safety of the user browsing experience, regardless of the browsed website;

- the browser experience is already hindered by the multiplicity of the processes performed within the browser and it would be unreasonable to impose further

processing overhead to the browser. A proxy acts as a delegate and dedicated hardware can be used to support intensive processing to accelerate additional processing overhead involved by the analysis.

The (sak_mis) proxy is designed to be implemented as an ICAP[47] proxy. The Internet Content Adaptation Protocol (ICAP) has been designed to delegate value-added transformations to surrogate web servers in order to "adapt" web contents to the destination users. In fact, ICAP was originally designed to be deployed on the content provider side where it leverages the highly distributed environment. However, recent applications comprise virus scanning and content filtering, which can be deployed, for example, on the client-side for corporate purposes. The purpose of (sak_mis) is similar: providing JavaScript analysis of HTTP communications for a group of users. In (sak_mis), the ICAP proxy is responsible for prefetching, which is a common proxy feature. The deobfuscation and decision stages, which are out of the scope of expertise of the proxy, are delegated to an ICAP application server.

4.3 Countermeasures against Redirection and Cloaking

Since there is a possibility that a malicious script may be scattered and concealed across several web pages, the *prefetching* stage proactively gathers contents of interest from detected inclusion and link locations (3rd requirement). Here, fetching is an arbitrary process directed to a few selected content types and not an automated sequential process as done by web engines. Indeed, it is ineffective to conduct analysis on a single script, more so when the script is obfuscated using an encoding transformation and the decoding key is absent. A few simple cloaking techniques that can thwart execution-based analysis, by leveraging controls, do expose obfuscated codes or links in the source code. It is, therefore, only a matter of arbitrarily following the attack target path that allows to trigger an attack. Obviously, the system is still vulnerable to advanced cloaking techniques to a certain extent:

- server-side cloaking, especially IP cloaking, which has been noted as the far most resilient to anti-cloaking techniques, can not be circumvented by (sak_mis). For example, once it has been discovered that (sak_mis) is operating in front of a given IP range, nothing can prevent this IP range from being blacklisted by malware distribution networks. However, aside from any intents of performing malware analysis, this does work as a good deterrent;

- fingerprinting where, instead of triggering controls within the same web page,

the browser fingerprint is sent back to the server to be processed. Here, the fingerprint of a browser is the concatenation of several (boolean or not) tests done to identify the type of the browser and the presence of some plugins. The malicious code is finally included in the web page if the browser fits the targeted profile. Although a proxy can falsely assume different browser personalities by arbitrarily tweaking HTTP headers, knowning which personality will trigger the attack is not apparent. It is recommended that (sak_mis) be transparent to avoid any misleading on the nature of the user-agent (5th requirement). That way, it will not be ignored by content providers and prevent any hindrance to the user experience. With the objective of ensuring the user's protection, deterring attack websites from sending malicious contents is sufficient.

Prefetching stands up as a limited, yet effective, solution against cloaking techniques. There is a concern that prefetching might disrupt the user experience in modern web applications. This should be assessed through a comprehensive survey, which is out of the scope of the present proposal.

4.4 Detecting Obfuscation

Another original feature of (sak_mis) is its consideration towards the obfuscation issue, in accordance with what Provos [115] concluded: that is, obfuscation does not make a good indicator of malice since benign web pages also make use of such techniques. Therefore, (sak_mis) attempts to go beyond obfuscation by actually recovering the original cleartext script and decides on its malice. The reader may question the relevance of considering the obfuscation as an issue, since obfuscated contents need to be deobfuscated before exploitation as outlined in the typical attack scenario (cloaking and redirection, deobfuscation, environment preparation, exploitation). Yet, it is the entanglement of these different stages in complex attacks, which spans across several domains, that make the mere hooking of JavaScript execution being potentially ineffective at preventing exploitation. Other side-effects include the following:

- when cloaking controls are implemented in the web page script contents, it is possible to arbitrarily trigger the code. However, execution could possibly miss such control and not trigger the expression of malicious contents;

- executing script contents might trigger attack contents and therefore requires isolation, leading to the design of offline analysis platforms. Therefore, code

reaching the end-user client is necessarily downloaded a second time (this happens in SpyProxy [100], for example).

It is important to notice that most offline analyzers are execution-based, which supports the present claim of a high causality between execution side-effects and isolation / offline design.

For these reasons, applying a static approach seems advantageous over dynamic approach, which lacks in code coverage and safety. However, obfuscation cannot be cancelled statically in cases where the script is encoded or encrypted. This downside stimulated the proposal to emulate the deobfuscation stage. Although several previous research projects offered ways to detect obfuscation [89, 69] or automate deobfuscation [32, 118, 36], it never provided both functionalities. The present proposal actually describes techniques to tackle these two tasks that are later addressed in Sections 7 and 9.

In particular, Section 7 introduces project *ob_asti* (Obfuscation and Abstract Syntax Tree Identification), which attempts to characterize obfuscated scripts by identifying recurring abstract syntax tree (AST) expressions of obfuscating transformations. It has been observed that obfuscation has grown complex and has been heavily used in web malware. Following the monetization of attacks, a real "blackmarket" has emerged with several attack toolkits being developed with the objective to be sold. One common feature of attack toolkits is payload obfuscation. Analysts have witnessed an important number of similarly obfuscated contents, supposedly due to the prominence of a few toolkits (or conversely, the lack of competitive ones). In project ob_asti, we assume that even though string randomization makes obfuscated scripts look different, they indeed share similar structures that can be uncovered by comparing their respective AST. Also, we assume that tools can be classified by identifying characteristic structures in ASTs that are specific to a given tool or obfuscating transformation. Ultimately, it may be possible to speculate on which obfuscating transformations are more recurrent in malicious scripts, and which are in benign scripts.

4.5 Reversing Obfuscation

The second challenge is that of deobfuscation, which is akin to the one of detecting obfuscation. Deobfuscation is the process of cancelling obfuscation (4th requirement), that is, to simplify the code to a form that allows to infer its intent. Constrained to an execution-less environment, it is not possible to apply most of the techniques outlined in previous research works [32, 118]. Reversing obfuscation results impossible with purely

static and formal methods. Project *u_adjet* concentrates on automating deobfuscation through the emulation of decoding routines. In particular, emulation is done by using deduction rules in a membership equational logic framework, namely Maude[25].

Section 9 further discusses this project and stresses actual differences between binary and scripting language obfuscations. Based on observations in cancelling binary obfuscation, project u_adjet attempted to apply similar formal methods to the issue of deobfuscation. Failing to do so, it was deduced that execution was partly needed. In a bid to preserve the integrity of the 2nd requirement, emulation was seen as an acceptable trade-off. Moreover, the formalism (equational membership logic) used in u_adjet is consistent with earlier empirical trials and results in this proposal. In fact, the basic assumption is not different from predecessors in the field: obfuscated contents will eventually be deobfuscated to allow their execution. This can also be interpreted as a termination property for deobfuscation in JavaScript malware. u_adjet stands for User-agent agnostic Automated Deobfuscation of JavaScript by Emulation.

4.6 Analysis of Deobfuscated Script

In the end, every module in (sak_mis), apart from this one, can be regarded as a supporting module to provide an acceptable input to analysis. The goal of (sak_mis) is indeed to provide analysis and decision of scripting malware. However, as it has been often stated, analysis works best on unobfuscated JavaScript. *mi_oos*, which stands for Modeling Intentions of unObfuscated Object-oriented Scripts, deals with reversing the intents planted by the developer in the exploit code of a web page. The models described in Section 11 are actually hybrids of object and sequence UML diagrams, which pertain to two distinct diagram categories: structure diagrams and behavior diagrams respectively. However, the resulting diagram does not thoroughly follow UML specifications and stands as a way to represent intents in a way that can be computed and detected by machines.

The project mi_oos is one of the rare occurrences of static program analysis implementation to web scripting languages. It is usually regarded as more practical to apply execution-based approaches since it supports the whole analysis cycle (fetching, deobfuscation, analysis). On the contrary, applying static analysis requires some adjustments that threaten the 1st requirement. One of the biggest challenges is to maintain the time overhead acceptable to the user to make (sak_mis) usable.

4.7 Recursive Characteristic of the System

On a related note, the reader may have questioned the whole flow of the (sak_mis) proxy given that malicious scripts are often obfuscated through several layers of obfuscation and redirection. Direct consequences are that each deobfuscation may yield more obfuscated or linked contents, which calls for more deobfuscation or prefecthing respectively. This is a feature of (sak_mis), even though it may have been missed in previous tools. However, experienced analysts are well-aware that this characteristic accounts for most of the tediousness of analyzing JavaScript malware. (sak_mis) hopefully features key-technologies that are able to deal with recursive processing, such as prefetching and the Maude framework itself. One point of discussion is to what extent the tasks of prefetching and deobfuscation can be decorrelated. And therefore, how reasonable and sustainable a solution it is to assign each task to a dedicated device. The latter point of discussion is, however, left as an open question to the reader, but it can be, with no doubt, speculated that the trade-off between hardware dedication and time overhead might be difficult to assess, given the multiplicity of existing environments.

4.8 Summary

Based on requirements we drew from reviewing the current attack landscape and the drawbacks of current detectors, we presented the architecture of our proposal in this section. The three functional modules collaborate to provide a mostly execution-less analysis of client-side JavaScript contents in order to detect any malicious intents.

The implementation design relies on an ICAP proxy that intercepts HTTP communication between the end user and the World Wide Web. The ICAP proxy delegates part of the processing to application servers that provide specialized functions that are out of the proxy's areas of expertise. In particular, deciding on the malice of a JavaScript is done on an the intentions extracted from unobfuscated JavaScript code. Unobfuscated JavaScript code is not trivial to obtain without relying on executing JavaScript code. To overcome this obstacle, the first two modules of the (sak_mis) proxy provide obfuscation detection and cancellation, respectively. The details of each module, as well as their underlying background knowledge, are described in subsequent sections after defining the concept of intention in the following section.

5. The Concept of Intention

> My primary interest is not with computer security. I am primarily interested in writing software that works as intended.
>
> Wietse Venema

The behavior of a program can be described by the relationship between the input and the output of the executing program [125]. Therefore, it is possible to draw the normal behavior profile of a program by observing possible outputs of execution. Regardless of the implementation of such profiles, anomaly detection engines are tailored to flag malicious behaviors by detecting any deviation from these normal behavior profiles.

On the contrary, *intention* could be defined as an execution-less behavior in opposition to the term *behavior*. Since it is not possible to make correspond inputs and outputs, intention can only be described by the syntax of the program. However, this does not give any indication concerning the meaning of the program. In a bid to augment such approach, the present proposal draws on advances in semantics. (sak_mis) proposes to express JavaScript programs as hybrid object-sequence diagram. While syntactic analysis can help decompose a program into a set of objects interacting with each other, the role of each object and the flow of data between these objects cannot be deduced without knowledge of the domain, here the programming language reference. The approach is quite liberal and actually borrows from action semantics [102] which has been designed with the aim to make formal semantic specifications better reflect basic concepts of a language, rendering more accessible to developers. In action semantics, the meaning of a programming language is defined by mapping program phrases to *actions*, which are semantic entities that incorporate the performance of computational behavior.

The concept of intention itself originated in philosophy and has been studied and formalized within the well-known model of Belief-Desire-Intention (BDI) [17]. This model was the basis of an extensive research in artificial intelligence, especially agent theory. One of the most influential contributions to this area is the following proposition from Cohen and Levesque [28]:

> Intention is choice with commitment.

Cohen and Levesque develop a formal theory for rational behavior that revolves around the BDI model. Intention of an agent **x** to perform an action **p** is defined as the persistent goal of the agent **x** to reach the state of believing that it will realize the action **p**, which eventually it does. A persistent goal is defined as the belief that the action **p** is false, that is, the action **p** has not been realized. As this first assumption holds, agent **x** chooses worlds in which action **p** will be later performed. Agent **x** is committed to performing action **p** unless it drops on the idea of performing **p** because he believes either that **p** is achieved or that **p** will never be. A comprehensive explanation of this theory is available in [28]. Throughout this dissertation, we will use both the words *intention* and *intent* interchangeably although there is a slight nuance[1].

5.1 Related Work

Among other research goals, later works in agent theory have focused on how machine agents could mimic another agent behavior and especially what role intention can play in the agent behavior. However, discerning intention in human agents is not trivial and raises several issues[7]:

- what kind of information about intentions is actually available in the surface flow of agents' activity?
- which aspects of this structure can be detected?
- what kinds of additional information, if any, might be needed to account for inferring intentions and purposes of agents' activity?
- how can skills be acquired to infer intentions?

Baldwin and Baird also outlined the fact that humans, mostly adults, are able to process continuous action (activity) streams in terms of hierarchical relations that link smaller-level intentions with intentions at higher levels. This is quite challenging for machines at the current state-of-the-art. Nonetheless, research work in imitation has later showed that actually understanding intention was possible for an artificial system. Using a learning method in which the imitator keeps track of the intentions of the initiator, the imitator is able to reproduce a behavior after repeated trials[77]. The computational model relies on a blocks world in which goals are expressed as relations between the blocks. Although the contextual environment is absent from their

[1] *Intention* has a more general meaning referring to a plan someone has in mind while *intent* is more specific and connotes more deliberation.

experiment, a good imitation was achieved, in terms of goals. They did it by focusing solely on the goal, without enforcing the spatiotemporal order of the different steps. However, what works well for imitation in robots may not be applicable elsewhere. When dealing with computer systems, it seems on the contrary that a particular set of steps, ordered in space and time, is necessary to define different intentions toward fufilling a same goal. In fact, an agent, usually human, may reconsider fulfilling a goal.

To account for that fact, successful incursions of intention can be found in computer security literature. For example, BINDER[35] is a host-based extrusion detection software that leverage the intention of a user to detect malware activity. The detection engine needs no prior knowledge of the system and infers user's intentions from the user's activity. Mouse and keyboard events are collected and mapped with processes' activity. Assuming a process is likely to be active shortly after receiving user input, processes that deviate this simple rule may be seen as anomalous and thereby not user-intended. This is more likely to be an extrusion attempt from a malware. The system is also able to learn from false positives and to correct the threshold time for a given process, that is, the delay time a process should not exceed after receiving user inputs to be considered *user-intended*. On a related note, some proposals to counter CSRF attacks also partly rely on heuristics conveying the user's intention [80, 91].

5.2 Representation of the Intent of a Developer in a Program

In the present proposal, the focus is on the developer. (sak_mis) aims to uncover the intents the developer has attached to her program. Thus, it is expected that if an attacker has introduced malice in her program, such malicious intents should be apparent in part of the program. Ways of concealing intents have been discussed previously (cloaking, redirection, obfuscation).

Capturing the intents of a developer is actually one of the early stages of software developments and the most well-known method is probably the usage of UML diagrams. UML (Unified Modeling Language) is used for communication, that is, capturing knowledge (semantics) about a subject and expressing knowledge (syntax) regarding the subject [5]. It can be used complementarily along formal methods to elaborate program specifications but UML is far more favored by developers since it outputs visual models.

Another way a developer can convey her intention is obviously by choosing an explicit naming policy as well as common design patterns. In object-oriented languages, a design pattern designates a recurring programming problem and the common design

towards a solution to this problem. A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design [56]. UML object diagrams are often used to represent design patterns.

Later in software development, a developer can explicit further on her intent by embedding annotations to her code for other developers to read. Comments, as they are universally known, allow to explain complex pieces of code and have also been used in recent programming languages for the automated generation of documentation while being typically ignored by compiler and interpreters.

Comments are useful when several developers share the same code. However, when they design distinct modules independently, they may not have full access to the internals of another module, and should only know how to communicate with this module. The developer needs to know what are the expected inputs and their types, and also what are the expected output of the module they wish to communicate with. Such information is referred to as *interface* or *protocol*.

In this dissertation, we give a tentative definition of the concept of intention:

Definition 1. *The intention of a developer in a program fulfills a programming goal through a sequence of actions*

The goal itself is semantic and specifies a human concept that can be rendered by a valid combination of actions. Actions are defined syntactically by grouping code of a program that collectively achieve a particular functionality.

Here, we have seen some of the many ways for a developer to express her intent and we have, as well, attempted to give our own definition. Now we may have a look at how another developer, with few or no prior knowledge of a program, can understand the computational intent of a program and recover the original intent of a developer.

5.3 On Reverse Engineering

While UML diagrams are used to convey the developer's intents into the program's specifications, reverse engineering is a set of techniques to recover the developer's intents from a given program. Reverse engineering is usually conducted to obtain missing knowledge, ideas, and design philosophy when such information is unavailable [46]. Most common usage of reverse engineering include analyzing the output of binary programs to design an adequate interface, and observing malware to understand how they work. Also, it is used for decompiling and analyzing assembly of commercial

software for purposes as well-intentioned as finding vulnerabilities or as ill-intentioned as cloning protected functionalities.

Another way of thinking places reverse engineering in a program comprehension or maintenance context where it is used to understand what the program is doing. This is true for legacy programs whose code may be available but difficult to understand due to their complexity, especially when some refactoring or extension is targeted. Although the most accurate description of the behavior of a software is its source code, reverse engineering techniques provide a way to extract higher-level views of the system [137]. Knowledge obtained through these reverse engineering techniques are often expressed using the de-facto standard modeling language of UML. Common UML tools provide a "reverse engineering" functionality that generates UML diagrams from source code analysis.

In fact, many researchers got interested into recovering and expressing knowledge by ways of UML diagrams from the source of object-oriented programs. Early works attempted to detect common Gang of Four's design patterns [56] in object-oriented programs by building repositories of common patterns occurring in program codes and mapping these (automatically or manually) to design patterns: the first example was SPOOL [85]. Since class diagrams in UML express the design of the program and sequence diagrams the behavior of the objects of the system, contributions have concentrated on these types of diagrams, with Reveal [94] focusing on the former. Reveal builds class diagrams based on information provided by a parser that builds a symbol table of the analyzed program by identifying name occurrences and definitions, and storing such information with their context. Other class diagram extraction proposals include [135] that combines:

- a static object diagram computation, which exploits the object-flow graph (OFG), that is, the flow propagation to transmit information about the objects that are created up to the fields that reference them;
- a dynamic object diagram computation based on traces obtained from test case executions.

However, a survey [88] at that time showed that many tools failed to adequately represent design abstractions and could only produce simple class diagrams due to a sizable semantic gap between the modeling language, UML, and the programming language. Resulting software models fail to represent the abstract program semantics needed for high-level comprehension, prompting the proposal of mappings [131] between UML and

the programming language (C++) based on syntactic and semantic information of the programming language and domain knowledge of programming conventions.

On the other hand, proposals that reverse-engineer sequence diagrams attempt to express the behavior of the analyzed program with the static approach being more accurate, by representing the whole set of outputs and dynamic approach being more precise with an exact matching between a given input and a given behavior. Static approaches for extracting sequence diagrams can be mainly divided between OFG-based [136] and control-flow based [121]. In the former approach, the objects created by the program and the ones accessible through program variables are inferred from the code. Then, each call to a method is resolved in terms of the source object and the target object involved in the message exchange. By propagating flow inside the OFG, it is possible to statically approximate the objects created by a program and their interactions. The latter approach maps intraprocedural flow of controls to UML using UML extensions that defines a richer set of control-flow primitives for sequence diagrams [107].

Dynamic approaches are often similar in that they instrument the source code to generate different execution traces, which information is then used to build sequence diagrams. In [39], the control-set primitives used in [136] are leveraged to combine several basic sequence diagram into high-level sequence diagrams. In fact, a first step generates several basic sequence diagrams through the execution of the instrumented source code that yields a state vector before and after each message. The state vector is a vector of variables that represent the state of the system at runtime. A similar approach is used in Fujaba [126], a reengineering test suite where the dynamic pattern instance recognition [145] leverages the *combined fragments* of UML 2.0 to combine sequence diagrams that consider only object interactions, yielding a program *slice* of the program's method call trace. In [18], traces are obtained by executing an instrumented version of the source code that generates statements about methods (entry, exit, signature, target object class, identifiers, and arguments), conditions (kind of statement), loops (kind of statement, loop condition, end of loop), and distributed information (client remote calls, server executions).

Performing reverse engineering techniques to generate UML diagrams has been extensively researched upon but often tend to stick to common design patterns as defined by the Gang of Four [56] and usually fail to capture all types of patterns. In particular, the lack of precision of static methods might not be too much of a drawback as generated models are fuzzy, allowing to match several different implementations to the same model, embracing polymorphism.

5.4 Summary

Intention is a human concept that has been extensively studied in philosophy and agent theory. It is also informally used in software science to designate the purpose of a program.

In this section, we have explored different usages of intention in software science. Namely, we have seen how developers communicate on the intentions their software should fulfill by the use of standardized models. We have also discussed how the intents of programs are recovered through reverse engineering techniques.

We advocate the idea that the intention of a program is actually the reflection of the intention of its programmer. The intention of a program denotes a purpose it has been programmed for and the subsequent actions taken to fulfill this purpose. Contrary to behavior, which the expression of a program for a given input, intention is readily available in the source code but may be obscured. One way to conceal the intentions of a program is obfuscation, which is addressed in the next section.

6. On Obfuscation

> Uncertainty is the only certainty there is, and knowing how to live with insecurity is the only security.
>
> John Allen Paulos

According to [8], the goal of program obfuscation is to make a program unintelligible while preserving its functionality:

Definition 2. *An obfuscator $\mathcal{O}$ is a "compiler" that takes as input a program P and produces a new program $\mathcal{O}(P)$ that satisfies the two following conditions:*

- *(functionality) $\mathcal{O}(P)$ computes the same function as P*
- *(polynomial slowdown) the description length and running time of $\mathcal{O}(P)$ is at most polynomially larger than that of P*
- *(unintelligible) any information that can be extracted from the text of $\mathcal{O}(P)$ can be extracted from the input-output behavior of $\mathcal{O}(P)$. This property is also known as the "virtual" blackbox property*

It is both a way for attackers to preserve the intellectual property of their payloads and a way for exploit kit users to prevent analyzers from learning noticeable patterns by hindering the comprehension of a program. An acceptable obfuscator will often produce, for a reasonable cost (in space and time), a program difficult to understand to a human analyst and also difficult to undo by an automated deobfuscator.

In this section, we will use the following definition of an obfuscating transformation (inspired by [29, 76]):

Definition 3. *A transformation $\mathcal{T} : P \rightarrow P'$ is an obfuscating transformation if it satisfies the following property*

- *it is semantics-preserving: $\forall P, [\![P']\!] = [\![P]\!]$*

and the obfuscated program P' exhibits the following good properties:

- *it is potent: let $E(P)$ be the complexity of P; P' is a potent obfuscated program if: $E(P') > E(P)$*
- *it is resilient: it is difficult to reconstruct P from P'*

- *it is efficient: P' computes the same function as P with a reasonable time/space overhead*

While an inverse transformation may not always exist, a deobfuscator $\mathcal{D}$ can be roughly defined as the semantics-preserving transformation $\mathcal{T} : P' \rightarrow P$.

In this section, we will describe related work in the domain of object-oriented program obfuscation, as a countermeasure for developers to prevent code theft. We will also cover the currently used obfuscating transformations often found in JavaScript malware and briefly explain methods to cancel obfuscation, that is *deobfuscation*.

6.1 Taxonomy of Obfuscation Techniques

Readers not familiar with obfuscation may take a look at the comprehensive taxonomy compiled by Collberg et al. [29]. In this survey, they operated a systematic classification of obfuscating transformations of Java bytecode. Assuming that obfuscation is performed as a mean of intellectual property protection, they classify several obfuscating transformations according to the kind of information it targets (layout, data, control) and also assess the quality of these transformations.

6.1.1 Measures

Collberg et al. rightfully observe that, given enough time and space, an opponent is able to deobfuscate even complex obfuscation schemes. Their classification also include comments on how to assess the quality of the obfuscation in terms of *cost* added to the processing, the *resilience* to deobfuscation, and the *potency* of the obfuscating transformation, that is, its capacity to confuse a human analyst.

The *potency* measures the "confusion" added by the obfuscating transformation for a human analyst. They propose the computation of potency based on related work in the field of software complexity metrics. It is generally understood that the complexity of an obfuscated program increases with the effects of one or several transformations over a function, a class, or the entire program. Complexity metrics usually depend on the occurrence of some constructs such as the number of predicates or the level of nested conditionals for a function, the inheritance depth or the number of subclasses for a class, and the length or the data structure complexity for a program. The scales for the potency measure are $\langle$ *low*, *medium*, *high* $\rangle$.

On the other hand, they do not elaborate much on the computation of the *resilience* but stipulates it is function of the time needed to automate a deobfuscator able to effectively reduce the potency of a transformation $\mathcal{T}$ and the execution time and space

required by such deobfuscator. The scales for the resilience measure are ⟨ *trivial*, *weak*, *strong*, *full*, *one-way* ⟩. A transformation is *trivial*, *weak*, *strong*, *full* if a deobfuscator can *crack* it by a *local*, *global*, *inter-procedural* or *inter-process* static analysis, respectively. A *one-way* transformation cannot be reversed.

Finally, the cost of an obfuscating transformation is the execution time/space overhead of the obfuscated program compared to the original. The scales for the cost measure are ⟨ *free* (constant), *cheap* (linear), *costly* (polynomial), *dear* (exponential) ⟩.

It should be noted that these measures are estimations. Similar to cryptographic transformations is the interest in designing transformations that are free or cheap to apply but dear to cancel. On the contrary, obfuscation produces an executable program.

6.1.2 Layout Transformations

Lexical transformations impact the layout of the obfuscated program. Such transformations comprise ones that are easy to apply, yet difficult or even impossible to cancel. They include:

- identifiers scrambling: identifiers are replaced by generic or randomized identifiers;
- comments removal;
- formatting removal: strips the code from any formatting (space, indentation, etc.).

These transformations are one-way since formatting or comments can never be recovered. The potency of layout transformations is overall low, but scrambling identifiers or removing comments are more potent (medium and high, respectively) since they contain a great deal of pragmatic information.

These transformations are often considered as *surface* transformations, obfuscating the concrete syntax of the program, in comparison to *deep* transformations (control-flow and data transformations) that actually modify the structure of the program [138].

6.1.3 Control Transformations

These transformations affect the control flow of the obfuscated program in terms of *aggregation*, *ordering* or *computations*. An important feature that impact the resilience of control-flow altering transformations is the presence of *opaque constructs*:

Definition 4. *Opaque constructs can be divided into two distinct cases:*

- *A variable V is an opaque variable if it has a property q, which is known at obfuscation time, but which is difficult to deduce for a deobfuscator*
- *A predicate, or boolean expression, P is opaque if its outcome is known at obfuscation time, but difficult to deduce for a deobfuscator*

Control aggregation transformations interfere with logical computation aggregation of instructions by splitting up programs or grouping together instructions that do not share any logic. These transformations break the *procedural* abstraction of a program but are not extremely potent per se, though their potency increase when combined. They include:

- inlining: a method call is replaced with its own body and the method definition is removed;
- outlining: instructions that do not necessarily share any logic are grouped together in a subroutine. This is best used with inlining;
- interleaving: code from distinct procedures are merged together into a single procedure and an extra parameter is used to distinguish calls to each procedure;
- method cloning: a single method is replicated and each version is obfuscated using a different transformation, giving the impression of several different methods. Method dispatch is used to select between different versions at runtime;
- loop transformations designate a set of transformations that increase the complexity of a loop by either breaking the iteration space into smaller nested blocks (blocking), replicating the loop body one or multiple times (unrolling) or breaking the body of the loop into independent loops (fission). They performed better resilience when combined.

These transformations are easy to apply and highly resilient (except for loop transformations), but not one-way since they may leave traces of the original control flow. Potency of loop transformations is low compared to method inlining/outlining (medium) while the quality of method interleaving/cloning depends on the quality of the opaque predicates they used.

Control computation transformations tamper with control flow by inserting dead or irrelevant code, as well as low-level code that has no equivalent at high-level in order to hide the real control-flow. They include:

- dead or irrelevant code insertion: predicates that increase the complexity of the obfuscated program without really impacting the control-flow. Dead code is usually found in the branch of the predicate that is never evaluated while irrelevant code is a predicate that always evaluate to the same value;
- loop condition extension: where a termination condition is obfuscated with irrelevant code;
- non-reducible flow graph conversion: when a language compiles to a virtual machine or native code that is more expressive than itself, it is possible to include instruction sequences that have no equivalent with any source language construct (language-breaking). For example, Java can only express *structured* control-flow that will translate to a *reducible* flow graph while Java bytecode that results from the compilation of Java source code, can express *arbitrary* control flow, which can produce a *non-reducible* flow graph [2];
- programming patterns removal: this transformation replaces calls to standard libraries or usage of well-known programming patterns by custom implementations;
- *table interpretation*: sections of code are are converted into a different virtual machine code and interpreted with the corresponding virtual machine, embedded within the obfuscated application;
- code parallelization: where multiple threads are put to contribution to run independent sections of code.

These transformations often heavily increase the space dimension of the obfuscated program and offer good potency and strong resilience. However, low-level code transformations such as table interpretation and code parallelization are often costly. Code insertion, loop condition extension and non-reducible flow graph conversion highly depend on the quality of the employed opaque predicates.

Control ordering transformations tamper the *locality* of pieces of code, that is, the physical closeness of logically related items in a program. Transformations target every level of locality from terms within expressions to methods within classes, to classes within files, etc. They include:

- statement order randomization: this transformation can be applied at different level of locality to alter the order of independent statements. This is best used combined with method inlining/outlining;

- loop reversal.

