UNIVERSITÉ DE LORRAINE

École doctorale IAEM Lorraine

Modèle de protection contre les codes malveillants dans un environnement distribué

THÈSE

présentée et soutenue publiquement le 11 Mai 2015

pour l'obtention du

Doctorat de l'Université de Lorraine

(mention informatique)

par

Thanh Dinh TA

Composition du jury

Rapporteurs :	Juan Caballero	Assistant research professor - IMDEA-Software
	Mourad Debbabi	Professor - Concordia University
Examinateurs :	Guillaume Bonfante	Maitre de conférences - Université de Lorraine
	Eric Freyssinet	Colonel de gendarmerie - Gendarmerie Nationale
	Stephan Merz	Directeur de recherche - INRIA Nancy
Directeurs de thèse :	Jean-Yves MARION	Professeur - Université de Lorraine
	Guillaume Bonfante	Maitre de conférences - Université de Lorraine

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Mis en page avec la classe thesul.

Remerciements

First, I would like to express my gratitude to my thesis advisor Jean-Yves Marion who gives me opportunities to pursuit the Ph.D program. Without him, I would not have any chance to realize my longing to do research, and this thesis would not complete. With his immense knowledge and tolerance, he shows me clearly essences of confused problems, and notices me which problem I should focus on, as well as which one I could safely omit.

I would like to thank my thesis co-advisor Guillaume Bonfante who always encourages me to think about new ideas, rectifies me when I seem go in the wrong direction, but also lets me pursuit independently my own ideas. I learned from him that, a fancy idea will be useless if one does not seriously realize it. He tolerates my mistakes and simply considers that they are lessons for me. He teaches me also that one should never underestimate the importance of simple ideas.

I would like to thank Romain Péchoux who guides me at the first stage of the thesis. He gives me his precious experiences in doing research that one must keep tenacity. He lets me pursuit my own ideas, encourages me when I fall back into despair, and tolerates me when my ideas lead to nothing. I learned also from him that ideas should be described as simple as possible.

I would like to thank my colleagues Aurélien Thiery, Hugo Férée, Hubert Godfroy, Fabrice Sabatier, Joan Calvet. Each of them has participated, directly or indirectly, into my studies. They pull me out of my corner, shares with me their ideas and experiences, both in life and in doing research.

I would like to thank members of CARTE. During my thesis, I do not feel alone, instead I have experienced a warmly atmosphere as a member of this group.

Thank to my parents who always encourage me and believe that I can do something useful. Though do not interfere in, they follow me in any decision I made. They share with me both weal and woe.

Finally, thank to my family. I have reserved very little time for them during the thesis. Thank to my wife who is always with and supports me. I learned from her that everything have always a deserved value, and we will finally recognize that, sooner or later. Thank to my son, who always requests me to play with him and always accepts my refuses, most of ideas of mine occur in playing some games with him.

Pour mon père qui m'a appris à faire des mathématiques, et pour ma mère qui m'a appris beaucoup d'autres choses.

Contents

Chapter 1 Introduction **1**
1.1 Malware detection and challenges . . . 2
1.1.1 Malicious code detection . . . 2
1.1.2 Challenges . . . 4
1.2 Plan and contributions of the thesis . . . 7

Chapter 2 On malware communications **11**
2.1 Introduction . . . 11
2.2 Automated malware analysis . . . 12
2.2.1 Limits of the static and the dynamic analysis . . . 12
2.2.2 Approach from software testing and code coverage . . . 16
2.3 Message format extraction . . . 17
2.3.1 Current research . . . 18
2.3.2 What is the "message format"? . . . 24
2.3.3 Our approach: message classification . . . 26

Chapter 3 Mathematical modeling **29**
3.1 Notations and terminology . . . 29
3.1.1 Graphs . . . 29
3.1.2 Labeled transition systems . . . 31
3.2 Prediction semantics of label transition systems . . . 35
3.2.1 Π-equivalence . . . 36
3.2.2 Observational prediction . . . 43
3.2.3 Labeled transition system reduction . . . 51
3.3 Algorithms . . . 62
3.3.1 Compact labeled transition system construction . . . 63
3.3.2 Π-equivalence verification and construction . . . 63

Chapter 4 Concrete interpretation **69**

4.1 Notions and terminology 69
4.1.1 Low-level representation 71
4.1.2 Assembly code representation 72
4.1.3 Execution of programs 75
4.2 Stepwise abstraction 81
4.2.1 From execution traces to the execution tree 82
4.2.2 From the execution tree to the message tree 89
4.2.3 From the message tree to the initial LTS 96
4.2.4 From the initial to the final LTS 97
4.2.5 Message classification by the final LTS 98
4.3 Implementation and experimentation 100
4.3.1 Implementation 100
4.3.2 Experimentation 102
4.3.3 Limits 117
4.3.4 Case of ZeuS bot 119

Chapter 5 Behavioral obfuscation **125**

5.1 Introduction 125
5.2 Behavior modeling 127
5.2.1 Motivating example 127
5.2.2 Basic categorical background 128
5.2.3 Syscall interaction abstract modeling 130
5.2.4 Process behaviors as paths and semantics 134
5.3 Behavioral obfuscation 137
5.3.1 Ideal detectors 137
5.3.2 Main obfuscation theorem 138
5.3.3 Obfuscated path generation through path replaying 140
5.3.4 Graph-based path transformation 142
5.4 Implementation and experiments 145
5.5 Obfuscated path detection 148

Chapter 6 Conclusions and perspectives **151**

6.1 Conclusions 151
6.2 Perspectives 153

Bibliography **155**

List of Figures

2.1 Bootstrap codes of Slackbot 14
2.2 A HTTP response message 19
2.3 From input messages to observable behaviors 26

3.1 Graphs and morphisms 31
3.2 An exemplified LTS 33
3.3 Abstraction procedure 38
3.4 LTS with π-equivalences 39
3.5 LTS in Example 3.2.1 40
3.6 LTS in Example 3.2.2 41
3.7 Observational predictions of a LTS 44
3.8 Trivial invalid observational prediction 45
3.9 Construction of the target LTS and the surjective complete morphism . . 46
3.10 Choosing an image between p_i(s) for a new state q' 49
3.12 Original LTS $\mathscr{A} = \mathscr{A}_1 = \mathscr{A}_1' = \mathscr{A}_1''$ 56
3.13 Reduction sequence $\mathscr{A}_1 \sqsubseteq_{op} \mathscr{A}_2 \sqsubseteq_{op} \mathscr{A}_3 \sqsubseteq_{op} \mathscr{A}_4$ 56
3.14 Reduction sequence $\mathscr{A}_1' \sqsubseteq_{op} \mathscr{A}_2' \sqsubseteq_{op} \mathscr{A}_3'$ 57
3.15 Reduction sequence $\mathscr{A}_1'' \sqsubseteq_{op} \mathscr{A}_2''$ 57
3.16 LTS having normal forms with different number of states 59
3.17 Sub LTS(s) 60
3.18 Intersected reduction sequences 61

4.1 Control flow diversion using `NtContinue` 73
4.2 Simple execution tree 80
4.3 Initial labeled transition system of `wget` with chosen finite length of 60 . 83
4.4 Execution tree of `wget` with the finite length of 25 86
4.5 Execution tree where only CFI(s) in traces are kept 87
4.6 Simplified execution tree of P 89
4.7 Message tree $\mathcal{M}(P)$ of P 89
4.8 Normalized message tree of `wget` with limit length 60 97
4.9 Initial LTS of `wget` with limit length 60 98
4.10 LTS(s) of `wget` with limit length 65 99
4.11 Final LTS of `wget` with limit length 90 101

4.12 Code coverage running time 107
4.13 Exponential growth of CFI(s) in execution trees 108
4.14 Different CFI(s) process the same group of bytes 109
4.15 Final labeled transition systems 110
4.16 Invalid message classification given by a LTS 112
4.17 Unbranchable CFI because of supplemental factors 113
4.18 Unbranchable CFI because of redundancy 113
4.19 Initial labeled transition system of `links` abstracted from execution trees of different trace's limit lengths 114
4.20 Execution tree of `links` of limit length 100 115
4.21 Input message dependent CFI(s) of `links` 116
4.22 Manual reverse engineering of the input message parser of `links` 117
4.23 Selecting process for injecting codes 120
4.24 Injecting codes into the opened process 121
4.25 Activating codes 121
4.26 Final LTS of ZeuS with limit length 350 122

5.1 Inductive construction 136
5.2 Obfuscated path construction 140
5.3 String diagrams 143
5.4 Registry initializing string diagram 146
5.5 File copying string diagram 146
5.6 File copying original string diagram 147
5.7 Code injecting string diagram 148
5.8 Code injecting obfuscated string diagram 148

Listings

1.1 Counter example of AntiM . . . 3
1.2 Hiding malicious behaviors using a 3SAT-opaque predicate . . . 4
1.3 Environment-dependent malware . . . 5
1.4 `W95/Regswap` . . . 6
1.5 A variant of `W95/Regswap` . . . 6
2.1 Bootstrap codes of Slackbot backdoor . . . 13
2.2 Trojan's main thread . . . 15
2.3 Second thread . . . 15
2.4 String matching using the Knuth-Morris-Pratt algorithm . . . 22
2.5 Processing of `wget` in searching for the line-feed character . . . 23
2.6 Processing of `curl` in searching for the line-feed character . . . 23
2.7 Simple type propagation . . . 25
3.1 Surjective complete morphism construction algorithm . . . 48
3.2 Construct $\mathscr{A}[i]$ by modifying $\mathscr{A}[i-1]$ . . . 50
3.3 Label transition system reduction algorithm . . . 63
3.4 π-equivalence verification algorithm . . . 64
3.5 π-equivalence construction algorithm . . . 66
4.1 Incorrect disassembly . . . 73
4.2 Correct disassembly . . . 73
4.3 `Trojan.Zbot` . . . 74
4.4 Effect of the control flow instruction . . . 76
4.5 "HTTP" string matching . . . 77
4.6 Sequence of executed instructions with input `‘‘HTa...’’` . . . 77
4.7 Exception `syscall` . . . 78
4.8 Control-flow depends on the initial state . . . 79
4.9 Execution tree construction algorithm . . . 84
4.10 Indirect dependency . . . 90

4.11 Dynamic order parsing . . . 94
4.12 Normalized message tree construction algorithm . . . 95
5.1 File unpacking . . . 127
5.2 Registry initializing . . . 127
5.3 Internal computation . . . 133
5.4 Syscall invocation . . . 133
5.5 Path X_1 . . . 135
5.6 Path X_2 . . . 135
5.7 Path $X_{1\text{-}2}$ . . . 140
5.8 Replay $rep(X_2)$ of X_2 . . . 141
5.9 Path X_3 . . . 144
5.10 Path X_4 . . . 144
5.11 Path X_5 . . . 145
5.12 File copying . . . 146
5.13 Code injecting . . . 147
5.14 Replacing NtReadVirtualMemory . . . 148

List of Tables

4.1 Number of CFI(s) in the execution tree of (the `HTTP` parser of) `wget` . . . 104
4.2 Number of CFI(s) in the execution tree of `links` 105
4.3 Number of CFI(s) in the execution tree of `ncftpget` 106

Chapter 1

Introduction

Nowadays, computer viruses are not anymore a hobby of curious students or smart hackers like they were more than 30 years ago. The viruses, or in a more general term, *malwares* come from simple mass variants which have no specific targets and try to propagate as much as possible, to very sophisticated variants having specific targets and trying to be stealth as much as possible. Such sophisticated malwares are written and used by organized crime groups and armed forces to steal private information or to control remotely the information infrastructure of their specific targets. The purposes of crimes may be just money, but the armed forces aim to prevail over oppositions in cyberwars.

For simple mass malwares, the reports of anti-virus companies [52, 109, 126, 154] confirm that they receive about 100.000 new samples per day, in 2014. Fortunately, according to these reports, most of these samples can be classified into a much smaller number of families, each contains samples of similar identification. That is the reason why the signature matching methods in current detection engines [125] still work in detecting such kind of malwares. However, the reports warn that malware authors start spending time to perfect their products. Consequently, we have little by little hope to see the situation where malware authors do not learn from failures of their products [31].

Serving much more difficult and specific purposes, the targeted malwares are also more sophisticated, both in size and functionality. Some recent reports about state sponsored/organized crime groups malwares (e.g. Stuxnet, Flame, Gauss, Zeus, Citadel, Reign [60, 61, 73, 82, 93, 94, 131, 132, 133]) confirm that such malicious codes can stay invisible for very long time under marketed antivirus softwares. Most of them can be recognized only under some very specific technical analysis which requires lots of manual intervention, others are in fact invisible until they are detected by chance [61].

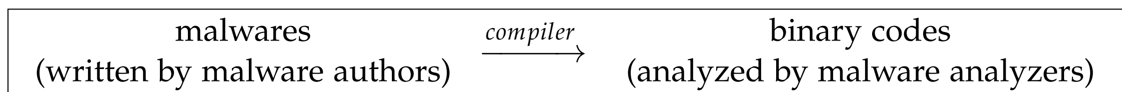
The defense methods from malware threats can be classified into two main classes: broader and deeper defense. Roughly speaking, the former tries to prevent the infection of malwares as soon as possible by extending the detection to many channels (e.g. Email, Web, USB drive, etc.) on which the computer can be infected. The latter tries

to recognize the existence of malwares even when they have landed on the computer and may be equipped of stealth techniques. Both of them are deployed to construct layered malware protection systems [112].

1.1 Malware detection and challenges

Such a layered architecture is necessary for designing protection systems, but it is not sufficient. That is because of there are always vulnerabilities which malwaress can penetrate into systems. Here, we do not discuss the vulnerabilities of the protected system, indeed we simply mention the mistakes (i.e. vulnerabilities) of the protecting system itself: it cannot detect the potential malicious codes.

To get a brief view of the difficulties that a malware detection system have to deal with, we may think of the following schema which illustrates the asymmetricity[1] between malwares and the protection against malwares



Under the viewpoint of malware authors, malwares are just programs existing under the source code forms. Today, malwares are written normally in some high-level programming languages, under the viewpoint of malware analyzers, malwares are programs existing under the binary code form.

Here the compiler, which transforms source codes to binary codes, plays also as an *obfuscator*: it removes all abstract structures (e.g. data types, module structures, function call flows, structured non-local control flows, etc) which exist only in high-level programming languages. These abstract structures are essential ingredients help understanding the semantics of the program.

Also in this schema, we implicitly omit the definition "what is a malware?". That is because of *a malware is just a program*, and this program is malicious or not, does not depend on the program itself, this depends instead on how and where it is used.

1.1.1 Malicious code detection

In this schema, there is mostly no problem with malware writers because, say, *malwares are just programs*. But malware analyzers have to do reversely, and the situation becomes very different. Given a program under binary form, ideally the analyzers first should understand what the program does, namely understand the semantics of the program, then may give a report about its suspicious actions (if exist).

[1] Another asymmetric view between malware writers and malware analyzers is presented in [165]. Here the authors are inspired from the asymmetricity between the encryption and the decryption in asymmetric cryptography.

Impossibility result Theoretically, the first requirement is impossible in general because of the Rice's theorem: *verifying whether an arbitrary program has some nontrivial property is undecidable* [134]. For example, in general, it is not possible to state that a sample will behave like a known malware or not, or it is not possible to state that a program will ask for some input from Internet.

Example 1.1.1. Suppose that there exists an "anti-malware" AntiMal which can always state correctly whether a program sends stolen private information of users to Internet, then the following program M will use (the supposed existence of) AntiMal itself to deny the existence of AntiMal.

```
M()
{
  if (AntiMal(M) == yes) does_nothing();
  else send_stolen_data();
}
```

Listing 1.1: Counter example of AntiM

Here we get a contradiction about which answer the anti-malware AntilMal states about M. Indeed,

- if $AntiMal(M) == yes$, namely $AntilMal$ states that M will send stolen data, but from the code of M above then M will not send stolen data (it does nothing),
- if $AntiMal(M) == no$, namely $AntiMal$ states that M will not send stolen data, but from the code, then M will send stolen data.

Remark 1.1.1. The construction of M is self-reference but well-defined because of Kleene's recursion theorem [149]. The contradiction above uses the self-reference argument which is a popular techniques in proofs about the undecidability.

Malicious codes detection in practice

However, it is not because the problem is undecidable that we should simply forget about it. In practice, researchers develop some techniques which solve partially these issues. The clearest example is the existence of antivirus softwares which are intended to detect known malwares in new samples.

Concretely, since the notion of "malicious programs" is too general and can be only loosely defined, people try to characterize programs by some specific features (e.g. data types [100, 102], control flow graphs [14], library/system call graphs [90], self-modifying codes, suspicious encrypt/decrypt functions [32], network communication [105]). All of them are extracted by analyzing binary codes or by observing the execution of binary codes in some protected environment.

Whenever such features are revealed, the malware detector will verify if they satisfy some pre-determined signatures. The exact signature matching technique, though can be easily bypassed, is still popularly deployed [125, 155]. Recent signatures use

code templates [35], program behaviors [9, 65, 90] together with model-checking [9, 152], or the morphology of control flow graphs [14, 15].

Other techniques There is an important class of program characterizing techniques employs advances from *machine-learning* [91, 135]. Mining on large databases (about thousands) of programs with some specific features (e.g. system calls imported in the program and their arguments [135], n-gram representation of binary codes [91]), the machine-learning based techniques aim to, first clustering the database into different classes which separate malicious from benign programs, as well as separate different families of malwares. Next, classifying an unknown program into some clustered classes.

1.1.2 Challenges

Malwares writers and security researches have been proposing many methods to bypass both the code analysis and the feature verification. These methods are commonly called *obfuscation* [38]. Obviously, malwares can take benefit from obfuscation methods to protect themselves.

Obfuscation against code analysis

This kind of obfuscation ranges from *code hiding* to *semantics hiding*. Roughly speaking, the former consists of techniques that conceal the actual codes of malwares, whereas the latter consists of techniques that conceal the actual behaviors of malwares.

Malwares use *code hiding* techniques to prevent their codes from being analyzed. Beside techniques preventing *dynamic analysis*: anti-debugging, anti-dumping, etc. There are techniques preventing *static analysis*, one of the most popular trick is to use *code packers* [164]. That means the actual codes are packed when the malware is not activated, and they are unpacked to execute only when the malware is activated. Several reports [6, 20, 74] show that there are about $60 - 90\%$ malicious samples are packed using different packers.

The techniques of *semantics hiding* are little more formal, they prevent analyzers from understanding real behaviors of programs, even when the (binary) codes are available. One of the first proposed technique is to use *opaque predicates*, that are conditions in the program deciding the execution flow the program should follow. Malware writers can use a condition which is very hard to evaluate precisely. An example is the NP-opaque predicate [103, 117, 160] given in the following example.

Example 1.1.2. The malware M implements a 3-SAT formula $E(x_1, x_1, \ldots, x_n)$ of n variables that is not satisfiable for all values of x_i(s), and let it be the condition deciding whether M activates malicious behaviors.

```
M()
{
```

```
  set of boolean variables X = {x_1, x_2, ..., x_n};
  unsatisfiable 3-SAT instance E(X);
  if (E(X)) does_nothing();
  else does_malicious_things();
}
```

Listing 1.2: Hiding malicious behaviors using a 3SAT-opaque predicate

From the code in Listing 1.2, M does always malicious things since the formula $E(X)$ is always unsatisfiable. However, any *static analyzer* (cf. Remark 1.1.2) must solve $E(X)$ to understand the possible behaviors of M, this problem is known in NP-complete in general [43].

Remark 1.1.2. In the case of obfuscation using opaque predicates, if the analyzer let the program M execute, then it knows immediately that M exhibits malicious behaviors, such an analyzer is dynamic. On the contrary, a static analyzer does not execute the program M.

A more sophisticated technique is to use *cryptographic triggers* or the *identity based encryption* [99, 139, 145, 162, 165]. That means malicious behaviors will be triggered or not depending on the environment of the infected host. Such a malware is environment-dependent, and in fact in some environment the program is not malicious. Since the triggered conditions of the program are protected by secure cryptographic algorithms, it is very hard to determine in which environment the program activates malicious behaviors.

Example 1.1.3. The program M exhibits malicious behaviors or not depending on the value of some specific registry key. This value depends on the host where M locates, consequently on some hosts M is indeed benign. If the `hash` function is secure, then it is very hard to know the registry key value that makes M exhibit malicious behaviors.

```
M()
{
  if (hash(specific_registry) != stored_hash) does_nothing();
  else does_malicious_things();
}
```

Listing 1.3: Environment-dependent malware

It is worth noting that the obfuscation techniques discussed above are not only theoretical propositions, some of them have been deployed already in real world malwares. For example, the malware Stuxnet will check the registry value `NTVDM Trace` of a specific registry key[1] to decide whether it should infect the host or not [61], another example is the malware Gauss will look for some special file names inside the folder `%PROGRAMFILES%` and use them as keys to decrypt and active malicious payloads [73]. Both of them are well-known state sponsored malwares which are just revealed recently.

[1] `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\MS-DOS Emulation`

Obfuscation against feature verification

In Section 1.1.1, we have discussed that the practical detection verifies the existence of some pre-determined signatures constructed from specific features (e.g. codes, control flow graph, system calls, etc). Then, even when the code analysis is success and required features are collected, there is another problem about the *reliability of signatures*: the program is actually malicious but the signature-based verification concludes that the program is benign, or vice-versa [34].

Example 1.1.4. The pieces of codes in Listings 1.4 and 1.5 are given in [156] to illustrate the metamorphic obfuscation technique of the `W95/Regswap` virus. They are extracted from two different variant of this virus, both actually do the same thing but the registers usage has been substituted to bypass the syntactic pattern detection.

```
pop    edx
mov    edi,0004h
mov    esi,ebp
mov    eax,000Ch
add    edx,0088h
mov    ebx,[edx]
mov    [esi+eax*4+00001118],ebx
```

Listing 1.4: `W95/Regswap`

```
pop    eax
mov    ebx,0004h
mov    edx,ebp
mov    edi,000Ch
add    eax,0088h
mov    esi,[eax]
mov    [edx+edi*4+00001118],esi
```

Listing 1.5: A variant of `W95/Regswap`

To exploit this drawback of the feature verification, malware writers will use some *obfuscating transformation* [40] to transform a malware (which may be already detected) to a new variant which has the same behaviors (we then say that the transformation is semantics preserving). This new variant has different features from the original one, and become undetected again. Metamorphic codes, which are used widely in the community of malware writers [156], are concrete examples of such a obfuscating transformation.

Summary We have introduced briefly the current malware threats, their challenges and the detection approaches. The impossibility result have denied the existence of a detector that can detect all malicious programs. Considering both the efforts of malware writers trying to bypass the detection, and the efforts of malware analyzers trying to detect stealth techniques, one may recognize that they raise a *co-evolution* of both sides of the front-line [119]. The thesis focuses on a narrow problem stated below, hoping that it can contribute to the efforts of malware detection.

1.2 Plan and contributions of the thesis

The thesis focuses on a class of malwares, called *bots*, which collaborate together to create a network of bots, named a *botnet*. In analyzing a bot sample, malware analyzers have to face up with a problem that this sample will not exhibit its malicious behaviors if it does not receive some special messages from other bots.

Contributions In this thesis, we observe that this problem can be solved using ideas of two different research problems. The first one is *code coverage* [22, 118], and the second one is *input message format extraction* [26, 28, 45, 101, 161, 163]. Concretely, in the context of malicious codes detection, the thesis gives two main contributions.

- First, by exploring different execution traces of a program when it consumes different input values, we give an efficient method for the problem of *code coverage*. This contribution aims to activate dormant functions of malwares in a botnet, even when the malwares are isolated from their network. Here, it is worth noting that such malwares only exhibit their malicious behaviors only when they receive some specific input messages.

  Obviously, a more simpler approach is to analyse a bot sample while letting it to communicate with other bots. However, this approach does not work if the botnet is isolated or corrupted. Additionally, we may disclose to other bots that this sample is under examined. Consequently the botnet may detect that it is under surveillance, and it may deploy resistance measures.

  Another approach is to let the bot sample to execute in a simulated environment (e.g. in a lab simulating Internet). This approach requires network and computer equipment. The contribution gives an alternative lightweight approach.

- Second, by classifying messages by their corresponding execution traces, we give a total new method for the problem of *message format extraction*. One may recognize that understanding the format of input messages will reinforce the dormant functions activation, which is the aim of the first contribution. So the two contribution are not independent of each other.

  Moreover, the message format can be used also as a kind of *detection signature*: two programs have strong relation if their consumed input messages have the same format. Here, one can recognize that we have followed the "you are what you read" slogan.

Plan The techniques used in this thesis employ directly *the semantics of programs*; we do not consider statistical approaches (e.g. machine learning). Concretely, the plan of the thesis is as follows:

We present in Chapter 1 the state of the art of malware creation/detection co-evolution. By giving a schema about the viewpoint of malware authors versus malware analyzers, we show challenges to the malicious code detection, and we suggest that these challenges rise mostly from the asymmetry between two viewpoints.

In Chapter 2, we discuss the message format extraction problem and its relation with the malware detection. We present in detail some limits of static and dynamic analysis in dealing with botnets, and next present a new approach that comes from software testing. The application of this new approach in activating dormant functions of malwares leads to an approach for the problem of *input message format extraction*. Next, we review the state of the art of this problem, then introduce some formal analysis. These analysis lead to a formal notion of message format, called message classification. Finally, we state a concrete interpretation for this formal notion.

In Chapter 3, we set up the mathematical framework for the thesis. This chapter begins by recalling some (as well as introducing new) notions and concepts about graphs and labeled transition systems (abbr. LTS(s)). The next section is the main part of this chapter, we develop new results about the equivalence of states inside a LTS, and consequently about how to construct a more compact LTS as a *predictive model* of an original LTS. Not only showing the existence of this predictive model, in the last section, we present algorithms constructing the model.

In Chapter 4, we present in detail a concrete interpretation for the abstract model given in Chapter 3. This interpretation shows how the shape of the input message can be recognized by classifying different execution traces, obtained by the execution of the program in processing different values of the input message. To map this concrete interpretation into the abstract model of LTS, we use a procedure called *stepwise abstraction*. At the final section of the chapter, we present a practical implementation and experimental results on real-world programs; and show the theoretical and practical limits of our method.

In Chapter 5, we present a formal model and constructive results about the behavioral obfuscation. It supports our arguments discussed above about the obfuscation against pattern verification. There are currently no work studying how malwares can bypass the behavioral detection, whereas many work describes how behavioral detection can successfully detect malwares. Though there may exist some reasons to believe that behavioral detection is immune from obfuscation (e.g. such a pattern is obtained from dynamic analysis, and (or) it characterizes behaviors of the program at a high-level, etc.), we show it can be obfuscated.

In Chapter 6, we summary the methodology used throughout the thesis, mostly about the subjects of Chapters 2 and 4, and how it is located in the general context of reverse engineering. We discuss also the perspective of using message classification, or concretely the final LTS resulted from the stepwise abstraction procedure, as a pattern of a program. We propose finally some directions to improve the current limits of our practical implementation.

Publications The work in Chapters 2 to 4 has been partially published in [19] as a full paper, in [17] as a short paper, and in [18] as a poster. The work in Chapter 5 has been partially published in [127] as a full paper.

Chapter 2

On malware communications

In Section 2.1, we introduce briefly the communication between malwares in botnets, and explain why malware detection has relation with the *input message format extraction* problem. In Section 2.2, we discuss some limits of classic analysis methods in dealing with malwares, and introduce the current analysis trend. In Section 2.3, we first review current approaches in reversing the sending/receiving message formats of unknown binary codes, and show some of their drawbacks. Next, we discuss how to give a formal definition of the message format and present our own point of view. Finally, relying on this point of view, we state our approach for the input message extraction problem.

2.1 Introduction

Historically, the input message format extraction tracks back to problems of *protocol reverse engineering* - the method aiming first to reproduce the technical specification of proprietary protocols, then this specification will be used to implement compatible applications. One of the most well-known example may be the reversing of the proprietary protocol SMB realized by Andrew Tridgell, and it takes him about 12 years to do that [158]. However, the principal ideas of his method are still found in the modern methods [26, 28, 41, 45, 163]. Beside the original motivation of understanding proprietary protocols, there is another one which arises from recent research dealing with modern malwares.

New malwares and new challenges There is a new generation of malware in which each of them does not operate alone, instead they collaborate and form a network of "peer-to-peer" malwares that operates under the command of several masters. This kind of architecture is called *botnet* where each malware is called *bot* and masters are called *bot-masters*. We can mention here at length various botnets (though such a list will be obsoleted quickly), notable examples recently are Mega-D, Zeus, Citadel, BetaBot, etc. Some of them are even developed under the open source model (e.g.

ZeuS, Citadel) and that accelerates hugely the development of new variants and samples (e.g. Gameover ZeuS is a bot developed from ZeuS).

To understand behaviors of a bot (i.e. a malware of a botnet), an approach is to replay the bot sample in a safe environment. But a challenge is that *the sample may not be fully activated without the communication with other bots in their network*. The malware needs sending/receiving messages which may include data and (or) control commands, and some of its functions are activated only when particular messages are received. The malwares having such a characteristic are classified into a more general class: *trigger-based malwares* [22] where each activates its malicious behaviors only if some conditions in the execution environment are satisfied.

To deal with this challenge, the analyzers need to activate the dormant malicious functions inside the codes of the sample. Moreover this activation must be coherent with the input of the sample, namely the dormant functions must be activated by some input values. We observe that by understanding the *input message format* that consists of the structure and the semantics of messages, the dormant code activation can be treated.

2.2 Automated malware analysis

The analysis of malwares focus by default on the binary codes: malwares samples exist always in the binary form except very rare cases where their sources are disclosed[1]. The traditional malwares analysis approaches can be classified into either static or dynamic. In static analysis, the binary codes of malwares are analyzed but the malwares are not really activated. Whereas in dynamic analysis, malwares are activated in a protected environment (e.g. sandboxes, virtual machines).

Concerning malicious codes in general, the survey in [57] shows the limits of static analysis, that come from the incapacity in handling obfuscated codes (namely almost all malwares). For the trigger-based malwares and botnets in particular, the researches in [22, 118] show also the limits of dynamic analysis: one cannot analysis malicious codes of a malware if they are not activated. Below, we discuss in little more detail about these limits.

2.2.1 Limits of the static and the dynamic analysis

Static analysis Regardless of many advances and a long history of research, the current techniques of *static analysis* are not scalable enough for obfuscated codes, namely for most of malicious codes: this approach is limited in both practical and theoretical aspects.

First, most of current malwares (about $75 - 90\%$) [6, 20, 74, 136] are protected by some packers (cf. Remark 2.2.1) but there is no method that can handle packed codes

[1] Even in these cases, the binary forms of malwares always occur first, their sources are public much later when they have largely propagated and been recognized.

directly: they need to be unpacked first. *Second*, the static unpacking needs always a correct identification of packing algorithms, again there is no result in this direction, current results are limited to distinguish packing codes from packed codes [42, 50, 53].

Third, even when the codes can be unpacked successfully, there are lots of obfuscation tricks counteracting the *abstract interpretation* based analysis by making it incomplete (cf. Note 2.2.1), for example the opaque predicates [46, 49]. Finally, in a more general context, the situation is even worse because there are constructive obfuscation methods making the complexity of the static analysis become NP-complete [103, 117], even on unpacked codes.

Remark 2.2.1. Code packing is a technique to generate gradually, at running time, the codes of the program [74]. At loading time, the program is in its packed form consisting of a small piece of codes, called the bootstrap codes, and of packed data. The bootstrap codes will, at running time, unpack (e.g. decompress, decrypt) the packed data into executable codes which, in turn, will be executed.

Example 2.2.1. The piece of codes in Listing 2.1 is extracted from a sample of the Slackbot backdoor (cf. also Figure 2.1)[1]. This malware is packed using a customized version of the UPX packer [124], and this piece of codes is the malware's bootstrap codes which restore the original codes from the packed codes.

```
...
0x408760    pushad
0x408761    mov esi,0x4070ae                    /* packed codes sec.   */
0x408766    lea edi,dword ptr [esi+0xffff9f52] /* unpacked codes sec. */
0x40876c    push edi
0x40876d    or ebp,0xffffffff
0x408770    jmp 0x408782
...
0x408778    mov al,byte ptr [esi]               /* copy and unpack     */
0x40877a    inc esi
0x40877b    mov byte ptr [edi],al
0x40877d    inc edi
...
0x4088ae    popad
0x4088af    jmp 0x4011cb                        /* jump to the OEP     */
...
```

Listing 2.1: Bootstrap codes of Slackbot backdoor

The packed codes (i.e. data) are located in the region starting at the address `0x4070ae`, data are first copied into the region starting at the address `0x401000` and then are unpacked here. When the unpacking procedure finishes, the malware jumps to its original entry point at `0x4011cb`, which is now in the unpacked code section. It is worth noting that the packed codes are not executable codes, disassembling them

[1] The analyzed backdoor sample has MD5 hash `365e5df06d50faa4a1229cdcef0ea9bf`, downloaded from `https://malwr.com`.

directly will fail or give nonsense instructions, they instead should simply be considered as data.

```
00408760  60                 pushad
00408761  BE AE 70 40 00     mov esi,slackbot.4070AE
00408766  8D BE 52 9F FF FF  lea edi,dword ptr ds:[esi-60AE]
0040876C  57                 push edi
0040876D  83 CD FF           or ebp,FFFFFFFF
00408770  EB 10              jmp slackbot.408782
00408772  90                 nop
00408773  90                 nop
00408774  90                 nop
00408775  90                 nop
00408776  90                 nop
00408777  90                 nop
00408778  8A 06              mov al,byte ptr ds:[esi]
0040877A  46                 inc esi
0040877B  88 07              mov byte ptr ds:[edi],al
0040877D  47                 inc edi
0040877E  01 DB              add ebx,ebx
00408780  75 07              jnz slackbot.408789
00408782  8B 1E              mov ebx,dword ptr ds:[esi]
```

Figure 2.1: Bootstrap codes of Slackbot

Note 2.2.1. [1] Abstract interpretation (abbr. AI) [44] is one of the major tools for static analysis. An AI procedure is *complete* if the abstract semantics of the program (i.e. $\texttt{Abs}(P)$) is also the one obtained from the concrete semantics (i.e. $\texttt{Con}(P)$) through the abstraction map (i.e. $\alpha(\texttt{Con}(P)) = \texttt{Abs}(P)$).

Normally an AI procedure should be *sound*, namely it requires at least $\alpha(\texttt{Con}(P)) \leq \texttt{Abs}(P)$, but not always complete. However, the soundness only is sometimes not enough (it needs to be also complete), for example in solving the opaque predicates by the AI, the over-approximated results are indeed useless, they cannot specify exactly which branch should the execution follow.

Dynamic analysis This approach will execute and then examine codes in some protected environment (e.g. sandboxes, virtual machines). Because the codes are actually executed, this approach is immune from some obfuscation techniques which impede the static analysis (e.g. code packing, opaque predicate). A typical case is the current best results about code unpacking come from the dynamic approach [85, 138]: they can restore the original codes without taking care of the algorithms used in the packers.

But dynamic analysis can be impeded also, and one of the most popular techniques is anti-debugging [62, 146, 155]. In practical dynamic analysis, malwares are executed in some protected environment (e.g. debuggers, sandboxes), so the malware will try to detect whether it is executed under such an environment, and stops executing if it is the case. Another anti-debugging trick is shown in the example below, the

[1] We use *notes* and *remarks* to give supplemental information. There is no strict different between a note and a remark, but a note is normally used to give more important information.

malware sample will execute its actual codes in another thread and lets the main thread suspended forever.