These transformations do not obscure much of the code but are very resilient since they are one-way. However, they are usually easy to carry out but sometimes require the application of data dependency analysis to confirm the legality of some reorderings. Conversely, data dependency analysis will be helpful, though time consuming, in comprehending a program obfuscated using such techniques.

6.1.4 Data Transformations

Data structures are also the target of obfuscating transformations that alter their storage, encoding, aggregation or ordering.

Data storage and encoding transformations make use of “unnatural” storage and encoding of dynamic as well as static data. They include:

- encoding conversion: data is represented following a different encoding;
- variable promotion: promoting a variable from a specialized storage class to a more general class;
- variable splitting: boolean and restricted-range variables can be split into a finite number of variables. Subsequently, built-in operations should be replaced by adequate operations with regards to the new representation;
- procedural conversion: static data are converted into procedures that generate these data.

These transformations are strongly resilient since they leverage the multiplicity of interpretation of a same piece of code. Resilience of these transformations often increase with the cost involved in performing these. Encoding transformations such as encoding conversion, variable splitting, or converting static data to procedure depend on the complexity of the encoding function.

Data aggregation transformations erase or hide data structures used in the original program. These transformations include merging variables, literals or classes to obscure the code and prevent recovering the data structures. They include:

- scalar variable merging: arithmetic operations on scalar variable allow to merge the storage of two scalar variables;

- array restructuring: similarly, we can merge two arrays, but it is also possible to split an array in several others, or fold an array on several dimensions, or even flatten this array to less dimensions;

- inheritance relations alteration: the hierarchy can be modified by refactoring two independent class as children of a common bogus class, or inserting a bogus class into the hierarchy, or even splitting a class into two consecutive classes in the hierarchy.

Array transformations are usually weak like loop transformations. Collberg et al. could not capture the potency of these transformations using the metrics founds in related work, which miss to capture the effects of structure transformations. They speculate that these transformations deprive the analyst from pragmatic information and hence contribute to obscure the program. On the other hand, modifying the inheritance relations or the class hierarchy suffer from low resilience, but this resilience increases when transformations are combined.

Data ordering transformation are simple transformations that randomize the order of declarations in the source code, whether it is the order of variables in an array (similar to array transformations), or the order of methods and instance variables in a class (similar to statement reordering).

6.1.5 Primitives on Deobfuscation

Deobfuscation, which can be thought as the transformation $\mathcal{T} : P' \to P$, should be seen as a simplification process that attempts to reveal the original intent of an obfuscated program. In particular, due to the one-way nature of certain transformations, it is never possible to recover the original program P.

As a matter of fact, the goal of deobfuscation is rather to cancel the effects of obfuscating transformations. In their taxonomy [29], Collberg et al. briefly consider some analysis techniques employed to *undo obfuscating transformations*. Although lexical transformations cannot be cancelled because they are essentially one-way, they do not contribute to obscuring the semantic structure of the program [138]. Therefore, analysis concentrates on deep transformations that affect the structure of the obfuscated program.

Of particular interest are opaque constructs, which are considered the most difficult part of the deobfuscation process. Indeed, by considering that opaque constructs involve the insertion of additional dead code, an obfuscated program can be seen as the mixture

of the original program with dead or irrelevant code generated by the obfuscation. The approach is two-fold, although it can be partly applied, and consists in *identifying* and *evaluating* opaque constructs:

- opaque construct identification consists in extracting the statements computing an opaque variable or an opaque predicate. It is suggested that pattern matching will work for local predicates (contained in a single basic block) but this can be easily evaded by avoiding known opaque constructs or syntactically mimicking constructs found in the original program to add more confusion. An alternative is the use of program slicing that allows to collect statements contributing to an opaque value, even if they are interspersed across the program. Program slicing can be hindered using parameter aliases or extra syntactic dependencies [76]. A third method is *statistical analysis*, which analyzes runtime characteristics of instrumented obfuscated program in order to detect recurring predicate values. This can be prevented by designing non-deterministic opaque predicates;
- opaque construct evaluation consists in optimizing code by detecting dead code to be removed and moving code duplicates (hoisting), after propagating the opaque values. This propagation is possible using data-flow analysis but more powerful techniques, such as theorem proving, are suggested by the survey. Collberg notes, however, that this can be thwarted by theorems known to be difficult to prove.

Overall, static deobfuscation is very costly and is often hindered by opaque constructs that are only evaluated at runtime.

These contributions from Collberg [29] and Barak [8] have been the basis for more work in programming language and bytecode obfuscation and deobfuscation, but no other work did a comprehensive survey of obfuscation as these ones, to the best of our knowledge. Obfuscation is, obviously, not limited to the above methods. However, they are not all applicable to the obfuscation of interpreted scripts in web environments.

6.2 Obfuscation of Web-based Scripts

Obfuscation has been used heavily in web malware campaigns to hide redirection payload injection, thus evading automated detection by pattern matching. Redirection, that is redirecting a user's browser from one web page to another automatically, can be achieved through the use of an HTTP status code of the type `3XX` [50]. This mechanism occurs before any content is actually downloaded. On the other hand, attackers make use of redirection to evade crawlers or fool the user by displaying what seems

```
var e = eval;
hidden = e("document.getElementById('hidden').innerHTML");
```

Figure 7. Aliasing of the eval() function

an innocuous web page while malicious scripts run in the background. In that case, redirection is achieved *locally* in the simplest way, by injecting an `iframe`, an HTML frame, that will load contents from a remote server, potentially from a different domain. JavaScript is employed to hide the injected payload through a countless number of methods described below.

A few surveys have studied the trends in obfuscating web contents using JavaScript: Chellapilla and Maykov [20] reported on the prevalent use of obfuscation in redirection spam campaigns (to confuse web page indexing) while Craioveanu [33] presented a survey on the *server-side polymorphism* or how to dynamically construct polymorphic malware using server-side scripting. A later contribution [70] described additional techniques to prevent analysis of scripting malware. These surveys concur that most obfuscation techniques witnessed in the wild are based on string manipulation and custom encoding methods to conceal any traces of string or substring that would be detected by common signatures.

6.2.1 Lexical Transformations

As described by Collberg, these transformations attempt to remove pragmatic information to the user such as comments, formatting, and explicit variable names. Variable names are often shortened or randomized. JavaScript native objects can also be replaced with random names in order to conceal their use to a human analyst. This practice is called *variable aliasing* and can be applied several times to create dummy variables that all point to a same location. In Figure 7, the critical function `eval()` is concealed as another function.

These transformations form the bulk of JavaScript *compressors* or *minifiers* as they are known. These programs can significantly reduce the size of a script, hence the bandwith consumption, improving the performance of a web site [70]. These techniques are widely used in popular web sites and therefore do not indicate malice.

```
foo = "fro";
bar = "mStr";
foobar = foo+bar;
a = String[foobar+"ingCha"+"rCode"](97);
```

Figure 8. Concatenating strings to invoke the fromCharCode() method

```
foobar = "h314i2205i2350dd2350e234n325".replace(/[0-9]/g,"");
```

Figure 9. Example of character substitution

6.2.2 String Manipulation

Analogous to data transformations described above for compiled languages, there is a variety of transformations that affect the encoding and storage of literals in JavaScript, and especially, string literals. As explained in Section 2, JavaScript has the particularity of providing a runtime evaluation operator, named `eval`. `eval` parses an expression as a JavaScript statement and executes it. This is one common *executable sink* among some described in Section 2. In general, executable sinks use data as code, which allows such behaviors as dynamically generating a program at runtime by concatenating string literals and evaluating the result. Conversely, a program can then be deconstructed into strings in order to conceal it.

The most common transformation on strings is *string concatenation*. It allows to evade signature-matching by slicing a string into several pieces that seem benign and concatenating these to retrieve the original string. Figure 8 displays an example where a function is finally executed after recovering its name by concatenating several other strings. On the other hand, *string splitting* is a way to store several strings as a single string, which will be then split to generate the original program statements.

String literals can be further obscured by injecting garbage. *Character substitution* is a technique where the original string is recovered by using the `replace()` to substitute garbage characters. Figure 9 illustrates such technique on the string `"hidden"`, which is interspersed with numbers.

All these simple transformations are to be combined to produce better obfuscation. They are often used together in custom encoding schemes to obfuscate long strings or a whole program.

```
s = unescape(%22te%22%20%2B%20%22st%22); // evaluates to "te" + "st"
s = eval(s); // "test"
t = "\x74\x65\x73\x74" // "test" in ISO-8859-1
u = \u0074\u0065\u0073\u0074 // "test" in Unicode
```

Figure 10. The string test written in different encodings

```
str = "qndy'mh)(:" // snippet
str2="";
for (i = 0; i < str.length; i ++){
    str2=str2+String.fromCharCode(str.charCodeAt(i)^1);
};
eval(str2);
```

Figure 11. Custom encoding based on a simple XOR

6.2.3 Encoding Schemes

In JavaScript, different character encodings can be used to make non-ASCII strings portable. An example of some available encodings is presented in Figure 10.

Notice the use of the `unescape` function that recovers `escape`d characters. This encoding is used to encode URLs that only accepts a limited number of characters.

The surveys on JavaScript obfuscation [20, 33, 70] also exhibit custom encoding obfuscations. They usually produced long obfuscated strings along with a decoding function, also known as the *decoding routine* to recover the original string. An example can be found in Figure 11 where the obfuscated string is decoded using a simple XOR operation (the numeric literal 1 can be considered the *deciphering key*). Other examples are described in [33] and in particular, two implementations of symmetric encryption that uses a mono-alphabetic substitution scheme in which every occurrence of a particular plaintext message unit is replaced by a ciphertext unit. Within the two exhibited schemes in Craioveanu's article, one has the deciphering key encoded as two permutation look-up tables located just before the deobfuscation function. While the second scheme hides the decryption key using an additional obfuscation technique. Implementations of popular encrypting and encoding schemes such as RSA or Base64 have been witnessed in JavaScript.

Some encoding do not process strings on a per-character basis but rather substitute locations in a string by symbols. String encoding is used in packers such as the

Dean Edwards' packer [43]. Encoding schemes usually feature an obfuscated string and a decoding routine as we mentioned earlier. The obfuscated string itself is not executable, contrary to an obfuscated program and it is necessary for it to be decoded, leading to dynamic code generation at runtime. The decoded code is often `eval`ed in a manner dubbed *eval unfolding*, that is, self-generation (using the `eval` construct) of new executable code, but can be also injected to the HTML document through the `document.write` function. Code injected in the HTML document is queued with other scripting contents and will possibly be executed later. The potency of custom encoding schemes is quite high, but the resilience may be low in the case a human analyst has access to the decoding routine and the decoding key if it exists.

JavaScript obfuscation techniques seem to cover few obfuscating transformations in the taxonomy of Collberg, mainly data encoding and storage ones. Indeed, as an interpreted language[2], JavaScript is limited in the range of transformations it can implement. In particular, low-level constructs are not available and multi-threading is not supported, which makes obfuscation relying on process parallelization impossible to achieve. Complexity is increased through the combination of several transformations to obfuscate the whole or part of a script. Additionally, the task of deobfuscation is made even more tedious for a human analyst when a program is obfuscated through several layers of obfuscation. Such practice prompted the development of tools to automate deobfuscation.

6.3 Reversing/Cancelling JavaScript Obfuscation

As demonstrated by Barak et al.[8] and at another level by Craioveanu[33], obfuscation is not a problem that cannot be overcome. Collberg suggested in [29] that given enough time and space, a determined attacker could cancel obfuscation. Barak later formally demonstrated the impossibility of the *virtual blackbox* property of obfuscation. It is simply impossible to make a program completely *unintelligible*. It should be conceded, however, that deobfuscation by static analysis is difficult or even impossible in certain cases. In particular, JavaScript obfuscation, which leverages at-runtime dynamic code generation, is quite impervious to static examination.

Deobfuscation is, therefore, needed to increase the accuracy of analysis. For example, although packers are an easy-to-spot occurrence, they are not necessarily an indicator of malice since packers are also used by benign web pages. In this section,

[2] Just-in-Time compiling of JavaScript may actually extend the range of available obfuscating transformations

will be covered the techniques used to deobfuscate JS code, the different tools we can use to automate such task and finally some related researches tackling the obfuscation of Web 2.0 active code.

6.3.1 Manual Approach

Manual methods are most of the time dynamic analysis methods that aim to reveal the code that has been obfuscated by simply executing the suspicious code. It can be simply done using a JS engine or debugger to execute an instrumented version of the suspicious obfuscated script. Instrumentation, in this case, is often limited to the hooking of critical sinks such as `eval()` or `document.write()`. A more popular method by way of substituting executable sinks with simple print instructions in order to reveal the malicious code once deobfuscated[104]. The JS engine should be run outside of a browser to prevent any exploitation. JavaScript external engine implementations such as Rhino[16] or SpiderMonkey[45] are widely used by researchers. However, these engines are often limited by the fact they do not implement any browser context such as the DOM or other APIs. It is not rare that a deobfuscation task halts on an unknown variable that is in fact a reference to an object in the HTML document or an API method. Analysts often have to declare statically the missing context in order to pursue analysis. Obviously, manual deobfuscation is a quite time-consuming and tedious work, and researchers often resort to self-developed tools to automate such task. Hopefully, there exists Malzilla[128] that offers some nice functionalities in manipulating JS code such as remote script downloading, string substitution, common encoding evaluation, etc. Nonetheless, deobfuscation remains an interactive task where the user needs to decide what to do next. In recent years, researchers became interested in automating the detection and deobfuscation of obfuscated JS scripts.

6.3.2 Automated Deobfuscation

To the best of our knowledge, there is very little literature on the subject of obfuscated JavaScript deobfuscation or simplification. Obfuscation itself is not even considered a problem by most researchers: either it is viewed as a sign of malice and then the obfuscated script is flagged as such and its execution is prevented; or it is considered trivial as it is expected to deobfuscate itself during an attack, and the JS engine is instrumented to analyze the output of `eval`ed contents.

The former mindset is no more alive nowadays but has produced some results in the identification of obfuscated JavaScript. While the latter concentrates on the `eval`

construct to extract traces of executed snippets of code that were previously obfuscated.

Assuming obfuscation is an indicator of malice makes the detection of obfuscated strings a sufficient criteria to block the incriminated script. Choi et al. [24] proposed a straightforward method to distinguish obfuscated and unobfuscated strings: string pattern analysis. They observed that obfuscated strings would differ from unobfuscated strings in terms of character frequency and distribution, as well as their length. Relying on these three metrics, they designed a set of rules to cover as much obfuscation patterns as possible. Interestingly, their rules were not able to capture an `eval` unfolding string. An extension of Choi's work [86] with a more thorough evaluation exhibited interesting results, though they still conflate obfuscated scripts with malicious ones. Later proposals leverage on machine learning methods in order to draw models, as generic as possible, of obfuscated malicious scripts from large datasets. In an approach similar to [24], Likarish et al. [89] attempted to codify visual differences between benign and obfuscated scripts. They selected the normalized frequency of 50 JavaScript keywords and symbols as features as well as 15 other features that describe an obfuscated script in lexical and string statistical terms. In particular, one feature measured the percentage of human readable strings in the script, which resulted to be one of the features most correlated to malicious (here, obfuscated) scripts along with the scripts' average string length, average number of characters per line, percentage of whitespace and the use of the keyword `eval`. They compared different machine learning methods in terms of performance and noted a maximum detection rate of 89.5% on real-world data. They deplored few false positives that revealed to be packed benign scripts. Another classification proposal [69] actually attempted to design a classification method that they claim would be resilient to obfuscation. Using different classes of features that describe DHTML documents in terms of native JavaScript functions (154), HTML document level features, or even ActiveX objects, they compared several learning methods for a maximum accuracy of 96.14% when using all feature classes. What they call "resilience" to obfuscation, however, is actually the ability of their classifier to classify obfuscated scripts as malicious. A later statistical classifier [84] extends the results of Zozzle [36] (explained later in this section) where abstract syntax trees (AST) of JavaScript samples are traversed to construct features that characterize obfuscated JavaScript programs. Zozzle actually classifies heap-spray attacks by extracting hierarchical features consisting of a *context* (branching conditional, loop, etc.) and a variable string. Zozzle associates these features with a probability to indicate malice. Similarly, Kaplan et al. [84] use this method to classify JS samples between obfuscated and unobfuscated JS programs. Obviously, the features they obtain match better the

strings that are usually witnessed in obfuscated JS programs. The results of such transformations were described earlier in this section. Their features can actually span one to several level in the AST for a low false positive rate (about 1%) and a relatively low false negative rate (about 5%), with an overall processing time of 5 megabytes per second. Kaplan's approach is actually the one that is the closer to ours as it also shares the assumption that obfuscation is not an indicator of malice. Previous proposals had mixed results: although some perform well at detecting obfuscation, they either miss obvious instances such as `eval` unfolding routines or fail to precisely extract obfuscated strings for further deobfuscation.

On the other hand, assuming that an obfuscated script necessarily unpacks itself to run, it is sufficient to execute the obfuscated script to the point where it will get deobfuscated. Since analysis is much more accurate on unobfuscated scripts [36], recent web malware analysis frameworks rely on a deobfuscator. In Zozzle [36], a machine-learned based classifier sort JavaScript samples based on features extracted from malicious JavaScript *code contexts*. A *context* is a fragment of JavaScript that is passed to the `Compile` function of the JavaScript engine. This function is invoked whenever `eval` is called or a new script is included with an `<iframe>` or `<script>` tag. The Zozzle deobfuscator, which makes use of the Detours binary instrumentation library [72], intercepts calls to the `Compile` function and can therefore observes code at each level of its unpacking, just before being executed. Such deobfuscator also exists as a browser plugin like the JavaScript Deobfuscator [110] for Firefox browsers. Alternatively, hooking of the `eval` function can be carried out at the JavaScript level, instead of the engine level. Such approach is taken by jsunpack [65], a popular JavaScript automated analysis web site.

Such tools have been widely used by analysts and have performed relatively well since they automate a quite time-consuming task. However, these dynamic approaches sometimes failed against anti-analysis tricks. Also, they often need to execute the analyzed sample several times (as jsunpack does) to consider multiple paths and extend their code coverage. Moreover, relying on code instrumentation, especially the hooking of critical sinks for trace collection, might be detected by anti-analysis techniques prompting the halt of the deobfuscation stage.

6.4 Anti-analysis Techniques

Not only obfuscated scripts unfold through several layers of obfuscation but, at one or several layers, the obfuscated code may be divided in different parts. Each part

```
<script src="foo.js"></script>
<script>
b=document.getElementById("bar").innerHTML;
</script>
...
<div id="bar">bar</div>
...
<script>
alert(a+b);
</script>

[foo.js]
var a="foo";
```

Figure 12. Example of a multi-partite scripts split into script blocks and remote sources

may be obfuscated by a different transformation or encoder. The different pieces of the obfuscated script can even be delivered from sources external to the scripting environment. That way, it is possible to conceal code that is obviously obfuscated from a human analyst by distributing it to a linked resource, a script or an iframe inclusion, or even having it stored within the HTML document. Deobfuscation can be further hindered by using anti-analysis techniques that detect the analysis environment or that constraint the execution of the obfuscated script on specific browser personalities.

6.4.1 Multi-partite Scripts

Security products often tend to analyze web pages by downloading relevant objects such as scripts and processing each object in isolation. Howard [70] observed that the task of deobfuscation by a security scanner is complicated when the script is split between separate script objects since components from each object may be required for successful detection. There are many locations where a script object can be concealed: the HTML document, linked script files, embedded frames from remote origins but also PDF files. In Figure 6.4.1, the contents to be `alert`ed are split between a remote script and an element of the HTML document. Strings concealed within the HTML document can be recovered using the `getElementById`, `getElementsByName` or `getElementsByTagName` functions. The `innerHTML` property actually returns the contents of an HTML element

```
function x(UW,P){
if(!P){P='[obfuscated data]';}
var W;
var VM='';
for(var G=0;G<UW.length;G+=arguments.callee.toString().replace(/\s/g,'').length-535){
W=(P.indexOf(UW.charAt(G))&255)<<18|(P.indexOf(UW.charAt(G+1))&255)<<12|
(P.indexOf(UW.charAt(G+2))&255)<<(arguments.callee.toString().replace(/\s/g,'').length-533)|
P.indexOf(UW.charAt(G+3))&255;VM+=String.fromCharCode((W&16711680)>>16,(W&65280)>>8,W&255);
}
eval(VM.substring(0,VM.length-(arguments.callee.toString().replace(/\s/g,'').length-537)));
}
x('[obfuscated data]');
```

Figure 13. Context-dependent obfuscation using arguments.callee

(the strings contained between the element opening and closing tags).

It is important that a deobfuscator gathers enough scripting contents to conduct analysis.

6.4.2 Opaque Constructs and Cloaking

Some obfuscating transformations benefit from an additional twist in order to thwart automated or manual deobfuscation. In these cases, unpacking will not start if a condition is not satisfied. To implement such anti-analysis techniques, we can make use of opaque constructs, as defined by Collberg previously in this section.

In fact, even multi-partite scripts make use of opaque constructs. For example, failing to analyze the "foo.js" file in Figure 6.4.1 will render the variable `b` opaque. Another occurrence makes a combined use of `arguments.callee()` (see Figure 13), which returns the body of the function being executed, with the `location.href` attribute, which is the referrer web site from which the malicious code has been invoked. On one hand, the `arguments.callee()` trick prevents an analyst from modifying the running code by referring to the original function. On the other hand, the referrer `location.href` is used as part of the decryption key in order to prevent analysis from being performed outside of the place for which it was designed to be run, that is, the web page that called the malicious script. This trick is effective against offline automated deobfuscators such as WEPAWET [31] or jsunpack [65].

Such techniques are often used to distinguish real browsers from automated web malware detection:

```
var e1 = new Date();
var x = e1.getSeconds();
setTimeout(decr,1025);

function decr() {
var e2 = new Date();
var k = e2.getSeconds();
var xX = k - x;

xorkey = 245 + xX
...
```

Figure 14. Anti-emulation technique based on the time delay to trigger the decoding routine

Definition 5. *Cloaking is a set of techniques that attempt to present different contents depending on some specific criteria.*

Cloaking targets behavioral discrepancies between real browsers and detectors. In fact, not only does it detect an analysis environment, it can also deliver some innocuous contents to the detector for the web page to be classified as legitimate. Most of the time, cloaking only prevents the expression of malicious code within the script by using an adequate opaque construct. In Figure 14, `xX` is an opaque variable for any analysis environments that do not emulate time delay functions for performance reasons. Without knowing the value of `xX`, deobfuscation cannot be performed.

Other common missing functionalities indicative of a possible emulator are exception handling, DOM interaction, event handling, etc. A deobfuscator should be able to provide such functionalities or at least emulate these to evade cloaking.

6.5 Summary

In web scripting languages, the range of available obfuscating transformations is restricted compared to richer languages: obfuscation tools or transformations witnessed in the wild are usually limited to lexical transformations, string manipulation and custom encoders or packers. In fact, attackers are interested in the high potency and low application cost of such transformation rather than their resilience.

Deobfuscators are few and often leverage the property that obfuscated programs in JavaScript unpack themselves to execute. Recently, more complex obfuscating transformations have demonstrated resilience to automated deobfuscation performed in emulation environments. This can be explained by a lack of anticipation from deobfuscator developers concerning the ability of obfuscated programs to defend themselves. Cloaking techniques are slowing down execution-based JavaScript deobfuscators and malware detectors that neglect the resilience of obfuscation.

In this dissertation, we take an opposite point of view in considering obfuscation as an obstacle to the analysis of web malware. Therefore, we advocate the necessity of prior deobfuscation to produce simplified code that will be easier to analyze and decide upon. In the following sections, we will successively describe techniques we designed to identify obfuscation patterns, extract obfuscated contents and deobfuscate these.

7. Identifying Hierarchical Structures Related to Obfuscation

> They who can give up essential liberty to obtain a little temporary safety, deserve neither liberty nor safety.
>
> Benjamin Franklin

As stated in Section 6, obfuscation has been extensively used by attackers, but it is not restricted to a malicious usage. In fact, developers also heavily resort to obfuscation when they wish to protect their intellectual property or compress voluminous scripting contents. Therefore, it would be counterproductive to consider every obfuscated script as malicious. Moreover, some obfuscation techniques, employed by attackers, are also used in legitimate code, and conversely, some commercial obfuscation tools, used to protect legitimate contents, are also used to hide malicious contents [70]. Deobfuscation is therefore required to provide malware detection without risk of false negatives.

Current solutions rely on executing obfuscated scripts without any reasoning, assuming the script unpacks itself before executing. However, anti-analysis techniques are able to cloak deobfuscation and to appear legitimate in the eyes of a browser emulator. In order to circumvent such drawbacks, we proposed to suppress any side-effect involved by the execution of JavaScript (JS) code. Such design choice prevents decoding routines from revealing their contents, which cannot be dealt with static examination alone.

This section describes *ob_asti*, the module responsible for detecting and extracting obfuscated contents from analyzed web pages. We argue on the reasons that motivate such design and the technological choices we have made such as the representation of JS programs as abstract syntax trees (ASTs). Then, algorithms borrowed to *arbology* are introduced to explain how subtree matching is performed. The accuracy of our implementation is evaluated through a preliminary experiment on malicious samples collected by Mitsuaki Akiyama and his team from NTT Information Sharing Platform Laboratories. We further discuss possible extensions to increase the recall of our method by relaxing the representation of subtrees.

7.1 Detecting Obfuscation

In Section 6, we have briefly described some past proposals on detecting obfuscation based on the assumption that an obfuscated program is malicious. In particular, Choi et al. [24] concentrated on the patterns of obfuscated strings. Their metrics attempted to capture the differences with plain text JavaScript code. If they were able to detect some types of encoded or stored strings, they were not able to identify the decoding routine of the `eval` unfolding encoder, which code is similar to legitimate JavaScript code. Features used in classifiers described in [89] and [69] could distinguish obfuscated samples and unobfuscated ones with a high accuracy relatively to the learning dataset. However, these two classifiers cannot indicate which part of the code contributed to obfuscation if the obfuscated code is mixed with legitimate code.

With the aim to extract obfuscated contents from web pages, it is important to precisely identify where in the code are located obfuscated contents. If obfuscation patterns are varied, we can observe that they are decrypted by decoding routines who look similar to each other. Moreover, variable or function identifiers are not reliable because of the heavy use of aliasing and scrambling, making string pattern or statistical methods ineffective. On the other hand, important structures may have been preserved in most cases in order to preserve the semantics of the obfuscation. In particular, obfuscating transformations that encode script code as strings, make use of different kinds of data structures to store the code. Upon deobfuscation, the decoding routine needs to access and iterate on those structures in order to retrieve the original script that will be evaluated.

In this dissertation, we assume that designing an efficient obfuscation (both potent and resilient with a low obfuscation cost) is difficult. Therefore, many attackers rely on obfuscation tools whose output guarantee a good potency in little time but which may not be very resilient. However, by combining these tools, a better resilience can be achieved. Still, we assume that hierarchical patterns that are characteristic of obfuscated code can be found. Abstract syntax trees are able to express such hierarchical patterns with accuracy. Our proposal is therefore two-fold here:

- to learn the hierarchical patterns of obfuscation from obfuscated samples;
- to detect hierarchical patterns, characteristic of obfuscation, in candidate samples.

Our proposal therefore focuses on encoding routines, which are prevalent in current attacks, and not on the string feature, which highly vary from one sample to another. Moreover, we propose to precisely identify part of the code that contribute to obfus-

cation, instead of computing statistical features that do not give any indication on the location of the obfuscated contents.

7.2 Overview

ob_asti extracts obfuscated contents from web pages analyzed by (sak_mis), providing input to the u_adjet deobfuscator. Because deobfuscation can be seen as a dual process, identification and evaluation, it is necessary to isolate parts of the JavaScript code involved in the obfuscation process. The second part, *evaluation*, deals with the simplification of the obfuscated program to a deobfuscated form, which is described in Section 9.

ob_asti takes as input a mixture of HTML and JS contexts originating from the same web page. This mixtures comprises the contents of the original web page, plus all linked or referred scripts and suspicious contents that are prefetched in the previous step of our system, as described in Section 4. The extractor articulates around two main stages (see Fig. 15):

- a learning stage: collected samples of obfuscated JS programs are transformed into ASTs. Recurring patterns consistent with obfuscation are extracted and added to our knowledge base;
- a detection stage: any aggregated web contents forwarded by the proxy is transformed into an AST. The AST is matched with our knowledge base and matched subtrees are subsequently extracted and fed to the deobfuscator.

As with all learning-based approaches, the accuracy of detection is dependent on the learned models. In our proposal, learning is based on hierarchical features of obfuscated programs that may not vary from one sample to another, given that they have been obfuscated with the same encoder.

7.3 Obfuscating Transformations

According to previous surveys and contributions on JavaScript obfuscation [33, 70, 24, 86, 84], the most recurring obfuscating transformations are to be string manipulation and string encoding techniques, with the latter often being a combination of the former. Therefore, by reducing the scope of transformations to the ones in Table 3, a classifier may be able to detect the majority of obfuscated JS programs. The list of *syntactical categories and symbols* given in Table 3 is not exhaustive.