Example 2.2.2. The pieces of codes in Listings 2.2 and 2.3 are extracted from a sample of a Trojan-Downloader malware[1]. It uses multiple threads to defeat the debugger. Starting from the main thread, the malware creates another thread (cf. the instruction at `0x402335`), and then actives this thread (cf. the instruction at `0x402473`) by calling the Windows™API `QueueUserAPC` whose address is stored in `ebx` (cf. the instruction at `0x402443`).

```
0x402423  call ds:IsDebuggerPresent
0x402429  xor esi,esi
...
0x402335  call ds:CreateThread
0x40243B  mov edi,eax
...
0x402443  mov ebx,ds:QueueUserAPC
...
0x402458  mov ebp,ds:SleepEx
...
0x40236c  push esi
0x40246d  push edi
0x40246e  push 0x401870
0x402473  call ebx
...
0x40247c  push 1
0x40247e  push 0xffffffff
0x402480  call ebp
...
```

Listing 2.2: Trojan's main thread

```
0x401870  push ebp
0x401871  mov ebp,esp
0x401873  and esp,0fffffff8h
0x401876  sub esp,0xc04
0x40187c  push ebx
0x40187d  push esi
0x40187e  push edi
0x40187f  nop
0x401880  push eax
0x401881  pop eax
0x401882  inc ecx
0x401883  dec ecx
0x401884  push ebx
0x401885  pop ebx
0x401886  nop
0x401887  sar edi,cl
0x401889  adc al,0xd8h
0x40188b  jbe 0x4018f6
...
```

Listing 2.3: Second thread

The important codes executed by the second thread are started at `0x401870` (cf. the instruction at `0x40246e`), and is shown in Listing 2.3. The main thread will end by calling the Windows™API `SleepEx` (see the instruction at `0x402480`) which will wait forever because the passed argument is `0xffffffff` (see the instruction at `0x40247e`). Consequently a debugger tracking the main thread will be blocked forever here,

[1]The analyzed sample has MD5 hash `7fd2f990cd4f2ed0d1ca12790f51ea2c` and is obtained from `http://vxheaven.org`. Several technical analysis about anti-debugging techniques used by this sample are given, but we are not agree with their conclusions. For example, the analysis in [107] says that the API `IsDebuggerPresent`, `SleepEx`, `GetTickCount`, `GetCurrentProcessId` are used to detect whether the malware is traced by a debugger, but this conclusion is misleading in this case. Because of some programming bugs in the sample, these API(s) do not help anti-debugging.

whereas the second thread continues executing.

We can observe also that, in this case, malware authors have inserted junks in these codes (e.g. `push` and `pop` consecutively `eax`, increase and decrease consequently `ecx`, etc.), may be with an intention to make confused analyzers.

An inherent limit of the dynamic analysis is that it is incomplete. Since only the executed codes are examined, the dynamic approach does not give any information about the inactivated codes. This limit makes dynamic analysis seriously incompetent in analyzing malwares of botnets, because a malware sample may activate its malicious behaviors only when it receives some particular messages from bot masters.

Summary On one hand, static analysis is *complete but imprecise* in analyzing obfuscated codes; namely it will cover all possible behaviors of the examined program, but it is not scalable enough for the obfuscated codes. On the other hand, dynamic analysis, say, if we accept that the anti-debugging techniques are not "essential", then it is *precise but incomplete*; namely it gives precise information for executed codes, but it cannot give any information for dormant codes. These limits of static and dynamic analysis make these approaches insufficient in analyzing bots, which consist of both obfuscated and dormant codes.

2.2.2 Approach from software testing and code coverage

There is a new approach proposed recently by some pioneer researchers, e.g. A. Moser et al. [118], D. Brumley et al. [22, 23] and J. Caballero et al. [26]. This approach combines several recent advances coming from automated software testing [29, 69, 70] and automated theorem solvers [71].

Initially, automated testing works on source code because it has been essentially designed to detect software bugs, as a phase in software engineering. The testing procedure aims at exploring all execution traces of a program: all functions of the program should be tested, thus need to be activated by some input. Any trace constraint (i.e. the required conditions so that the program with follow this trace) is encoded as a conjunction of control conditions in the code, by *symbolic execution* [88], the control conditions are resolved, the solutions consist of input values satisfying the conjunction.

As previously discussed, incompleteness is an inherent disadvantage of dynamic analysis. But the *concolic execution* method from the automated software testing can help improving this. It is straightforward to see that one (i.e. concolic execution) complements the other (i.e. dynamic analysis) in activating dormant codes of malwares, except that concolic execution works on source code.

Some authors (e.g. Brumley et al [22]) have first lifted binary codes to a high-level representation, named *immediate representation* (abbr. IR) codes [23], and then apply the concolic execution on IR codes. This is also the current trend in the automated binary codes analysis, for that we refer to the discussions in [7, 151].

Remark 2.2.2. Between dynamic analysis and automated software testing, there is an approach called *fuzz testing* [153], which has a rich history in software vulnerability research, and is still very popular and effective. Very roughly speaking fuzz testing is automated software testing without automated theorem solvers. It generates automatically random inputs for testing.

Note 2.2.2. It is worth noting that symbolic execution has been introduced by J. C. King [88] since 1976, but it is not quite scalable in practice because the complexity of the total symbolic conjunction formulae. Until the work of P. Godefroid et al. [69] where the concepts of "concolic execution" are introduced at the first time: this is a combination of total symbolic execution and the concrete random testing. Whenever a condition is not resolvable by any reasoning theory implemented in the SMT solver, it is replaced by a concrete value calculated from the current program state. In the general case, the concolic execution is still not complete, but in practice, it is much better that symbolic execution.

2.3 Message format extraction

The input message format extraction is just one among many problems of the reverse engineering. In *automated protocol reverse engineering*, researchers face up with the problem of understanding the structure and the semantics (i.e. meaning) of messages received by a program: to recognized the states of the protocol state machine, the semantics of messages must be well determined. The largely agreed conclusion is that the message format extraction should be the first and crucial step of the protocol reverse engineering [26, 28, 41, 45].

In the context of the automated botnet analysis, the message format extraction together with the traditional dynamic analysis, can help understanding functions of bot as well as activate the dormant functions. That is because once the format of inputs are recognized, the functions can be activated by changing systematically the input value.

Obviously, this process is also the method of the automated software testing as discussed above, except that the input format extraction is not a problem. The more ambitious goal in the automated botnet analysis, similar with the automated protocol reverse engineering, is to understand completely the communication protocol between bots. However, this thesis is a first step towards a protocol analysis, and we will focus on message analysis, more strictly *we will consider only the input message extraction in this thesis*.

Below, we first give a review for the current methods used in the automated message format extraction researches. This review, aside from serving as a brief discussion for the state of the art, gives an intuition about the format of the message. Here we want to emphasize that *there is not yet a unique understanding about the message format*, researchers give different ideas about which properties should be considered as

the structure of a message, and though these ideas are sharpened over time, the current methods are still more or less experimental. We discuss next some drawbacks of these methods, and finally, state some objective ideas about the current methods, then give briefly our approach.

2.3.1 Current research

As discussed at the beginning of Section 2.1, the initial research is interested in understanding and reimplementing proprietary protocols (e.g. [66, Fritzler et al. with AIM/OSCAR protocol], and [158, Tridgell et al. with SAMBA protocol]). These work are done mainly by manual effort, and this approach is still effective today for sophisticated protocols (e.g. the reverse engineering of Skype [12, 110] or Dropbox [87]), though it is also a very time consuming task.

The number of new malwares samples per day is more than 100.000 in 2014, as confirmed by security reports of anti-virus companies [109, 126, 154]. Only a small part of them is bots, and even a smaller part of these bots needs some special manual treatment [52], but the number is still huge. Because of this massive production of new samples, the manual approach becomes sooner or later insufficient in confronting with malwares. So the interest of this thesis is the *automated approach*, which aims at an “once and for all” treatment.

The automated message format extraction, or more generally the automated protocol reverse engineering, has a relatively short history in comparison with the reverse code engineering. The first work is given in [45], and the currently most remarkable results may be ones given in [26, 27]. This automated approach is currently in its infancy, the obtained results are not yet as accurate as the manual approach in complicated cases. It techniques can be criticized in many ways, and seriously speaking, it is not yet completely automated. Nevertheless only these methods can keep up with the massive production of new malwares in the future.

Current point of views about message formats In the first part of this section, we will present current methods in automated message format extraction. The common point of them is that the authors consider intuitively that the format of messages should consists of *fields* and *separators* between fields (cf. Example 2.3.1), etc. as well as a *hierarchical structure* where a message may wrap other messages of lower layer protocols.

Example 2.3.1. The HTTP response message in Figure 2.2 consists of fields and separators as follows

$$\underbrace{\texttt{HTTP/1.0 301...}}_{field_1}\ \overbrace{\texttt{\textbackslash r\textbackslash n}}^{separator}\ \underbrace{\texttt{Location...}}_{field_2}$$

The difference is in their technical treatments in recognizing these elements. The current techniques can be classified into two main direction:

```
48 54 54 50 2F 31 2E 30 20 33 30 31 20 4D 6F 76 HTTP/1.0 301 Mov
65 64 20 50 65 72 6D 61 6E 65 6E 74 6C 79 0D 0A ed Permanently..
4C 6F 63 61 74 69 6F 6E 3A 20 68 74 74 70 3A 2F Location: http:/
```

Figure 2.2: A HTTP response message

1. mining a set of exchanged messages, and

2. tracing how the program processes a message.

Below, we will review the main ideas of each direction.

Message mining

The first direction, pioneered mostly by W. Cui et al. [45] and G. Wondracek et al. [163], aims at extracting the information from network messages. Since the technical methods used in that work are quite similar, we review, for example, the method used in [45].

Basically, they consider that a raw message consists of fields, which can be *text* (i.e. ASCII printable characters) or *binary* data: at the low-level they are called *tokens*. To find *text tokens*, they first find *text segments* by considering consequences of ASCII characters that are separated by binary bytes. Inside each text segments, the text tokens are detected by considering sequence of bytes separated by *delimiters* that are space or tab characters. To find *binary tokens*, the procedure above is not applicable because delimiters for binary tokens are very hard to recognize, so they simply consider *each binary byte as a binary token*.

They propose the *token pattern* as a upper-level representation of the raw message, such a pattern has form

$$(\text{direction}, \text{type of token}, \text{type of token}, \dots)$$

where the direction specifies that the message goes from client to server or reversely. To classify analogous messages into a pattern, these messages are compared using a *byte-wise alignment procedure* based on the Needleman-Wunsch algorithm [120].

The hierarchical structure of messages is also considered. Here, that means a message of some protocol may wrap another message of a lower protocol. They assume that the wrapping can be recognized by a special type of token, called *format distinguisher*. To detect a format distinguisher token, the messages are examined to find out a common unique value lower than some threshold (which is experimentally calculated). Moreover such a value is a token only if, recursively, the sub-message (i.e. the wrapped message) is also a message.

Binary code leveraging

The second direction consists of techniques leveraging the binary program receiving messages. The first treatment is pioneered by J. Caballero et al. [28], using dynamic

analysis, the authors extract information by observing how the program processes the message. This original method is reused and improved to handle messages with hierarchical architecture in [26, 101].

Because the appeared techniques do not work with encrypted messages, the authors in [161] identify the decrypted phase from the normal message parsing phase, that allows applying next the method having been developed in [26].

Since all sequent research apply the original ideas in [28], we review here the techniques used in this work. The authors first consider that the message has a (flat) structure as a sequence of *fields*, each is characterized by 5 attributes:

- start position, i.e. the start position of the field in the message,
- length type, i.e. whether the field is of fixed or variable length (possible values: 1. `fixed`, or 2. `variable`),
- boundary, i.e how the field ends[1]. There are the following possible values: 1. `fixed`: the field is of the fixed-length, 2. `direction`: the length of field is determined by another field, 3. `separator`: the field ends by a separator (i.e. some constants marked the boundaries of fields),
- direction, i.e. whether the field contains information about the position of other fields,
- keyword value, i.e. the value of the field (when it is determined as a keyword).

With the flat structure of the message, its fields will be extracted completely if the length of each field and the separators are identified. Since the length of a given field can be determined either by some value hard-coded inside the binary code or from another field of attribute *direction* (i.e. a direction field), then extraction problem is now reduced to identify *direction fields* and *separators*.

The identification of a direction field is realized by a specific tainting procedure, which determines whether this field is used (by the program) in accessing another field. The identification of a separator is realized by determining whether the field is compared again other bytes of another field.

Summary Two directions discussed above, beside the differences in technical treatment, have different perspectives about the message format. The former considers that the similar patterns learned from a set of real communicated messages, will characterize the message format, whereas the latter considers that the different behaviors of program in processing different parts of a message, will bring out the message format.

[1]The *boundary* attribute is different from the *length type*, it determines how the program find the end of the field, in other words the length type is a consequence of the boundary.

More importantly, in the latter direction, the role of the program processing message suggests also that *the format (i.e. semantics) of messages should be given by the program, and not the messages themselves*.

Drawbacks

We can classify the drawbacks of the current techniques into two categories: 1. the difficulties in applying them in the context of malicious code analysis, and 2. the inaccuracies of these techniques.

Application context difficulties The minimal condition for the operation of the *message mining* techniques is that they must be able to capture a large enough set of *real messages* exchanged between the examined malwares and other malwares in the botnet. But with the sink-holed botnets, namely the botnets whose the bot masters have been disrupted or isolated from Internet (cf. Remark 2.3.1), the communication between bots is collapsed until there are no more sending/receiving messages.

In this situation, it is impossible to capture enough messages that are needed for the mining procedure. In other words, it is impossible to study malwares without their living botnet. On the contrary, the *binary code leveraging* techniques work with a single input message, and they even do not require that this is a real message, so they are not affected by the discussed situation.

Remark 2.3.1. In real-world, some botnets can be disrupted using the DNS sink-hole technique [24]. This mitigation does not require fully understanding the communication protocol or the format of exchanged messages, instead some network related measures will be used to redirect the IP addresses of the hosts (mostly the bot masters) to non-existent addresses or to the addresses of some controlled host. These IP addresses are extracted normally from (configuration files, binary codes, etc.) of bots.

However, modern botnets can use several techniques to bypass the DNS sinkhole (e.g. TDSS/TDL4 [95], Kraten [137]). One of the most popular technique is to use Domain Generator Algorithms (abbr. DGA) [3]. Using this technique, the bot will generate randomly many domain names, and try to connect to several domains hoping that some of them are locations of the new masters.

Inaccuracies To criticize the inaccuracies of techniques, it is not surprising to show the cases where they do not work yet (even in their determined problem scope) so we only give several observations.

First, as pointed out also in [28, 101], the essential limitation of the *message mining* techniques comes from the lack of semantics information in the network messages. That means no matter what the improvement in the mining methods, the avoidance of the programs leads always to the fact that these techniques cannot say anything about how the programs consume the messages. But without the existence of programs sending/receiving messages, the messages themselves are just meaningless data.

For example, rationally an input message extraction technique should give a result which semantically fits in with one given by the program. But we can think of a scenario where a client sends a massive number of HTTP messages to a FTP server. Since the receiver does not understand the protocol, it simply rejects all received messages. In other words, under the server's point of view, these messages are meaningless.

Now in this scenario, we consider a message mining procedure works on these messages. Since it bases on a message analysis and not a programs analysis, it may extract irrelevant regularities (from the program's point of view). And in this case, the procedure may still conclude that the messages contain some structure, and that is obviously a false positive.

Second, the current techniques in *binary code leveraging* depends strictly on some artificial assumptions of how the receiving programs treat the message, but these assumptions are not always satisfied. For example, the detection of a separator follows the assumption that there will be some string constant being compared byte-by-byte (from the first to the last byte of the pattern) in a loop with the message. But this assumption is not warranted where some non-popular string matching algorithms are deployed.

We propose below two real-world examples for which the detection of separator given in [28] does not operate. The first is extracted from an implementation of the Knuth-Morris-Pratt string matching algorithm [89]. The second is extracted from the binary codes of the programs `curl` and `wget`.

Example 2.3.2. The piece of codes[1] in Listing 2.4 describes the operation of the Knuth-Morris-Pratt string matching algorithm [89]. The pattern buffer is located at the addresses determined by the `dword` value at the address `[ebp-0x10]` and the code uses a counter, determined by the `dword` value at the address `[ebp-0x28]`, to specify which byte of the pattern buffer is currently used to compare with the bytes of the target buffer. Similarly, the target buffer is located at the address determined by `[ebp-0x2c]`, the counter of the target is determined by `[ebp-0xc]`.

Each byte of the pattern is extracted by "`movsx eax,byte ptr [ecx+eax]`", whereas the byte of the target is extracted by "`movsx ecx,byte ptr [edx+ecx]`". One is compared with the other by "`cmp eax,ecx`".

```
kmp_loop: mov   al, 0
          cmp   [ebp-0x28], 0xffffffff
          ...
          mov   eax, [ebp-0x28]          /* ebp-0x28: pattern counter */
          mov   ecx, [ebp-0x10]          /* ebp-0x10: pattern buffer  */
          movsx eax, byte ptr [ecx+eax]
          mov   ecx, [ebp-0x2c]          /* target counter            */
          mov   edx, [ebp-0xc]           /* target buffer             */
          movsx ecx, byte ptr [edx+ecx]
          cmp   eax, ecx                 /* compare pattern-target    */
          setnz bl
```

[1]This piece of codes is extracted from the compiled code (by the clang compiler version 3.6). The source code, originally in C, is modified version of the implementation taken from [98].

```
          ...
update:   mov   eax, [ebp-0x28]
          mov   eax, [ebp+eax*4-0x24]   /* partial matching table    */
          mov   [ebp-0x28], eax         /* update pattern counter    */
          jmp   kmp_loop
```

Listing 2.4: String matching using the Knuth-Morris-Pratt algorithm

Until there, everything follows perfectly the assumption. But when we examine how the counter is updated (starting from the label `update`), we see that *the counter of the pattern does not increase sequentially*: it is updated by a *partial matching table* located at the address determined by the `dword` value at the address `[ebp-0x24]` (i.e. there is a pointer dereferencing here). The codes from the label `update` are actually equivalent with `i = table[i]` where `i` is the pattern counter.