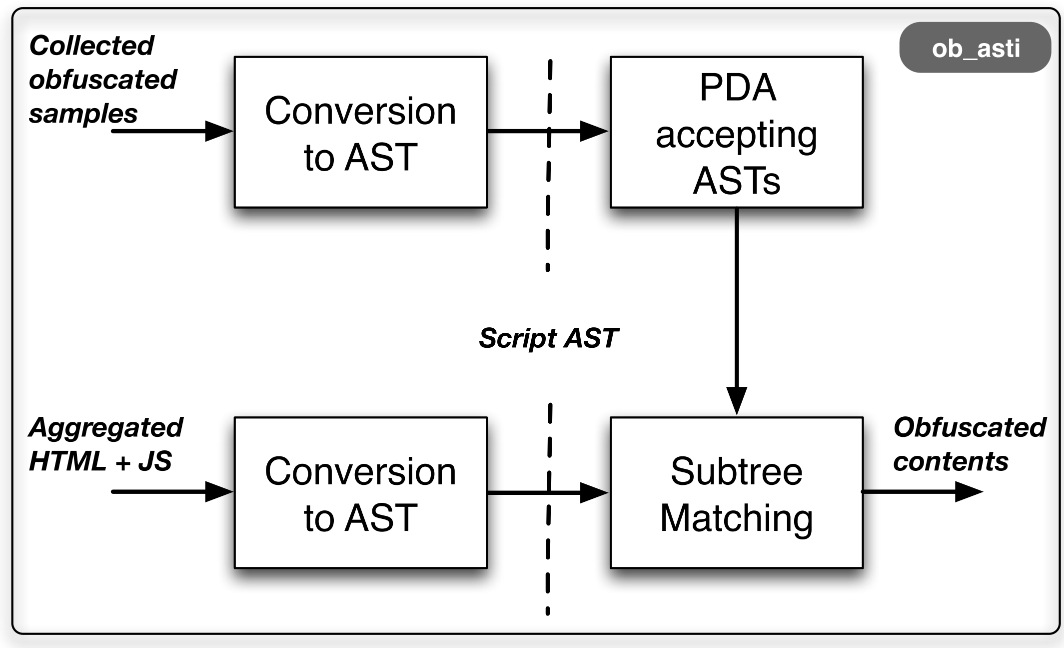

Figure 15. Workflow of the ob_asti obfuscation extractor

technique	Collberg class [29]	syntax
variable aliasing	data storage	assignments
concatenation	data storage	`+`, `concat`
DOM reference	data aggregation	`getElementsById`
substitution	data encoding	`replace`
basic encoding	data encoding	`unescape`
encryption	static data to procedure conversion	`eval`, `fromCharCode`, arithmetic ops, loops

Table 3. Common obfuscation techniques occurring in JS malware

Assuming attackers rely on popular attack toolkits that use the same obfuscation tools, it can explain the recurring patterns witnessed by the above-mentioned surveys. Given that discrepancies between two samples may arise from the use of identifier scrambling, we can assume that the samples have similar, or even identical, syntactical structures. By abstracting the samples at the syntactical level, we can reveal such hierarchical similarities.

7.4 Abstract Syntax Trees in JavaScript

In our approach, we consider the abstract syntax trees (ASTs) of scripts to analyze. An AST is a tree-based representation of a program code obtained by *parsing* or syntactical analysis. Syntactical analysis is the process of identifying the constituents of a statement according to the grammatical rules that define the forms of all permitted statements [19].

Figure 16 displays a sample AST for the following sample JS program:

```
for(i=0;i<str.length;i++){
  str2 = str2 +
  String.fromCharCode(str.charCodeAt(i)^1);
  str3 = str3 + str3};
eval (str2);
```

The loop statement contains two instructions represented by two paths stemming from the `LOOP` node: one being the actual decoding using the `String.fromCharCode` function and the other path being a dummy operation on an unrelated variable.

Since we are concerned with reducing the entropy of script contents for the purpose of pattern matching, it becomes necessary to abstract the script code in order to get rid of the randomization introduced by the identifiers and values. The AST we employ here differs in some aspects to similar approaches [118, 36, 84].

The abstract syntax described below is based on the syntax of JavaScript. More information can be found in ECMAScript specification [42].

Lexical tokens are represented as follows:

- whitespace, semicolons and comments are not represented.
- tokens: 3 different types to represent variables, numeric literals and string literals, respectively.

```
<token> ::= <ident-name>
```

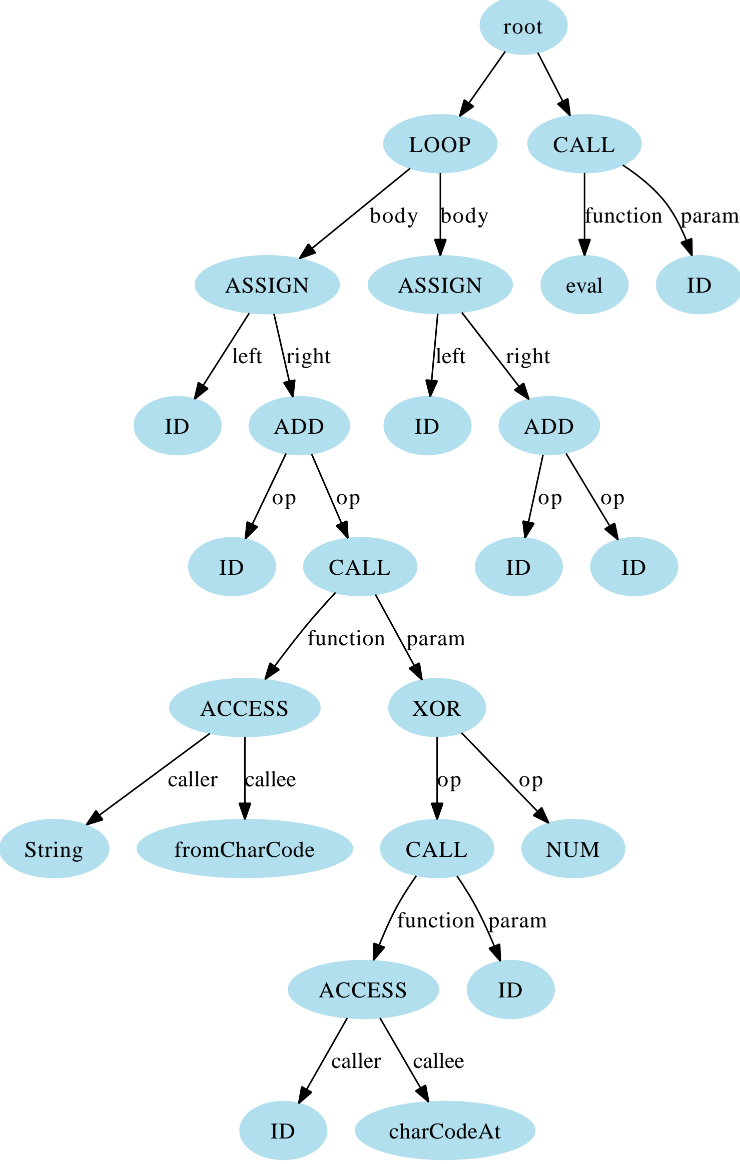

Figure 16. Abstract syntax tree of a sample loop

```
            | <numeric>
            | <string>
```

- variables: all identifiers are abstracted.

```
<ident> ::= <ident-name> "but not" <reserved>
<ident-name> ::= ID
```

- reserved keywords comprise all JavaScript keywords, which are reserved symbols and cannot be used as variable names, as well as future reserved keywords and the `NULL` literal and boolean values. Not all keywords are represented in our abstract syntax. Future keywords are out of the scope here.

```
<reserved> ::= <keyword>
             | <future-reserved>
             | <null>
             | <boolean>
<keyword> ::= "one of" BRANCH | BREAK | CATCH | CONTINUE | FUNCTION
            | IN | LOOP | NEW | RET | THIS
            | THROW | TRY | TYPEOF | VAR | WITH
```

`BRANCH` represents all conditional branching (`if`, `switch` or the ternary operator). `LOOP` represents all loop predicates (`for`, `while` and derivatives). Notable omissions include `else` and `then` predicates.

- literals are discarded and replaced by generic types.

```
<literal> ::= <null>
            | <boolean>
            | <numeric>
            | <string>
            | <regexp>
<null> ::= NULL
<boolean> ::= BOOL
```

```
<numeric> ::= NUM
<string> ::= STR
<regexp> ::= REGEXP
```

All string and numeric literals are abstracted to generic types `STR` and `NUM`, respectively. `BOOL` accommodates the boolean values `true` and `false`.

Expressions are represented as follows:

- primary expressions in JavaScript can be composed solely of the `this` keyword, a variable, a literal, an array or an object literal, or a group of expressions.

```
<prim-expr> ::= THIS
              | <ident>
              | <literal>
              | <array>
              | <object>
              | (<expr>)
<array> ::= ARY_LIT
<object> ::= OBJ_LIT { }
           | OBJ_LIT { <property-list> }
<property-list> ::= <property>
                  | <property-list> , <property>
<property> ::= <property-name> : <assign-expr>
<property-name> ::= ID
                  | STR
                  | NUM
(<expr>) ::= PARENTHESIS { <expr> }
```

Arrays are abstracted to `ARY_LIT` only, with no more information on the literals contained in the array. On the contrary, objects are containers (containing a block) that holds zero, one or several properties. The properties are expressed in the form of an assignment. Finally, the grouping operator `( )` is represented as a block labeled `PARENTHESIS`.

- left-hand side expressions comprise constructors and function call expressions.

```
<lhside-expr> ::= <new-expr>
                | <call-expr>
<new-expr> ::= NEW { new-expr }
             | NEW { <member-expr> , args }
<call-expr> ::= CALL { <member-expr> , args }
              | CALL { <call-expr> , args }
<member-expr> ::= <prim-expr>
              | <func-expr>
              | ACCESS { <member-expr> , <ident-name> }
<args> ::= ( )
         | ( <arg-list> )
<arg-list> ::= <assign-expr>
             | <arg-list> , <assign-expr>
```

The `ACCESS` label abstracts access operators such as `obj.prop` or `obj["prop"]`. Calls to constructors or other functions can include string chains of multiple objects and properties and sub-properties.

- function expressions include any native function ranging from `eval()` to `String.fromCharCode()`. We monitor the call of a large number of non-user-defined functions ranging from JavaScript core functions to DOM API functions, to form submission functions, to mathematical functions.
- postfix expressions define postfix operators:

```
<postfix-expr> ::= <lhside-expr>
                 | POSTINC { <lhside-expr> }
                 | POSTDEC { <lhside-expr> }
```

Postfix operators increment or decrement the result of an expression after evaluation.

- almost every operator is represented in our abstract syntax and this include unary operators, multiplicative operators, additive operators, bitwise shift operators, relational operators, equality operators, binary bitwise operators, binary logical operators and assignment operators. Every single operator (`OP` in what follows)

is defined but we could consider a unique abstract operator for a whole class. Unary operators are represented as follows:

```
<unary-op> ::= OP { <operand> }
```

Binary operators are represented as follows:

```
<binary-op> ::= OP { <operand> , <operand> }
```

Assignment operators, in particular the simple assignment (=), are represented as follows:

```
<assign-op> ::= ASSIGN { <left-operand> , <right-operand> }
```

Operands can usually be either an expression, a literal, an identifier or an operation. In particular, `left-operand`s are identifiers or expressions that evaluate to a left-hand side expression.

Statements are defined as follows:

- blocks contain a list of statements. They form the hierarchical unit of our abstract syntax tree. All statements belonging to the same block are represented as subtrees stemming from the root node of this block.

```
<block> ::= { <stmt-list> }
<stmt-list> ::= <stmt>
              | <stmt-list> <stmt>
```

- variable statements introduce new variables to the program.

```
<var-stmt> ::= VAR { <var-decl-list> }
<var-decl-list> ::= <var-decl>
                  | <var-decl-list> , <var-decl>
<var-decl-list> ::= ID [[= <assign-expr>]]
```

Variable statements often declare several variables at once. Variables are always assigned the `undefined` value at creation, but can be assigned an initial value subsequently.

- conditional statements encompass `if` statements, as well as, `switch` statements and ternary operators.

```
<cond-stmt> ::= BRANCH ( ) { <stmt-list> , <stmt-list> }
              | BRANCH ( ) { <stmt-list> }
```

The condition is discarded and each branch (`then` and `else`) are subtrees of the `BRANCH` node.

- iteration statements deal with all kinds of loops while discarding the loop conditions (initialization, stop condition, update).

```
<loop-stmt> ::= LOOP { <stmt-list> }
<continue-stmt> ::= CONTINUE
                  | CONTINUE { ID }
<break-stmt> ::= BREAK
               | BREAK { ID }
```

Statements containing `continue` and `break` keywords are optional.

- return statements redirect the control-flow outside the function where it is called.

```
<return-stmt> ::= RET
                | RET { <expr> }
```

- with statements execute statements within the block as computations within another object lexical environment.

```
<with-stmt> ::= WITH { <expr> , <stmt-list> }
```

- labelled statements are seldom used and are only used with continue and break statements.

```
<label-stmt> ::= LABEL { ID , <stmt-list> }
```

- try/catch statements allow for exception handling within JavaScript.

```
<try-stmt> ::= TRY { <stmt-list> , <catch-stmt> }
<catch-stmt> ::= CATCH ( ) { <stmt-list> }
```

Function definitions allow to declare named and anonymous functions with or without parameters:

```
<func-decl> ::= FUNC { ID , [[<param-list> ,]] <func-body> }
<func-expr> ::= FUNC { [[ID , <param-list> ,]]
<param-list> ::= ID
               | <param-list> , ID
<func-body> ::= <stmt-list>
```

Each JavaScript file is parsed into an abstract syntax tree that follows such specification. A great deal of simplification has been introduced in such abstraction in order to reduce as much as possible the generation and traversal of the AST. Patterns to learn are therefore shorter, but may be imprecise at times. A slight twist in our approach is that we retain some of the native objects and functions of the scripting language whenever used in order to reinforce the semantics of the constructs.

7.5 Matching Subtrees to Detect Obfuscation

As stated earlier, considering JavaScript programs as ASTs, obfuscated contents within a JavaScript program may represent part or whole of the program. Therefore, obfuscated contents can be viewed as a subtree in the program's AST. In this dissertation, a subtree is not to be confused with a subtree as defined in graph theory, that is, a graph whose set of edges (or links) and vertices (or nodes) are subsets of the edges and vertices of a given tree.

Definition 6. *For a tree T, a (bottom-up) subtree T' consists of a node from T and all its descendants.*

The sets of nodes, N', and links, L' of T' are subsets of the sets of nodes and links of T, respectively: $N' \subseteq N$ and $L' \subseteq L$

Moreover, the order of siblings in T is preserved in T'.

In this dissertation, we explore some methods to detect the presence of obfuscation as a subtree within a JavaScript program's AST.

7.5.1 AST Paths

Expressing subtrees is not straightforward and there is a high risk of information loss whenever a tree is transformed into another representation. In particular, our first attempt was to consider each path of a given AST, that is, the result of traversing an AST from root to leaf without visiting the same node twice. Such approach is prone to combinatorial explosion since each time a node has children, a number of paths equal to the number of children will be generated. For example, here are some abstract paths generated from the AST displayed in Figure 16, during a *pre-order* traversal, that is, visiting the child nodes starting from the leftmost first:

```
LOOP->ASSIGN->ID
LOOP->ASSIGN->ADD->ID
LOOP->ASSIGN->ADD->CALL->ACCESS->String
LOOP->ASSIGN->ADD->CALL->ACCESS->fromCharCode
...
```

The `root` node has been deliberately omitted.

Detecting a common subtree is not trivial and is actually done in two steps:

- an obfuscation pattern, as represented by a subtree, is most probably the intersection of several abstract paths, since children are separated during traversal. Detecting an obfuscation pattern involves detecting all abstract paths.
- an obfuscation pattern, as represented by a subtree, is not necessarily rooted to the AST's root, and more likely to be nested. This means that the subtree is a subpath of an abstract path.

Manipulating such structures is costly and the ordering of the subtree can be lost, without knowledge of the syntax. In particular, if only one occurrence of a path is retained in order to reduce the combinatorial explosion, a program is then represented as a set of unique abstract paths, which does not presume of the ordering of these paths.

7.5.2 AST Fingerprinting

In a similar fashion, with the collaboration of Dr. Miyamoto from the University of Tokyo and Mitsuaki Akiyama from NTT Information Sharing Platform Labs [99],

we studied how to quickly detect obfuscation, and in particular, similarly obfuscated programs. Classification of obfuscated programs is possible employing this method.

AST fingerprinting (ASTF) has originally been proposed by Chilowicz et al. [23] as a method for plagiarism detection in computer programs.

Definition 7. *A fingerprint of the subtree t rooted at node A is the tuple:*
$(w(t), \mathcal{H}(t), p(t), parent(t))$
where:

- $w(t)$ *is the weight of the subtree t, that is, the number of nodes in this subtree (root node A included);*
- $\mathcal{H}(t)$ *is a hash value reflecting the structural properties of the subtree t;*
- $p(t)$ *is a pointer to the root node of t in the AST, that is node A;*
- $parent(t)$ *is a pointer to the parent node of t.*

If the subtree has n child subtrees, $t_1, \cdots t_n$, a hash value of this subtree, ($\mathcal{H}(t)$), is $\mathcal{H}(r(t) \cdot \mathcal{H}(t_1) \cdots \mathcal{H}(t_n))$ where $\mathcal{H}$ is a cryptographic hash function, such as SHA-1 [103] or MD5 [120]. Note that the hash variables are calculated in bottom-up manner, that is, leaves are calculated first.

For a given JavaScript program, a repository of ASTFs can be computed comprising a fingerprint for each given subtree. To reduce the size of such repository, it is reduced to unique occurrences of ASTFs. In [99], samples were clustered using a matrix approach. By merging all ASTF repositories, a database of all unique ASTFs occurring in a dataset is built. A sample can then be characterized by the absence or the presence of one or several occurrences of a given ASTF. Samples are grouped around commonly shared ASTFs.

ASTF is an interesting approach for clustering samples based on the subtrees they include. Samples bearing signs of obfuscation can then be detected through such method. However, generating every ASTF is time consuming, which is not appropriate for fast detection.

7.5.3 Subtree Matching by Deterministic Pushdown Automaton

As we observed previously, it is generally a bad idea to transform the subtree structure to a string-based structure, since there is an inevitable loss of information. However, string matching algorithms are straightforward and cheap.

The requirements for an obfuscation detector that can precisely extract obfuscated contents are:

- learn obfuscation patterns;
- model patterns into something that can be matched;
- apply a matching algorithm to candidate ASTs to quickly detect the presence of learned obfuscation patterns;

The first step is to gain knowledge about obfuscated JavaScript programs. Only surveying literature will not give a practical view of current obfuscation techniques used in JavaScript. Therefore, the choice of a learning-based approach seems appropriate to provide up-to-date knowledge to our system.

The second requirement prompts the design of a model complete enough to accommodate several obfuscation patterns, and compact enough to be usable. Previous attempts have generated costly systems in terms of space and memory, and possibly time, with some loss of information. A model able to retain the subtree structure of learned obfuscation patterns would save space, and be more accurate.

Finally, the last requirement is related to the second. However, while it is acceptable that learning obfuscation patterns takes a little time to compute, the technique adopted for pattern matching should be fast to be usable in a runtime context. Essentially, it is bound to the model chosen to represent subtrees.

Arbology [96] is such discipline that attempts to apply algorithms inspired from string processing algorithms to tree structures. In particular, arbology makes use of linear notations of trees such as *prefix* or *postfix* notations since trees are *linearized* through any sequential algorithm. Flouri et al. [53] demonstrate some interesting properties on linear notations and especially :

Theorem 1. *Given a tree t and its prefix notation $pref(t)$, all subtrees of t in prefix notation are the substrings of $pref(t)$.*

The reciprocal proposition is not always true.

By analogy with well-known string processing algorithms that make use of finite state machines, Flouri et al. adopted the PDA as the model of computation for the processing of linear notations of trees, obtained by recursive traversal.

Definition 8. *A nondeterministic pushdown automaton (nondeterministic PDA) is a seven-tuple*

$M = (Q, \mathcal{A}, G, \delta, q_0, Z_0, F)$
where:

- Q *is a finite set of states*
- A *is an input alphabet (the same used to generate the language used in trees)*
- G *is a pushdown store alphabet, an alphabet specific made of symbols to used exclusively in push and pop operations on the pushdown store*
- δ *is mapping from* $Q \times (\mathcal{A} \cup \varepsilon) \times G$*, the set of PDA configurations, into a set of finite subsets of* $Q \times G*$*, the set of images by* δ
- $q_0 \in Q$*, the initial state*
- $Z_0 \in G$*, the initial contents of the pushdown store*
- $F \subseteq Q$*, the set of final accepting states*

Algorithms to construct a subtree matching pushdown automaton (PDA) were proposed in [53, 52]. In particular, they generalize an algorithm that matches all possible subtrees of a tree in its prefix notation, to an algorithm that matches a set of given trees within a tree. The construction of such multiple subtree matching PDA is done in three steps:

1. construction of a PDA accepting a set of trees $P = t_1, t_2, ..., t_m$ in their prefix notation;
2. construction of a nondeterministic subtree matching PDA for a set of trees $P = t_1, t_2, ..., t_m$ in their prefix notation;
3. transformation of an input-driven nondeterministic PDA to an equivalent deterministic PDA.

Definition 9. *A pushdown automaton* $M = (Q, \mathcal{A}, G, \delta, q_0, Z_0, F)$ *is deterministic if it holds:*

- *there is at most one allowed transition from any PDA configuration:*

  $\forall q \in Q, a \in \mathcal{A} \cup \{\varepsilon\}, Z \in G, |\delta(q, a, Z)| \leq 1$

- *an* ε*-transition is possible from* q *with* Z *on top, only if no other transition is possible:*

  $\forall q \in Q, a \in \mathcal{A}, Z \in G, \delta(q, a, Z) = \emptyset$ *and* $|\delta(q, \varepsilon, Z)| \leq 1$

Contrarily to finite automata, where a well-known method exists to convert nondeterministic finite automaton (NFA) to an equivalent deterministic finite automaton (DFA) that recognizes the same language, an equivalent deterministic pushdown automaton does not exist for some nondeterministic pushdown automata. However, Melichar demonstrates the determinisation of *input-driven* pushdown automata [96], among other classes of PDA.

Definition 10. *A pushdown automaton is input-driven if each pushdown operation is determined only by the input symbol.*

JavaScript ASTs constructed following the syntax described previously satisfy the requirements of being *rooted*, *directed*, *labelled*, *ranked* and *ordered*, leading to the applicability of pushdown automata theories [96]:

- ASTs are *rooted*, *directed* and *ordered* as derived from parse trees that bear these properties. (1) The root does not have any antecedent node; (2) all other nodes have one and only one antecedent; (3) there is just one path from the root to any node of the AST; (4) children of any node are ordered.
- Each node of the AST bears a label as described in the syntax above;
- The alphabet used in ASTs is *ranked* but some expressions such as function expressions or function calls accommodate a variable number of body statements or parameters, respectively, which leads to a varying *arity*, that is the number of children of their corresponding node in the AST. This can represent a challenge in implementation.

Confident in the ability of a pushdown automaton to address the issue of subtree matching in JavaScript ASTs, we implemented such tool and subsequently evaluated the accuracy of such approach.

7.6 Implementation

In the first step, we generate AST fingerprints (ASTFs) for each JS file present in our learning dataset. Each JS file is parsed and the corresponding AST is generated by applying the abstraction described in the previous section. The obtained AST is then processed by a Perl script that computes its fingerprint as explained in Section 7.5.2. The output of such processing provides a list of ASTFs corresponding to the subtrees composing the script's AST. We assume that recurring fingerprints, among our

dataset samples, denote recurring subtrees. These subtrees should be the expression of obfuscating transformations provided our dataset is composed of obfuscated JS files. Fingerprints varying for slight variations of the code, this ensures that only invariant subtrees are learned. However, some false negatives can occur when confronted with "variants", and this should be looked for manually.

Subtree selection is done manually. After clustering samples of the learning datasets around unique occurrences of ASTFs, we merge clusters containing similar samples (according to their concordance rate). Among the recurring ASTFs of each cluster, we pick one or several subtrees that will form our knowledge base of obfuscation patterns.

The second step of this system is to apply principles of arbology described previously to match the obfuscation patterns we extracted from the learning dataset. We implemented a tool to generate a deterministic subtree matching pushdown automaton from an XML file that represents the obfuscation patterns in the form of ASTs. The tool sequentially applies the three algorithms developed by Flouri et al [52]. The output of such tool is the set of states, transitions and final states of the subtree matching PDA.

To perform matching on prefix notation of candidate ASTs, we modified a Python implementation of a finite state machine to make it to accommodate a stack (actually a simple counter that adds or subtracts node arities). The program first loads the transitions in memory and initializes the stack. Then, the JavaScript program, in which we wish to detect obfuscation patterns, is transformed to an AST and its prefix notation is fed to the pushdown automaton simulator. Upon matching, the matched subtree is returned by the program.

All prototypes manipulating ASTs and computing transitions for the PDA were implemented in Ruby using the johnson [9] library, a library that manipulates JavaScript code. All prototypes computing ASTFs and similarity between sets of ASTFs were contributed by Dr. Daisuke Miyamoto and implemented in Perl. The subtree matching PDA prototype is based on a finite state machine prototype written in Python.

7.7 Experiment

The preliminary experiment evaluates jointly the ASTF and subtree matching methods in characterizing a dataset of obfuscated JavaScript programs, in particular ones collected from malicious websites. Using ASTF, we are able to discover recurring subtrees we assume to be characteristic of obfuscating transformations. To that extent, human knowledge is also involved. We also evaluate the accuracy and performance of subtree

matching using a PDA on a set of subtrees relatively large compared to the dataset of ASTs it is extracted from.

7.7.1 Dataset Description

We used URLs listed by MalwareDomainList.com (MDL) [95], a website that publishes a large URL blacklist related to malware infected web pages. We collected HTTP sessions on different domains on February 8th, 14th and 16th, 2011. The data collection procedure is as follows: a client honeypot crawls all the URLs on the list and detects malicious ones. The honeypot's environment runs Internet Explorer 6.0 on a Windows XP SP2 platform [3], software versions known to include exploitable vulnerabilities. Vulnerable versions of Adobe Reader, Flash Player, JRE, WinZip, and QuickTime plug-ins were also installed on the system. During the anti-Malware engineering WorkShop (MWS) [74], this dataset has been distributed to academic and industrial researchers for research promotion.

Additionally, we added another dataset comprised of JS files from randomly chosen 25 domains among Alexa Top 100 [4]. This dataset serves mostly the role of control set.

7.7.2 Evaluation

In this preliminary experiment, we evaluated the precision of our methods by attempting to identify, in a dataset of obfuscated scripts, obfuscating transformations learned in a previous dataset. As suggested here, we applied the two techniques in Sections 7.5.2 and 7.5.3 at two distinct stages. We assume obfuscation patterns may be recurring among samples from a dataset of obfuscated scripts. Moreover, hierarchical syntactic structures of these scripts may be similar. A classification is therefore possible. Obviously, learning methods also suffer from limitations: we cannot guarantee the completeness of a dataset, and any error in the selection of relevant patterns of obfuscation will impact the accuracy of the system. We will see how it affected our results and discussed the reasons and the perspectives.

Using ASTFs, we calculated the concordance rate between every pair of files in the learning dataset (Day 1 (February 8th) of the MWS dataset). From these concordance rates, we were able to regroup samples, including duplicates, in 16 clusters as shown in Figure 17. From each cluster, we manually picked one or several subtrees based on the recurrence of its corresponding ASTF and some additional criteria:

- we excluded subtrees rooted at the root of an AST, as it is not a *proper* subtree;

- we excluded subtrees of which ASTF weight (the size of the subtree) is less than 7;
- we excluded subtrees of which ASTF weight is more than 150, in order to reduce the number of transitions of the automaton to be built;
- we tried to pick one subtree for a given cluster when the set was made up of non-duplicates and the subtree was the most recurring one among samples;
- we tried to pick several subtrees when no subtree were outstandingly recurring in a cluster. 2 or 3 subtrees would be taken based on their location in the AST or on their length.

Important to the performance of the PDA is not the number of subtrees it accepts, but the length of the subtrees. Longer subtrees reduce the number of false positives but increase the time overhead. During the learning stage, we extracted 32 recurring subtrees representing invariant parts of or a whole obfuscation technique from the 16 clusters. These 32 subtrees were as small as 7 nodes and as big as 127 nodes. Building the subtree matching pushdown automaton generated 37,730 transitions.

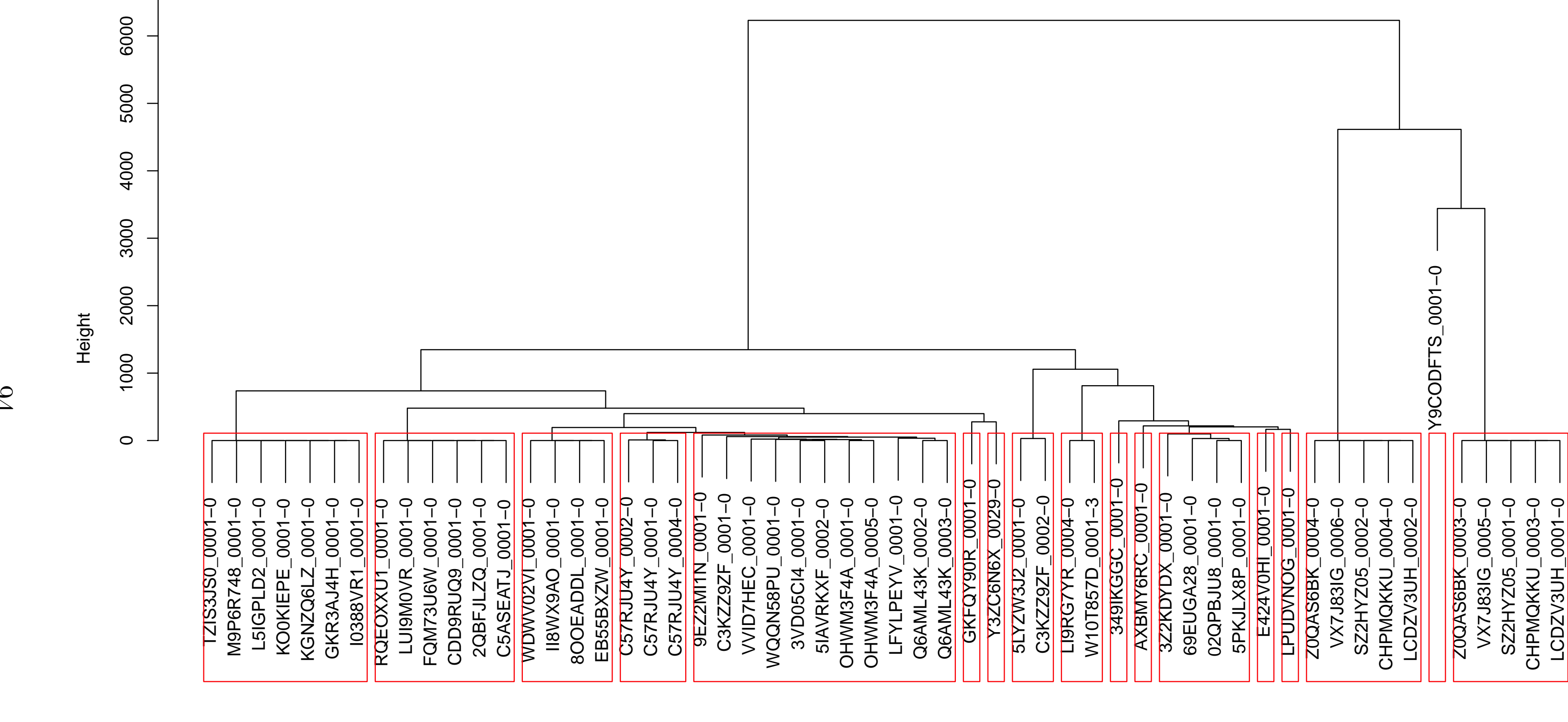

Figure 17. Day 1 samples clustered by Ward's method based on their ASTF distribution

For every testing dataset, we extracted scripting contents from HTML contents through HTTP transaction filtering and aggregation of scripts. The scripts obtained were then transformed into ASTs of which prefix notation was subsequently fed to the PDA. Results are displayed in separate Tables 4, 5 and 6 for the Day 2, Day 3 and Alexa 25 datasets, respectively.

In each table, we expressed the learned subtrees on two lines. *Subtree ID* indicates one of the 32 individual subtrees that expresses part or a whole obfuscating transformation. *Obfuscation ID* (from A to P) indicates one of the 16 obfuscating transformations around which we clustered samples of the learning dataset. We further regrouped matched samples according to the subtrees that were matched. We call *obfuscation pattern*, or simply *pattern*, the combination of 1 or several obfuscating transformations. Over the 3 testing datasets (Day 2 and 3, Alex 25), we identified 19 different obfuscation patterns (pattern ID 1 to 19). For each pattern we discovered in the testing phase, we counted the number of matched samples. In particular, we can observe that some patterns are exclusive to one day or the other: patterns 6, 11 and 12 for Day 2 (see Table 4); patterns 16, 17 and 18 for Day 3 (see Table 5). Obviously, the domains visited are different but this can also indicate that new tools were used to generate these obfuscated scripts.

subtree ID	02QPBJU8_0001-0/0	2QBFJLZQ_0001-0/0	2QBFJLZQ_0001-0/1	2QBFJLZQ_0001-0/2	5LYZW3J2_0001-0/0	5LYZW3J2_0001-0/1	8OOEADDL_0001-0/0	9EZ2MI1N_0001-0/0	9EZ2MI1N_0001-0/1	9EZ2MI1N_0001-0/2	C3KZZ9ZF_0001-0/0	C3KZZ9ZF_0001-0/1	C57RJU4Y_0001-0/0	C57RJU4Y_0001-0/1	CHPMQKKU_0003-0/0	CHPMQKKU_0003-0/1	GKFQY90R_0001-0/0	GKR3AJ4H_0001-0/0	GKR3AJ4H_0001-0/1	LI9RG7YR_0004-0/0	LI9RG7YR_0004-0/1	LI9RG7YR_0004-0/2	OHWM3F4A_0001-0/0	Q6AML43K_0002-0/0	Q6AML43K_0002-0/1	VVID7HEC_0001-0/0	VVID7HEC_0001-0/1	Y3ZC6N6X_0029-0/0	Y3ZC6N6X_0029-0/1	Y3ZC6N6X_0029-0/2	Y9CODFTS_0001-0/0	Y9CODFTS_0001-0/1	samples matched
obfuscation ID	A	B			C		D	E			F		G		H		I	J		K			L	M		N		O			P		
1	X																																8
2																								X	X								7
3																		X	X														11
4																									X								4
5																												X	X	X			2
6										X																							1
7											X	X																					1
8					X	X																											2
9																				X	X	X											2
10		X		X																					X								1
11													X	X									X										2
12													X	X																			1
13																							X										11
14																	X																1
15																										X	X						1

Table 4. Matched samples of Day 2 dataset clustered by obfuscation patterns

In Day 2 dataset (see Table 4), 55 samples out of 82 were found to contain subtrees we learned, displaying 15 different obfuscation patterns. Patterns do not correspond to a single obfuscating transformation or tool but may also be the combination of several obfuscation techniques. Indeed, obfuscation techniques perform better when combined, and obfuscated programs are themselves altered through several layers of obfuscation. Therefore, a program may be obfuscated once using one technique, and further modified through another transformation. A program may also be obfuscated by several techniques targeting distinct parts of the program as it is the case for the pattern 11, which combines obfuscation techniques G and L. It is also possible to detect unknown encoding techniques. In fact, by splitting an obfuscation technique in several parts represented by distinct subtrees, it may be possible to detect reused parts of an encoding routine in another obfuscation technique. For example, pattern 10 shows a case where two subtrees of obfuscation technique B and one subtree of M are combined into a new encoder. As for negative samples, it is important to distinguish cases where files were incomplete (around 5 samples), which prevented detection, from samples of which obfuscating transformation was previously unknown, that is, was not learnt during the learning stage. After investigation, 17 of the negative samples are false negatives representing 12 obfuscation techniques that were not present in the learning dataset. Finally, 5 samples were true negatives, samples from legitimate websites that were wrongly mixed in the malicious samples dataset. Since there were no false positive, we only computed the recall: $\frac{\text{true positives}}{\text{true positives}+\text{false negatives}} = 0.76$.

subtree ID	02QPBJU8_0001-0/0	2QBFJLZQ_0001-0/0	2QBFJLZQ_0001-0/1	2QBFJLZQ_0001-0/2	5LYZW3J2_0001-0/0	5LYZW3J2_0001-0/1	8OOEADDL_0001-0/0	9EZ2MI1N_0001-0/0	9EZ2MI1N_0001-0/1	9EZ2MI1N_0001-0/2	C3KZZ9ZF_0001-0/0	C3KZZ9ZF_0001-0/1	C57RJU4Y_0001-0/0	C57RJU4Y_0001-0/1	CHPMQKKU_0003-0/0	CHPMQKKU_0003-0/1	GKFQY90R_0001-0/0	GKR3AJ4H_0001-0/0	GKR3AJ4H_0001-0/1	LI9RG7YR_0004-0/0	LI9RG7YR_0004-0/1	LI9RG7YR_0004-0/2	OHWM3F4A_0001-0/0	Q6AML43K_0002-0/0	Q6AML43K_0002-0/1	VVID7HEC_0001-0/0	VVID7HEC_0001-0/1	Y3ZC6N6X_0029-0/0	Y3ZC6N6X_0029-0/1	Y3ZC6N6X_0029-0/2	Y9CODFTS_0001-0/0	Y9CODFTS_0001-0/1	samples matched
obfuscation ID	A	B			C		D	E			F		G		H		I	J		K			L	M		N		O			P		
1	X																																9
2																								X	X								9
3																		X	X														11
4																									X								8
5																												X	X	X			1
7											X	X																					1
8					X	X																											2
9																				X	X	X											2
10		X		X																					X								1
13																							X										6
14																	X																1
15																										X	X						1
16															X	X																	46
17									X	X																							1
18																															X	X	1

Table 5. Matched samples of Day 3 dataset clustered by obfuscation patterns

The Day 3 dataset (see Table 5) had 100 samples out of 116 that contained around 15 different obfuscation patterns. The samples in Day 3 dataset share many similarities with the Day 2 datasets and only a few new patterns were found. Similarly, among the negative samples, 4 were found not to contain a decoding routine, which is our target, therefore preventing detection. 12 samples did not return any match as the 6 obfuscation patterns they displayed were not in the learning dataset. The recall is therefore of 0.89.

subtree ID	02QPBJU8_0001-0/0	2QBFJLZQ_0001-0/0	2QBFJLZQ_0001-0/1	2QBFJLZQ_0001-0/2	5LYZW3J2_0001-0/0	5LYZW3J2_0001-0/1	8OOEADDL_0001-0/0	9EZ2MI1N_0001-0/0	9EZ2MI1N_0001-0/1	9EZ2MI1N_0001-0/2	C3KZZ9ZF_0001-0/0	C3KZZ9ZF_0001-0/1	C57RJU4Y_0001-0/0	C57RJU4Y_0001-0/1	CHPMQKKU_0003-0/0	CHPMQKKU_0003-0/1	GKFQY90R_0001-0/0	GKR3AJ4H_0001-0/0	GKR3AJ4H_0001-0/1	LI9RG7YR_0004-0/0	LI9RG7YR_0004-0/1	LI9RG7YR_0004-0/2	OHWM3F4A_0001-0/0	Q6AML43K_0002-0/0	Q6AML43K_0002-0/1	VVID7HEC_0001-0/0	VVID7HEC_0001-0/1	Y3ZC6N6X_0029-0/0	Y3ZC6N6X_0029-0/1	Y3ZC6N6X_0029-0/2	Y9CODFTS_0001-0/0	Y9CODFTS_0001-0/1	samples matched
obfuscation ID	A	B			C		D	E			F		G		H		I	J		K			L	M		N		O			P		
13																							X										18
19									X																								1

Table 6. Matched samples of the Alexa 25 dataset clustered by obfuscation patterns

dataset	nbr of samples	matched	not matched	tp	tn	fp	fn	precision	recall	accuracy
Day 2	82	55	27(5)	55	5	0	22(5)	100%	76.3%	83.3%
Day 3	116	100	16(4)	100	0	0	16(4)	100%	89.2%	89.2%
D2 + D3	198	155	43(9)	155	5	0	38(9)	100%	84.2%	84.6%
Alexa 25	148	19	129	8	129	11	0	42.1%	100%	92.5%
All	346	174	172(9)	163	134	11	38(9)	93.6%	84.9%	88.1%

Table 7. Recapitulative table of the preliminary experiment

As a way to control the accuracy of the identification, we also tested the Alexa dataset (see Table 6) that contained both unobfuscated and obfuscated benign samples and it resulted that 19 samples were detected as obfuscated out of 148. And among these 19 matched samples, 18 displayed the same pattern. After further investigation, it resulted that 8 of these files were compressed and did display some obfuscation. The true negative rate is therefore 0.92. This does not contradict our goal, which is to detect and extract obfuscated contents, whatever the intent is. Indeed, the intent, malicious or not, is not predictable from the sole obfuscated script itself and needs further analysis after deobfuscation.

Overall, identifying trees has been quite accurate (88%) on the testing and control datasets. Day 2 comprised JS samples quite different from the learning dataset that was much more similar to Day 3 dataset. Table 7 summarizes the numerical results (figures between parentheses were not counted as they represent incomplete samples). Samples matched in Alexa 25 dataset express the fact that the obfuscation type L may not be relevant (7 nodes, the smallest subtree of the learning set) while the obfuscating transformation denoted by the pattern ID 19 only relies on a single subtree (partial matching of obfuscation type E). In fact, much more credit should be given to obfuscating transformations involving every subtree of a given obfuscation ID.

On the other hand, pattern IDs also give some insights on the accuracy of our manual feature selection. Partial matching of obfuscating transformation, where one subtree is left out, may indicate that this particular subtree may not be relevant to the corresponding obfuscation ID. Some subtrees were also never matched, which indicate that they are not indicators of obfuscation or that they just did not occur in testing datasets. On the contrary, subtrees occurring in the control dataset might either indicate that a learned obfuscating transformation has been detected or that the subtree is actually not an indicator of obfuscation.

7.7.3 Performance

Considering the high number of transitions, it was expected that the process may take time and it is a challenge to design the smallest and most expressive set of subtrees able to characterize obfuscating transformations of a given dataset. We ran our automaton on top of a Mac OS X platform with a Dual-Core Intel Xeon (2 x 2.66GHz) with 8GB of DDR2 RAM. The script is written in Python and is far from being optimized. Nonetheless, the script processed the 82 samples from Day 2 with an average time of 243ms, the 116 of Day 3 in 244ms in average and the Alexa dataset containing 148

samples in 38.44 seconds, which represents an average processing time of nearly 260ms. As expected, the Alexa dataset, containing much bigger JavaScript files, took a greater time to be processed by the PDA but still remains in an acceptable range. Nonetheless, the PDA can scale with a large number of subtrees to be matched provided we limit their size.

7.8 Discussion

7.8.1 Other Obfuscating Transformations

Although the techniques described in this section are applicable to a large list of transformations, it better captures the hierarchical structures displayed in current widespread encoders and packers. Other types of transformations may also display characteristic structures but with a higher false positive probability. For example, opaque predicates may not be distinguished from legitimate control branching in the actual AST representation since control branchings are stripped off from the condition statement. Another example is variable aliasing, which is not very potent per se, but can result in a large amount of unnecessary code. These two examples are definitely better addressed through different techniques more in relation with the next step of deobfuscation. Detecting these transformations is actually more difficult in that they resemble legitimate code, syntactically speaking, which makes AST-based detection ineffective.

7.8.2 Irrelevant Code

One problem inherent to obfuscated programs is the presence of irrelevant code in the form of dummy instructions, that is, instructions that do not affect the true flow of execution but rather are here to misdirect an analyst. It affects both vertical (insertion of intermediate nodes) and horizontal (insertion of sibling nodes or subtrees) distances. However, since our PDA uses prefix notations, it simplifies the insertion of dummy instructions to the insertion of dummy symbols in a prefix notation string. One proposal we can make to tackle vertical insertion of dummy nodes relies on the principles of automata theory. Our PDA is governed by transition rules generated from learned subtrees. If we upgrade each state's transitions with a transition function that accommodates unexpected input symbols to remain on the same state, as would do an exception clause in a regular expression, we can achieve a more flexible subtree matching. Nonetheless, this transition function cannot accept infinite input words and we

would need to define a limit.

On the other hand, horizontal insertions can introduce a whole subtree, or even several subtrees that are irrelevant to the program's control flow. It becomes necessary to ignore whole subtrees from the subtree matching procedure, which may not be scalable. Using induced subtrees, instead of bottom-up subtrees, can reduce the impact of the insertion of whole branches in subtrees of interest. In fact, our proposed method lacks flexibility since it is an exact match. In particular, bottom-up subtrees, as defined in [22], are subtrees comprising all descendants of a given node in a tree. Such subtree represents a whole block in a program and it is not possible to arbitrarily select only some instructions in the block, if they are not consecutive. If the dummy instructions are regrouped at the end of the block, they can be ignored by truncating the prefix notation of the subtree.

Figure 18 shows a bottom-up subtree of the example AST presented in Figure 16. This bottom-up subtree represents the first instruction of a loop body. While it is possible to truncate this subtree, it is not possible to represent partial paths, that is, representing other instructions of this block in a bottom-up subtree, requires each instruction to be completely represented, starting from the first instruction. With induced subtrees [22], that is, subtrees stemming from a given node, from which leaf nodes are arbitrarily removed, it is possible to ignore part of the code, and preserve tree structures that are really specific of an obfuscating transformation. We can represent one or several paths, truncated or not, stemming from a given node. Figure 19 shows an induced subtree of the example AST in Figure 16. This subtree represents two non-consecutive truncated instructions from a loop body block.

Pruning a subtree involves cutting one or several paths from the root node of the subtree, or cutting the remaining of a path starting from a given node. This can be done at random, but may likely be more efficient if we have identified invariant nodes of an obfuscation pattern. The obfuscation pattern can thereby be reduced to its invariant nodes.

An additional proposal to fight against the discrepancies between slight variations of AST-based subtrees is to consider simplifying the AST at some point. Kamizono et al. [83] who experimented with abstract syntax trees in a similar approach to ours, noticed as well the lack of flexibility of the AST. In particular, they compared ASTs of slightly different samples and remarked that the resulting ASTs differed from a small amount of nodes. The obfuscation technique used in both samples was the same decoding routine but one of the two used an additional parameter. We speculate that a more data-flow sensitive representation should cover up for such differences. As a

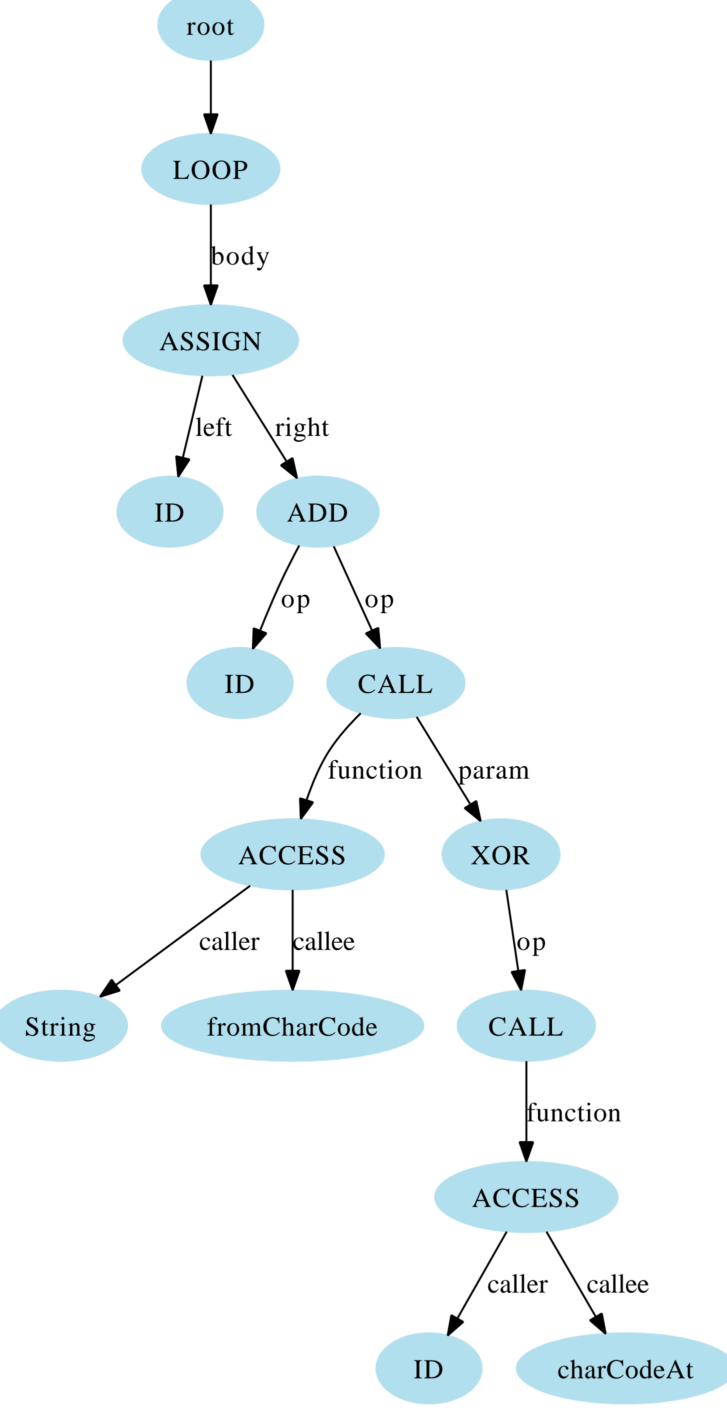

Figure 18. Bottom-up subtree of the example eval unfolding

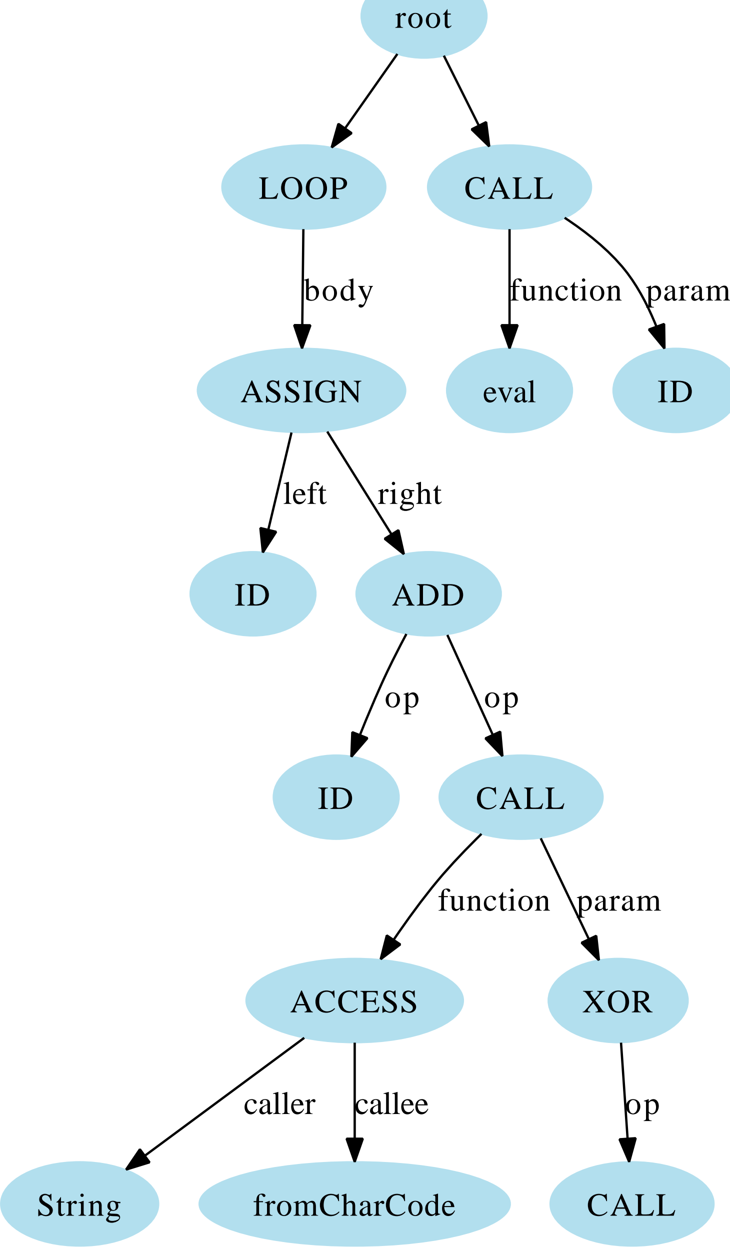

Figure 19. Induced subtree of the example eval unfolding

matter of fact, by deriving abstract semantic graphs (ASGs) from ASTs, it is possible to rearrange ASTs. However, the actual AST representation has some limitations against such rearrangements since function parameters are discarded.

7.8.3 Feature Selection

As for subtree selection, there were cases where representative subtrees were not obvious, so we had to arbitrarily agree on some rules as listed in Section 7.7.2. In the method we presented, subtrees were selected from some unique samples after clustering samples according to their ASTFs. We speculate co-clustering could have been used to identify fingerprints from recurring subtrees. In fact, we also considered using an automated tool to achieve such daunting task: Varro [93] is an open-source tool able to discover frequently occurring subtrees in a set of trees. Varro introduces condensed canonically ordered trees for efficiently discovering frequently recurring unordered subtrees. Though it minimizes memory use so that moderately large treebanks are tractable on commonly available hardware, the worst case memory performance is $O(nm)$ where n is the number of vertices in the treebank and m is the largest frequent subtree found in it. Because of these drawbacks, it resulted unsuitable to efficiently process programming language-based treebanks.

An appropriate feature selection able to automatically extract subtree invariants is still a challenge in this research.

7.9 Summary

Related works in obfuscation detection concentrated on string heuristic features or statistical features of obfuscated strings with more or less success. Previous work usually targeted encryption or encoding schemes where some patterns are recurring. However, by combining these transformations with less potent techniques such as variable aliasing, it is possible to thwart statistical features.

In order to cancel the effects of string randomization, we propose to concentrate on hierarchical structures of obfuscated code. This approach rather targets decoding routines than the obfuscated strings but allows to precisely extract code that will be used during the deobfuscation stage.

Abstract syntax tree is a representation that abstracts out variable and function names that are the target of string aliasing while preserving the structure of the program. Detecting an obfuscating transformation is then seen as detecting the presence of characteristic subtrees, representing the given obfuscation, in a program's AST.

In this section, we successfully applied abstract syntax tree methods to scripting languages, allowing us to cluster similar obfuscated contents regardless of the original code being encoded, and regardless of string measures. Learning is an adaptive approach but necessitates continuous update. Our subtree matching system has been able to quickly classify samples based on the type of obfuscating transformations the sample implements. We also demonstrated there were some trends in co-occurrence of some obfuscating transformations to encode different parts in a code, and its variation over time. The small number of combinations discovered let us speculate that a few automated tools are used to generate most of the samples we collected.

Our approach suffers from some limitations, the main one being the low flexibility of the representation used. Bottom-up subtree can only accommodate exactly identical subtrees and may generate false negatives. It also suffers from common drawbacks inherent to learning systems, namely the necessity of frequent updates of the patterns to match, and the influence of the learning dataset on the learned patterns. Especially in web security, due to the transience (also known as *shortlivedness*) of malicious web pages, it is difficult to obtain comprehensive datasets of malicious web pages. Additionally, the sophistication of cloaking techniques has also hindered the collection and the sharing of malicious samples datasets.

Finally, an appropriate subtree selection still needs to be implemented to reduce false positives. This is left as future work though we started tackling this issue using co-clustering techniques based on ASTFs. We may benefit from larger datasets, but this also implies more tedious dataset processing and analysis.

Precisely extracting obfuscated contents is essential to the deobfuscation stage since it reduces the amount of code to be processed. The two next sections describe our attempts to emulate the deobfuscation stage and automate this processing.

8. Automated Deduction

> Mathematical reasoning may be regarded rather schematically as the exercise of a combination of two facilities, which we may call intuition and ingenuity.
>
> — Alan Turing

Deobfuscation, in particular considering opaque predicates, can be seen as a dual process of identification and evaluation of obfuscated contents that contribute to the concealment of the original code [29]. In the actual context of source code analysis, deobfuscation is a simplification process that can benefit from code optimization techniques such as control-flow graph reduction, dead code removal, constant value propagation and folding, as well as other heuristics. In this dissertation, we are interested in automating such process in order to provide realtime deobfuscation to our proposed system.

Binary obfuscation is well documented in literature and former approaches to automated deduction of binary program deobfuscation has attracted our attention. With the aim to develop automated reasoning approach in our proposal, we review here some notions important to the comprehension of equational reasoning and rewriting systems.

8.1 First-order Logic

Reasoning is a human ability to make inferences [114], associated with *cognition*, the faculty to process information and apply knowledge. Automated reasoning is the area of computer science dedicated to produce computer systems that automate the process of reasoning. The goal of such discipline was originally to mechanize different forms of reasoning, but it has been often associated to deductive reasoning as practiced in mathematics and formal logic. In fact, it is best seen as [114]:

> Providing an algorithmic description to a formal calculus so that it can be implemented on a computer to prove theorems.

The choice of the deduction calculus or logic is dependent on the problem domain, that is, the class of problems to solve. They are many options, the simplest one being *propositional logic*[3] from which other logics derive. In particular, *first-order logic* is

[3] Propositional logic will not be detailed here but the reader can refer to any introductory literature to formal logic.

distinguished from propositional logic by its use of *terms*, *predicates* and *quantifiers*.

What follows is a brief reminder on first-order logic. First-order logic deals not only with facts, as propositional logic does, but also with objects and relations. The objects are denoted using *terms*.

Definition 11. *Let L be a language of first-order logic. L is defined by the triplet $(\mathcal{C}, \mathcal{F}, \mathcal{P})$ where:*

- *$\mathcal{C}$ is the set of individual constants*
- *$\mathcal{F}$ is the set of function symbols, each with an arity ≥ 1*
- *$\mathcal{P}$ is the set of predicate symbols, each with an arity ≥ 1, including the* equality *predicate noted $=$*

A term of L is either:

- *a variable $x \in \mathcal{V}$ ($\mathcal{V}$ is the fixed infinite set of symbols called variables)*
- *a constant $c \in \mathcal{C}$*
- *a functional relation $f(t_1, ..., t_n)$ where $f \in \mathcal{F}$ is an n-ary function and $t_1, ..., t_n$ are terms of L*

A ground term *is a variable-free term.*

Predicates are applied to terms to form atomic formulas or atoms:

Definition 12. *If $p \in \mathcal{P}$ is a predicate and $t_1, ..., t_n$ are terms of L, then:*
$p(t_1, ..., t_n)$ is a formula of L.

Additionally, the combination of formulas, using logical connectives, is also a formula.