More practically, even in the case the separator is a single character, the assumption above is still not warranted. We can see that in the second example.

Example 2.3.3. The code snippet in Listing 2.5 is used by `wget` to search for the line-feed character inside a HTTP message. The message's buffer has its address stored in `eax`, its length stored in `edx`. The register `bl` stores the value `0xa` (which is the ASCII code of line-feed).

The comparison starts from the beginning of the message, each byte is compared with the line-feed character (by the instruction `cmp byte ptr [eax], bl`) until a match is found, and while the number of compared byte is still less than the message length (that verified by the instruction `cmp eax, edx`). This kind of search fits perfectly with the assumption.

```
loop:     cmp byte ptr [eax], bl        /* bl = 0x0a (line-feed)   */
          jz  ln_found                  /* line-feed detected      */
          inc eax
          cmp eax, edx                  /* edx = message's length */
          jb  loop
          ...
ln_found: ...
```

Listing 2.5: Processing of `wget` in searching for the line-feed character

However, also searching for the line-feed character, `curl` uses a faster but more sophisticated searching algorithm as given in Listing 2.6. The message's buffer has its address stored in `edx`, its length stored in `eax`.

Scanning from the beginning of the message, each 4 bytes is xor-ed with `0x0a0a0a0a` (i.e. 4 line-feed characters), the result is used together with two "magic" values `0x7efefeff` and `0x81010100` to verify whether it contains a byte of value `0x00`, and if yes then calculate the position of this byte[1].

```
loop:     mov ecx, dword ptr [edx]
```

[1]The detail description of this fast searching algorithm can be referenced in the file `memchr.asm` of the library `msvcrt` of Microsoft Windows™SDK.

```
          xor ecx, ebx              /* ebx = 0x0a0a0a0a              */
          mov edi, 0x7efefeff
          add edi, ecx              /* edi = ecx - 0x81010101        */
          xor ecx, 0xffffffff
          xor ecx, edi
          add edx, 0x4
          and ecx, 0x81010100
          jnz LN_found
          sub eax, 0x4             /* eax = number of unexamined bytes */
          jnb loop
          ...
LN_found: ...
```

Listing 2.6: Processing of `curl` in searching for the line-feed character

We can observe that in the case of `wget`, the assumption about the comparison of the line-feed constant with bytes of the message is satisfied: one can recognize that there is a specific constant (this is the value `0xa` of line-feed stored in `bl`) that is compared with every bytes of the message. But it is not in the case of `curl`, without a prior knowledge about this fast searching algorithm, one cannot be sure which is the interested constant (this is neither `0x7efefeff` nor `0x7efefeff` and nor `0x81010100`).

Summary The assumptions taken by the techniques discussed above are not deduced naturally from the programs and the input messages, they are instead subjective assumptions of the authors. In some cases, the examined objects (i.e. programs and input messages) satisfy these assumptions, but there will be many cases where they do not. The drawbacks discussed above urge that there should be a new method which tries to avoid as much as possible artificial assumptions.

2.3.2 What is the "message format"?

There is a question that is implicitly avoided until there and, as far as we understand, also in other researches, this is "what is the format of a message?" The rigorous definition of the format is not yet raised as a crucial requirement (though it can lead to controversial conclusions). For the malware writers, the message format specifies a convention between senders and receivers. But this convention is hidden from the malware analyzers who must reconstructing it.

Being motivated by some real-world protocol specifications, the current research consider an intuition that some basic elements, like *fields*, the *separators*, as well as *subfields* (that give a hierarchical structure), etc. constitute the message format. Normally, different authors will give different intuitions about the correlation between these elements, such an intuition is not ready yet to be generalized.

We have claimed a general conception in Section 2.3 that the (input) format extraction is to understand the structure and semantics of messages (received) by a program. Unfortunately such a conception is not clear enough because one can ask

a very similar question "what is the structure and (or) the semantics of messages?"; then obviously we have no meaningful conclusions here. But we propose the following analysis.

Message format as a mathematical model We may look for an analogy by thinking of an input message as a normal input of a program, and the program (as a function) should have some specific but unknown input *data type*. Then, message format extraction has relations with *data type reconstruction* from the binary codes [100, 102]. It is worth noting that data type is in fact a *mathematical model* which can be defined formally, whereas data structure concerns a *concrete implementation* of data type which cannot. So message format extraction can be understood as *searching for mathematical model of the message that fits a concrete implementation*.

But for data type reconstruction, the "searching" principle is at least *well defined* because types are concretely implemented following the convention of the high-level programming language. Then, for example, types can be reconstructed from some *type-sink* (i.e. an execution point in the program where types can be determined immediately, e.g. a return value or arguments of a well-known system call [102]) using a *type propagation* procedure, which is indeed a simple form of the *type unification* procedures used in type inference algorithms [113].

```
func:      push   ebp
           mov    ebp, esp
           lea    eax, [ebp+0x8]          /* eax=ebp+0x8 */
           mov    ecx, [eax]
           add    ecx, [eax+0x4]
           movsx  eax, byte ptr [eax+0x8]
           add    ecx, eax
           mov    eax, ecx
           pop    ebp
           retn
```

Listing 2.7: Simple type propagation

Example 2.3.4. The code snippet in Listing 2.7 is a toy example illustrating the well-defined principles of the type propagation. It describes a function in processing its arguments. Since the value of `eax` is assigned to be `ebp+0x8` by the instruction `lea eax, [ebp+0x8]`, we can recognize that there are 3 arguments located respectively at `[ebp+0x8]`, `[ebp+0xc]` and `[ebp+0x10]` on the stack. Let us consider the instruction

`mov ecx, [eax]`

The technical specification of `mov` says that if the target is a 32-bit register (here it is `ecx`) then the source must be a 32-bit register or a 32-bit memory value, so in the instruction above `[eax]` must be a 32-bit memory value: a *type-sink* is detected. There is also a *type propagation*: the known type of `ecx` in the `mov` instruction allows determining the type of `[eax]`. We have the type of the argument at `[ebp+0x8]` is

`dword`, similarly the type of the argument at `[ebp+0xc]` and at `[0x10]` is respectively `dword` and `byte`.

In Example 2.3.4, we can observe that the type reconstruction can follow the well-defined principles of the type inference. Whereas for the input message format extraction, such principles do not exist: given an input message which is assembled by some specific but unknown convention, the program can use an arbitrary algorithm to disassemble the message.

2.3.3 Our approach: message classification

The previous claim "the message format is a mathematical model of the message that fits a concrete implementation" must be concretized. Let us consider a program P and a set M of inputs of P, we interpret concretely

- the "concrete implementation" as some observable behavior r of P,
- the "mathematical model" (i.e. the message format) as some *equivalence relation* $R \subseteq M \times M$ in the set of messages, and
- the "fitting" means a matching between R and the equivalence up to r

  $$\left(m, m'\right) \in R \Longleftrightarrow P\left(m\right) \approx_r P\left(m'\right)$$

  where $P\left(x\right)$ is the observable behaviors that P exhibits along the execution with the input value x. In other words, if $\left(m, m'\right) \in R$ then the execution of P on input m exhibits the same observable behavior r as the one on input m'.

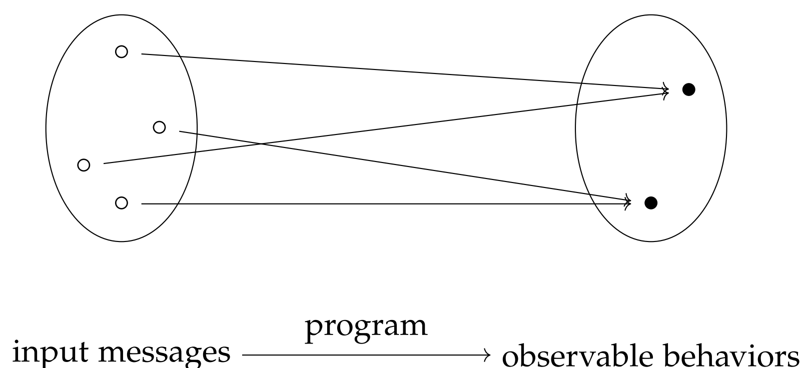


Figure 2.3: From input messages to observable behaviors

Relying on the interpretation above, we observe that the format of an input message is indeed the structure of the set (of all input messages). In other words *reversing the input format is reconstructing the structure of the input set*. Because the program may raise different observable behaviors of sort r in processing different input messages,

the structure of the set of input messages can be understood by studying the set of observable behaviors (cf. Figure 2.3).

Concretely, we state that the structure can be described (or at least approximated) by a *labeled transition system* (abbr. LTS). Given a program receiving input messages, we provide a LTS which will categorize these messages according to their corresponding *execution traces*, as a concrete interpretation of observable behaviors.

Actually, we will show that one retrieves the shape of messages from the LTS. More importantly, we do not make any hypothesis on the input message processing algorithm of the program. Because the shape (i.e. format) of messages is simply a consequence of the categorization, we use the notion of *input message classification* instead of the "input message format extraction". This formalization avoids problematic terms such as "fields", "separators", "keywords", etc. that we met together with the classic notion of "message format".

Summary In the last section of this chapter, we have reviewed the current trends in message format extraction and stated out our approach. This approach relies actually on a different viewpoint of the message format. Here, we try to avoid artifical assumptions and intuitive terms existing in other approaches. In the next chapter, we will develop the mathematical framework of the approach

Chapter 3

Mathematical modeling

In this chapter, we develop the theoretical framework that supports our approach about input message classification. The first part of this chapter is reserved to recall some specific mathematical notation about graphs and labeled transition systems. Those are classic branches of mathematics and computer science. We extract only a small subset as the minimal knowledge needed to present the framework.

In the second part, we develop upon these concepts new theoretical results that will be reused in the next chapter, where the concrete problem is discussed. We also try to give the intuitive but precise understanding of these results in our concrete context, so all given examples are actually motivated from our concrete problems.

In the last part, we give algorithms that constructs some structures given in the second part. These algorithms will be used in the next two chapters, where we give a concrete interpretation and a practical implementation.

The results of this chapter have been presented partially in [19]. This is joint work with Guillaume Bonfante and Jean-Yves Marion.

3.1 Notations and terminology

The concepts introduced below can be found in several standard textbooks and survey papers, for example [75, 77, 114, 140]. We refer to [77, 114] for a systematic representation while to [75, 140] for a short review.

3.1.1 Graphs

Graph theory is a classic branch of mathematics, which has many applications in computer science, the graphical models and the reasoning on them appear almost everywhere. In the following, we review some algebraic notations of graphs and functions between graphs.

Definition 3.1.1 (Directed graph)**.** A directed graph $G(V,E)$ consists of two distinguished finite sets: V of *vertices* and E of *edges*, together with total functions

$s, t \colon E \to V$, which specify respectively the source and the target of any edge.

$$E \underset{t}{\overset{s}{\rightrightarrows}} V$$

The set of edges and vertices of G are denoted by $E(G)$ and $V(G)$, respectively.

Remark 3.1.1. There exist in general multiple edges between a pair of the source and the target vertex. Such a graph is called *multi-graph* to distinguish from a more familiar definition where there is no more than an edge between any pair of vertices. From now on, graphs are understood as multi-graphs without further mentions.

For any vertex $v \in V$, if an edge e satisfies $s(e) = v$ then e is called an *outgoing edge* of v, and if $t(e) = v$ then e is called an *incoming edge* of v. If an edge e is both an outgoing and incoming edge of some vertex v then e is called a *loop-back* at v.

A *path* in the directed graph is a sequence of consecutive edges $p = e_1 e_2 \ldots e_n$ so that $t(e_i) = s(e_{i+1})$ for all $1 \leq i < n$, next depicted as

$$p = v_1 \xrightarrow{e_1} v_2 \xrightarrow{e_2} \ldots \xrightarrow{e_n} v_{n+1}$$

The functions s, t have then natural extensions $s, t \colon P \to V$ on the set P of paths. Given some path p as above then $s(p) = v_1$ and $t(p) = v_{n+1}$ and write $v_1 \xrightarrow{p} v_{n+1}$. We call also v_1 the source and v_{n+1} the target vertex of p.

Definition 3.1.2 (Graph homomorphism)**.** Given graphs G_1, G_2, a graph homomorphism $h \colon G_1 \to G_2$ consists of functions:

$$h_V \colon V(G_1) \to V(G_2) \text{ and } h_E \colon E(G_1) \to E(G_2)$$

satisfying

$$h_V(s(e)) = s(h_E(e)) \text{ and } h_G(t(e)) = t(h_G(e))$$

for any $e \in E(G_1)$, in other words they make the following diagram commutative:

$$\begin{array}{ccccc} V(G_1) & \xleftarrow{s} & E(G_1) & \xrightarrow{t} & V(G_1) \\ \downarrow{h_V} & & \downarrow{h_E} & & \downarrow{h_V} \\ V(G_2) & \xleftarrow[s]{} & E(G_2) & \xrightarrow[t]{} & V(G_2) \end{array}$$

Remark 3.1.2. Given a graph homomorphism $h \colon G_1 \to G_2$, we will omit the subscripts in functions h_V, h_E and write them simply as h when the meaning is clear from the context. Also, a graph homomorphism is called shortly as morphism.

Special graph morphisms For a graph G, let $E(G)|_{v\to}$ denote the set of all outgoing edges of any vertex $v \in V(G)$. Given a (graph homo)morphism $h \colon G_1 \to G_2$ and a

vertex $v \in V(G_1)$, there is a function

$$h|_{v\rightarrow} : \; E(G_1)|_{v\rightarrow} \rightarrow E(G_2)|_{h(v)\rightarrow}$$

that is the natural restriction of $h_E \colon E(G_1) \rightarrow E(G_2)$ on $E(G_1)|_{v\rightarrow} \subseteq E(G_1)$, then:

Definition 3.1.3 (Correct, complete and surjective graph morphism)**.** The morphism $h \colon G_1 \rightarrow G_2$ is called correct (resp. complete) if the restriction $h|_{v\rightarrow}$ is injective (resp. surjective) for all $v \in V(G_1)$. We say also that h is surjective if the function $h_V \colon V(G_1) \rightarrow V(G_2)$ is surjective.

Example 3.1.1. There are some morphisms from G_1 to G_2^i (for $i = 1, \ldots, 4$) and their properties are as follows

- the morphism $h_1 \colon G_1 \rightarrow G_2^1$ mapping $v_i \mapsto v_i^1$ for $i = 1, \ldots, 4$, and $e_1, e_2 \mapsto e_1$ and $e_i \mapsto e_i$ for $i = 3, 4$ is neither correct (since e_1 and e_2 in G_1 have the same image) nor complete (since e_2 in G_2^1 has no inverse image).
- the morphism $h_2 \colon G_1 \rightarrow G_2^2$ mapping $v_i \mapsto v_i^2$ and $e_i \mapsto e_i$ for $i = 1, \ldots, 4$ is correct but not complete (since e_5 in G_2^2 has no inverse image).
- the morphism $h_3 \colon G_1 \rightarrow G_2^3$ mapping $v_i \mapsto v_i^3$ for $i = 1, 2$ and $v_3, v_4 \mapsto v_3^3$ is both correct and complete.

Notice that the morphisms h_1 and h_3 are surjective but h_2 is not (since v_5^2 in G_2^2 has no inverse image).

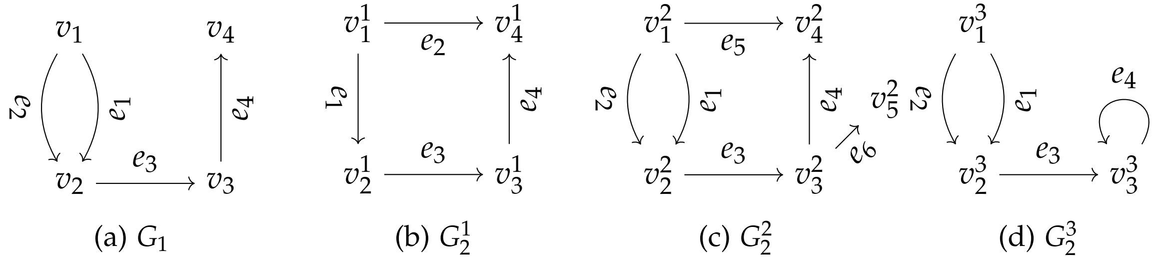


Figure 3.1: Graphs and morphisms

The following result is derived directly from the definition of graph morphism and the definition of correct, complete, surjective graph morphisms.

Proposition 3.1.1. *The composition of two morphisms is a morphism. Moreover, the composition of two complete (resp. correct) morphisms is a complete (resp. correct) morphism.*

3.1.2 Labeled transition systems

The states of a system and transitions between states are important concepts in computation. An abstract model of them is studied in the theory of labeled transition

systems [114], which is a richer model of the graph. In the following, we give notations for a special class of labeled transition systems where each is deterministic and has a special, unique state called the initial.

Definition 3.1.4 (Labeled transition system)**.** A deterministic labeled transition system $\mathscr{A}\langle Q, \iota, \Sigma, \delta \rangle$ consists of

- a finite set Q of states, a special $\iota \in Q$ is qualified as the initial state,
- a finite set Σ of alphabet (i.e. labels),
- a partial function $\delta\colon Q \times \Sigma \to Q$ with respect to Σ, called the transition function.

The set of state, the initial state, the alphabet and the transition function of a LTS $\mathscr{A}$ are denoted respectively by $Q(\mathscr{A})$, $\iota(\mathscr{A})$, $\Sigma(\mathscr{A})$ and $\delta(\mathscr{A})$.