Logical connectives include $\vee$ (disjunction), $\wedge$ (conjunction), $\neg$ (negation), $\Rightarrow$ (implication) and $\Leftrightarrow$ (equivalence). Quantifiers are the universal quantification noted $\forall$ and the existential quantification noted $\exists$.

The facts are represented by *sentences*.

Definition 13. *A sentence is either:*

- *an atom*
- *a formula using connectives*

- *a closed formula (to which it may also mostly refers in some conventions)*

A formula is closed if it contains no free variables*.*

A free variable *is a variable that is not bound by a quantifier. For example, in the following, y is bound while x is free:*

$\exists y\ p(x, y)$

Having established what a language and a sentence are in first-order logic, we can introduce the notion of theory. A theory is a set of sentences in a formal language.

Definition 14. *A theory T in first-order logic is the pair (L, Γ) where Γ is a set of sentences of L, called* axioms*.*

Theorems derive from axioms, or other theorems, by rules of deduction *or demonstrated by a* proof*. Unlike theorems, axioms are accepted without proofs.*

8.2 Equational Logic

A common ground for mathematicians and logicians, *equational reasoning* is deemed the simplest and most powerful formal method [106]. It has also garnered interest from computer scientists: functional programs are essentially sets of equations (typically with higher-order functions) and the execution of such programs is then some kind of equational reasoning [113]. A set of equations is also called an *equational system*.

Definition 15. *An equation is an expression of the form:*

$s = t$ where s and t are terms.

An equational system E is a theory in equational logic of the form (L, Γ) with L a first-order logic language with no predicates except $=$ and Γ a set of equations of L.

Reasoning from a set of equations derives new equations from the existing equations and axioms of E. These new equations are formed by replacing occurrences of a given term s by another term t following an equality $s = t$. $s = t$ is therefore a justification for the reasoning step that led from the original equation to the new equation. Such replacement is called *substitution*.

Definition 16. *Let $L = (\mathcal{C}, \mathcal{F}, \mathcal{P})$ be a language of first-order logic and $\mathcal{T}$ the set of terms of L, a* substitution *of L is a total function $\sigma : \mathcal{V} \rightarrow \mathcal{T}$*

If $v_1, ..., v_n$ are variables of $\mathcal{V}$ and $t_1, ..., t_n$ are expressions of $\mathcal{T}$, then a substitution σ is a set of mappings $\{v_1|t_1, ..., v_n|t_1\}$ between variables and expressions.

The application of σ to an expression E_i of L is written $E_i\sigma$ and is defined recursively by:

- *if $x \in \mathcal{V}$, $x\sigma = \sigma(x)$*
- *if $c \in \mathcal{C}$, $c\sigma = c$*
- *if $f \in \mathcal{F}$ an n-ary function and $t_1, ..., t_n$ are terms of L, $f(t_1, ..., t_n)\sigma = f(t_1\sigma, ..., t_n\sigma)$*
- *if $p \in \mathcal{P}$ an n-ary predicate and $t_1, ..., t_n$ are terms of L, $p(t_1, ..., t_n)\sigma = p(t_1\sigma, ..., t_n\sigma)$*
- *if A and B are formulas of L, $(\not A)\sigma = \not(A\sigma)$ and $(A \Rightarrow B)\sigma = (A\sigma \Rightarrow B\sigma)$*

Additionally, we define matching *for s and t terms of L, and A and B formulas of L:*

- *s* matches *t if there is a substitution σ of L such that $s = t\sigma$*
- *A* matches *B if there is a substitution σ of L such that $A = B\sigma$*
- *if s matches t then s is an* instance *of the* pattern *t*

An instance of an expression E_i, noted $E_i\sigma$ is therefore received from E_i by simultaneously replacing all occurrences of v_i by the respective t_i for $1 \leq i \leq n$. A sequence of instances with their justification, that is the substitution, is a chain of equational reasoning. In particular, if for a set of expressions $\{E_1, ..., E_n\}$, there is a substitution σ such as:

$E_1\sigma = ... = E_n\sigma$

then σ is called a *unifier*.

Definition 17. *for s and t terms of L and A and B formulas of L,*

s and t unify if there is a substitution σ of L such that $s\sigma = t\sigma$.

Similarly, A and B unify if there is a substitution σ of L such that $A\sigma = B\sigma$.

The unifying substitution is called a unifier*.*

Additionally, a most general unifier (mgu) of s and t is a unifier σ such that, if there is unifier σ' of s and t, then there is a substitution τ such that, for all $x \in \mathcal{V}$:

$x\sigma' = (x\sigma)\tau$.

In an equational theory $T = (L, \Gamma)$, the fundamental rules of *inference* are *substitution* and *replacement*:

- if $T \models s = t$ ($s = t$ is true in T or, T satisfies the equation $s = t$), then $T \models s\sigma = t\sigma$ for every substitution σ of L
- if $T \models s = t$, then $T \models u = u'$ where u' is obtained by replacing one occurrence of s in u by t

This is enough to reasoning, that is deriving logical consequences of an equational system. However, this is inefficient since we do not know how to choose the substitutions. It becomes necessary to find restrictions of these inference rules that are still capable of deriving all equational consequences of an equational system [113].

8.3 Rewriting Systems

The central idea of term rewriting systems (TRS) is to orient an equation $s = t$ into a rule $s \rightarrow t$ indicating that instances of s may be replaced by instances of t, but not vice-versa [113].

Definition 18. *A term rewriting system of L is a set $\mathcal{R}$ of rewrite rules of L.*

A rewrite rule of L is a directed equation $s \rightarrow t$ of L such that all variables of t are contained in s.

Applying a rewrite rule $s \rightarrow t$ to a term u is done in two steps. First, an occurrence of a term v is found in u that matches s. Then, by noting the matching substitution σ, a term u' is obtained by replacing the occurrence of v in u by $t\sigma$. The result of the application of the rewrite rule $s \rightarrow t$ to the term u is therefore the term u'.

A rewrite system $\mathcal{R}$ defines a *rewriting relation* or *reduction relation* $\rightarrow_{\mathcal{R}} \subseteq \mathcal{T} \times \mathcal{T}$ as the smallest relation containing $\mathcal{R}$ closed under substitution and replacement:

- if $s \rightarrow_{\mathcal{R}} t$, then $s\sigma \rightarrow_{\mathcal{R}} t\sigma$ for every substitution σ of L;
- if $s \rightarrow_{\mathcal{R}} t$, then $u \rightarrow_{\mathcal{R}} u'$ where u' is obtained by replacing one occurrence of s in u by t.

Additionally, we define the following closures for the rewriting relation $\rightarrow_{\mathcal{R}}$:

- $\rightarrow^*_{\mathcal{R}}$, the reflexive-transitive closure, that is the smallest reflexive and transitive reduction relation that contains $\rightarrow_{\mathcal{R}}$;
- $\leftrightarrow_{\mathcal{R}}$, the symmetric closure, that is the smallest symmetric reduction relation that contains $\rightarrow_{\mathcal{R}}$;
- $\leftrightarrow^*_{\mathcal{R}}$, the reflexive-symmetric-transitive closure, that is the smallest reflexive, symmetric and transitive reduction relation that contains $\rightarrow_{\mathcal{R}}$.

For two terms s and t, $s \rightarrow^*_{\mathcal{R}} t$ means that there is finite number, even null, of rewrites from s to t. $s \leftrightarrow_{\mathcal{R}} s$ means that the terms s and t can be rewritten to each other.

Similarly, $s \leftrightarrow^*_{\mathcal{R}} s$ means that the terms s and t can be rewritten to each other in a finite number of steps.

A *derivation* for the rewrite $\rightarrow_{\mathcal{R}}$ is a sequence of the form $t_1 \rightarrow_{\mathcal{R}} ... \rightarrow_{\mathcal{R}} t_n$. A term t is then said *reducible* if there is a term u such that $t \rightarrow_{\mathcal{R}} u$; otherwise it is said *irreducible*. When a term cannot be rewritten (or reduced) anymore, it has reached a *normal form*:

Definition 19. *A term s of L is in* normal form, *relative to $\mathcal{R}$, if there is no term t such that $s \rightarrow_{\mathcal{R}} t$*

Additionally, t is a normal form *of s relative to $\mathcal{R}$ if $s \rightarrow^*_{\mathcal{R}} t$ and t is in normal form, relative to $\mathcal{R}$.*

8.3.1 Properties

Rewriting systems, as any logical systems, may satisfy properties of *soundness* and *completeness*. A term rewriting system is *sound* with respect to its equational theory if for all pair of terms (s, t), the rule $s \rightarrow_{\mathcal{R}} t$ implies that the theory satisfies the equation $s = t$. Similarly, a term rewriting system is said *complete* with respect to its theory for all pair of terms (s, t), if the fact that the theory satisfies the equation $s = t$ implies a reflexive-symmetric-transitive closure for the reduction relation on the set of terms.

A rewriting system also has specific properties:

- it is *Church-Rosser* if and only if:

  $\forall s, t \in L,\ s \leftrightarrow^*_{\mathcal{R}} t \Leftrightarrow \exists u,\ s \rightarrow^*_{\mathcal{R}} u$ and $t \rightarrow^*_{\mathcal{R}} u$

- it is *confluent* if and only if:

  $\forall s, t, u \in L,\ u \rightarrow^*_{\mathcal{R}} s$ and $u \rightarrow^*_{\mathcal{R}} t \Rightarrow \exists v,\ s \rightarrow^*_{\mathcal{R}} v$ and $t \rightarrow^*_{\mathcal{R}} v$

- it is *noetherian* or *finitely terminating* if and only if there is no infinite chain of reductions $t_1 \rightarrow_{\mathcal{R}} t_2 \rightarrow_{\mathcal{R}} ...$

- it is *normalizing* if and only if every term has a normal form

- it is *convergent* if and only if it is both terminating and confluent

8.3.2 Theorems

Here, we state some known theorems. Their demonstration is out of the scope of this dissertation.

Theorem 2. *A term rewriting system $\mathcal{R}$ is Church-Rosser if and only if it is confluent.*

Actually, the above equivalence can be extended to *semi-confluence*:

Definition 20. *A relation $\mathcal{R}$ is semi-confluent if and only if:*
$\forall s, t, u \in L,\ u \to_{\mathcal{R}} s$ *and* $u \to^*_{\mathcal{R}} t \Rightarrow \exists v,\ s \to^*_{\mathcal{R}} v$ *and* $t \to^*_{\mathcal{R}} v$

The above theorem has the following corollary [6]: if $\to_{\mathcal{R}}$ is confluent and $s \leftrightarrow^*_{\mathcal{R}} t$ then

1. $s \to^*_{\mathcal{R}} t$ holds if t is in normal form and,

2. $s = t$ if both s and t are in normal form.

From that corollary, it goes on that: if $\to_{\mathcal{R}}$ is confluent, then the normal form of a term of L exists.

Since every term has at least one normal form if $\to_{\mathcal{R}}$ is normalizing, it follows that: if $\to_{\mathcal{R}}$ is confluent and normalizing, every term has a unique normal form.

Proving properties of confluence, termination or normalization are well-studied topics in the field of rewriting systems. In the remaining of this section, we introduce Maude, a reflexive language that supports both an equational logic and a rewriting logic.

8.4 Maude as a Rewriting Framework

Maude is a declarative language based on rewriting logic, which has its underlying equational logic as a parameter. A Maude program is a logical theory, and a Maude computation is a logical deduction using the axioms specified in the theory [26]. Maude provides two classes of modules, namely *functional modules* included in a broader class of *system modules*. This inclusion reflects the sublogic inclusion in which *membership equational logic* is embedded in *rewriting logic* [26].

In the remainder of this section, we will only cover functional modules and their underlying logic, membership equational logic. System modules are also of interest, but are not covered in this dissertation since we did not developed further on this class of modules, although we feel the present dissertation can be extended using arbitrary rewrite rules implemented in rewriting theories.

8.4.1 Equational Membership Logic

Maude functional modules are based on an extension of *order-sorted equational logic* called *membership equational logic*. This means that terms in equational systems do

not belong to a single set but to distinct sorts, which are analogous to object types in programming languages. These sorts are ordered, that is, there exist an ordering relation between sorts, with subsetting and supersetting relations. Sorts are further regrouped under *kinds* at a more abstract level. For example, the kind `Number` can apply to a term. But it may be proven that a term inhabits specific sorts like `Nat` for a natural number, or `Int` for an integer.

A signature Ω in membership equational logic is a triple (K, Σ, Π) where (K, Σ) is a K-kinded signature (where K is a set of sorts) and Π is a restricted signature of predicates, so that Π only consists of unary predicates. A theory in membership equational theory is a pair (Ω, Γ) where Ω is a signature in membership equational logic, and Γ is set of sentences on the signature. In [27], Clavel et al. stipulate that atomic formulas in membership equational logic are either equations or sort membership assertions of the form $t : s$ where the term t has kind k and s belongs to a sort S_k. General sentences are Horn clauses on these atomic formulas, quantified by finite sets of K-kinded variables. That is, they are either conditional equations of the form:

$(\forall X)\ t = t'$ if $(\bigwedge_i u_i = v_i) \wedge (\bigwedge_i w_j : s_j)$

or membership axioms of the form:

$(\forall X)\ t : s$ if $(\bigwedge_i u_i = v_i) \wedge (\bigwedge_i w_j : s_j)$

In [97], Meseguer demonstrates the equivalence of membership equational logic with many-sorted Horn logic with equality. He derives from this equivalence that all results for many-sorted Horn logic with equality hold true for membership equational logic, as a sublogic of the former. In particular, he states that since soundness and completeness of the rules of deduction has been proven for order-sorted Horn logic with equality, and since many-sorted Horn logic with equality is a special case of the former, then an immediate corollary is the soundness and completeness of usual rules of deduction for a theory (Ω, Γ) in membership equational logic:

- **reflexivity**: $\Gamma \vdash_\Omega t = t$
- **symmetry**: $\Gamma \vdash_\Omega t = t' \rightarrow \Gamma \vdash_\Omega t' = t$
- **transitivity**: $\Gamma \vdash_\Omega t = t'$ and $\Gamma \vdash_\Omega t' = t'' \rightarrow \Gamma \vdash_\Omega t = t''$
- **congruence**: $\Gamma \vdash_\Omega t_1 = t'_1$ and ... and $\Gamma \vdash_\Omega t_n = t'_n$
  $\rightarrow \Gamma \vdash_\Omega f(t_1, ..., t_n) = f(t'_1, ..., t'_n)$
- **membership**: $\Gamma \vdash_\Omega t = t'$ and $\Gamma \vdash_\Omega t : s \rightarrow \Gamma \vdash_\Omega t' : s$
- **modus ponens**

Theorem 3 (Soundness and Completeness)**.** *For any atomic sentence* φ*,*

$\Gamma \vdash_{\Omega} \varphi \Leftrightarrow \Gamma \models_{\Omega} \varphi$

More detailed explanations on membership equational logic can be found in [27] and [14].

8.4.2 Functional Modules

Functional modules are equational theories in membership equational logic with data types and operations on them. As stated previously, Maude's equational modules are based on membership equational logic, and therefore supports multiple sorts, subsort relations, as well as, operator overloading and assertions of membership in a sort. In Maude, functional modules are declared as follows:

```
fmod <ModuleName> is <DeclarationsAndStatements> endfm
```

Declarations include the importation of other modules and the specification of sorts, operators and variables to be used in the functional module. While statements are a list of (conditional) equations and membership axioms.

Sorts are declared using the keyword `sort` followed by one or several sorts, in no particular order, and ending with a point. An optional subsort declaration specifies the (partial) order of subsorts if any, in the form:

```
sort Zero Nat .
subsort Zero < Nat .
```

The declaration of operators feature zero, one or several sorts in input and a single sort for the output. Several operators sharing the same signature can be declared at once using the `ops` keyword:

```
op zero : -> Zero .
op s_ : Nat -> Nat .
ops _+_ _*_ : Nat Nat -> Nat .
```

Variables are declared using the keyword `var` in association with a sort. Several variables of the same sort can be declared altogether using the keyword `vars`:

```
var Z : Zero .
vars I J K : Nat .
```

Computation in a functional module is accomplished by using the equations as rewrite rules until a *canonical*, or normal, form is found. The equations must therefore satisfy the requirements of being Church-Rosser (that is, confluent), terminating and sort-decreasing. This guarantees that reductions using these equations will lead to a unique canonical form, and that this canonical form will be assigned a sort that is smaller than any other sort in this functional module.

Statements can be distinguished between unconditional and conditional statements. Unconditional equations are declared using the keyword `eq`:

```
eq <Term-1> = <Term-2> [<StatementAttributes>]
```

Both terms must have the same kind. Additionally, any variable appearing in the right-hand term must also appear in the left-hand term, in order for the equation to be executable. This requirement is actually relaxed for conditional equations as we will see below. Equations can generally be specified in three different ways [26]:

1. in the above-mentioned style, in which they are assumed to be executable as simplification rules from left to right;

2. in the above-mentioned style, but with the `[nonexec]` attribute, making the equation non executable, and therefore not subject to executability requirements;

3. as equational attributes of specific operators.

In fact, an operator can be declared with an attribute telling Maude that it satisfies a certain property, such as associativity (using `[assoc]`) or commutativity (using `[comm]`). Such attribute should not be expressed as an equation in the specification, because it would be redundant if declared along the equational attribute itself, or if declared alone as an equation, it would alter the specification's operational semantics. A simple example is declaring the commutativity attribute as an equation, which yields a non-terminating chain of equational simplifications.

Unconditional membership axioms are declared using the keyword `mb`:

```
mb <Term> : <Sort> [<StatementAttributes>]
```

They specify that a term has a given sort. As equations, they optionally have statement attributes.

Conditional equations and membership axioms differ from unconditional ones by the presence of one or a conjunction of equational conditions in the right-hand of the equation (where `/\` represents the logical connective $\wedge$):

```
ceq <Term-1> = <Term-2>
if <EqCondition-1> /\ ... /\ <EqCondition-n>
[<StatementAttributes>]

cmb <Term> : <Sort>
if <EqCondition-1> /\ ... /\ <EqCondition-n>
[<StatementAttributes>]
```

These equational conditions are constituted of individual equations and memberships, of which concrete syntax is either of the following three variants:

1. ordinary equations `t = t'`. These equations are operationally interpreted as usual, that is, for the given substitution σ, $t\sigma$ and $t'\sigma$ are both reduced to their canonical form and compared for equality;

2. *matching* equations `t := t'`. These equations are mathematically interpreted as ordinary equations but are operationally treated in a special way, and therefore must satisfy special requirements. Variables in the term `t` do not necessarily appear in the left-hand side of the conditional equation and are actually instantiated when executing the equation by matching the term `t` with the canonical form of the term `t'`. `t` must be a *pattern* in order to decide equality. A term `t` is a pattern with respect to its functional module, if for any well-formed substitution σ, such that, for each variable x in its domain, the term $x\sigma$ is in canonical form with respect to the equations of the functional module, $t\sigma$ is also in canonical form;

3. abbreviated Boolean equations of the form `t` with `t` a term in the kind `[Bool]`, abbreviating the equation `t = true`. These equations are just a special case of ordinary equations.

8.4.3 Properties of Functional Modules

Functional modules are equational theories of the form $(\Sigma, E \cup A)$ in membership equational logic, with E the set of (conditional) equations and membership axioms specified as statements, and A the equations specified as equational attributes in operators (as seen above). Ground terms in the signature Σ form a Σ-algebra T_Σ. The *initial model* for the theory is the Σ-algebra defined by the equivalence classes of ground terms modulo $E \cup A$, denoted $T_{\Sigma/E\cup A}$. By adding a set of variables X as constants to the signature

Σ, we define a term algebra $T_\Sigma(X)$ where now the terms may have variables in X. Given a set of variables X, each having a given kind, a *substitution* is a kind-preserving function $\sigma : X \to T_\Sigma$. Such substitutions may be used to represent assignments of terms in T_Σ to variables in X. It is demonstrated [26] that such substitutions can be extended to homomorphic functions on terms of the form $\sigma : T_\Sigma(X) \to T_\Sigma$, also denoted substitution. Given a term $t \in T_\Sigma(X)$, corresponding to the left-hand side of an oriented equation, and a subject ground term $u \in T_\Sigma$, we say that t *matches* u if there is a substitution σ such that t and u are syntactically equals. A term t *rewrites* a term t' using a Σ-equation of the form $l = r$ if there is a subterm of t, at a given position, which is matched by l, and t' is obtained by replacing the matched subterm in t. This step of *equational simplification* is denoted by $t \to_E t'$ where the possible equations for rewriting are chosen from E. Let $\to_E^*$ be the reflexive-transitive closure of $\to_E$.

A set of equations E is *confluent* when any two rewritings of a term can always be *unified* by further rewriting:

if $t \to_E^* t_1$ and $t \to_E^* t_2$, then $\exists t'$ such that $t_1 \to_E^* t'$ and $t_2 \to_E^* t'$.

A set of equations E is *terminating* when there is no infinite sequence of rewriting steps:

$t_0 \to_E t_1 \to_E t_2 \to_E ...$

If E is both *confluent* and *terminating*, a term t can be reduced to a unique *canonical* form.

The last important property is the one of *sort-decreasingness*, that is, given a confluent and terminating set of equations E, the canonical form obtained by simplifying a term t by the equations of E should have the *least sort possible* among the sorts of all the terms equivalent to it by the equations of E. Additionally, it should be possible to compute this least sort from the canonical form itself, using only the operator declarations and the membership axioms.

8.4.4 The Reduction Command

`reduce` is the rewriting command of the functional modules that causes the specified term to be reduced using the equations and membership axioms in the given module:

```
reduce {in <Module> :} <Term> .
```

8.5 Summary

Maude is a declarative language that supports both equational and rewriting logic. In this section, we were particularly interested in functional modules that are based on

membership equational logic.

Membership equational logic is a type of equational logic that is an extension of order-sorted equational logic, and is also equivalent to many-sorted Horn logic with equality. To understand basic notions of *rewriting*, *substitution*, *matching* and *unification*, we reviewed notions on equational logic, as well as first-order logic.

Maude functional modules are equational theories in which computation (also known as *reduction*) is accomplished by using equations as rewrite rules until a canonical form is found. This is possible if the set of equations is confluent and terminating, as we have seen.

By considering a program as a set of equations, we assume it is possible to determine a reduction of the program to a normal form, provided the program is confluent and terminates. The next section discusses our attempt to apply such reasoning to the unpacking of encoding schemes, which are bound to deobfuscate, revealing the original unobfuscated program.

9. Automated Deobfuscation of Script Contents using Rewriting-based Emulation

> Avoiding danger is no safer in the long run than outright exposure. Life is either a daring adventure, or nothing.
>
> Helen Keller

As noted in Section 6, obfuscation is reversible, given enough space and time [29]. Web malware usually features obfuscated JavaScript programs that will eventually unpack during a deobfuscation stage, according to Cova et al. [32]. We further observed in Section 7 that most JavaScript obfuscation tools are encoders (or packers) that rely on the dynamic generation of additional code. Assuming obfuscated contents eventually unpack themselves, state-of-the-art analysis environments provide instrumented browser emulation that hooks the execution of JavaScript to extract deobfuscated JavaScript code for analysis purpose.

Despite the popularity of such approach, it has been suffering from a few common drawbacks. In particular, emulating a browser does not only require to provide a realistic environment including a `window` and a `document`, but also may require to instantiate several browser personalities. A personality is the combination of a specific browser version and a set of specific plugins. Rajab et al. [117] pointed out the difficulty for an administrator to maintain such system. Additionally, anti-analysis techniques have evolved [70] making emulators to fail against sophisticated cloaking techniques.

As seen in Sections 2 and 6, attacks are varied and all obfuscation techniques do not rely on code unfolding or encoding schemes but may rather combine simple techniques to interleave deobfuscation and exploitation stages. To increase code coverage over dynamic approaches, we advocate the implementation of an execution-less deobfuscation approach. Indeed, deobfuscation is needed since obfuscated code is rarely decidable. In this section, we mainly consider encoding schemes that are, in most cases, bound to deobfuscate. By considering deobfuscation as a terminating process, we can safely assume that it will yield a deobfuscated program, which is a simplified form of the obfuscated program. We will subsequently describe our reasoning approach on encoding schemes and how to implement deobfuscation using the Maude language as a way to emulate packed programs.

9.1 Deobfuscation

Evaluation of obfuscated contents is difficult to achieve using only static examination of code. For this reason, current web malware detectors employ execution-based approaches. While they rightfully consider obfuscated contents as undecidable, they assume that packed contents are bound to deobfuscate. Proposals such as Zozzle [36] elicit an instrumentation of critical sinks, similar to what is done in the JavaScript Deobfuscator [110], following the methods developed by Nazario [104]. Current proposals implement instrumented browser emulation environments where the `eval` function is hooked to reveal the code being executed.

While this approach works globally well, we may disagree on some points:

- execution may fail to capture malicious code on a single pass when predicates force the execution to take an irrelevant path. In fact, dynamic approaches are often criticized for their lack of code coverage;

- browser emulators are based on JavaScript engines and are therefore prone to vulnerabilities inherent to the engine. Depending on the degree of containment, such execution may not be safe;

- instrumented environments hook the `eval` function but some obfuscation techniques do not rely on such function to conceal malicious intents. Such transformations have the potential to evade analysis environments that employ statistical or learning methods to classify scripting malware.

From its inception, our research has been inspired to reverse the malicious intents of obfuscated scripts with an approach different from what has been achieved so far, especially on two specific points: (1) apply static methods as much as possible; (2) consider obfuscation as not decidable. As presented in Sections 2 and 3, current web malware detectors only apply one or the other, but never both. However, deobfuscation is difficult to achieve with static methods only. However, Collberg et al. suggest that it may be partly possible in [29]: they speculate that *partial evaluation* may be applied to simplify obfuscated programs. In a similar fashion, we propose to interpret the code of the decoding routine of a JS packer. While emulation of the JS language can be achieved through dedicated APIs of another language, we decided to explore a much formal approach.

By considering a program as a set of equations, we attempt to deduce its outcome through simplification. This is a simple and straightforward approach that relies on deduction.

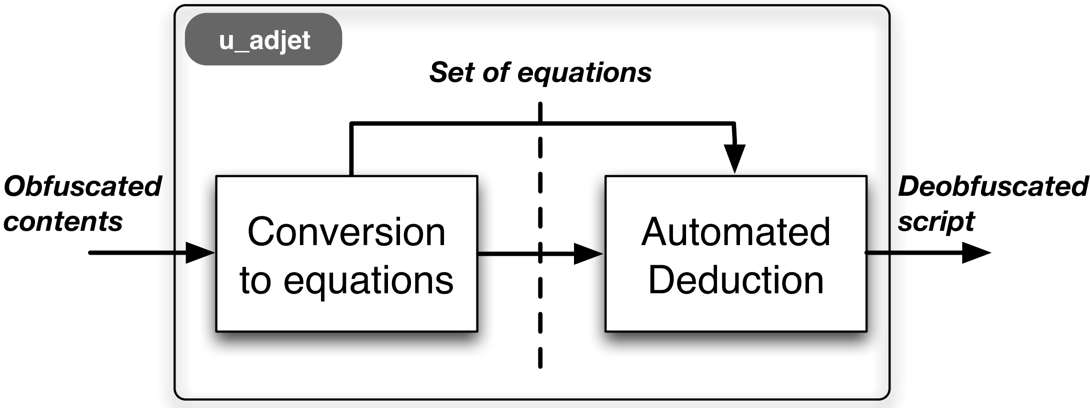

Figure 20. Workflow of the u_adjet deobfuscator

9.2 Overview

u_adjet is a deobfuscator that relies on the Maude rewriting framework to emulate the instructions of an obfuscated JS program. Provided we have extracted instructions relevant to obfuscation, u_adjet performs the conversion of instructions to equations parsable by Maude. Maude will then provide *rewrite* these equations to produce a simplified version of the JS program (see Fig. 20).

Similar to current approaches, we first concentrate on encoding/packing schemes, which presently represent the majority of obfuscated contents. However, automated deduction is obviously not restricted to a subset of obfuscation techniques. Encoding schemes are merely considered as a proof-of-concept here to introduce a novel approach to the automation of deobfuscation.

In particular, encoding schemes can be seen as a special case where obfuscation and deobfuscation coexist. In fact, the deobfuscation part modifies (here, rewrites) the obfuscated part. Our intuition is that by mechanizing the deobfuscation part, we can automate this process and simply introduce the obfuscated part as input to the system, in order to produce deobfuscated code.

Algorithm 1 gives a more detailed description of the processing that takes place just after prefetching. Once the employed obfuscation scheme has been detected (by ob_asti), the obfuscated contents are extracted and the deciphering routine is converted into a Maude functional module. Upon deobfuscation, an additional step verifies

Algorithm 1 Automated deduction of script instructions

```
1: obfstring, decroutine = extract(script)
2: decroutine = functionalize(decroutine)
3: fmod.decls,fmod.stmts = convert(decroutine)
4: output = Maude.reduce(fmod,obfstring)
5: if output contains links then
6:    output += prefetch(links)
7: end if
8: if output is obfuscated then
9:    script = output
10:   repeat from line 1
11: end if
```

whether deobfuscation is still needed.

An advantage of Maude is that it employs term-indexing techniques to achieve high speeds of rewriting[123]. However, most JavaScript encoding schemes employ an imperative style of programming, in particular using loops. It requires additional processing to transform a JS program to a functional one. The conversion of JS code to Maude functional modules is further detailed in the next section.

9.3 Automated Deduction using Maude

We were suggested to use Maude[25], a rewriting framework based on *membership equational logic*. In [98], Meseguer observed that equational and rewriting logics were suited to specify the four different program styles that result from the cartesian product $\{imperative, declarative\} \times \{sequential, concurrent\}$ where:

- *imperative* programs, described as "involving *commands* changing the state of the machine to perform a task", are opposed to *declarative* programs, which "give a mathematical axiomatization of a problem (as opposed to low-level instructions on how to solve it)";
- *sequential* programs, which run "sequentially" and "for each input yield an answer or loop", are opposed to *concurrent* programs, which run "in parallel" and may "yield many different answers, or no answer at all, in the sense of being *reactive systems* constantly reacting with their environment".

In particular, Meseguer notes that equational logic is very well suited to give executable axiomatizations of imperative sequential languages and refer the reader to [58] for further details on how to reason on such programs using equational logic.

Actually, he demonstrates the use of functional modules in Maude[25] for the axiomatization of declarative sequential programs. As a matter of fact, the declarative aspect of Maude is appropriate and such declarations should therefore be reflected in imperative programs if we wish to use functional modules for their axiomatization. JavaScript is actually a special case itself since it is a multi-paradigm language: in particular, it supports functional programming style, which is a kind of declarative programming style. As we have seen in Section 8, functional modules satisfy the *membership equational logic*, therefore equations must be confluent, terminating and sort-decreasing.

9.3.1 Obfuscation as a Subset of JavaScript

Our goal is to express decoding routines as rewrite rules to simplify the obfuscated string, which represents the packed program. We expect the rewriting framework, here Maude, to perform the simplification process, hopefully yielding a canonical form of the packed program, which would be deobfuscated (see Fig. 21). Although JavaScript is imperative, and mostly sequential as a programming language (as it is not a concurrent one per se), we can reason on it in terms of equations since it also supports functional programming. However, JavaScript is dynamically typed, which may be an obstacle for producing the declarative part of a functional module. On the contrary, generating the statements seems to be straightforward (see Alg. 1).

As stated previously, we are not attempting to specify JavaScript as a whole but only obfuscated programs. Moreover, we presently restrict the scope of obfuscated programs to those that are encoded using an encoder/packer style of obfuscation. This particular obfuscation technique is the most popular and displays a common pattern including a decoding routine, and an obfuscated string on which the decoding routine is applied. In this case, the `reduce` command is applied to the obfuscated string, which will be reduced using the equations of the decoding routine. This is analogous to the emulation of the said decoding routine.

9.3.2 Axiomatizing JavaScript Packers in Maude

To axiomatize JavaScript code, and especially the subset of functions used in packers, we leverage the functional properties of JavaScript. Indeed, functional programs are well-suited to reasoning and have some interesting properties. In particular, imperative

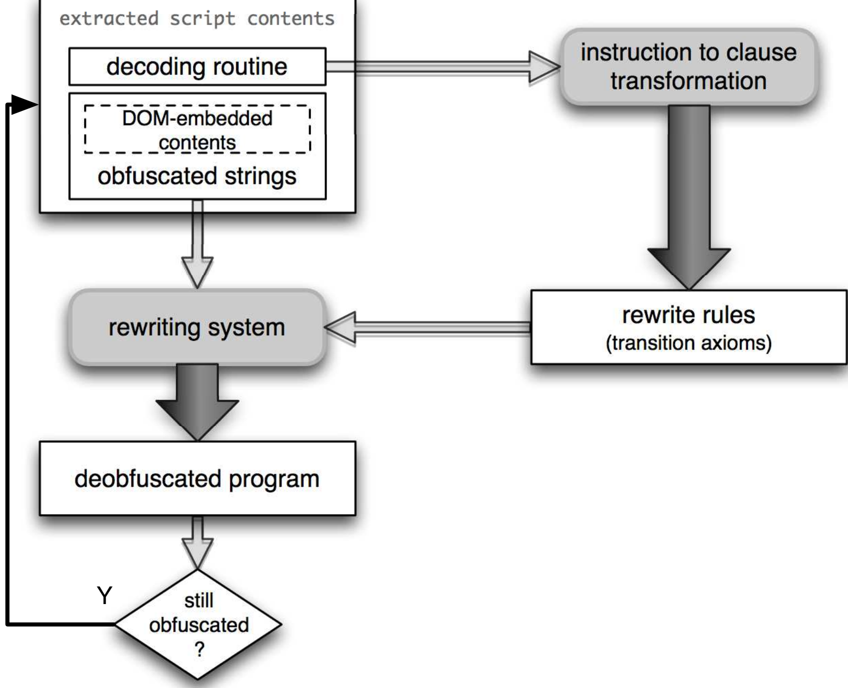

Figure 21. Decomposition of a packed program and conversion of the decoding routine

programs have side-effects during execution since they change the program state [71], as we noted earlier. As a matter of fact, imperative programs lack *referential transparency* since the behavior of a program is not only function of the input (as it is with functional programs) but also function of the state of the executing program. Other characteristics of functional programs are [11]:

- closures and higher-order functions: closures designate the inner functions (functions defined within a function) with a *free variable* binded to the outer function; higher-order functions can take other functions as arguments or even return functions as output argument.

  These two constructs actually allow to modularize programs in an elegant way;

- recursion: in functional programming, recursion is used as a control flow mechanism.

In order to leverage Maude's equational logic to deduce the outcome of a JS decoding routine, it is therefore necessary to convert the JS decoding routine to a functional equivalent.

Here, we describe a summary algorithm to deal with simple JavaScript programs (the `functionalize` function in Algorithm 1): The algorithm is still a tentative to

Algorithm 2 Conversion from imperative to functional JavaScript

1: $vars = []$
2: $p' = recursionalize(p)$
3: **for** each instruction i in p' **do**
4:   **if** i is a function call **then**
5:     $i' = closure(i)$
6:     $v = i.vars$
7:     **if** v in $vars$ **then**
8:       i' is closed in the last function using v
9:     **else**
10:       $vars.push(v)$
11:     **end if**
12:   **end if**
13: **end for**
14: VARS = $vars$
15: EQNS = $p'.i'$

provide a conversion for all JS programs, but can be restricted only to the conversion of loops to recursive functions for decoding functions of common JS packers.

Functional modules are thereby used to emulate deobfuscation: variables and objects are mapped to sorts in Maude, instructions are emulated through equations. Computation is realized by using these equations as rewrite rules applied to the obfuscated strings, until a canonical form is found, that is, the deobfuscated script. Rewrite rules define transitions from one state to another, prompting the *deduction* of the resulting state from the initial state by *reduction*. Such logical framework allows the generation of concrete semantics from concrete input, but based on formal operations.

More concretely, we describe how to evaluate such programs in Maude in the remainder of this section.

As for sorts, it is possible to include all sorts, regardless of the ones actually used in the program. All variables declared, as well as undeclared, should be declared using the

keyword `var`, followed by an appropriate identifier in uppercase and an appropriate sort, that is the variable type. Such type may be opaque. The closed functions we generated using Algorithm 2 should be declared as operators and control flow is performed by conditional equations. All variables occurring in a function are reflected in the sort signature of the operator. As with variables, some types may not be known before execution.

Types of JavaScript have some equivalent in Maude sorts, along with the predefined operators:

- Boolean variables are expressed using the `Bool` sort and the functional modules `TRUTH-VALUE` that instantiates the values `true` and `false`, and `TRUTH` that declares operators for equalities and inequalities as well as a complete branching control construct `if_then_else_fi` where the underscores are placeholders for terms;

- Numeric variables can be expressed through a number of sorts but we privilege the `INT` functional module that declares the sorts `Int` and `NzInt` (for non-zero integers). `Int` extends the `Nat` sort representing natural numbers with a unary minus operator. It provides all sorts of operators to compute arithmetical operations on integers, as well as bitwise operations and tests;

- Floating-point numbers can be represented using the sort `Float` and the module `FLOAT`, which provides all the functions used in JavaScript such as `ceil` or `floor`. Additionally, `Float` provides trigonometric operators usually defined by the object `Math` in JavaScript;

- String literals also benefit from a predefined `String` sort declared in the `STRING` functional module. It supports the substring operation (`substr`), the string length operation (`length`), string concatenations (`_+_`), conversions between ASCII code and characters (`ascii` and `char`) but no advanced regular expression manipulation such as provided by the JavaScript function `replace`.

Although, it is possible to extend Maude functional modules in order to support a better emulation of JavaScript capabilities, we may be still constrained in some ways, especially when dealing with regular expressions. Additionally, the declarative style of Maude needs some further adaptations of the JavaScript code being emulated. These adaptations as well as the processing of regular expressions are treated in a preprocessing and postprocessing stages detailed later.

```
output := default
for (counter := init; counter > stop; update(counter))
    output := f(counter)
```

Figure 22. Pseudo-code of a loop performing the function f()

```
function f(counter):
    if counter = stop then
        return default
    else
        return f(update(counter))
```

Figure 23. Pseudo-code of a recursive function f()

9.3.3 Loop Conversion

Decoding routines usually make use of loops to repeat their processing on the string they decode. Conversion to functional programming requires that loops be transformed to recursive functions.

Since u_adjet "translates" JavaScript instructions to Maude equations, preprocessing tasks include the transformation of any loop (as depicted in Figure 22) to a recursive function (as depicted in Figure 23).

The transformation from a loop to a recursive function is a trivial algorithm that takes as input a function `f()` (applied to an initial value `default`) iterated within a loop that runs from an initial value `init` to a final value `stop`, the pace being defined by the function `update`. The resulting recursive function `f()` has a default condition decided by the value of `stop` and recurs on values paced by the function `update()`. This recursive function is first called on an initial value `init`.

Maude functional modules allow to express the recursion by using conditional equations with one case being the stop condition and the other the actual processing.

9.3.4 Postprocessing

The translation is challenging to automate for the reason that translating the decoding routine to a functional module operator requires the knowledge of the number and type of all variables that are used in the decoding routine. Not only should be provided the type of the input variables as well as the output, but also the type of each variable

participating to the process of the decoding routine. Similarly, these variables are also translated within the functional module and should be named and typed accordingly. It is actually the dynamic typing characteristic of JavaScript that constitutes a challenge here where the type of a variable may not be known in advance. However, this case may often happen when the variable is defined from a value originating from outside the script, which may be resolved prior to deobfuscation thanks to prefetching or resolving DOM-embedded contents. As stipulated above, some preprocessing are needed prior to emulating the deobfuscation of obfuscated contents. In particular, Algorithm 1 mentions a function `extract` responsible for extracting the obfuscated strings and the decoding routine from the script contents. This function corresponds to the processing of ob_asti which goal is to detect obfuscation patterns and extract the related instructions from the script code. As we stated in Section 7, not only is it convenient to detect recurring patterns of obfuscation, but it is also possible to design a deobfuscation method that may be faster than executing the decoding routine, by ignoring irrelevant code for example. Therefore, against the actual obstacles of converting JavaScript code into functional module, we argue that given a specific obfuscation pattern, we can design a specific Maude functional module to handle this pattern, with the appropriate declarations of variables and sorts.

On the other hand, Algorithm 1 also indicates that links may be generated after processing, prompting the `prefetch` function. This function is responsible for commanding the (sak_mis) proxy to request the pages from the generated links and inline the script contents of the response back to the script being processed.

9.4 Example

Here, we present an example of the application of u_adjet to a simple malicious JavaScript sample to demonstrate its feasibility and its accuracy.

This example (see Fig. 24) is a simple *eval unfolding* featuring a single loop that deciphers an obfuscated string via XOR operations. The script is included into an HTML file that displays a 404 error page to an unsuspecting user.

The proposed system first extracts the script contents, that is, the instructions comprised between the `<script>` tags, and parses the contents. The parse tree is analyzed to detect the obfuscation scheme. Here, the obfuscation scheme uses a loop to process the obfuscated string. This loop is converted to a recursive function whose body is the aforementioned string processing script. The Maude system readily provides a predefined functional module that defines the string data type as well as operators

```
<html>
<title>404 Not Found</title>
</head><body>
<h1>Not Found</h1>
<p>The requested URL /index.php was
not found on this server.</p>
<p>Additionally, a 404 Not Found
error was encountered while trying to use
an ErrorDocument to handle the request.</p>
<hr>
</body></html><script language=JavaScript>
str = "qndy'mh)(:" // the obfuscated string is
// abbreviated for the purpose of brevity
str2=""; for (i = 0; i < str.length; i ++){
str2=str2+String.fromCharCode(str.charCodeAt(i)^1); };
eval(str2);</script></html>
```

Figure 24. Original HTML code

```
fmod TEST is
  protecting INT .
  protecting STRING .
  op test : Int String String -> String .
  var I : Int .
  vars S1 S2 : String .
  ceq test(I,S1,S2) = S2 if length(S1) <= I .
  ceq test(I,S1,S2) = test((I + 1),S1,S2)
  + char(ascii(substr(S1,(length(S1) - I - 1),1)) xor 1)
    if I < length(S1) .
endfm
```

Figure 25. Maude functional module

to manipulate string objects: the `fromCharCode()` function is mapped to Maude's `char` operator, which converts an ASCII code to the corresponding character; the `charCodeAt()` function is emulated by the combination of two basic operators, `ascii`, the inverse of `char`, and `substr`, the substring operator. The result of the conversion to a Maude functional module is displayed in Figure 25. The workflow of the recursion is realized through conditional equations. The obfuscated string `str` as well as the empty string `str2` are inputs to the Maude system and are going to be rewritten by the functional module we generate (Fig. 25).

9.5 Discussion

9.5.1 Performance

In our approach, we wish to maximize code coverage by adopting a rather static approach for script analysis. This needs prior deobfuscation of analyzed samples. This preprocessing stage surely incurs some delay but emulating the deobfuscation stage using the Maude framework may alleviate the time overhead. Indeed, our approach features some time-saving points to minimize the delay to a certain extent:

- the prefetching stage anticipates the fact that several snippets of code will be gradually downloaded to the client-side;
- we expect the conversion stage between script contents and Maude functional modules to be fast as we are mapping script objects to Maude predefined data types and operators (although the loop transformation implies an additional delay as we have seen);
- the Maude rewriting system performs relatively well compared to time spent loading rich Web applications: it demonstrated processing time not exceeding 100 milliseconds for the few samples we tested on an Intel Core 2 Duo platform (2.5GHz) with 4GB of memory running the Maude 2.5 engine.

The time overhead does not take into account several intermediate processing such as conversion time between JS instructions and Maude functional modules, and only rewrites a single obfuscation. We are aware that this result needs to be confirmed through extensive implementation and testing.

9.5.2 Termination

Another issue is the termination of the deobfuscation process. While Maude functional modules should satisfy the requirements of being Church-Rosser and preferably terminating, JavaScript programs can not guarantee termination. We assume that deobfuscation in the case of JS programs is necessarily terminating in order to provide executable JS code. However, we can imagine an attacker trying to carry out a denial of service on u_adjet by crafting an infinite loop in the deobfuscation stage. If this is the case, we can think of two ways to prevent a denial of service. Either, we check for a trivial infinite loop during the loop to recursion transformation, or we set a recursion limit to prevent an infinite loop.

9.5.3 Other Obfuscation Techniques

As we have seen in Section 6, there are numerous obfuscation techniques besides custom encoding/packing. Most of the simple techniques such as string concatenation or variable aliasing may be simplified using Maude. One big challenge in obfuscation and cloaking techniques is the opaque predicate issue. When an opaque predicate is represented by a difficult theorem, it may be possible to prove it using Maude. In general, we speculate that Maude can resolve the problem of opaque predicates, which would help simplify obfuscated programs by removing irrelevant code.

For other obfuscation techniques, the actual challenges is to isolate functions, generate closures for these and links these through their bounded variables, building higher-order functions. The tentative algorithm we showed (Alg. 2 is a starting point to formalize the conversion of JS code to functional equivalent in order to apply equational reasoning on obfuscated JS.

By specifying rules, we may design deobfuscation techniques for different classes of obfuscation. We think that more research can be done in applying rewriting logic using Maude system modules.

9.6 Summary

(sak_mis) relies on a deobfuscation module to provide unobfuscated code for analysis. With the constraint of avoiding direct JavaScript execution, we took on automating the emulation of JavaScript instructions involved in the deobfuscation stage of web attacks.

We proposed to perform automated deduction using a rewriting logic framework. Maude rewriting framework, based on membership equational logic, offers sound and complete rules of deduction [97]. The richness of this framework provides an executable environment for most of object-oriented programming languages such as web scripting languages. Additionally, the Maude system has also been hailed for its good time performance [123], which our proof-of-concept also demonstrated. However, the declarative style of Maude hinders the one-to-one mapping between script languages such as JavaScript and Maude language.

There are still challenges left to completely automate deobfuscation. In particular, quantifying the time overhead of a multi-staged deobfuscation. With its integration within (sak_mis), the u_adjet deobfuscator may suffer from additional time delays. It will be necessary to spot potential bottlenecks in the system where improvements can be achieved.

Hopefully, the output of the module should be the canonical form of the deobfus-

cation stage, that is, a code equivalent to the original code that was obfuscated. That way, analysis can be safely performed to reverse the intents present in the original attack code. The theoretical background that permits to associate intents, or human concepts, to the syntactic pieces of deobfuscated code will be described in next section.

10. On Program Comprehension

> Every program has (at least) two purposes: the one for which it was written and another for which it wasn't.
>
> Alan J. Perlis

Program Comprehension is a domain of computer science that focuses on source code maintenance methods and theories. It is necessary to facilitate maintenance, reengineering, code reuse, documentation, reverse engineering, extension of existing software systems, etc. [75].

This discipline aims to explain a program, its structure, its behavior, its effects on operational context and relationships to its application domain [12].

10.1 The Concept Assignment Problem

Based on the source code, it is always possible to extract tokens from parsing, but this is somehow limited in its expression, in particular qualitatively. In fact, a human being will struggle to understand a program if the terms used are not human-oriented. To achieve this level of expression involves a great deal of knowledge of the application domain.

Parsing source code yields formal, mostly structural and syntactic features, which are adequate for machine-based automated treatment. This allows to represent computational intents at a low-level of abstraction, close to the source code. Biggerstaff et al. [12] outlines the recognition of *programming-oriented concepts* through parsing.

On the other hand, *human-oriented concepts* are expressed more informally, in terms that may be ambiguous. Expressing computational intent is therefore a complex process involving analysis, experimentation, inference and semantic mapping between domain concepts and operations expressed on literals and data structures.

Lastly, binding the two worlds of concepts, that is, discovering human-oriented concepts and assigning them to their realizations within a specific program is the *concept assignment problem* [12].

Biggerstaff et al. further elaborated that there were no algorithm or set of inference rules for recognizing human-oriented concepts. Another issue is that it appears there is a paradigm shift between the two worlds of concepts in the kind of features used for recognition, as well as, the nature of the processing required:

- programming-oriented concepts are expressed in terms of formal feature or by ways of deduction on those features;
- recognizing human-oriented concepts is more like a “decryption” problem since *a priori knowledge* is needed to infer ambiguous tokens (natural language tokens).

Biggerstaff et al. have proposed a suite of tools, DESIRE (DESign Information Recovery Environment), to support an intelligent agent (a human being) in strategies to assign concepts to portions of codes. They identify two main tasks:

- identify entities and relationships based on formal information, as well as some informal information such as grouping and association clues;
- assign these entities to known (or newly discovered) domain concepts leveraging on domain knowledge.

DESIRE has been used in several scenarios that bootstrap program comprehension either based on suggestive data names, patterns of relationships (regarding the abstract architecture or framework of the program) or the user’s experience.

Biggerstaff et al. have concluded that concept assignment is a difficult problem and that neither recognition, nor assignment can be completely automated. Partial automation is possible but this problem necessarily relies on an a priori knowledge base that comprises a great deal of information on the application domain and typical program architectures.

10.2 Concept Recognition and Program Slicing

Reverse engineering is the process of extracting knowledge or design blueprints from anything man-made [46]. Reversing techniques are often used by security vendors to trace every step a malicious program takes and assess the damage it could cause. With compiler-generated code, it is often difficult to determine the developer’s original intentions. Although it may be easier to recover the true intents of the source code, it is generally expressed at a low-level of abstraction, usually syntactic as we have seen previously.

Concept recognition, as defined by Biggerstaff et al. [12], uses a finite set of pattern templates to recognize concept signatures. Simplest and most elemental patterns are recognized first before being integrated into larger-grained, composite concepts. Parsing, as a process of tokenizing source code instructions, is considered a degenerate case of concept recognition.

Discovering concepts is not trivial and as the above description suggests, it needs to be bootstrapped in some way. Experiences with software debugging led Mark Weiser to observe that a large computer program is better understood when broken into smaller pieces [144]. Program decomposition was a sanctioned practice for program design, back in the 1970s. Weiser proposed another complementary method, *program slicing*, that would be applied *a posteriori* to the written code, instead of during design stage. Originally a program slice is an executable subset of a program comprised of instructions relevant to a given criterion. The slicing criterion is generally a pair ($< i, V >$) where i is the number of the statement at which to observe and V is the set of variables to observe. The program slice would generally include any statement that has an effect on the value of the observed variables at statement i. Slices should satisfy two properties:

- the slice should have been obtained from the original program by statement deletion;
- the slice should preserve the behavior of the original program, as observed through the window of the slicing criterion.

Before and after Weiser generalized program slicing, many different algorithms have been proposed to approximate slices based on dataflow analysis, which allows to find pieces of code that influence a particular behavior [144]. Although finding the minimal slice is deemed a difficult problem [38], a comprehensive survey on program slicing methods [148] indicates that non-executable slices are often smaller, and thus more helpful to program comprehension. The same survey goes further on by observing that program comprehension, among other applications, only needs non-executable slices. One particular method [109], based on Program Dependence Graph (PDG) [49], has been proposed by Ottenstein and Ottenstein to compute a program slice, for a given variable, as simply the set of statements that influence the value of this variable. This method finds a slice by walking back the PDG and performs in linear time.

Common slicing methods usually require a starting statement to perform slicing. On the contrary, decomposition slices [55] are a set of program slices that capture all relevant computations involving a given variable, leading to a "direct" decomposition of a program into two (or more) components. A decomposition slice for a variable v is therefore the union of the program slice for v at all points that output v and at the last statement of the program.

Not every variable in a program is outputted at a statement, making the decomposition slice as proposed by Gallagher and Lyle [55] inaccurate for such cases. Based on

the observation, Icuma et al. [73] proposed to define a decomposition slice for a variable v as the union of program slices for v at the statements that define a value to v, that is, assertions to v and inputs on v. Interestingly, this new decomposition slicing method can make use of PDG, constructing a slice in linear time.

10.3 Program Categorization

There are indeed a few works on text categorization that have explored source code classification issues such as [139]. But to the best of our knowledge, there is only one work [90] on categorizing web scripts, in particular JavaScript programs. The authors actually indicate the lack of contributions in this field as a motivation to their work. Along their survey, they made several observations:

- programming languages are generally designed to be open-ended and largely task-agnostic;
- unlike natural language texts, program source code is unambiguous to the compiler and has exact syntactic structures;
- syntactic information and some language-specific semantic information, could be important and useful for classification.

Similar to [139], Lu and Kan examined potential categories for JavaScript program: they decided to adopt the 54 categories used by a popular JavaScript programming tutorial website. However, they regret that the categories are developer-oriented and not consumer-centric. They also deplore the shortness of provided programs used for categorization, since these are sample programs.

Therefore, they proposed to perform categorization on JavaScript *functional units*, rather than on the entire page's scripts. A functional unit, or simply unit, is defined as a JavaScript instance, combined with all of (potentially) called subprocedures. Based on these automatically extracted units, Lu and Kan proposed 33 discrete categories based on functionality.

They perform categorization using several different approaches on functional unit tokens:

- syntactic analysis based language tokens that distinguish expression operators, URLs, HTML tags, etc. These are counted and use as syntactic features;
- code metrics comprising classic complexity metrics as well as structural similarity count and builtin function count;

- code reusing edit distance;
- DOM-related program comprehension features generated by both static and dynamic analysis.

They tested their methods on 1.7 million web pages from over 11000 distinct servers. Their study confirm that a large part of scripts from their corpus were copies or simple modifications. While text categorization baseline performs well, program analysis features greatly improved the performance.

10.4 Summary

Since static analysis is best performed on source code, we strived so far to provide readable code. In such state, reversing intentions should be a straightforward task. State-of-the-art code analysis methods have a potential to extract useful information from the source code that can help decide on the malice of a program. In particular, we are interested in abstracting a program to concepts understandable to humans, not only to the benefit of human analysts but to map the multiplicity of implementations to a same intention.

The theory of concept assignment takes interest in detecting independent pieces of code that perform a distinct function, and associating these to a human concept. Concept assignment benefits from advances in program analysis and has integrated some existing tools to achieve its goal. In particular, program slicing and program dependence graph are two such techniques that provide a modular view of programs on which we can base our decomposition. There were promising results in the decomposition of JavaScript code and the categorization of code excerpts in Lu and Kan's proposal [90]. However, these results are not applicable in our case since these work concentrated on consumer-centric categories, and such perspective differs from ours. Additionally, their features only accommodate a subset of JavaScript functionalities, namely, interactions with the Document Object Model.

Our purpose necessitates an alternative decomposition of programs that are closer to the developer's design. More details on how concept assignment is implemented in mi_oos are given in the following section.

11. Knowledge-based Static Intentional Analysis of Unobfuscated JavaScript Code

> Those who stand for nothing fall for anything.
>
> Alexander Hamilton

In order to capture the intentions of an unobfuscated program without relying on the semantics of the variable naming/aliasing, which suffers from string randomization, we propose to better rely on the functionalities demonstrated by the code of the program through the clustering of pieces of that code around programming objects.

In this section, we cover the two main steps to express the computational intents of a program in order to decide on the malice of such program: program slicing and object categorization.

11.1 Motivation

Modeling intentions of a program is not quite like capturing its behavior since we are not interested in the actual execution outputs of the analyzed program but rather into the operations that are actually present in its source code. This approach allows an analyzer to prevent execution side-effects, particularly, parameterized execution where only a part of the code is executed. This results in a partial expression of the code that can be harmful. Here, intentions are modelizations of what the code intends to do, or rather what it is intended for. This is a straightforward view of the code that expresses the code's true nature. It is based on the object-oriented paradigm, common to many languages used in web programming, and in particular, client-side web scripting languages such as JavaScript, VBScript or ActionScript. As the subject of our case study, and because of its importance in the AJAX framework as well as its frequent use in Web-based malware, we chose JavaScript to illustrate our point.

As a reminder, JavaScript is a prototype-based scripting language that bears the property of being a duck-typed, functional language. As a matter of fact, both functional and object-oriented decomposition are eligible for JavaScript (as well as for JScript, Microsoft's variant and ActionScript, both being based on ECMAScript) but not for VBScript. We stated earlier in Section 10 that this research aims to model analyzed scripts as objects interacting, in order to infer what are the script's main functionalities.

11.2 Related Works on Web Script Analysis

Protecting the client-side, in particular the browser, from getting exploited by client-side web malware has risen as the best solution to ensure a safe browsing experience to the user. Indeed, since it is not possible to protect every web server in the world, not to mention attacker-owned domains, there is always a possibility that a user ends up visiting a web page containing malicious scripts. As outlined in Section 2, there are actually a few ways to prevent exploitation, the most straightforward being simply to disable JavaScript. But these approaches either lack in usability or in performance. In this section, we will cover many efforts towards designing browser-based solutions or extensions, as well as, preliminary researches similar to ours. One significant contribution will be featured for each category.

11.2.1 Machine Learning Based

Cujo [118] is one of the last example of machine-based learning web malware detectors. Cujo specializes in detecting drive-by download attacks by learning lexical features (static analysis) and execution traces (dynamic analysis). Authors of Cujo observe that current countermeasures suffer from either of two shortcomings: some approaches are too specific (for example, detecting only heap-spray) while more general approaches induce a performance overhead too prohibitive to be usable.

Cujo is a proxy-based solution that extracts generic features, in this case sequence of q words (noted q-grams), from both static and dynamic analyses, generating unified reports. Cujo uses Support Vector Machines (SVM) to build a hyperplane that separates two vector classes representing contiguous reports (similarity is calculated on q-gram mapping) of benign web scripts on one hand and malicious ones on the other hand. Cujo offers additional explanation to the detection patterns by reporting on how much a feature participate into detection. The combination of both analysis types also offer additional resilience to Cujo since a malware should attempt to circumvent both analyses to evade Cujo.

Cujo outperforms current anti-virus products and enables detecting 94% of the drive-by downloads with few false alarms and a median run-time of 500 ms per web page, which is hardly perceived by the user.

These figures actually represent a sizable indicator to assess the performance of our own detector.

11.2.2 Emulation Based

JSAND [32] is an emulator-based JS analyzer embedded in WEPAWET [31] that focuses on vulnerabilities exploited in browser plugins and ActiveX controls. JSAND authors have observed that web attacks often perform four typical stages: redirection and cloaking, deobfuscation, environment preparation and exploitation. Therefore, JSAND builds models for anomaly detection based on ten features characterizing events occurring during these four stages. Among these ten features, those characterizing later stages are deemed *necessary* since an attack does not occur without environment preparation and exploitation, while the former stages provide *useful* features that are not required to perform an attack but allow an attacker to hide malicious code from detectors.

JSAND is implemented using HtmlUnit [15], which allows to emulate a browser environment and assume several browser personalities, as well as arbitrary system environments and configurations. Additionally, JSAND enhances HtmlUnit with anti-cloaking techniques such as a forced-execution model that detects which functions of the script were not invoked at runtime, and subsequently call these. JSAND also provides an exploit analysis functionality that is able to classify exploits using a naive Bayesian classifier. JSAND outperforms antiviruses and honeypots but may be evaded using fingerprinting techniques, as described in Section 3.