In other words, a labeled transition system (abbr. LTS) is a finite state automaton but without the notion of final states. We have another notion of "ending" states which we call *terminal states*. Given a LTS $\mathscr{A}\langle Q, \iota, \Sigma, \delta \rangle$, a state $q \in Q$ is called *terminal* if there is no letter $c \in \Sigma$ so that $\delta(q, c)$ is defined. There exists always a unique initial state in $\mathscr{A}$ but there may exist multiple (or may not exist any) terminal states.

Remark 3.1.3. The standard definition of labeled transition systems [114] takes no interest in the notion of initial state, such a state is interested rather in the theory of automata [78]. The definition above of LTS(s) is a hybrid, because we are interested in the unique existence of the initial state (which exists always in automata but not in LTS(s)), but we do not keep such an interest for the terminal states.

Reached states and extended transition function For any states $q, q' \in Q$ and a letter $c \in \Sigma$ satisfying $\delta(q, c) = q'$, we say that there is a transition labeled c from q to q', it is denoted by

$$q \xrightarrow{c} q'$$

The LTS $\mathscr{A}$ reads words $w = c_1 \cdot c_2 \cdot c_2 \cdots c_n \in \Sigma^*$, starting from the initial state ι and following the sequence of transitions:

$$(\iota = q_1) \xrightarrow{c_1} q_2 \xrightarrow{c_2} q_3 \xrightarrow{c_2} \dots$$

where $q_{i+1} = \delta(q_i, c_i)$. Since δ is a partial function, $\delta(q_i, c_i)$ may be undefined at some index i, consequently $\mathscr{A}$ may not always read completely the word w. If w is read completely then the last state of the sequence will be called a *reached state* and denoted by $\delta(\iota, w)$, if not then $\delta(\iota, w)$ is undefined. This notation coincides with the natural extension of the transition function

$$\delta\colon Q \times \Sigma^* \to Q$$

which is a partial function to words in Σ^*, defined as follows

$$\delta(q_1, c_1 \cdot c_2 \cdot \cdots \cdot c_n) = \begin{cases} q_{n+1}, & \text{if } q_{i+1} = \delta(q_i, c_i) \text{ for all } 1 \leq i \leq n \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

Whenever $\delta(q, w)$ is defined for some word $w \in \Sigma^*$, the function $Tr(q, w)$ is defined to return the sequence of states occurring on the sequence of transitions.

Example 3.1.2. The LTS $\mathscr{A}$ in Figure 3.2 has $Q(\mathscr{A}) = \{q_1, q_2, q_3, \perp\}$, $\iota(\mathscr{A}) = q_1$, $\Sigma(\mathscr{A}) = \{a, b, c\}$ and the transition function $\delta(\mathscr{A}) = \delta$ is defined by

$$\begin{aligned} &\delta(q_1, a) = q_2, \delta(q_1, b) = q_3, \delta(q_1, c) = \perp \\ &\delta(q_2, a) = q_1, \delta(q_2, b) = q_3, \delta(q_2, c) = \text{undefined} \\ &\delta(q_3, a) = \perp, \delta(q_3, b) = \text{undefined}, \delta(q_3, c) = \text{undefined} \\ &\delta(\perp, a) = \text{undefined}, \delta(\perp, b) = \text{undefined}, \delta(\perp, c) = \text{undefined} \end{aligned}$$

The function δ is extended to words in $\{a, b, c\}^*$, for example $\delta(q_1, a \cdot a \cdot b) = q_3$ and $Q(q_1, a \cdot a \cdot b) = q_2 \cdot q_1 \cdot q_3$. There is only one terminal state in $\mathscr{A}$, that is $\perp$.

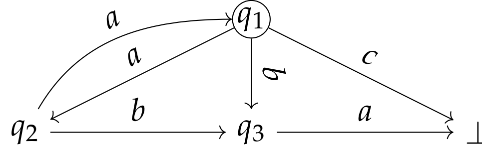


Figure 3.2: An exemplified LTS

Remark 3.1.4. Hereafter, we use the circle to mark the initial state of LTS(s) in illustrative figures. The letter $\perp$ (with or without scripts) serves to identify terminal states.

Remark 3.1.5. A state q in the LTS $\mathscr{A}$ is called reachable if there is a word $w \in \Sigma^*$ so that $\delta(\iota, w) = q$, otherwise q is called unreachable. The unreachable states are not interesting because they do not participate to the reading procedure of the LTS, and can be safely removed accordingly. From now on, we suppose that the considered LTS(s) contain only reachable states, except explicitly mentioned.

Definition 3.1.5 (Underlying graph of an LTS)**.** Given an LTS $\mathscr{A}(Q, \iota, \Sigma, \delta)$, there corresponds a graph $G(E, V)$ determined by

- the set of vertices $V = Q$,
- the set of (directed) edges $E = \{q \xrightarrow{c} q' \mid \delta(q, c) = q'\}$,
- $s(e) = q$ and $t(e) = q'$ if $e = q \xrightarrow{c} q'$ for some $c \in \Sigma$,

and is called the underlying graph of $\mathscr{A}$, denoted by $G(\mathscr{A})$.

LTS morphisms The concept of LTS morphisms is derived directly from the (underlying) graph morphism, and is defined as follows.

Definition 3.1.6 (LTS morphism)**.** Given LTS(s) $\mathscr{A}_1$ and $\mathscr{A}_2$, a LTS morphism $h\colon \mathscr{A}_1 \to \mathscr{A}_2$ is a graph homomorphism $h\colon G(\mathscr{A}_1) \to G(\mathscr{A}_2)$ satisfying

1. $h\left(q \xrightarrow{c} q'\right) = h(q) \xrightarrow{c} h(q')$,
2. $h(\iota_1) = \iota_2$,

then $\mathscr{A}_1$ and $\mathscr{A}_2$ are called respectively the source and the target of h. In other words, the graph homomorphism respects letters in transitions.

Remark 3.1.6. From now on, we call LTS morphism shortly morphism when the meaning is clear from the context.

Since the LTS morphism is a graph morphism, the *complete*, *correct* and *injective* properties can be applied to it (cf. Definition 3.1.3). Concretely, for a LTS $\mathscr{A}\langle Q, \iota, \delta, \Sigma\rangle$, let $Q|_{q\to}$ denote the set of all outgoing transitions of q. Then given a LTS morphism $h\colon \mathscr{A}_1 \to \mathscr{A}_2$ and a state $q \in Q(\mathscr{A}_1)$ there is a natural restriction of h on q

$$h|_{q\to}\colon\ Q(\mathscr{A}_1)|_{q\to} \to Q(\mathscr{A}_2)|_{h(q)\to}$$

Definition 3.1.7 (Correct, complete and surjective LTS morphism)**.** The morphism $h\colon \mathscr{A}_1 \to \mathscr{A}_2$ is called correct (resp. complete) if the restriction $h|_{q\to}$ is injective (resp. surjective) for all state $q \in Q(\mathscr{A}_1)$. We say also h is surjective if the function $h_Q\colon Q(\mathscr{A}_1) \to Q(\mathscr{A}_2)$ is surjective.

Proposition 3.1.2. *A LTS morphism is always correct. The composition of two morphisms is a morphism.*

Proof. The former can be verified directly from Definitions 3.1.6 and 3.1.7. The latter is a direct consequence of Proposition 3.1.1. □

Lemma 3.1.1. *Given LTS(s) $\mathscr{A}_1\langle Q_1, \iota_1, \Sigma_1, \delta_1\rangle$ and $\mathscr{A}_2\langle Q_2, \iota_2, \Sigma_2, \delta_2\rangle$, suppose that there is a morphism $h\colon \mathscr{A}_1 \to \mathscr{A}_2$ then: if $\delta_1(\iota_1, w)$ is defined then so is $\delta_2(\iota_2, w)$, and $\delta_2(\iota_2, w) = h(\delta_1(\iota_1, w))$.*

Proof. The lemma can be proved by structural induction on w. For the base step, that is $w = \epsilon$ (i.e. the empty word), we have $\delta_1(\iota_1, \epsilon) = \iota_1$ and $\delta_2(\iota_2, \epsilon) = \iota_2 = h(\iota_1)$ directly from the definition.

$$\begin{array}{ccc} \delta_1(\iota_1, w) & \xrightarrow{c} & \delta_1(\iota_1, w \cdot c) \\ {\scriptstyle h}\downarrow & & \downarrow{\scriptstyle h} \\ \delta_2(\iota_2, w) & \xrightarrow{c} & \delta_2(\iota_2, w \cdot c) \end{array}$$

For the inductive step, suppose that $\delta_1(\iota_1, w)$ is defined (and unique) and $\delta_2(\iota_2, w) = h(\delta_1(\iota_1, w))$ for some word w. If $\delta_1(\iota_1, w \cdot c)$ is defined for some letter c, namely there is a unique transition

$$\delta_1(\iota_1, w) \xrightarrow{c} \delta_1(\iota_1, w \cdot c)$$

then there is a unique state $h(\delta_1(\iota_1, w \cdot c))$ satisfying the transition

$$h(\delta_1(\iota_1, w)) \xrightarrow{c} h(\delta_1(\iota_1, w \cdot c))$$

We have $h(\delta_1(\iota_1, w)) = \delta_2(\iota_2, w)$, then $\delta_2(i_2, w \cdot c)$ is uniquely defined and $\delta_2(i_2, w \cdot c) = h(\delta_1(\iota_1, w \cdot c))$. □

Proposition 3.1.3. *Given LTS $\mathscr{A}_1$ and $\mathscr{A}_2$, if there is a morphism $h\colon \mathscr{A}_1 \to \mathscr{A}_2$ then h is unique.*

Proof. We first note that all states of $\mathscr{A}_1$ are reachable[1]: for any state $q \in Q(\mathscr{A}_1)$ there is a word w so that $\delta_1(\iota_1, w) = q$. By Lemma 3.1.1, if $\delta_1(\iota_1, w)$ is defined then $\delta_2(\iota_2, w)$ is also uniquely defined, and

$$h(q) = h(\delta_1(\iota_1, w)) = \delta_2(\iota_2, w)$$

That means $h(q) = \delta_2(\iota_2, w)$ for any morphism h, in other words, h is unique. □

Summary The concepts presented above are mathematical background needed to develop the next results about modeling the behaviors of a program receiving different values of an input message. We are now ready to discuss some structures inside a LTS as well as relations, given by morphisms, between LTS(s).

3.2 Prediction semantics of label transition systems

In this second part, we introduce two notions. The first one, called *π-equivalence*, concerns a special equivalence relation of the set of states of a LTS. The second one, called *observational prediction* (or *behavioral prediction*), concerns a special relation between LTS(s). Both of them are used to construct a model that abstracts the concrete behaviors observed from finite execution traces of a program.

The advantage of this model is that it is constructed from finite behaviors, but it can be used to "predict" arbitrarily long behaviors. Some results deduced from this model are constructive and they allow us providing algorithms constructing practical objects in the next chapter.

Obviously, these notions do not come from vacuum. They indeed come from our subjective viewpoint in trying to propose a model that explains the concrete behaviors. So beside presenting the formal results, we give also some initial intuitions leading to them, that hopefully helps showing their naturalness.

[1] In Remark 3.1.5, we have supposed that all states in a LTS are reachable.

3.2.1 Π-equivalence

Given a LTS $\mathscr{A} = \langle Q, \iota, \Sigma, \delta \rangle$, the transition function $\delta \colon Q \times \Sigma \to Q$ can be thought of as a reaction function describing the behavior of $\mathscr{A}$ as a *reactive system*: at a state $q \in Q$, depending the received input letter c, the system moves to the state q'

$$q \xrightarrow{c} q'$$

as determined by δ as $(q, c) \mapsto q'$. Under this viewpoint, we are led to an important concept in the theory of reactive systems: *bisimulation and bisimilarity relation* between states of labeled transition systems [114, 140].

Definition 3.2.1 (Bisimulation and bisimilarity)**.** Given a LTS $\mathscr{A} = \langle Q, \iota, \Sigma, \delta \rangle$, a bisimulation in $\mathscr{A}$ is a relation $\mathscr{R} \subseteq \Sigma \times \Sigma$ satisfying: if $(q_1, q_2) \in \mathscr{R}$, then

- for any transition $q_1 \xrightarrow{c} q_1'$ from q_1, there is also a transition $q_2 \xrightarrow{c} q_2'$ so that $(q_1', q_2') \in \mathscr{R}$, and
- that is also true for any transition from q_2.

The bisimilarity is determined by the unique union of all bisimulations.

Internal viewpoint of prediction

The bisimulation/bisimilarity relation is used to describe the behavioral equivalence between processes in the theory of reactive systems [114], but the development leading to the notion of *π-equivalence* given below comes from a slightly different viewpoint. That reveals also the difference between the two notions: standard bisimulation/bisimilarity and π-equivalence.

Practically, we observe different execution traces of a program by feeding it with different values of the input message, by this way we can construct initially a LTS that reads all tested values. More importantly, there is a correspondence between the reading sequences of the LTS and the traces of the program: we say that *this LTS simulates the program*.

But the problem is that the observed traces, practically, must be finite, so the simulation given by the constructed LTS is validated only for finite traces. Such an LTS has a shape of tree, and thus is very limited: *it cannot predict anything*.

To get a more general LTS, we make a thesis that there are states in the initial LTS that are equivalent[1], so if we can abstract them as a single state then the obtained LTS is more general: *it can predict the behavior of arbitrarily long traces*. The abstraction procedure will verify some conditions about states:

1. they must have the same behaviors inside the finite (i.e observed) traces, and

[1]This is no surprising because by observing the reading procedure of any (long enough) word on an LTS, one can recognize that there are always repeated states. The formal statement of this phenomenon when the LTS is a finite state automaton is the *pumping lemma* [78, 149].

2. if they are considered to be equivalent, abstracting them as a single state does not lead to any "observed contradiction". That means the abstract LTS (i.e. the LTS obtained from the abstraction procedure) may contain infinite traces, but any infinite trace must contains a unique finite trace in the original LTS which ends at a terminal state (i.e. ends at conventional state, from that further behaviors are considered to unobservable with respect to the original LTS).

The second condition reveals also the difference between the standard notion of bisimulation/bisimilarity and the one of π-equivalence. The abstracted LTS, while keeps the same behaviors with the original LTS on observed traces, can "predict" also the behaviors of terminal states in the original LTS. We may note that the terminal states are states where, because of our limited observation, we cannot observe yet their behaviors. They are not states where we can assure that there are no behaviors.

In summary, the abstracted LTS (constructed by merging equivalent states of the original LTS) is not equivalent with the original LTS. Instead, it should be able to *predict* some unobservable behaviors caused by the limited observation capability of the original LTS. Thus, the standard notions of bisimulation and bisimilarity cannot be used directly. Furthermore, we can prove that no existing notion of process equivalence [68, 128] can capture this prediction semantics. As far as we known, *such kind of semantics is not discussed anywhere*.

Relative prediction It is worth noting that *the prediction is relative*: it is validated with the information obtained from observed finite traces but it may be invalidated when we receive more information by observing longer finite traces. In this case, the abstraction procedure must be restarted with the new information. We can observe such a case in the following example.

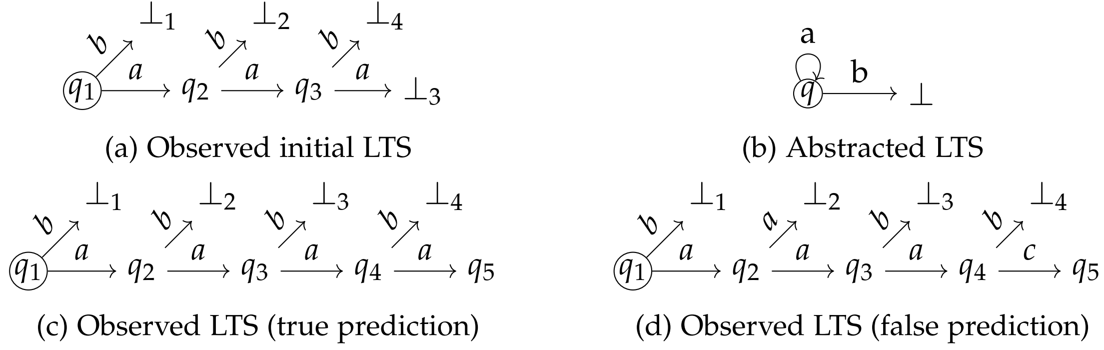
Example 3.2.1. The abstraction procedure starts with the initial LTS in Figure 3.3a, obtained by observing traces with the maximal length is 3. The LTS can distinguish traces of the program when receiving different input values, for example the traces

$$Tr\left(q_1, a \cdot a \cdot a \cdot \Sigma^*\right), Tr\left(a \cdot b \cdot \Sigma^*\right), Tr\left(b \cdot \Sigma^*\right)$$

are mutually distinguished. We may arrive also at the conclusion that some states of the LTS are equivalent, for example

$$s_1 \sim s_2 \sim s_3 \sim \bot_3 \text{ and } \bot_1 \sim \bot_2 \sim \bot_4$$

then we constructs an abstract LTS (cf. Figure 3.3b) as a predictive model. This prediction will be valid in the case we observe traces with the maximal length is 4 and obtain the LTS in Figure 3.3c, but it will be invalid if we obtain the LTS in Figure 3.3d.



(a) Observed initial LTS

(b) Abstracted LTS

(c) Observed LTS (true prediction)

(d) Observed LTS (false prediction)

Figure 3.3: Abstraction procedure

Formal development

Motivated by this intuition, we state the formal definition of this equivalence relation, called the π-equivalence, as follows.