11.2.3 Abstract Syntax Based

Zozzle [36] is an in-browser mostly static JS malware detector. Zozzle attempts to improve on past proposals on four points:

1. accelerate detection for the sake of usability (in a browser);
2. overcome the issue of obfuscation for static analyzers;
3. lower the false positive rate to prevent harming the user experience on benign websites;
4. offer runtime analysis to overcome transience issues that plague URL-based detectors.

Zozzle satisfies its first requirement by its localization as part of the browser's JavaScript engine, it is run at parsing time, greatly improving its time analysis and therefore imposing little time overhead to the user. Its hooking into the JavaScript

engine also permits Zozzle to address the issue of obfuscation as it has direct access to the final, expanded version of the JavaScript code.

Zozzle has an extremely low false positive rate of 0.0003% thanks to its highly precise detector trained on deobfuscated JavaScript code. In fact, Zozzle cannot deal with obfuscated JavaScript, but the training phase is done on JavaScript code that has been previously deobfuscated by an "augmented" browser that extracts and collects fragments of JavaScript. The Zozzle deobfuscator collects information about which context leads to other code contexts. Once contexts have been labeled (benign or malicious), features indicative of benign or malicious intents are extracted by leveraging the hierarchical structure of the JavaScript AST. Features consist of two parts: a context (loop, condition, etc.) and a text (a substring of the AST node). To improve performance, feature extraction is restricted to specific AST nodes: expressions and variable declarations. It is further improved by keeping only the most contributing features. The machine learning method employed in Zozzle is a Bayesian classifier that distinguishes benign and malicious JavaScript programs.

11.2.4 Symbolic Execution Based

Rozzle [87] proposes to address the issue of cloaking, as outlined by [117], that is, fingerprinting techniques used to limit exploit triggering to targeted environments only, and thus preventing possible detection. Cloaking also allows to evade automated crawlers.

As Rajab et al. [117] pointed out, cloaking techniques hinder the processing of every type of web malware detector. In particular, for emulators that have the potential to impersonate every type of browser personality, every type of system environment, it becomes a tedious task for an administrator to maintain all browser/system instances. As for Rozzle, it claims to successfully replace multiple VM instances in a browser with no runtime overhead.

Rozzle aims to fulfill 3 major objectives concerning multiple execution of malicious JavaScript:

- lower the overhead of detection-driven execution of multiple malware paths;
- improve the effectiveness of detectors;
- lower the CPU and memory overhead.

Rozzle is an enhancement or amplification technology, designed to improve the efficacy of both static and runtime malware detection. Rozzle augments the semantics

of a regular JS interpreter by introducing additional statements in the execution that corresponds to symbolically executed paths. A later stage sees Rozzle concretizing output statements for symbolic values according to a concretization policy. This allows Rozzle to reflect multiple program outcomes in a single modified execution. Main challenges of Rozzle concern the symbolic values, which are maintained on the heap: looping on a symbolic value, DOM interaction, limiting the heap size, etc.

Rozzle has been successfully implemented in Chakra, the Internet Explorer 9 JavaScript execution engine and exhibits detection performance far better than past static or dynamic detectors, since its multiple execution allows to eventually match the environment targeted by an attack.

11.3 Inferring Intentions to Detect Web Malware

In this dissertation, we take a radically different approach to the above-mentioned related work. Consistent to the requirements enunciated in Section 4, we propose a novel way to represent malicious intents concealed in Web malware. In particular, we are interested in formalizing such intents rather than interpreting an observed behavior through the scope of statistics or heuristics. As a matter of fact, most of the approaches described above rely on execution traces that allow to capture some behavioral features of the program. While execution-based approaches impose some delay to the analysis, we are more concerned by the incomplete view it may give. By analogy, execution-based analysis behaves like a blackbox, and behaviors observed at the output are functions of the input. While this is totally acceptable in cases where we do not have access to the code, we want to point out that, at this stage of analysis (after deobfuscation), the source code of the malicious script is available. Inferring intention is therefore a whitebox analysis on the script source code.

A second characteristic of intention is its abstract representation. Using program comprehension methods introduced in Section 10, we attempt to associate programmatic constructs, at the syntactic level, to human concepts, at the semantic level. The representation we employ here is the widely used UML diagram specification. In particular, we leverage information produced during analysis to model objects and transitions of the program. The resulting UML diagram is not standard and is actually a hybrid of the sequence and the object diagram specifications.

Finally, our approach also abides by the concept of semantic unification that seeks out to determine whether two terms share the same meaning [112]. Concretely, two implementations of a same program are considered to bear the same intention and

```
poexali();
function poexali() {
var ender = document.createElement('object');
ender.setAttribute('id','ender');
ender.setAttribute
('classid','clsid:BD96C556-65A3-11D0-983A-00C04FC29E36');
try {
var asq = ender.CreateObject('msxml2.XMLHTTP','');
var ass = ender.CreateObject("Shell.Application",'');
var asst = ender.CreateObject('adodb.stream','');
try {
asst.type = 1;
asq.open('GET','http://attacksite//attack.php',false);
asq.send();
asst.open();
asst.Write(asq.responseBody);
var imya = './/..//svchosts.exe';
asst.SaveToFile(imya,2);
asst.Close();
} catch(e) {}
try {
ass.shellexecute(imya);
} catch(e) {}
} catch(e) {}
}
```

download
storage
execution

Figure 26. Sample program clustered following slicing.

should then be modeled with the same representation. Concept assignment actually prones a fuzzy modeling where many programmatic concepts can be assigned to a same human concept.

We detail our approach in the remainder of this section.

11.4 Program Slicing

mi_oos accepts JavaScript code that has been previously deobfuscated by u_adjet and applies a static code analysis whose goal is to express the computational intents of the JS program. This is done thanks to two combined approaches: program slicing and object categorization (or concept assignment).

As stated in Section 2, dynamic approaches are usually applied to the analysis of dynamic scripts. Contrary to these previous works, we propose a static approach based on principles of whitebox reverse engineering. Since we have access to the source code, there is no need for a traces analysis. Compilation techniques are seldom consid-

ered since scripting languages are interpreted, though contrary to compiled programs, scripts are readily readable. Based on these observations, we borrowed some ideas from program analysis to design a straightforward program slicing algorithm based on data flow analysis. This is unlike previous slicing proposals based on control-flow analysis on Java programs[122].

The algorithm is detailed in Algorithm 3 and is a block-based forward flow tracing technique. It parses the script from the entry point of a block to its exit point, statement by statement. The notations used in Algorithm 3 are the ones used in [144]: V designates the set of variables of a statement, DEF_n the set of variable defined at a statement, REF_n the ones referred at this statement. The program is gradually decomposed into several object-based slices as it is browsed: for each statement, if an unknown object is found, that is, a variable is being defined, a new object cluster (a slice) is created to store subsequently affected statements by the instance variable. If instead, the variable is referred, there might be a cluster existing for this object, and thus the statement is stored in this one. Obviously, several variables can be involved in a single statement, and the statement is concurrently added to each variable's cluster. This would later denotes an interaction between these objects. At the end, most of the instructions are present in as many slices as there are independent object instances. These slices are sets of script statements that do not fully comply to Weiser's definition[144]. In particular, we do not consider control statements: both branches are just seen as sequential statements instead of parallel ones. Thus this may lead to a behavior different from the original program. Since, we are interested by the intention of the program, regardless on the branch taken, we only group (cluster) instructions related to a same object. Figure 26 gives an image of such clustering around 3 identified objects in our sample program, which is a typical drive-by download attack.

Another step is to find interactions between the object clusters. These interactions are actually flow transitions between objects and can be easily found by intersecting slices of the respective objects. The intersection is the set of statements where the dataflow of the program is passed from one object to another. The interaction is a directed edge and direction is decided on the nature of the operation in the statement. Deciding on the nature of a statement, and overall, of the object cluster is based on semantically classifying the language's core functions, objects, APIs and their methods. In the end, we obtain an abstract view of the script that can be rendered as a UML sequence diagram.

The model we obtain describes the interactions of the different objects constituting the program. For abstract as it can be, it gives a pretty accurate view of the activities

Algorithm 3 Forward minimal program slicing

```
slices = []
for n ∈ statements do
    for v ∈ V_s do
        if v ∈ DEF_s then
            s = slice.new(v)
            s.add(n)
            slices.add(s)
        end if
        if v ∈ REF_s then
            s = slices.getSlice(v)
            s.add(n)
        end if
    end for
end for
for s ∈ slices do
    v = slices.getIndex(s)
    s.label = v.category
    labels = []
    for f ∈ FUNC_s do
        labels.add(s.label, f.category)
    end for
    if labels.length = 1 then
        s.label = labels[0]
    end if
end for
```

carried out by the program. As the readers might question, it is obvious that such model does not directly indicate the malicious or benign nature of a script. Decision should be made by an expert, through supervised learning as discussed in Section 11.6.1.

11.5 Object Categorization

This represents a critical step in our system since it has to do with knowledge. It is not distinct from slicing since it actually characterizes the objects extracted during program slicing. Indeed, at parsing time of each statement, it is possible to retrieve the category of an object or function from a reference. Core and API functions would be looked up in a reference, and information on the kind of function as well as its data flow direction would be returned. From the example in Figure 26, the `SaveToFile()` function is labeled as filesystem access function. If labels, retrieved from methods invoked by the object (here, the slice criterion), are semantically different, then the label of the object itself will be used. From the example in Figure 26, the type of `asq` is `msxml2.XMLHTTP` and is labeled as an HTTP access object. The sequence diagram is then annotated

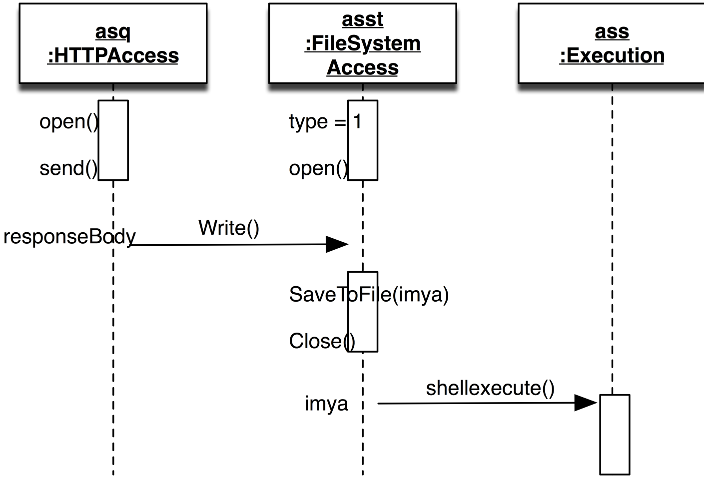

Figure 27. A resulting sequence diagram.

with the category labels for each object, as shown in Figure 27. The `ender` object, which is an object instantiator, is omitted as it is to be incorporated in each of the other objects.

An advantage of categorizing objects and labeling them is to further group two or several objects having the same role. This further reduces the size of the output model and can circumvent issues arising from the creation of duplicated objects in the program.

11.6 Discussion

This research experiences with some design choices that are seldom implemented. These design choices can be reduced to two distinct aspects that are often debated: (1) functional versus object-oriented; (2) dynamic versus static. This is not the purpose of this project to fuel the debate and we would rather elaborate on the merits and demerits of our own approach. Therefore, we chose to offer here, as a discussion, indicators of the feasibility and performance of our approach.

11.6.1 Learning and Comparing Models

In order to provide decision support, it is necessary that the models be comparable. In a further perspective of deciding on the nature, malicious or benign, of an unobfuscated

script, we need to gather knowledge on malicious and benign models of scripts. These two activities are indivisible and the solution rely on the ability to find a proper model representation.

Learning can be done using supervised learning methods where a human expert will classify learned models between malicious and benign. However, if an expert is needed, it is also possible to build an ontology of malicious script activities. These activities would then be illustrated by object-oriented models that will be used for comparison purposes.

Comparison may not be costly as the abstraction of the object-oriented model may reduce the number of entities to be compared. We expect that some flexibility should be introduced when comparing the transitions between two or more objects. Some Markov models can also be used to predict the transitions between objects and the probability of co-occurrence of a type of *objects* with another type of *objects*, in both benign and malicious scripts. One potential drawback of our method is the possibly low number of entities, which might render the model undecidable. Additional features might therefore be needed to motivate a decision.

In this project, we propose to adopt the UML sequence diagram as a way to naturally express the output of our analysis. These sequence diagrams are however very abstract to represent the slices, but are tractable for the representation of several object entities within a large program. Moreover, once objects have been labeled, there is no more need to know the contents of the slice. Attempts to classify and compare UML diagrams has also been proposed. Notably, since UML class diagrams are often used to represent design patterns, there have been attempts to formalize the expression of such models[54] for the automatic generation of the specifications of solutions to popular design patterns. As a more related example to our project, Wendehals and Orso[146] have proposed a method to recognize behavioral patterns using sequence diagrams for reverse engineering purposes. They argued that extracting class diagrams was not sufficient to fully comprehend a program being reversed, and that a more dynamic approach was needed. Behavioral patterns are compared by transforming the sequence diagrams to finite automata. Converting sequence diagrams to finite automata can be beneficial to our approach. Additionally, many diagrams can be represented by a single automaton.

Finally, deciding on the malice of a script's intentions necessitates to classify once more models that are deemed malicious and the ones deemed benign by an expert. Models being generic, we expect their number to be relatively small. Models that are hard to decide may be decidable by complementing the decision process with some heuristics. A study on the decidability of the concept of intention seems interesting

and is left as future work.

11.6.2 Implementation and Performance

With an abstract model featuring a small number of entities, we expect the comparison stage to be quick, provided that we dispose of sufficient knowledge. The major time overhead is obviously incumbent to the processing of our algorithm. Unfortunately, time performance drawn from Ruby script executions cannot indicate how mi_oos may perform. Since mi_oos performs after parsing, it is possible to interpose it, that is, to implement the algorithm at parsing time, which would save a non negligible amount of time. Such implementation have produced interesting results in past proposals such as Zozzle [36]. Therefore, we can expect some excellent performance here comparable to Zozzle (around 500ms).

11.7 Summary

Contrary to most related research work, we advocate the analysis of web malware through a static approach. More, we are applying in this research an alternative model to represent the intentions of a program.

Past research work usually concentrate on building statistical or heuristic models by learning features extracted from great amounts of data collected for extensive period of times. At best, static features are used to complement dynamic features in drawing a more accurate model. On the other hand, to complement detection of web malware, latest proposals attempt to thwart state-of-the-art techniques employed by attackers, such as cloaking. There is actually perspectives to develop novel solutions on the basis of future advances in symbolic execution.

In this section, we have proposed a slicing algorithm that allows to statically decompose a program into minimal object-oriented slices. These slices do not satisfy the definition of Weiser, but they form unitary pieces of code that instantiate a single concept. The model obtained from such decomposition can be further simplified by grouping similar concepts that may have been purposely split by the attacker. Each concept or group of concepts can then be associated to a human concept based on the objects or methods it comprises. Provided, the modeled program is unobfuscated, the intents of a developer are clearly expressed as a sequence or combination of concepts. Yet, it is difficult to decide whether the intents are malicious or not. By learning intents that are malicious and those that are benign, it is possible to classify the intentional model of a program from known ones. Additionally, this can be complemented by

heuristic features of possible malicious or benign behaviour, as well as, the input from alternative detectors such as signature-based (for any shellcodes) or reputation-based (for any URLs) to increase accuracy.

12. Conclusion

> Experience is what you get when you didn't get what you wanted.
>
> Randy Pausch
> The Last Lecture

12.1 Discussion and Summary of Contributions

Intent or intention are abstract concepts that originate from agent theory and philosophy. Though the two words denote a slight nuance, they both express that someone has something in mind, a plan: a goal (intent) with a course of actions (intention). Our definition conveniently blends the two definitions into one as a way to formalize the intent(ions) a developer puts in a program she writes.

Such expression is not unusual in software engineering where the intents of the developer can be conveyed through the realization of specification documents or the representation by models. In particular, we have discussed how the intents of a developer can be represented by using UML diagrams to express some concepts understandable by other developers or non-developers.

On the other hand, we have also discussed how an attacker can conceal malicious intentions in her program by using obfuscation. We have seen that these intentions are blurred syntactically or at a higher level, but that the semantics of the program must remain. Precisely, we propose to formalize intentions to bridge the syntactic and semantic representation of a program, and infer what is the program's original functionality.

In this dissertation, our domain of application is the security of the client-side in web 2.0 transactions. In particular, we have made a large review of security issues of the web browser and related technologies. We have argued that although JavaScript constitutes an important technological cornerstone of modern web applications, especially on the client-side, it is also its main vulnerability. This duality actually affects the design of usable solutions to counter web-based attacks as we have witnessed with the "practical" impossibility of disabling JavaScript. And unless modern web applications shift to another technology to provide client-side processing to web browsers, constraining or monitoring the execution of JavaScript programs will remain an issue for web 2.0 applications and clients.

In fact, we have also ruled out the relevance of deploying security countermeasures

on the server-side since it is not reliable. Indeed, such approach would also be constraining for the end-user since it would restrict her web experience to domains that subscribe to a security policy. We therefore advocated that web security countermeasures should be deployed preferably on the client-side to ensure a safe browsing experience. Incidentally, in parallel to our efforts, we have witnessed a shift in academic research to concentrate on the client-side, in these last 5 years. Still, the challenge of protecting the end-user has remained a goal difficult to attain, given the scarce computing resources in comparison to massive datacenters deployed in this era of cloud computing. As a matter of fact, analyzing JavaScript in the browser is a computing-intensive task, given that the browser should also provide HTTP/HTML processing.

That is why this dissertation also proposes to alleviate the browser from processing overloads by delegating the analysis to an external dedicated proxy. This position is not novel in terms of architecture but satisfies our requirements in terms of containment and usability: delegating the process has the dual advantage of freeing resources for the browser, preserving the quality of the user experience, and prevents any failure that would endanger the user if the analysis was performed in the browser.

This thesis presents research, in the area of program analysis and comprehension, that shows how intentions of a program can be formalized and used to express what a program is doing. We particularly focus on decomposing a program into unitary blocks that each express a particular functionality and how these blocks are combined to carry out a specific intention. A main obstacle to the inference of such intentions is their obfuscation using program transformations that target several layers of the source code: namely, the layout, the data and the control-flow. Moreover, some of the transformations are extremely resilient in that they are one-way, which means that the original program may never be recovered. Based on past results documented in the references attached to the present dissertation, we argue that while we may not recover the original program, it is still possible to recover a semantically-equivalent program. Such process, called deobfuscation, can be seen as dual: first, it is necessary to identify what is obfuscated; second, obfuscated contents should be evaluated to simplify the code.

To illustrate and validate these approaches, we investigated, in this thesis, three problems corresponding to three functionalities provided by our proposed system:

- Section 7 presents an alternative approach to obfuscation detection in JS programs that contrasts with current string-based and statistical approaches. As stated above, this is the first part of the dual deobfuscation process. The aim is

to precisely detect parts of the JavaScript program that are obfuscated in order to extract these for further processing. Based on the observation that obfuscation generates polymorphic code, we conclude that a more abstract measure of obfuscation was needed. In particular, we were inspired to use abstract syntax trees (ASTs) since we noticed that two programs obfuscated with the same obfuscation tool has similar, or even identical, abstract syntax structure.

Related work, in the field of web malware analysis, have stressed the prevalence of encoding/packing schemes in currently used obfuscation techniques. Encoding schemes display a two-part pattern with a decoding routine and an obfuscated string on which the decoding routine is applied to unpack the original script. Given its prevalence, we have been concentrating on detecting such patterns using abstract syntax trees. In particular, to fulfill the extraction goal of our approach, we needed to precisely detect the subtree in the AST that is characteristic of the obfuscation pattern.

By incorporating knowledge from a newly founded discipline named arbology, whose subject is the design of algorithms able to efficiently process tree structures, we implemented a pushdown automaton able to exactly match subtrees we have previously learned. The downside of this approach is its lack of flexibility leading to the classification of nearly-similar subtrees as negatives. We speculate that by extending the approach to accept induced subtrees expressing invariants between subtree variants, instead of bottom-up subtrees, we would be able to reduce the rate of false negatives;

- Section 9 presents a novel approach to JavaScript deobfuscation that does not rely on hooking the dynamic generation of additional code. Contrary to past proposals in web malware detection, we do not consider that obfuscated will seamlessly deobfuscate every time. Cloaking techniques witnessed in recent attack cases have been able to evade execution-based deobfuscation. There are several reasons that make dynamic detectors to fail on cloaked scripts, and in particular the side-effects of execution: values generated during execution may be captured by a specific predicate in the code that will either halt deobfuscation or trigger the generation of a benign code, eventually preventing detection.

  Additionally, static examination does not suffice to decide on the malice of an obfuscated script. This motivated us to find ways to cancel the obfuscation and allow our system to decide on the original source code, as most related approaches do.

In particular, related work operate by emulating the browsing environment and integrating modified versions of JavaScript engines. Since we do not assume the security of JavaScript, we proposed the radical approach to emulate JavaScript, or at least the subset of JavaScript used in obfuscation techniques. We further restrict our preliminary work to the emulation of encoding schemes.

With no prior work on execution-less JavaScript deobfuscation on which we could relate, we attempted to formalize the emulation of a decoding routine on the obfuscated string, as the rewriting of this obfuscated string through the decoding routine. This is akin to equational reasoning where a term can be reduced using equations belonging to an equational system. We were subsequently suggested to use the Maude language to axiomatize the process of unpacking of a JavaScript obfuscated string. In this special case of obfuscation, we are quite confident that the decoding routine will terminate yielding the original script that was obfuscated. We speculate on the extension of such reasoning on other obfuscation techniques we have reviewed in related work. In particular, Maude provides another class of modules, system modules, that embeds rewriting logic that provides the possibility to design arbitrary rules for rewriting;

- Section 11 presents an alternative approach to JavaScript categorization that attempts to associate programmatic concepts to more higher-level concepts, with the ultimate goal to distinguish benign intentions and malicious intentions. Web malware classification traditionally classifies execution traces using statistical, and more recently hierarchical features. Therefore variants that share the same intention but implemented differently may end classified in distinct classes. Additionally, a new variant (that displays a new implementation) of a known intention may evade detection.

  Using concept assignment, we propose that programs, implemented using different functions but holding the same intention, be represented with a unique model. This model is a labeled diagram built on the decomposition of the program into unitary objects that express a distinct functionality. We have designed a forward decomposition minimal and lightweight slicing algorithm to that end. A knowledge base that categorize objects and functions of a language into conceptual categories should allow for the labeling of the sliced objects. The final result is a hybrid object-sequence diagram that can than therefore be shared with human and machine agents. Moreover, we discuss the possible comparison and matching of models using a finite automaton as suggested in related work. We speculate

that this concept of intention can be extended to other languages, at least in the domain of client-side web security, but further to higher-order systems.

12.2 Avenues for Future Work

A number of refinements and extensions of the applications of the work presented in this dissertation are conceivable, as we already suggested in the corresponding sections.

The method to detect subtrees characteristic of obfuscation patterns may be refined by replacing the model of representation, the prefixed bottom-up subtree, by an induced subtree. However, *arbology* is based on properties of the prefix notation of bottom-up subtrees and therefore the application of a pushdown automaton to the model of induced subtrees may not produce the expected results. Therefore, further investigation is necessary to settle on this application. In the case, good properties of the prefix notation would not hold for the induced subtree, more investigation would be needed to come up with an alternative model of computation.

On a related note, we were also unable to come up with a feature selection to automatically learn subtrees that were recurring in a set of ASTs, provided the subtrees are hierarchical patterns of obfuscation. This last supposition is also debatable, and further extends the investigation to a method able to detect obfuscated strings and its related decoding routine. Capitalizing on related work in obfuscation detection, we would be able to detect obfuscated strings, but the extraction of this string and its related decoding routine would require tokenizing the script and analyzing each token individually (rejecting possible small ones). Once found, program slicing can be applied with the obfuscated string identifier as criterion, in order to detect all related statements in the script. This may or may not have the additional advantage of removing irrelevant code, depending on the nesting of the obfuscated string within the decoding routine.

The deobfuscation method we proposed only targets encoding schemes, which is the most prevalent obfuscating transformation in JavaScript. Other obfuscating transformations are not actually handled by our deobfuscator. However, the equational reasoning approach is promising, provided we can efficiently express obfuscated scripts as sets of equations. Moreover, in order to ensure good properties of confluence and termination, and therefore yield a canonical form, we may design specific rewrite rules using system modules that will ensure the above-mentioned properties. It is still early to tell whether this approach will be successful, but early results we obtained on a special case calls for generalization.

It is also interesting to see how we can make use of a rewriting system like Maude

to handle parallel instances of a same decoding routine, in particular to the rewriting of different versions of an obfuscated string. Doing so would benefit the evasion of cloaking techniques by parallelizing the computation of concurrent instances of a same obfuscated program, as it was already proposed using symbolic execution. In fact, Maude is used as an interpreter here and we can therefore attempt to apply compilation/optimization approaches. More precisely, a JavaScript compiler based on Maude can be envisioned as an insightful proof-of-concept.

Developing deobfuscation methods may also help to understand obfuscation of script contents. Our approach can be generalized to the survey of JavaScript obfuscation and the monitoring of trends, in order to forecast future obfuscation patterns. Of course, it will allow to have a view of currently-deployed obfuscation patterns: which obfuscation patterns are mostly found in malicious scripts, what is the probability of co-occurrence of one technique with another, an estimation of the number of obfuscation tools actually used, etc. Another interesting insight, and also a good discussion point, is the possibility to observe the gradual shift of obfuscation techniques used by malware to obfuscation techniques mostly used by benign JS programs, or even unobfuscated JS program patterns. It was actually speculated by Rajab et al. [117] that attackers will be confronted with a dilemma in trading off heavy obfuscation with detection evasion. Such estimations obviously require either of two things: large and up-to-date datasets, or a crawler able to evade cloaking techniques.

We speculate on the possibility to generalize the inference of intents for higher-order systems. Doing so will alleviate the need for mutual policy agreement, and will enhance model checking. Analysis of a system based on its intentions would generate a model of the actual functionalities embedded in the system, which can allow one to decide on the malice of such model. A first hint to such generalization is the trivial extension of intention models to other object-oriented programming languages. Indeed, program developed in these languages can be potentially decomposed using program slicing. A relevant debatable point is the decidability of intention, whether some intention is malicious or not. We have shown that intentions can be deduced but that they may not indicate actual malice. However, we can complement the analysis with other features such as URLs being used, the amont of memory being allocated, etc.

There is actually a particular issue with the proposed system: the heterogeneity of models used to represent the program along the different computational steps. Unifying the representation can be done along the refinement of the concept of intention, or as a gradual transformation of the intention model of an obfuscated program to the one of an unobfuscated program, and then, to the one of the higher-level concepts exhibited

by the unobfuscated program.

On a more general level, we may discuss the future of Web malware. There are actually tangible signs that JavaScript will still be prevalent in the years to come. On one hand, JavaScript has been the de-facto standard of client-side computation for some years, due not only to the presence of a builtin engine within common popular browsers, but also to due to the continuous development from vendors. As a matter of fact, JavaScript has gotten faster over the years, with now optimized versions of compilers that produce JavaScript bytecode. Such compilers may also suffer from vulnerabilities. A testament to the growing popularity of JavaScript is the existence of numerous instances of programming languages that use JavaScript as an intermediate language, such as Objective-J for Objective-C, Quby for Ruby, or Parenscript for Common Lisp. On the other hand, JavaScript is also plebiscited by the new HTML specification, HTML 5. Recent applications making use of heavy JavaScript programming merge new HTML 5 capabilities such as Canvas in order to produce sophisticated graphics or stunningly beautiful in-browser games, without relying on Flash or any other plugin. Obviously, future research will also need to concentrate on the network and cross-domain communication capabilities offered by HTML 5 and the ability of JavaScript to manipulate these to fulfill malicious intentions. An alternative scripting language being intensively developed by Google is the Dart language [59]. However, for compatibility purposes, Dart is provided with a compiler that compiles Dart to JavaScript.

A long-term prediction of future web technologies is difficult but efforts to advance a Web 3.0 (advocated by Tim Berners-Lee himself) as the Semantic Web has long been fueled by many contributions under the Semantic Web initiative [147]. It extends the network of hyperlinked human-readable web pages by inserting machine-readable metadata about pages and how they relate to each other. We can speculate that such metadata can be useful to the inference of intentions, by making the association of programming constructs with human concepts easier to achieve. It extends the network of hyperlinked human-readable web pages by inserting machine-readable metadata about pages and how they relate to each other. We can speculate that such metadata can be useful to the inference of intentions, by making the association of programming constructs with human concepts easier to achieve. Additionally, the Semantic Web initiative also concentrates on the development of ontologies. Given the fact that known Web malware vulnerabilities are referenced in vulnerabilities databases, we may be able to generate intention models from the descriptions of common exploits. Intentions being expressed as UML diagrams, which are universally used, it is also easy to

share. Moreover, human concepts are expressed as labels that can be translatd in many languages.

Finally, we have purposedly excluded the user from any decision-making in our approach. We are aware of the actual lack of web security education nowadays, which is particularly demonstrated in recent social network attacks. In fact, the user is often considered a vulnerability in many risk management approaches. Therefore, it is necessary to raise awareness among web users. Intention modeling that makes use of human concepts may be used to communicate on the malice of scripting contents present in web pages.