Definition 3.2.2 (π-equivalence)**.** Given an LTS $\mathscr{A} = \langle Q, \iota, \Sigma, \delta \rangle$, a π-equivalence in $\mathscr{A}$ is an equivalence relation $\sim \subseteq \Sigma \times \Sigma$ satisfying: if $q_1 \sim q_2$ then

1. either q_1 or q_2 is a terminal state, or

2. for any transition $q_1 \xrightarrow{c} q_1'$, there is also a transition $q_2 \xrightarrow{c} q_2'$ so that $q_1' \sim q_2'$,

Note 3.2.1. The notation of π-equivalence given above and the standard notation of bisimulation [114, 140] have an important difference. Consider some states q_1, q_2, q_3 satisfying $(q_1, q_2) \in R_1$ and $(q_2, q_3) \in R_2$

- if R_1 and R_2 are bisimulations then there exists always some bisimulation R_3 so that $(q_1, q_3) \in R_3$, but

- if R_1 and R_2 are π-equivalences then there may never exist any π-equivalence R_3 so that $(q_1, q_3) \in R_3$.

Technically, let us consider q_1, q_2, q_3 where q_2 is a terminal state but q_1, q_3 are not, then we have immediately $(q_1, q_2) \in R_1$ and $(q_1, q_3) \in R_3$, for some π-equivalence R_1 and R_3 because they satisfy the first condition of Definition 3.2.2. Now we can arrange q_1 and q_3 so that they do not satisfying the second condition, then there does not exist any π-equivalence R_3 so that $(q_1, q_3) \in R_3$.

The π-equivalence is actually an *approximation*: it tries to approximate a state of low-information by a state of high-information, here they are respectively q_2 and either q_1 or q_3. The intuition here is that q_2 has no observed behaviors (i.e. no outgoing transitions), such a state can be approximated by any state.

This difference leads to non-standard phenomenons. For example the union of all π-equivalences is not a π-equivalence in general, so the standard definition of bisimilarity (as the union of all bisimulations) can not be applied for the π-equivalence.

Example 3.2.2. From Definition 3.2.2, we can verify that the following equivalence relations

$$\begin{aligned}
\sim_1 &= \{\{q_2, \bot_1\}, \{q_1\}, \{q_4\}, \{\bot_2\}, \{\bot_3\}, \{\bot_4\}, \{\bot_5\}, \{\bot_6\}\} \\
\sim_2 &= \{\{q_4, \bot_1\}, \{q_1\}, \{q_2\}, \{\bot_2\}, \{\bot_3\}, \{\bot_4\}, \{\bot_5\}, \{\bot_6\}\} \\
\sim_3 &= \{\{\bot_1, \bot_2, \bot_3, \bot_4, \bot_5, \bot_6, q_1\}, \{q_2\}, \{q_4\}\} \\
\sim_4 &= \{\{q_2, \bot_1, \bot_2, \bot_3, \bot_4\}, \{q_1, \bot_5, \bot_6\}, \{q_1\}\}
\end{aligned}$$

in the LTS of Figure 3.4 are π-equivalences. It may worth noting that $q_2 \sim_1 \bot_1$ and $\bot_1 \sim_2 q_4$, but there is no π-equivalence relation $\sim$ so that $q_2 \sim q_4$, because q_2 has 3 transitions while q_4 has only 2 transitions.

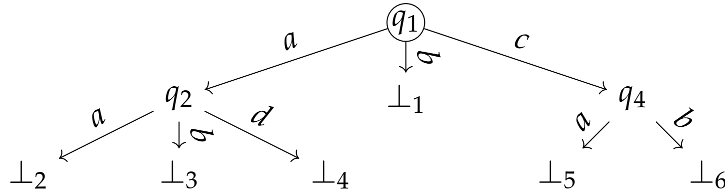


Figure 3.4: LTS with π-equivalences

Quotient labeled transition system From the definition of π-equivalence, we have a property: if there is a transition of some label c from a state q, then any state in the equivalence class $[q]_\sim$ either

1. has also a transition of label c, or
2. it must be a terminal state.

This is stated formally as follows.

Proposition 3.2.1. *Let $\sim$ be a π-equivalence in a LTS $\mathscr{A}$, given an equivalence class $[q]_\sim$ of some state $q \in Q(\mathscr{A})$. If there is some transition $q \xrightarrow{c} q'$ for some $c \in \Sigma(\mathscr{A})$ and some $q' \in Q(\mathscr{A})$, then for any $q_1 \in [q]_\sim$ either*

- *there is a unique state $q'_1 \in Q(\mathscr{A})$ so that $q_1 \xrightarrow{c} q'_1$ and $q' \sim q'_1$, or*
- *q_1 is a terminal state.*

Proof. This is direct from Definition 3.2.2: we note that $q \sim q_1$ since $q_1 \in [q]_\sim$, so if $q \xrightarrow{c} q'$ and q_1 is not a terminal state then there exists also the transition $q_1 \sim q'_1$ for some $q'_1 \sim q'$, and because the LTS $\mathscr{A}$ is deterministic then q'_1 is unique. □

From Proposition 3.2.1, if we identify equivalence states into a new single state and define a transition

$$[q]_\sim \xrightarrow{c} [q']_\sim$$

whenever there is some $q_1 \in [q]_\sim$ and $q_1' \in [q']_\sim$ so that

$$q_1 \xrightarrow{c} q_1'$$

then the "similar" transition

$$q \xrightarrow{c} q'$$

for some $q' \in [q']_\sim$ exists also on any non-terminal state $q \in [q]_\sim$. That makes the following construction of the quotient LTS is not ambiguous.

Definition 3.2.3. Given an LTS $\mathscr{A} = (Q, \iota, \Sigma, \delta)$ and a π-equivalence $\sim$ in $\mathscr{A}$. The LTS $\mathscr{A}/\sim$ constructed as follows:

- $Q(\mathscr{A}/\sim) = \{[q]_\sim \mid q \in Q(\mathscr{A})\}$, $\iota(\mathscr{A}/\sim) = [\iota(\mathscr{A})]_\sim$,
- $\delta(\mathscr{A}/\sim) = \{[q]_\sim \xrightarrow{c} [q']_\sim \mid q \xrightarrow{c} q' \in \delta(\mathscr{A})\}$, $\Sigma(\mathscr{A}/\sim) = \Sigma(\mathscr{A})$

is called the quotient LTS of $\mathscr{A}$ by $\sim$.

Example 3.2.3. We reconsider several LTS(s) described in Examples 3.2.1 and 3.2.2. First, we can verify that the following equivalence relations in the original LTS $\mathscr{A}_1$ (cf. Figure 3.5a)

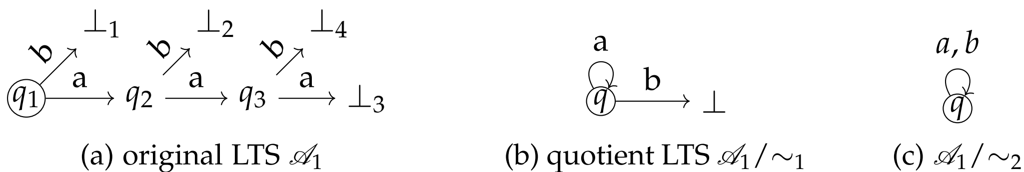
$$\sim_1 = \{\{q_1, q_2, q_3, \bot_3\}, \{\bot_1, \bot_2, \bot_4\}\}$$
$$\sim_2 = \{\{q_1, q_2, q_3, \bot_1, \bot_2, \bot_3, \bot_4\}\}$$

are π-equivalences. The quotient LTS $\mathscr{A}_1/\sim_1$ (cf. Figure 3.5b) is constructed by letting

$$q = \{q_1, q_2, q_3, \bot_3\} \text{ and } \bot = \{\bot_1, \bot_2, \bot_4\}$$

and similarly, the quotient LTS $\mathscr{A}/\sim_2$ (cf. Figure 3.5c) is constructed by letting

$$q = \{q_1, q_2, q_3, \bot_1, \bot_2, \bot_3, \bot_4\}$$



(a) original LTS $\mathscr{A}_1$ (b) quotient LTS $\mathscr{A}_1/\sim_1$ (c) $\mathscr{A}_1/\sim_2$

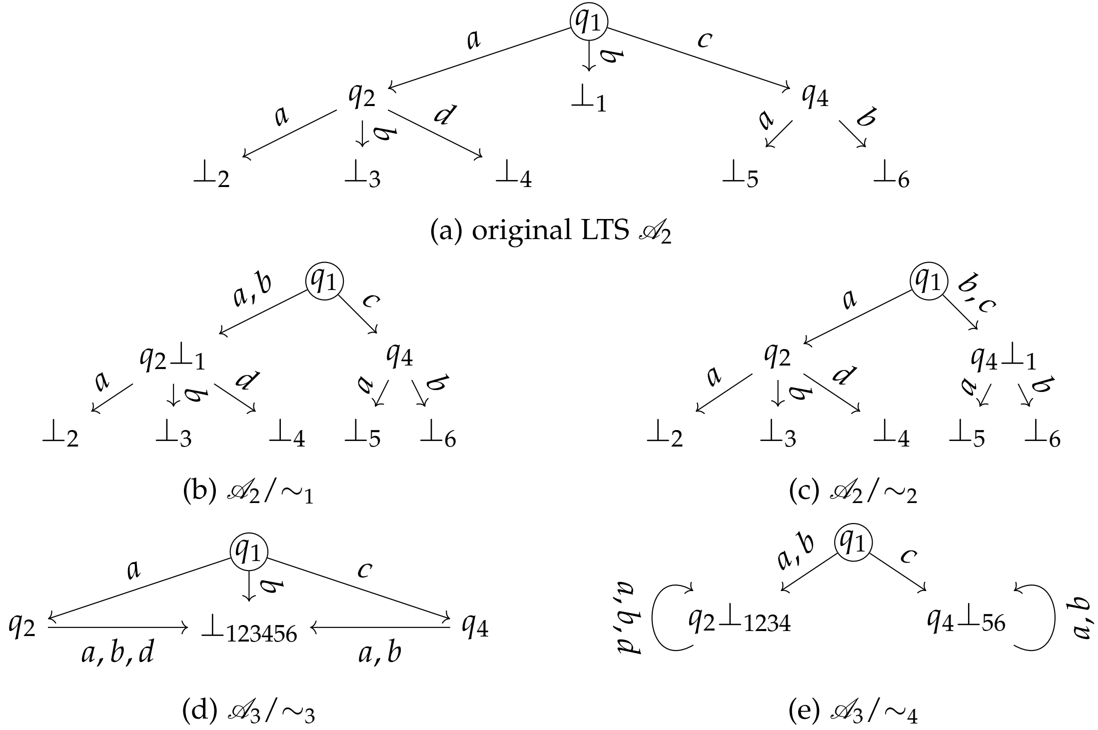
Figure 3.5: LTS in Example 3.2.1

Second, in the LTS $\mathscr{A}_2$ (cf. Figure 3.6a) we have verified in Example 3.2.2 that the following equivalence relations

$$\sim_1 = \{\{q_2, \bot_1\}, \{q_1\}, \{q_4\}, \{\bot_2\}, \{\bot_3\}, \{\bot_4\}, \{\bot_5\}, \{\bot_6\}\}$$

$$\sim_2 = \{\{q_4, \bot_1\}, \{q_1\}, \{q_2\}, \{\bot_2\}, \{\bot_3\}, \{\bot_4\}, \{\bot_5\}, \{\bot_6\}\}$$
$$\sim_3 = \{\{\bot_1, \bot_2, \bot_3, \bot_4, \bot_5, \bot_6\}, \{q_1\}, \{q_2\}, \{q_4\}\}$$
$$\sim_4 = \{\{q_2, \bot_1, \bot_2, \bot_3, \bot_4\}, \{q_1, \bot_5, \bot_6\}, \{q_1\}\}$$

are π-equivalences. They lead to the quotient LTS(s) $\mathscr{A}_2/\sim_i$ for $i = 1, \ldots, 4$ (cf. Figures 3.6b to 3.6e).



(a) original LTS $\mathscr{A}_2$

(b) $\mathscr{A}_2/\sim_1$

(c) $\mathscr{A}_2/\sim_2$

(d) $\mathscr{A}_3/\sim_3$

(e) $\mathscr{A}_3/\sim_4$

Figure 3.6: LTS in Example 3.2.2

Any equivalence relation on a set defines a function from the set to the quotient set, that maps any element to its equivalence class. Similarly, the π-equivalence defines a morphism from the original LTS to the quotient LTS, moreover this morphism is also complete as stated in Proposition 3.2.1.

Proposition 3.2.2. *Given a π-equivalence $\sim$, the function $h_\sim \colon \mathscr{A} \to \mathscr{A}/\sim$ that maps*

$$q \mapsto [q]_\sim \text{ and } q \xrightarrow{c} q' \mapsto [q]_\sim \xrightarrow{c} [q']_\sim$$

is a surjective complete morphism.

Remark 3.2.1. The function $h_\sim$ defined above is called the *derived morphism* of $\sim$.

Proof. The function $h_\sim$ is a surjective morphism by its construction. From Proposition 3.2.1, if there is a transition $[q]_\sim \xrightarrow{c} [q']_\sim$ and q is not a terminal state then there is also a transition $q \xrightarrow{c} q''$ for some $q'' \sim q'$, namely $h_\sim$ is complete. $\square$

Conceptually, a complete morphism will map equivalent states in the source LTS into a single state of the target and it keeps also all labeled transitions. Then unsurprisingly, a complete morphism determines also a π-equivalence, this is stated formally as follows.

Proposition 3.2.3. *Given a complete morphism $h\colon \mathscr{A} \rightarrow \mathscr{A}'$, then there is a π-equivalence $\sim$ in $\mathscr{A}$ whose the derived morphism $h_\sim$ satisfies $\mathscr{A}/\!\sim\, = h_\sim(\mathscr{A}) = h(\mathscr{A})$.*

Remark 3.2.2. In Proposition 3.2.3 and hereafter, two LTS(s) are considered equal if they are identical up to some relabeling of states.

Remark 3.2.3. In Proposition 3.2.3, the morphisms h and $h_\sim$ have different co-domains (one is $\mathscr{A}'$ whereas the other is $\mathscr{A}/\!\sim$) then normally we cannot write $h_\sim = h$. But with the convention about the equality of LTS(s) we can write unambiguously $h_\sim = h$. In general, given LTS morphisms $h_1\colon \mathscr{A}_1 \rightarrow \mathscr{A}_1'$ and $h_2\colon \mathscr{A}_2 \rightarrow \mathscr{A}_2'$, we can write unambiguously

$$h_1 = h_2$$

whenever $\mathscr{A}_1 = \mathscr{A}_2$ and $h_1(\mathscr{A}_1) = h_2(\mathscr{A}_2)$, the LTS equality notations follow Remark 3.2.2.

Proof. (of Proposition 3.2.3) Let $\sim$ be a relation in (states of) $\mathscr{A}$ defined by

$$q \sim q' \Longleftrightarrow h(q) = h(q')$$

then clearly $\sim$ is an equivalence relation. To prove that $\sim$ is a π-equivalence, we observe that, for any $q \sim q'$, if q' or q is a terminal state, then the conditions of the π-equivalence in Definition 3.2.2 is satisfied immediately. So it is sufficient to verify the conditions in the case both q and q' are not terminal.

We will verify that if $q \sim q'$ and both of them are not terminal states, then for each transition $q \xrightarrow{c} q_1$, there is a transition $q' \xrightarrow{c} q_1'$ so that $q_1 \sim q_1'$. Indeed, we have $h\left(q \xrightarrow{c} q_1\right) = h(q) \xrightarrow{c} h(q_1)$, since $h(q) = h(q')$ and h is complete, there exists a transition $q' \xrightarrow{c} q_1'$ so that

$$h\left(q' \xrightarrow{c} q_1'\right) = h(q') \xrightarrow{c} h(q_1') = h(q) \xrightarrow{c} h(q_1)$$

and then $h(q_1') = h(q_1)$, in other words $q_1' \sim q_1$. □

The following theorem, about the relation between the π-equivalence and the complete morphism, will combines the results of Propositions 3.2.2 and 3.2.3.

Theorem 1. *Given a morphism $h\colon \mathscr{A} \rightarrow \mathscr{A}'$, then h is complete if and only if there is a π-equivalence $\sim$ in $\mathscr{A}$ so that $h_\sim = h$.*

3.2.2 Observational prediction

The π-equivalence gives an *internal viewpoint* which describes in which conditions a state can be replaced by another state. This viewpoint has indeed a strong relation with an *external viewpoint* (or operational viewpoint) which describes how a LTS can predict correctly another LTS.

External viewpoint of prediction

The bisimulation gives an explanation for which cases we can say that two LTS(s) are behavioral equivalence. The following interpretation gives an explanation for which cases we can say that a LTS can be used as a *predictive model* of another LTS. One can recognize the analogy between two pairs (bisimilarity, behavioral equivalence) and (π-equivalence, behavioral prediction).

Let us consider LTS(s) $\mathscr{A}$ and $\mathscr{A}'$, in reading any word $w = c_1 \cdot c_2 \cdots c_n \in \Sigma^*$ we obtain two sequences of transitions:

$$(\iota = q_1) \xrightarrow{c_1} q_2 \xrightarrow{c_2} q_3 \xrightarrow{c_3} \ldots \xrightarrow{c_{i-1}} q_i \ldots$$
$$(\iota' = q'_1) \xrightarrow{c_1} q'_2 \xrightarrow{c_2} q'_3 \xrightarrow{c_3} \ldots \xrightarrow{c_{i-1}} q'_i \ldots$$

Suppose that both of them have read successfully a suffix $\overline{w} = c_1 \cdot c_2 \cdots c_{i-1}$, and $\mathscr{A}$ reaches the state q_i while $\mathscr{A}'$ reaches the state q'_i. There are some conditions that $\mathscr{A}'$ must satisfy to become a predictive model for $\mathscr{A}$, they are described as follows:

- if $\mathscr{A}$ can continue reading some letter c_i, namely the transition $q_i \xrightarrow{c_i}$ exists (the target of the transition, which is not important here, is omitted), then $\mathscr{A}'$ can also continue reading c_i, namely the transition $q'_i \xrightarrow{c_i}$ exists. Otherwise, if $q_i \xrightarrow{c_i}$ does not exist then neither does $q'_i \xrightarrow{c_i}$,
- if there is a transition $q'_i \xrightarrow{c}$ for some letter $c \in \Sigma$ and q_i is not a terminal state[1], then the transition $q_i \xrightarrow{c}$ exists. Because if it does not then we have the following ambiguity at q'_i: in using $\mathscr{A}'$ which hopefully is a correct prediction of $\mathscr{A}$, we can wrongly conclude that $\mathscr{A}$ can read the word $\overline{w} \cdot c$ (because $\mathscr{A}'$ can), but actually it cannot.

Intuitively, the reasons of conditions above may be clear if we think of the situation where $\mathscr{A}'$ *is used instead of* $\mathscr{A}$.

Formal development

The intuitive conditions about external viewpoint of prediction are stated formally in the following definition.

[1] We accept the case where q_i is a terminal state, such a state is implicitly considered unobservable, namely it has indeed transitions but we cannot observe.

Definition 3.2.4 (Observational prediction)**.** Given LTS(s) $\mathscr{A}\langle Q, \iota, \Sigma, \delta\rangle$ and $\mathscr{A}'\langle Q', \iota', \Sigma, \delta'\rangle$ having the same alphabet, then $\mathscr{A}'$ is called an observational prediction of $\mathscr{A}$ if

1. for any word $w \in \Sigma^*$, if $\delta(\iota, w)$ is defined then so is $\delta(\iota', w)$,
2. for any word $w \in \Sigma^*$ where $\delta'(\iota', w)$ is defined but $\delta(\iota, w)$ is not, there is a unique proper prefix $\overline{w}$ of w so that $\delta(\iota, \overline{w})$ is defined but $\delta(\iota, \overline{w})$ is a terminal state.

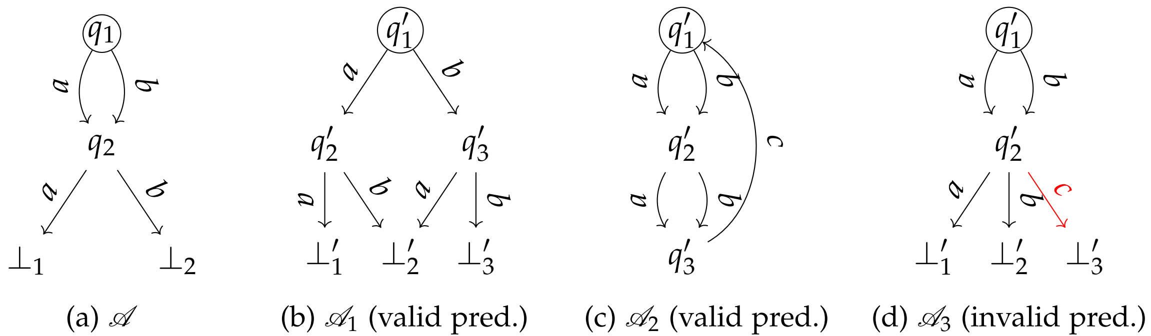
and we write $\mathscr{A} \sqsubseteq_{op} \mathscr{A}'$ (here op stands for observational prediction).

The first condition of Definition 3.2.4 is clear: $\mathscr{A}'$ must be able to read any word that $\mathscr{A}$ can read. The second condition can be thought of as: suppose there is a word w so that $\delta(\iota', w)$ is defined but $\delta(\iota, w)$ is not, that means if $\mathscr{A}'$ is used instead of $\mathscr{A}$ then there is an ambiguity at this word (because $\mathscr{A}$ cannot read w). But *this ambiguity is acceptable* because it is actually a *prediction*.

Concretely speaking, in reading w, the LTS $\mathscr{A}$ has read a *maximal prefix* $\overline{w}$ of w, then it reaches the state $\delta(\iota, \overline{w})$ having no outgoing transitions (i.e. a terminal state), such a state implies that $\mathscr{A}$ may reach further states in reading w, but these states are unobservable.

Example 3.2.4. The LTS(s) $\mathscr{A}_1$ and $\mathscr{A}_2$ (cf.Figures 3.7b and 3.7c) are observational predictions of $\mathscr{A}$ (cf. Figure 3.7a), but the LTS $\mathscr{A}_3$ (cf. Figure 3.7d) is not. Indeed,

- $\mathscr{A}$ and $\mathscr{A}_1$ have the same set of readable words $W = \{a, b, a \cdot a, a \cdot b, b \cdot a, b \cdot b\}$. $\mathscr{A}_2$ can read also any word in W, moreover if w is a readable word of $\mathscr{A}_2$ and $w \notin W$ (i.e. $\mathscr{A}$ cannot read w) then w has form $\overline{w} \cdot c \cdot w'$ where $\overline{w} \in \{a \cdot a, a \cdot b, b \cdot a, b \cdot b\}$ (i.e. $\delta(\iota, \overline{w})$ is a terminal state of $\mathscr{A}$),
- $\mathscr{A}_3$ can read $a \cdot c$ while $\mathscr{A}$ cannot, the word $a \cdot c$ has two proper prefixes a and ϵ, but neither $\delta(\iota, a)$ nor $\delta(\iota, \epsilon)$ is a terminal state of $\mathscr{A}$.



(a) $\mathscr{A}$ (b) $\mathscr{A}_1$ (valid pred.) (c) $\mathscr{A}_2$ (valid pred.) (d) $\mathscr{A}_3$ (invalid pred.)

Figure 3.7: Observational predictions of a LTS

The LTS $\mathscr{A}_2$ is also a prediction for $\mathscr{A}$: it can read the word $a \cdot a \cdot c$ while $\mathscr{A}$ can read only the prefix $a \cdot a$. Here it simulates the original one, and moreover it predicts that there may be a transition $\perp'_1 \xrightarrow{c} q'_1$.

Example 3.2.5. There is a detail that may lead to the misunderstanding about the observational prediction. That is one may think that any LTS has a trivial observational prediction consisting of a single state with multiple loop-back transition.

Consider the LTS inFigure 3.8, one may think that this LTS is an observational prediction of all LTS(s) in Figure 3.7. But that is not true, one can verify the conditions

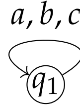


Figure 3.8: Trivial invalid observational prediction

of Definition 3.2.4. For example, the LTS in Figure 3.8 can read a word such that $c \cdot a \cdot b$ but no LTS in Figure 3.7 can read this word. Moreover there is also no prefix of $c \cdot a \cdot b$ so that any LTS in Figure 3.7 can reach a terminal state in reading this prefix.

Observational prediction and complete morphism The following proposition shows a relation between the complete morphism and the observational prediction.

Proposition 3.2.4. *Given LTS(s)* $\mathscr{A}\langle Q, \iota, \Sigma, \delta\rangle$ *and* $\mathscr{A}'\langle Q', \iota', \Sigma, \delta'\rangle$ *having the same alphabet, if there is a complete morphism* $h\colon \mathscr{A} \to \mathscr{A}'$ *then* $\mathscr{A} \sqsubseteq_{op} \mathscr{A}'$.

Proof. We verify the conditions of Definition 3.2.4. The first one can be derived directly from Lemma 3.1.1: if $\delta(\iota, w)$ is defined then so is $\delta'(\iota', w)$. To prove the second one, let us consider a word w where $\delta'(\iota', w)$ is defined but $\delta(\iota, w)$ is not, then we can always find a suffix

$$\overline{w} = c_1 \cdot c_2 \cdots c_{i-1}$$

(in the worst case $\overline{w} = \epsilon$) of w by letting $\mathscr{A}$ read w as

$$(\iota = q_1) \xrightarrow{c_1} q_2 \xrightarrow{c_2} \ldots \xrightarrow{c_{i-1}} q_i$$

and stop at q_i whenever there is no valid transition, then $\delta(\iota, \overline{w})$ is defined for every prefix w' satisfying $|w'| \leq |\overline{w}|$. Let $\mathscr{A}'$ read w, the reading sequence (given by Lemma 3.1.1) is

$$\left(\iota' = h(\iota)\right) \xrightarrow{c_1} h(q_2) \xrightarrow{c_2} \cdots \xrightarrow{c_{i-1}} h(q_i)$$

Let $c_i \in C$ be the letter so that w has form

$$\overline{w} \cdot c_i \cdots$$

such a letter exists always because $w \neq \epsilon$ (since $\delta(\iota, w)$ is not defined), so there is a transition $h(q_i) \xrightarrow{c_i}$ but the transition $q_i \xrightarrow{c_i}$ does not exist. Because h is complete, the appeared situation of transitions is possible only when q_i is a terminal state. That means $\delta(\iota, \overline{w}.c)$ is undefined for any $c \in \Sigma$. □

We can observe that the observational prediction is looser than the existence of a complete morphism. For example, in Example 3.2.4 the LTS $\mathscr{A}_1$ is an observational prediction of $\mathscr{A}$, but there is no morphism from $\mathscr{A}$ to $\mathscr{A}_1$, then the inverse of Proposition 3.2.4 is not true. However, we can construct from $\mathscr{A}_1$ an observational prediction $\mathscr{A}_1'$ having no more states than $\mathscr{A}_1$ and there is a surjective complete morphism from $\mathscr{A}$ to $\mathscr{A}_1'$ (cf. Example 3.2.6). In general, we have the following:

Theorem 2. *Given LTS(s) $\mathscr{A}$ and $\mathscr{A}'$ with $\mathscr{A} \sqsubseteq_{op} \mathscr{A}'$, then there exists a LTS $\mathscr{A}''$ having no more states than $\mathscr{A}'$ and a surjective complete morphism $h\colon \mathscr{A} \to \mathscr{A}''$.*

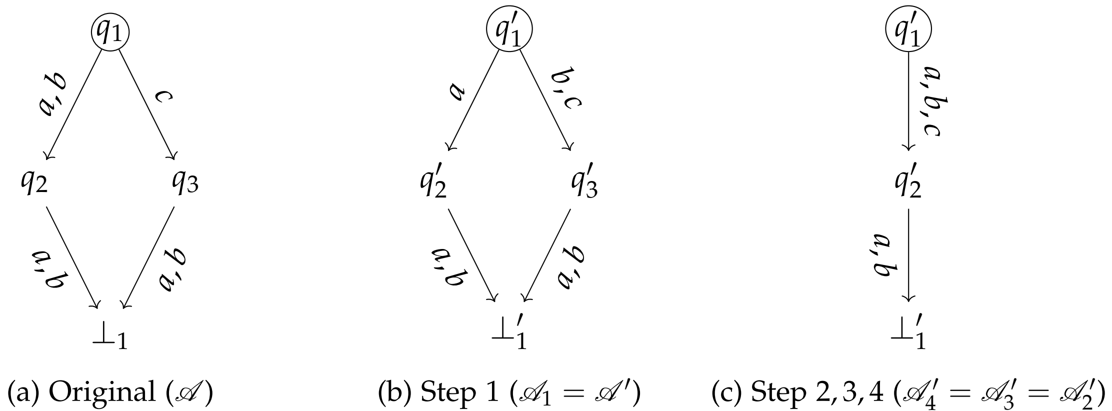
Before giving the proof for the theorem, we can get an intuition about how to construct the surjective complete morphism h and the LTS $\mathscr{A}''$ in the following example.

Remark 3.2.4. In Figure 3.9, we have drawn several "multiple labels" transitions (e.g. $q_1 \xrightarrow{a,b} q_2$ in Figure 3.9a). Each of them expresses actually *multiple transitions of single label* (e.g. $q_1 \xrightarrow{a,b} q_2$ expresses 2 transitions from q_1 to q_2 of labels a and b respectively), and not a *single transition of multiple labels*. We keep using this notation in latter figures.

Example 3.2.6. Let us consider the LTS(s) in Figure 3.9. The LTS $\mathscr{A}'$ (cf. Figure 3.9b) is an observational prediction of $\mathscr{A}$ (cf. Figure 3.9a), but we can prove that there is no surjective complete morphism from $\mathscr{A}$ to $\mathscr{A}'$. Indeed, suppose that there is such a morphism h, since q_1 and q_1' are initial state of $\mathscr{A}$ and $\mathscr{A}'$, we have $q_1' = h(q_1)$. Let us consider the transition $q_1 \xrightarrow{a} q_2$, its image must be

$$h\left(q_1 \xrightarrow{a} q_2\right) = q_1' \xrightarrow{a} q_2'$$

then we have $q_2' = h(q_2)$. But similarly, when considering the transition $q_1 \xrightarrow{b} q_2$, we will have $q_3' = h(q_2)$. This ambiguity about the image of q_2 means h cannot be a morphism.



(a) Original ($\mathscr{A}$) (b) Step 1 ($\mathscr{A}_1 = \mathscr{A}'$) (c) Step 2, 3, 4 ($\mathscr{A}_4' = \mathscr{A}_3' = \mathscr{A}_2'$)

Figure 3.9: Construction of the target LTS and the surjective complete morphism

The ambiguity above can be fixed by modifying the LTS $\mathscr{A}'$, for example we can first change the transition $q_1' \xrightarrow{b} q_3'$ into $q_1' \xrightarrow{b} q_2'$, and next $q_1' \xrightarrow{c} q_3'$ into $q_1' \xrightarrow{c} q_2'$,

and the result LTS is still an observational prediction of $\mathscr{A}$ (cf. Figure 3.9c) Based on this observation, we can construct the morphism h and modify the LTS $\mathscr{A}'$ in the following steps

- Step 1: keep $\mathscr{A}_1 = \mathscr{A}'$ (cf. Figure 3.9b), and let $h(q_1) = q_1'$,
- Step 2: observe that $q_1 \xrightarrow{a,b} q_2$, but $q_1' \xrightarrow{a} q_2'$ and $q_1' \xrightarrow{b} q_3'$, then
  - select an arbitrary state in $\{q_2', q_3'\}$, for example q_2',
  - replace $q_1' \xrightarrow{b} q_3'$ by $q_1' \xrightarrow{b} q_2'$,
  - there remains a transition of target q_3' in $\mathscr{A}_1$, that is $q_1' \xrightarrow{c} q_3'$, replace it by $q_1' \xrightarrow{c} q_2'$,
  - the state q_3' becomes now unreachable, remove it from $\mathscr{A}_1'$,

  consequently, $\mathscr{A}_1'$ becomes $\mathscr{A}_2'$ (cf. Figure 3.9c), and let $h(q_2) = q_2'$,
- Step 3: since $q_1 \xrightarrow{c} q_3$ and $q_1' \xrightarrow{c} q_2'$, keep $\mathscr{A}_3' = \mathscr{A}_2'$ and let $h(q_3) = q_2'$,
- Step 4: since $h(q_2) = h(q_3) = q_2'$, moreover $q_2 \xrightarrow{a,b} \perp_1$ and $q_3 \xrightarrow{a,b} \perp_1$, there are also transitions $q_2' \xrightarrow{a,b} \perp_1'$; simply keep $\mathscr{A}_4' = \mathscr{A}_3'$ and let $h(\perp_1) = \perp_1'$.

The morphism h is now completely defined

$$h(q_1) = q_1', \ h(q_2) = h(q_3) = q_2', \ h(\perp_1) = \perp_1'$$

and $\mathscr{A}''$ is obtained as $\mathscr{A}_3'$.

In the construction of the morphism h and the LTS $\mathscr{A}''$ above, we can assign for each step i a tuple $(Q_i, h_i, \mathscr{A}_i')$ which determine the "state" s_i of the construction, concretely

- $Q_i \subseteq Q(\mathscr{A})$ is the set of examined states in $\mathscr{A}$,
- $h_i \colon Q_i \to Q(\mathscr{A}')$ is the map, mapping partially each state of $\mathscr{A}$ to a state, defined by the construction, of $\mathscr{A}'$,
- $\mathscr{A}_i'$ is the constructed LTS at the result of this step.

For example, the following states are assigned for the construction steps in Example 3.2.6

- Step 1: $s_1 = (\{q_1\}, \{q_1 \mapsto q_1'\}, \mathscr{A}_1')$,
- Step 2: $s_2 = (\{q_1, q_2\}, \{q_1 \mapsto q_1', q_2 \mapsto q_2'\}, \mathscr{A}_2')$,
- Step 3: $s_3 = (\{q_1, q_2, q_3\}, \{q_1 \mapsto q_1', q_2 \mapsto q_2', q_3 \mapsto q_2'\}, \mathscr{A}_3')$,

- Step 4: $s_4 = (\{q_1, q_2, q_3, \perp_1\}, \{q_1 \mapsto q'_1, q_2 \mapsto q'_2, q_3 \mapsto q'_2, \perp_1 \mapsto \perp'_1\}, \mathscr{A}'_4)$.

and the construction can be presented then as a transition of states

$$s_1 \to s_2 \to s_3 \to s_4$$

The idea of the state transition will be used to construct the morphism and the target LTS in the general case. Given LTS(s) $\mathscr{A} = \langle Q, \iota, \Sigma, \delta \rangle$ and $\mathscr{A}' = \langle Q', \iota', \Sigma, \delta' \rangle$ sharing the same alphabet, and satisfying $\mathscr{A} \sqsubseteq_{op} \mathscr{A}'$, the algorithm in Listing 3.1 constructs the surjective complete morphism $h \colon \mathscr{A} \to \mathscr{A}''$ and the target LTS $\mathscr{A}''$. It is worth noting that the tuple $(Q[i], h[i], \mathscr{A}[i])$ in this algorithm implies the state s_i as discussed above.

```
MorphismConstruct(LTS(s) 𝒜 ⊑op 𝒜') {
  i ← 0;
  (Q[i],h[i],𝒜[i]) ← ({ι},{ι ↦ ι'},𝒜');
  while (∃q ∈ Q[i] | ∃q -c-> q' in 𝒜 and q' ∉ Q[i]) {
    (Q[i+1],h[i+1],𝒜[i+1]) ← (Q[i],h[i],𝒜[i]);
    i ← i+1;
    qs ← {q' | q -c-> q' in 𝒜 for some c ∈ Σ and q' ∉ Q[i]};
    foreach (q' ∈ qs) {
      Q[i] ← Q[i] ∪ {q'}; /* update Q[i] */
      cs ← {c | q -c-> q' in 𝒜};
      ps ← {p | h(q) -c-> p in 𝒜[i] for some c ∈ cs};
      p ← an arbitrary element of ps;
      h[i] ← h[i] ∪ {q' ↦ p}; /* update h[i] */
      foreach (p' ∈ ps \ {p}) { /* update 𝒜[i] */
        replace h(q) -c-> p' by h(q) -c-> p;
        removes unreachable states from 𝒜[i];
      }
    }
  }
  h ← h[i]; 𝒜'' ← 𝒜[i];
}
```

Listing 3.1: Surjective complete morphism construction algorithm

The proof of Theorem 2 bases on the termination and correctness of the algorithm. In the following assertions, we first prove that it terminates (cf. Corollary 3.2.1). Because if h is not complete for some i, then the LTS $\mathscr{A}[i]$ cannot be an observational prediction of $\mathscr{A}$, then we prove that the algorithms is correct by showing that the invariant $\mathscr{A} \sqsubseteq_{op} \mathscr{A}[i]$ is valid for all i (cf. Corollary 3.2.3).