References

[1] Adobe. JavaScript for Acrobat. Available at: `http://www.adobe.com/devnet/acrobat/javascript.html`.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., 1986.

[3] Mitsuaki Akiyama, Makoto Iwamura, Yuhei Kawakoya, Kazufumi Aoki, and Mitsutaka Itoh. Design and Implementation of High Interaction Client Honeypot for Drive-by-Download Attacks. *IEICE Transactions on Communications*, E93-B(5):1131–1139, 2010.

[4] Alexa. Top 500 Global Sites. Available at: `http://www.alexa.com/topsites`.

[5] Si Alhir. *UML in a Nutshell*. O'Reilly, September 1998.

[6] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[7] Dare A. Baldwin and Jodie A. Baird. Discerning intentions in dynamic human action. *TRENDS in Cognitive Sciences*, 5(4):171–178, 2001.

[8] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In *Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, pages 1–18. Springer-Verlag, 2001.

[9] John Barnette. johnson. Available at: `http://github.com/jbarnette/johnson/`.

[10] Adam Barth, Collin Jackson, and John C. Mitchell. Robust Defenses for Cross-Site Request Forgeries. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, 2008.

[11] Abhijit Belapurkar. Functional Programming in the Java Language. Available at: `http://www.ibm.com/developerworks/java/library/j-fp.html`, July 2004.

[12] Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas Webster. The concept assignment problem in program understanding. In *Proceedings of the 15th International Conference on Software Engineering*, pages 482–498. IEEE Computer Society Press, 1993.

[13] Rafael Bosse Brinhosa, Carlos Becker Westphall, and Carla Merkle Westphall. A Security Framework for Input Validation. In *Proceedings of the Second International Conference on Emerging Security Information, Systems and Technologies*, pages 88–92. IEEE Computer Society, 2008.

[14] Adel Bouhoula, Jean-Pierre Jouannaud, and José Meseguer. Specification and Proof in Membership Equational Logic. *Theoretical Computer Science*, 236(1-2):35–132, April 2000.

[15] Mike Bowler. HtmlUnit. Available at: `http://htmlunit.sourceforge.net`.

[16] Norris Boyd. Rhino: JavaScript for Java. Available at: `http://www.mozilla.org/rhino/`.

[17] Michael E. Bratman. *Intention, Plans, and Practical Reason.* CLSI Publications, 1999.

[18] Lionel C. Briand, Yvan Labiche, and Johanne Leduc. Toward the reverse engineering of uml sequence diagrams for distributed java software. *IEEE Transactions on Software Engineering*, 32(9):642–663, September 2006.

[19] Nigel P. Chapman. *LR Parsing: Theory and Practice*. Cambridge University Press, 1987.

[20] Kumar Chellapilla and Alexey Maykov. A Taxonomy of JavaScript Redirection Spam. In *Proceedings of the 3rd International Workshop on Adversarial Information Retrieval on the Web*, pages 81–88. ACM, 2007.

[21] Brian Chess, Yekaterina Tsipenyuk O'Neil, and Jacob West. JavaScript Hijacking. Technical report, Fortify, 2007.

[22] Yun Chi, Richard R. Muntz, Siegfried Nijssen, and Joost N. Kok. Frequent Subtree Mining - An Overview. *Fundamenta Informaticae*, 66:161–198, November 2004.

[23] Michel Chilowicz, Etienne Duris, and Gilles Roussel. Syntax Tree Fingerprinting: a Foundation for Source Code Similarity Detection. In *Proceedings of the 17th IEEE International Conference on Program Comprehension*, pages 243–247, May 2009.

[24] Younghan Choi, Taeghyoon Kim, Seokjin Choi, and Cheolwon Lee. Automatic detection for javascript obfuscation attacks in web pages through string pattern analysis. In *Proceedings of the 1st International Conference on Future Generation Information Technology*, pages 160–172. Springer-Verlag, 2009.

[25] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí Oliet, José Meseguer, and Carolyn L. Talcott. The Maude System. Available at: `http://maude.cs.uiuc.edu`.

[26] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott. *Maude Manual (Version 2.6)*. SRI International, January 2011.

[27] Manuel Clavel, Steven Eker, Patrick Lincoln, and José Meseguer. Principles of maude. *Electronic Notes in Theoretical Computer Science*, 4:65 – 89, 1996.

[28] Philip R. Cohen and Hector J. Levesque. Intention is Choice with Commitment. *Artificial Intelligence*, 42:213–261, 1990.

[29] Christian Collberg, Clark Thomborson, and Douglas Low. A Taxonomy of Obfuscating Transfromations. Technical Report 148, The University of Auckland, 1997.

[30] Eric Costello. Remote Scripting with IFRAME. Available at: `http://oreilly.com/pub/a/javascript/2002/02/08/iframe.html`, February 2002.

[31] Marco Cova, Christopher Kruegel, and Giovanni Vigna. WEPAWET. Available at: `http://wepawet.cs.ucsb.edu`.

[32] Marco Cova, Christopher Kruegel, and Giovanni Vigna. Detection and Analysis of Drive-by-Download Attacks and Malicious JavaScript Code. In *Proceedings of the 19th International World Wide Web Conference*, 2009.

[33] Cristian Craioveanu. Server-side script polymorphism: Techniques of analysis and defense. In *3rd International Conference Malicious and Unwanted Software*, pages 9 –16, October 2008.

[34] Douglas Crockford. The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627 (Informational), July 2006.

[35] Weidong Cui, Randy H. Katz, and Wai-tian Tan. BINDER: An Extrusion-based Break-In Detector for Personal Computers. In *USENIX Annual Technical Conference*, pages 363–366, 2005.

[36] Charlie Curtsinger, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. ZOZZLE: Fast and Precise In-Browser JavaScript Malware Detection. In *Proceedings of the 20th USENIX Security Symposium*, August 2011.

[37] David Dagon, Manos Antonakakis, Kevin Day, Xiapu Luo, Christopher P. Lee, and Wenke Lee. Recursive dns architectures and vulnerability implications. In *Proceedings of the 16th Annual Network & Distributed System Security Symposium*. The Internet Society, 2009.

[38] Sebastian Danicic, Chris Fox, Mark Harman, Rob Hierons, John Howroyd, and Michael R. Laurence. Static Program Slicing Algorithms are Minimal for Free Liberal Program Schemas. *The Computer Journal*, 48:737–748, 2006.

[39] Romain Delamare and Benoît Baudry. Reverse-engineering of uml 2.0 sequence diagrams from execution traces. In *Proceedings of the 20th European Conference on Object-Oriented Programming*, 2006.

[40] Stefano Di Paola and Giorgio Fedon. Subverting Ajax. 23rd Chaos Communication Congress, 2006.

[41] Ecma International. ECMAScript Documentation. Available at: `http://www.ecmascript.org/docs.php`.

[42] Ecma International. Standard Ecma-262: ECMAScript Language Specification. Available at: `http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf`.

[43] Dean Edwards. Packer. Available at: `http://dean.edwards.name/packer/`.

[44] Manuel Egele, Engin Kirda, and Christopher Kruegel. Mitigating Drive-By Download Attacks: Challenges and Open Problems. In *IFIP in Advances in Information and Communication Technology*. Springer, 2009.

[45] Brendan Eich. SpiderMonkey (JavaScript-C) Engine. Available at: `http://www.mozilla.org/js/spidermonkey/`.

[46] Eldad Eilam. *Reversing: Secrets of Reverse Engineering*. Wiley Publishing, Inc., 2005.

[47] Jeremy Elson and Alberto Cerpa. Internet Content Adaptation Protocol (ICAP). RFC 3507 (Informational), April 2003.

[48] Stefan Esser. CSRF protections are not doomed by XSS. Available at: `http://blog.php-security.org/archives/48-CSRF-protections-are-not-doomed-by-XSS.html`, November 2006.

[49] Jeanne Ferrante and Karl J. Ottenstein. A Program Form based on Data Dependency in Predicate Regions. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 217–236. ACM, 1983.

[50] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFC 2817.

[51] David Flanagan. *JavaScript: The Definitive Guide*, chapter 13, pages 332–338. O'Reilly, sixth edition, 2011.

[52] Tomáš Flouri, Jan Janoušek, and Bořivoj Melichar. Subtree Matching by Pushdown Automata. *Computer Science and Information Systems*, 7(2):331–357, April 2010.

[53] Tomáš Flouri, Bořivoj Melichar, and Jan Janoušek. Subtree Matching by Deterministic Pushdown Automata. In *International Multiconference on Computer Science and Information Technology*, pages 659–666, October 2009.

[54] Robert B. France, Dae-Kyoo Kim, Sudipto Ghosh, and Eunjee Song. A UML-Based Pattern Specification Technique. *IEEE Transactions on Software Engineering*, 30:193–206, March 2004.

[55] Keith Brian Gallagher and James R. Lyle. Using Program Slicing in Software Maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, August 1991.

[56] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[57] Jesse J. Garrett. Ajax: A New Approach to Web Applications. Available at: `http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications`, February 2005.

[58] Joseph Goguen and Grant Malcolm. *Algebraic Semantics of Imperative Programs*. MIT Press, 1996.

[59] Google, Inc. Dart. Available at: `http://www.dartlang.org`.

[60] Google, Inc. Google Safe Browsing API. Available at: `http://code.google.com/apis/safebrowsing/`.

[61] Jeremiah Grossman. Web 2.0 Pivot Attacks. Available at: `http://jeremiahgrossman.blogspot.com/2010/02/web-20-pivot-attacks.html`, February 2010.

[62] Jeremiah Grossman, Robert Hansen, Petko D. Petkov, Anton Rager, and Seth Fogie. *XSS Attacks - Cross Site Scripting Exploits and Defense*. Syngress, 2007.

[63] Jeremiah Grossman and T.C. Niedzialkowski. Hacking Intranet Websites from the Outside. Black Hat Conference USA, 2006.

[64] Robert Hansen. XSS (Cross Site Scripting) Cheat Sheet. Available at: `http://ha.ckers.org/xss.html`.

[65] Blake Hartstein. jsunpack - a Generic JavaScript Unpacker. Available at: `http://jsunpack.jeek.org`.

[66] Billy Hoffman. JavaScript Malware for a Gray Goo Tomorrow! ShmooCon, 2007.

[67] Billy Hoffman and Bryan Sullivan. *Ajax Security*, chapter 7, pages 175–200. Addison-Wesley, 2007.

[68] Billy Hoffman and Bryan Sullivan. *Ajax Security*. Addison-Wesley (Pearson), 2007.

[69] Yung-Tsung Hou, Yimeng Chang, Tsuhan Chen, Chi-Sung Laih, and Chia-Mei Chen. Malicious web content detection by machine learning. *Expert Systems with Applications*, 37(1):55–60, January 2010.

[70] Fraser Howard. Malware with your Mocha? Obfuscation and antiemulation tricks in malicious JavaScript. Technical report, Sophos, September 2010.

[71] Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21(3):359–411, September 1989.

[72] Galen Hunt and Doug Brubacher. Detours: Binary interception of win32 functions. In *Proceedings of the 3rd Conference on USENIX Windows NT Symposium*. USENIX Association, 1999.

[73] Nelson T. Icuma, Jingde Cheng, and Kazuo Ushijima. Decomposition Slicing Based on Program Dependence Graph. 全国大会講演論文集, 45(5):251–252, 1992.

[74] Information Processing Society of Japan. anti-Malware engineering WorkShop (MWS). Available at: `http://www.iwsec.org/mws/2011/`.

[75] International Conference of Program Comprehension. Available at: `http://www.program-comprehension.org`.

[76] Kirill S. Ivanov and Vladimir A. Zakharov. Program Obfuscation as Obstruction of Program Static Analysis. In *Russian Academy of Sciences Technical Report Series*, June 2004.

[77] Bart Jansen and Tony Belpaeme. A Computational Model of Intention Reading in Imitation. *Robotics and Autonomous Systems*, 54:394–402, 2006.

[78] John Jean. Facebook CSRF and XSS vulnerabilities. Available at: `http://www.wargan.com/facebook-multiple-vulnerabilities-051010.php`, October 2010.

[79] Martin Johns. On JavaScript Malware and Related Threats. *Journal in Computer Virology*, 4(3):161–178, August 2008.

[80] Martin Johns and Justus Winter. RequestRodeo: Client Side Protection against Session Riding. OWASP AppSecEU, 2006.

[81] Martin Johns and Justus Winter. Protecting the Intranet Against "JavaScript Malware" and Related Attacks. In *Proceedings of the 4th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 40–59, Berlin, Heidelberg, 2007. Springer-Verlag.

[82] Nenad Jovanovic, Engin Kirda, and Christopher Kruegel. Preventing Cross Site Request Forgery Attacks. In *Proceedings of the Second International Conference on Security and Privacy in Communication Networks*, 2006.

[83] Masaki Kamizono, Masata Nishida, Emi Kojima, and Yuji Hoshizawa. Categorizing Hostile JavaScript using Abstract Syntax Tree Analysis. In マルウェア対策研究人材育成ワークショップ *2011(MWS2011)*, pages 474–479, October 2011. in Japanese.

[84] Scott Kaplan, Benjamin Livshits, Benjamin Zorn, Christian Siefert, and Charlie Curtsinger. "nofus: Automatically detecting" + string.fromcharcode(32) + "obfuscated ".tolowercase() + "javascript code". Technical Report MSR-TR-2011-57, Microsoft Research, May 2011.

[85] Rudolf K. Keller, Reinhard Schauer, Sébastien Robitaille, and Patrick Pagé. Pattern-based Reverse-engineering of Design Components. In *Proceedings of the 21st International Conference on Software Engineering*, pages 226–235. ACM, 1999.

[86] Byung-Ik Kim, Chae-Tae Im, and Hyun-Chul Jung. Suspicious Malicious Web Site Detection with Strength Analysis of a JavaScript Obfuscation. *International Journal International Journal of Advanced Science and Technology*, 26:19–32, January 2011.

[87] Clemens Kolbitsch, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. Rozzle: De-Cloaking Internet Malware. Technical Report MSR-TR-2011-94, Microsoft Research, October 2011.

[88] Ralf Kollman, Petri Selonen, Eleni Stroulia, Tarja Systä, and Albert Zündorf. A study on the current state of the art in tool-supported uml-based static reverse engineering. In *Proceedings of the Ninth Working Conference on Reverse Engineering*, pages 22–. IEEE Computer Society, 2002.

[89] Peter Likarish, Eunjin Jung, and Insoon Jo. Obfuscated malicious javascript detection using classification techniques. In *4th International Conference on Malicious and Unwanted Software*, pages 47 –54, October 2009.

[90] Wei Lu and Min-Yen Kan. Supervised Categorization of JavaScript™using Program Analysis Features. In *Proceedings of the Second Asia Information Retrieval Symposium*, 2005.

[91] Ziqing Mao, Ninghui Li, and Ian Molloy. Defeating Cross-Site Request Forgery Attacks with Browser-Enforced Authenticity Protection. In *Financial Cryptography and Data Security*, pages 238–255. Springer-Verlag, 2009.

[92] Giorgio Maone. NoScript. Available at: `http://noscript.net/`.

[93] Scott Martens. Varro: An Algorithm and Toolkit for Regular Structure Discovery in Treebanks. In *Proceedings of the 23rd International Conference on Computational Linguistics: Posters*, pages 810–818. Association for Computational Linguistics, 2010.

[94] Sarah Matzko, Peter J. Clarke, Tanton H. Gibbs, Brian A. Malloy, James F. Power, and Rosemary Monahan. Reveal: a Tool to Reverse Engineer Class Diagrams. In *Proceedings of the Fortieth International Conference on Tools Pacific: Objects for Internet, Mobile and Embedded Applications*, pages 13–21. Australian Computer Society, Inc., 2002.

[95] MDL. Malware Domain List. Available at: `http://malwaredomainlist.com`.

[96] Bořivoj Melichar. Arbology: Trees and Pushdown Automata. In *Proceedings of the 4th International Conference on Language and Automata Theory and Applications*, pages 32–49. Springer, 2010.

[97] José Meseguer. Membership Algebra as a Logical Framework for Equational Specification. In *Selected papers from the 12th International Workshop on Recent Trends in Algebraic Development Techniques*, pages 18–61, 1997.

[98] José Meseguer. Software Specification and Verification in Rewriting Logic. *Models, Algebras and Logic of Engineering Software*, 2003.

[99] Daisuke Miyamoto, Gregory Blanc, and Mitsuaki Akiyama. A Consideration for Categorizing JavaScript Files based on Abstract Syntax Tree Fingerprinting. In マルウェア対策研究人材育成ワークショップ *2011(MWS2011)*, pages 462–467, October 2011. in Japanese.

[100] Alexander Moshchuk, Tanya Bragin, Damien Deville, Steven D. Gribble, and Henry M. Levy. Spyproxy: Execution-based detection of malicious web content. In *Proceedings of 16th USENIX Security Symposium*. USENIX Association, 2007.

[101] Benjamin Mossé. Browser-Rider. Available at: `http://code.google.com/p/browser-rider/`.

[102] Peter Mosses. *Action Semantics*. Cambridge University Press, 1992.

[103] National Institute of Standards and Technology. Secure Hash Standard. Available at: http://www.itl.nist.gov/fipspubs/fip180-1.htm.

[104] José Nazario. Reverse Engineering Malicious JavaScript. CanSecWest, 2007.

[105] Johnathan Nightingale. Fraudulent *.google.com Certificate. Available at: https://blog.mozilla.com/security/2011/08/29/fraudulent-google-com-certificate/, August 2011.

[106] John O'Donnell, Cordelia Hall, and Rex Page. *Discrete Mathematics Using a Computer*, chapter 2, pages 37–60. Springer, 2006.

[107] Object Management Group. Unified Modeling Language™(UML®). Available at: http://www.omg.org/spec/UML/.

[108] OpenDNS. PhishTank - Join the fight against phishing. Available at: http://www.phishtank.com.

[109] Karl J. Ottenstein and Linda M. Ottenstein. The Program Dependence Graph in a Software Development Environment. In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 177–184. ACM, 1984.

[110] Wladimir Palant. Javascript deobfuscator. Available at: https://addons.mozilla.org/en-US/firefox/addon/javascript-deobfuscator/.

[111] Petko D. Petkov. Google Gmail e-mail Hijack Technique. Available at: http://www.gnucitizen.org/blog/google-gmail-e-mail-hijack-technique/, September 2007.

[112] Charles Petrie. Pragmatic semantic unification. *IEEE Internet Computing*, 9(5):96–97, 2005.

[113] David A. Plaisted. *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 1, chapter 2, pages 274–364. Oxford University Press, 1993.

[114] Frederic Portoraro. Automated reasoning. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Summer 2011 edition, 2011.

[115] Niels Provos, Dean McNamee, Panayiotis Mavrommatis, Ke Wang, and Nagendra Modadugu. The ghost in the browser analysis of web-based malware. In *Proceedings of the First Workshop on Hot Topics in Understanding Botnets*. USENIX Association, 2007.

[116] Anton Rager. Advanced Cross Site Scripting - Evil XSS. Shmoocon, 2005.

[117] Moheeb Abu Rajab, Lucas Ballard, Nav Jagpal, Panayiotis Mavrommatis, Daisuke Nojiri, Niels Provos, and Ludwig Schmidt. Trends in Circumventing Web-Malware Detection. Technical Report rajab-2011a, Google, Inc,, July 2011.

[118] Konrad Rieck, Tammo Krueger, and Andreas Dewald. Cujo: Efficient Detection and Prevention of Drive-by-download Attacks. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 31–39. ACM, 2010.

[119] Ivan Ristić. Web Application Firewalls Primer. *(IN)SECURE*, (5):4–10, 2006.

[120] Ronald Rivest. The MD5 Message-Digest Algorithm. RFC 1321 (Informational), April 1992.

[121] Atanas Rountev, Olga Volgin, and Miriam Reddoch. Static Control-flow Analysis for Reverse Engineering of UML Sequence Diagrams. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 96–102. ACM, 2005.

[122] Atanas Rountev, Olga Volgin, and Miriam Reddoch. Static Control-flow Analysis for Reverse Engineering of UML Sequence Diagrams. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 96–102. ACM, 2005.

[123] Natarajan Shankar. Automated Deduction for Verification. *ACM Computing Surveys*, 41(4), October 2009.

[124] Chris Shiflett. Cross-Site Request Forgeries. Available at: `http://shiflett.org/articles/cross-site-request-forgeries`, December 2004.

[125] Kenneth Slonneger and Barry L. Kurtz. *Formal Syntax and Semantics of Programming Languages*. Addison-Wesley, 1995.

[126] Software Engineering Research Group, University of Paderborn. The Fujaba Tools Suite. Available at: http://www.fujaba.de.

[127] Alexander Sotirov. Heap Feng Shui in JavaScript. Available at: http://www.phreedom.org/research/heap-feng-shui/.

[128] Boban Spasic. Malzilla - Malware Hunting Tool. Available at: http://malzilla.sourceforge.net.

[129] StopBadware. badwarebusters.org - the stopbadware online community. Available at: http://badwarebusters.org.

[130] Dafydd Stuttard and Marcus Pinto. *The Web Application Hacker's Handbook*. Wiley, 2007.

[131] Andrew Sutton and Jonathan I. Maletic. Mappings for accurately reverse engineering uml class models from c++. In *Proceedings of the 12th Working Conference on Reverse Engineering*, pages 175–184. IEEE Computer Society, 2005.

[132] The Open Web Application Security Project. Available at: http://www.owasp.org.

[133] The Open Web Application Security Project. OWASP Top 10 - 2010. Available at: http://owasptop10.googlecode.com/files/OWASPTop10-2010.pdf.

[134] The Web Application Security Consortium. Available at: http://www.webappsec.org.

[135] Paolo Tonella and Alessandra Potrich. Static and dynamic c++ code analysis for the recovery of the object diagram. In *Proceedings of the International Conference on Software Maintenance*, pages 54–. IEEE Computer Society, 2002.

[136] Paolo Tonella and Alessandra Potrich. Reverse engineering of the interaction diagrams from c++ code. In *Proceedings of the 19th International Conference on Software Maintenance*, pages 159–. IEEE Computer Society, 2003.

[137] Paolo Tonella and Alessandra Potrich. *Reverse Engineering of Object Oriented Code*. Monographs in Computer Science. Springer, 2005.

[138] Sharath K. Udupa, Saumya K. Debray, and Matias Madou. Deobfuscation: Reverse Engineering Obfuscated Code. In *Proceedings of the 12th Working Conference on Reverse Engineering*, pages 45–54. IEEE Computer Society, 2005.

[139] Secil Ugurel, Robert Krovetz, and C. Lee Giles. What's the Code?: Automatic Classification of Source Code Archives. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 632–638. ACM, 2002.

[140] Anne van Kesteren. Cross-Origin Resource Sharing. Available at: `http://www.w3.org/TR/cors/`.

[141] Anne van Kesteren. XMLHttpRequest Level 2. Available at: `http://www.w3.org/TR/XMLHttpRequest2/`.

[142] David Y. Wang, Stefan Savage, and Geoffrey M. Voelker. Cloak and Dagger: Dynamics of Web Search Cloaking. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, pages 477–490. ACM, 2011.

[143] Web of Trust. Safe Browsing Tool — WOT (Web of Trust). Available at: `http://www.mywot.com`.

[144] Mark Weiser. Program Slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–449. IEEE Press, 1981.

[145] Lothar Wendehals. Specifying patterns for dynamic pattern instance recognition with uml 2.0 sequence diagrams. In *Proceedings of the 6th Workshop on Software Reenginering*, 2004.

[146] Lothar Wendehals and Alessandro Orso. Recognizing Behavioral Patterns at Runtime using Finite Automata. In *Proceedings of the 2006 International Workshop on Dynamic Systems Analysis*, pages 33–40. ACM, 2006.

[147] World Wide Web Consortium. Semantic Web. Available at: `http://semanticweb.org`.

[148] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. A Brief Survey of Program Slicing. *SIGSOFT Software Engineering Notes*, 30(2):1–36, March 2005.

Appendix

A. List of Publications

A.1 Journal

1-1. Gregory Blanc and Youki Kadobayashi, "A Step Towards Static Script Malware Abstraction: Rewriting Obfuscated Script with Maude", *IEICE Transactions on Information and Systems*, Vol. E94-D, No. 11, pp. 2159–2166, November 2011.

A.2 International Conference

2-1. Gregory Blanc and Youki Kadobayashi, "Towards Learning Intentions in Web 2.0", In *Proceedings of the 4th Joint Workshop on Information Security (JWIS 2009)*, Kaohsiung, Taiwan, August 2009.

2-2. Gregory Blanc and Youki Kadobayashi, "Towards Real-time JavaScript Deobfuscation for Analysis Purposes", In *Proceedings of the 5th Joint Workshop on Information Security (JWIS 2010)*, Guangzhou, China, August 2010.

2-3. Gregory Blanc and Youki Kadobayashi, "Towards Revealing JavaScript Program Intents using Abstract Interpretation", In *Proceedings of the 6th Asian Internet Engineering Conference (AINTEC 2010)*, Bangkok, Thailand, November 2010.

2-4. Gregory Blanc, Ruo Ando and Youki Kadobayashi, "Term-Rewriting Deobfuscation for Static Client-Side Scripting Malware Detection", In *Proceedings of the 4th IFIP International Conference on New Technologies, Mobility and Security (NTMS 2011)*, Paris, France, February 2011.

2-5. Gregory Blanc, Daisuke Miyamoto, Mitsuaki Akiyama and Youki Kadobayashi, "Characterizing Obfuscated JavaScript using Abstract Syntax Trees: Experimenting with Malicious Scripts", In *Proceedings of the 26th IEEE International Conference on Advanced Information Networking and Applications Workshops (WAINA 2012)*, Fukuoka, Japan, March 2012 (to appear).

2-6. Takeshi Takahashi, Gregory Blanc, Youki Kadobayashi, Doudou Fall, Hiroaki Hazeyama and Shinichiro Matsuo, "Enabling Secure Multitenancy in Cloud Computing: Challenges and Approaches", In *Proceedings of the 2nd Baltic Conference on Future Internet Communications (BCFIC 2012)*, Vilnius, Lithuania, April 2012 (to appear).

2-7. Noppawat Chaisamran, Takeshi Okuda, Gregory Blanc and Suguru Yamaguchi, "Trust-based VoIP Spam Detection based on Call Duration and Human Relationships", In *Proceedings of the 11th IEEE/IPSJ International Symposium on Applications and the Internet (SAINT 2011)*, Munich, Germany, July 2011.

2-8. Doudou Fall, Gregory Blanc, Takeshi Okuda, Youki Kadobayashi and Suguru Yamaguchi, "Toward Quantified Risk-Adaptive Access Control for Multi-tenant Cloud Computing", In *Proceedings of the 6th Joint Workshop on Information Security (JWIS 2011)*, Kaohsiung, Taiwan, August 2011.

A.3 Technical Report

3-1. Gregory Blanc, Mitsuaki Akiyama, Daisuke Miyamoto and Youki Kadobayashi, "Identifying Characteristic Syntactic Structures in Obfuscated Scripts by Subtree Matching", コンピュータセキュリティシンポジウム 2011 (CSS 2011), pp. 468–473, 2011 年 10 月.

3-2. 宮本大輔, Gregory Blanc, 秋山満昭, "抽象構文木を用いた Javascript ファイルの分類に関する一検討", コンピュータセキュリティシンポジウム 2011 (CSS 2011), pp. 462–467, 2011 年 10 月. (in Japanese).

3-3. Xin Wang, 岡田和也, Gregory Blanc, 奥田剛, 山口英, "Chord における Sybil Node 検知手法の提案と評価", コンピュータセキュリティシンポジウム 2010 (CSS 2010), pp. 723–728, 2010 年 10 月. (in Japanese).

3-4. 森久和昭, 神宮真人, 神田慎也, Gregory Blanc, 門林雄基, "通信ログを基にしたマルウェア配布オペレーションの抽出と可視化の試み", コンピュータセキュリティシンポジウム 2010 (CSS 2010), pp. 885–890, 2010 年 10 月. (in Japanese).

3-5. Noppawat Chaisamran, Gregory Blanc, Kazuya Okada, Takeshi Okuda, and Suguru Yamaguchi, "Basic Trust Calculation to Prevent Spam in VoIP Network based on Call Duration (Single Hop Consideration)", In *IEICE Technical Report, IA2010-51*, Vol. 110, pp. 1–6, November 2010.

3-6. 神宮真人, Gregory Blanc, 奥田剛, 山口英, "脆弱性がもたらす影響をトレース可能な遷移グラフの提案", コンピュータセキュリティシンポジウム 2011 (CSS 2011), 2011 年 10 月. (in Japanese).

Other Talks

4-1. Gregory Blanc, "Measurement Research to the Web Calamity's Rescue", *3rd CAIDA-WIDE-CASFI Joint Measurement Workshop*, Osaka, Japan, April 2010.

4-2. Gregory Blanc, "Web 2.0 Security", *Internet Summit of Africa, Web Workshop (jointly with AfNOG/AfriNIC/AfREN)*, Kigali, Rwanda, May 2010.

4-3. Gregory Blanc, "Naviguer sur le Web 2.0 en toute sécurité : analyse de code JavaScript côté client" (poster), *Journée Francophone de la Recherche*, Tokyo, Japan, November 2010. (in French).

B. Research Grants and Scholarships

2009-2010 French Ministry of Foreign and European Affairs

2008/2009 Lavoisier Japon Scholarship Program

18 months from January 2009

2011-2012 NEC C&C Foundation

FY 2011 Grant for Non-Japanese Researcher

12 months from April 2011