The state of the i-th construction step (i.e. the i-th execution of the `while`-loop block) is represented by a tuple $s_i = (Q[i], h[i], \mathscr{A}[i])$, the set of examined state $Q[i]$ is updated as similar to the set of examined vertices in a *breadth-first search* procedure: starting from $\{\iota\}$, new states will be added if they can be reached from the states of the current set.

Lemma 3.2.1. *For each state $q \neq \iota$ of the LTS $\mathscr{A}$, there exists an index i such that $q \notin Q[i]$ but $q \in Q[i+1]$.*

Proof. We observe that $Q[i] \subseteq Q[i+1]$ for all i, consequently if the index i exists (for a state $q \neq \iota$) then it is also unique, so it is sufficient to prove the existence of i. Since q is reachable from the initial state ι[1], let $w = c_1 \cdot c_2 \cdots c_n \in \Sigma^*$ be the shortest word satisfying $\delta(\iota, w) = q$. Because at each step i, new state (beside states of $Q[i]$) will be added into $Q[i+1]$ if they can be reached from $Q[i]$, then $\delta(c_1 \cdot c_2 \cdots c_i, \iota)$ will be added into $Q[j]$ for some $j \leq i$. Hence, q will be added into $Q[m]$ for some $m \leq n$. □

Corollary 3.2.1. *The algorithm in Listing 3.1 halts always, and $Q[i] = Q$ when it halts.*

Proof. It is sufficient to prove that the `while`-loop halts. Indeed, from Lemma 3.2.1 there is some index i so that $q \in Q[i]$ for all $q \in Q$, in other words $Q[i] = Q$. Moreover the `while`-loop will halt at an index i where all states, which are reachable from states of $Q[i]$, belong also to $Q[i]$ □

The construction of the algorithm in Listing 3.1 is similar to the construction of a surjective function from a function by modifying its co-domain. The image $h(q')$ of a state $q' \in Q$ is set by choosing a representative element q in the set of possible images ps.

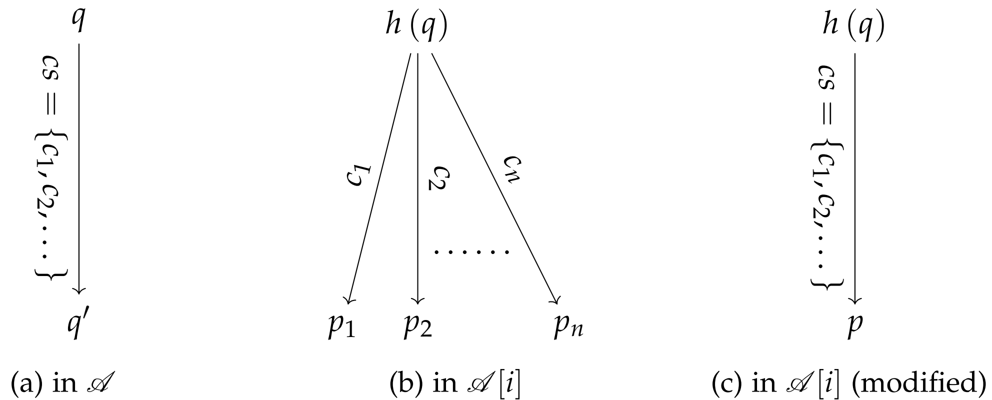
Example 3.2.7. We have assigned the image $h(q) \in Q(\mathscr{A}[i])$ for the state $q \in \mathscr{A}[i]$ (cf. Figure 3.10b). To assign the image for the new state q', we can chose a state p in the set

$$ps = \{p \mid h(q) \xrightarrow{c} p \text{ in } \mathscr{A}[i] \text{ for some } c \in cs\}$$

and replace

$$h(q) \xrightarrow{c_j} p_j \text{ by } h(q) \xrightarrow{c_j} p$$

for other states $p_j \in ps \setminus \{p\}$ (cf. Figure 3.10c).



(a) in $\mathscr{A}$ (b) in $\mathscr{A}[i]$ (c) in $\mathscr{A}[i]$ (modified)

Figure 3.10: Choosing an image between p_i(s) for a new state q'

[1] In Remark 3.1.5, we have supposed that all states of a LTS are reachable.

Lemma 3.2.2. *If $\mathscr{A}[i]$ (i.e. the LTS at the end of i-th `while`-loop) is an observational prediction of $\mathscr{A}$ then any word $w \in \Sigma^*$ that can be read from q' in $\mathscr{A}$, can be read also from any state $p_j \in ps$ in $\mathscr{A}[i]$. In other words, if $\delta(q', w)$ is defined then $\delta[i](p_j, w)$ is defined, where $\delta[i]$ denotes the transition function of $\mathscr{A}[i]$.*

Proof. Let $\overline{w}$ be the shortest word satisfying $\delta(\iota, \overline{w}) = q$. Consider the word $w' = \overline{w} \cdot c_i \cdot w$ where $c_i \in cs$ and w is a word that can be read from q' in $\mathscr{A}$, then w' can be read from ι in $\mathscr{A}$. Moreover $\mathscr{A}[i]$ is an observational prediction of $\mathscr{A}$, then w' can be read also from $\iota[i]$, consequently w can be read from any $p_j \in ps$. □

Corollary 3.2.2. *Let $\mathscr{A}[i+1]$ be the LTS obtained from $\mathscr{A}[i]$ by replacing each*

$$h(q) \xrightarrow{c_j} p_j \text{ by } h(q) \xrightarrow{c_j} p$$

where $p \in ps$ and $p_j \in ps \setminus \{p\}$. Then $\mathscr{A}[i+1]$ is also an observational prediction of $\mathscr{A}$.

Proof. Since $\mathscr{A}[i]$ is an observational prediction of $\mathscr{A}$, by Lemma 3.2.2, any word $w \in \Sigma^*$ that can be read from q' in $\mathscr{A}$, can be read also from any state $p_j \in ps$ in $\mathscr{A}[i]$. Then the sequence of transitions obtained in reading any word in $\mathscr{A}[i+1]$ is the same as the one in $\mathscr{A}[i]$ except that any transition $h(q) \xrightarrow{c_j} p_j$ (if it exists) is replaced by $h(q) \xrightarrow{c_j} p$, so $\mathscr{A}[i+1]$ is still an observational prediction of $\mathscr{A}$. □

Corollary 3.2.3. *The LTS $\mathscr{A}[i]$ at the end of the i-th `while`-loop block is an observational prediction of $\mathscr{A}$.*

Proof. We will prove by induction on i. For the base case $i = 0$, we have $\mathscr{A}[0] = \mathscr{A}'$, then this is immediately true. Suppose that the corollary holds already for $\mathscr{A}[i-1]$ for some index $i \geq 1$, we need to prove that $\mathscr{A}[i]$ is also an observational prediction of $\mathscr{A}$. Indeed, the LTS $\mathscr{A}[i]$ is modified from $\mathscr{A}[i-1]$ inside the i-th `while`-loop block by the commands in Listing 3.2 (cf. also Figure 3.10).

```
...
𝒜[i+1] ← 𝒜[i];  i ← i+1;
...
foreach (p′ ∈ ps \ {p}) {
   replace h(q) –c→ p′ by h(q) –c→ p;
   remove unreachable states from 𝒜[i];
}
```

Listing 3.2: Construct $\mathscr{A}[i]$ by modifying $\mathscr{A}[i-1]$

Since $\mathscr{A}[i-1]$ is an observational prediction of $\mathscr{A}$, by Corollary 3.2.2, the LTS $\mathscr{A}[i]$ (before removing unreachable states) is an observational prediction of $\mathscr{A}$. And after removing unreachable states, it is still an observational prediction of $\mathscr{A}$. □

Now we give the proof for Theorem 2, that is a consequence of Corollaries 3.2.1 and 3.2.3 and the construction of the algorithm in Listing 3.1.

Proof. (of Theorem 2) We observe the assignment $h \leftarrow h[i]$ and $\mathscr{A}'' \leftarrow \mathscr{A}[i]$ at the end of the algorithm in Listing 3.1. We will prove that the LTS $\mathscr{A}''$ and the morphism h satisfy the requirement of Theorem 2. Indeed,

- by Corollary 3.2.1, the algorithm halts and $Q[i] = Q$, so $h = h[i]$ is a complete morphism $h \colon \mathscr{A} \to \mathscr{A}''$,
- by Corollary 3.2.3, $\mathscr{A}'' = \mathscr{A}[i]$ is an observational prediction of $\mathscr{A}$.

Moreover, if a state in $\mathscr{A}''$ that is not an image $h(q)$ of any state $q \in Q$, then it can be removed also from $\mathscr{A}''$, in other words h is a surjective complete morphism. Since some states have been removed from $\mathscr{A}'$ then $\mathscr{A}''$ has no more states than $\mathscr{A}'$. □

Remark 3.2.5. There is a simpler proof for Theorem 2 though it is not constructive as the proof given above. For each state $q \in Q$, we first calculate the set[1]:

$$H(q) = \{q' \in Q' \mid \exists w \; \delta(\iota, w) = q, \delta'(\iota', w) = q'\}$$

Notice that $H(q) \neq \emptyset$ since $\mathscr{A} \sqsubseteq_{op} \mathscr{A}'$, then we select a representative state $q' \in H(q)$. We modify the LTS $\mathscr{A}'$ by replacing any transition $q_1 \xrightarrow{c} q'_1$ where $q'_1 \in H(q)$ and $q'_1 \neq q'$, by the transition $q_1 \xrightarrow{c} q'$. It can be proved that the modified LTS $\mathscr{A}'$ still satisfies $\mathscr{A} \sqsubseteq_{op} \mathscr{A}'$ but now $H(q) = \{q'\}$, then we define $h(q) = q'$.

Let us repeat the procedure above for other states, the complete morphism h is defined for all $q \in Q$, moreover any state of $\mathscr{A}'$ that is not touched in this calculation can be safely removed to obtain a LTS $\mathscr{A}''$ having no more states than $\mathscr{A}'$. Then $h \colon \mathscr{A} \to \mathscr{A}''$ is a surjective complete morphism.

Summary We have introduced the *internal/external views* of the concept "observational prediction" in Sections 3.2.1 and 3.2.2. We have proved Theorem 1 (under the internal viewpoint) and Theorem 2 (under the external viewpoint) which give relations between a π-equivalence and a surjective complete morphism, and between a surjective complete morphism and an observational prediction. These results will be used in the following part of the section.

3.2.3 Labeled transition system reduction

The notion of the π-equivalence and one of the observational prediction aim to describe behaviors of an LTS in predicting an initial LTS. The former takes an *internal view*, it says that if some states of the initial LTS are equivalent, they can be merged to form a LTS keeping the same information, and more interestingly the new LTS can predict some unobservable behaviors. Whereas the latter takes an *external view*, it says nothing about the relation between states, but it gives more general conditions that a LTS should satisfy to become a correct predictor for the initial LTS.

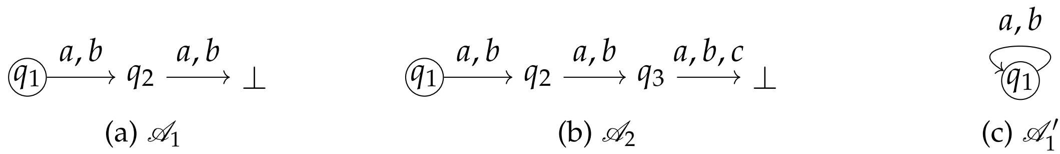
[1] Such a set is defined always because all states of $\mathscr{A}$ are reachable (as supposed in Remark 3.1.5).

What is a good predictive labeled transition system?

There is immediately a question *"how good the predictive LTS is?"* There are at least two requirements that come to mind 1. *the prediction capability*, and 2. *the compactness*. A predictor LTS may not be good if even though, it describes correctly the observed behaviors, it fails immediately where more behaviors are observed (cf. Example 3.2.8). Also, it is not good if it is voluminous (i.e. has more states than the original LTS), because any LTS is a trivial predictor of itself.

Example 3.2.8. We think of the LTS(s) $\mathscr{A}_1$ and $\mathscr{A}_2$ as partial observations of a unknown LTS $\mathscr{A}$, where $\mathscr{A}_2$ is more complete than $\mathscr{A}_1$. Suppose that we are given the LTS $\mathscr{A}_1$, we next create $\mathscr{A}_1'$ as the quotient of $\mathscr{A}_1$ by the π-equivalence $\sim = \{\{q_1, q_2, \perp\}\}$, then we may predict that $\mathscr{A}_1'$ is the unknown LTS $\mathscr{A}$.

However this prediction fails immediately given the more complete observation $\mathscr{A}_2$, because $\mathscr{A}_1'$ is not an observational prediction of $\mathscr{A}_2$. For example $\mathscr{A}_2$ can read the word $a \cdot b \cdot c$ while $\mathscr{A}_1'$ cannot.



(a) $\mathscr{A}_1$ (b) $\mathscr{A}_2$ (c) $\mathscr{A}_1'$

We have no answer yet for the requirement about the prediction capability, predicting correctly something which has not been observed completely yet is undecidable in general [134]. But for the requirement about the compactness, we will present a procedure constructing observational prediction LTS(s) which are more compact than the original LTS. In many cases, we can even obtain the "best compact possible form".

Remark 3.2.6. In considering the equivalence between LTS(s), the compactness requirement has been studied in researches about the optimization of DFA(s) [78]. The equivalence used in the DFA optimization means that two equivalent LTS(s) (i.e. two DFA(s) in this case) has the same language, it is then different from our definition of the observational prediction.

Observational prediction transitivity

The key idea of constructing a best compact LTS is to modify the original LTS by a sequence of observational prediction LTS(s). This is based on the following transitivity.

Proposition 3.2.5 (Observational prediction transitivity). *Given LTS(s) $\mathscr{A}_1, \mathscr{A}_2$ and $\mathscr{A}_3$, if $\mathscr{A}_1 \sqsubseteq_{op} \mathscr{A}_2$ and $\mathscr{A}_2 \sqsubseteq_{op} \mathscr{A}_3$ then $\mathscr{A}_1 \sqsubseteq_{op} \mathscr{A}_3$.*

Proof. Because $\mathscr{A}_1 \sqsubseteq_{op} \mathscr{A}_2$ and $\mathscr{A}_2 \sqsubseteq_{op} \mathscr{A}_3$, the three LTS(s) share the same alphabet, let

$$\Sigma = \Sigma(\mathscr{A}_1) = \Sigma(\mathscr{A}_2) = \Sigma(\mathscr{A}_3)$$

and let δ_1, δ_2 and δ_3 denote respectively the transition function $\delta(\mathscr{A}_1), \delta(\mathscr{A}_2)$ and $\delta(\mathscr{A}_3)$. We will verify the conditions in Definition 3.2.4 of the observational prediction.

For any word $w \in \Sigma^*$, because $\mathscr{A}_1 \sqsubseteq_{op} \mathscr{A}_2$, if $\delta_1(\iota_1, w)$ is defined then $\delta_2(\iota_2, w)$ is defined; because $\mathscr{A}_2 \sqsubseteq_{op} \mathscr{A}_3$, if $\delta_2(\iota_2, w)$ is defined then $\delta_3(\iota_3, w)$ is defined. So if $\delta_1(\iota_1, w)$ is defined then $\delta_3(\iota_3, w)$ is defined for any $w \in \Sigma^*$.

Let $w \in \Sigma^*$ be a word so that $\delta_3(\iota_3, w)$ is defined but $\delta_1(\iota_1, w)$ is not. If $\delta_2(\iota_2, w)$ is defined, since $\mathscr{A}_1 \sqsubseteq_{op} \mathscr{A}_2$, there is a unique prefix $\overline{w}$ of w so that $\delta_1(\iota_1, \overline{w})$ is a terminal state, and that gives immediately the result. If $\delta_2(\iota_2, w)$ is not defined, since $\mathscr{A}_2 \sqsubseteq_{op} \mathscr{A}_3$, there is a unique prefix $\overline{w}$ of w so that $\delta_2(\iota_2, \overline{w})$ is a terminal state. Since $\mathscr{A}_1 \sqsubseteq_{op} \mathscr{A}_2$, there must be two cases:

- if $\delta_1(\iota_1, \overline{w})$ is not defined (notice that $\delta_2(\iota_2, \overline{w})$ is defined) then there is a unique prefix $\overline{\overline{w}}$ of $\overline{w}$ so that $\delta_1\left(\iota_1, \overline{\overline{w}}\right)$ is a terminal state, and $\overline{\overline{w}}$ is also a prefix of w.

- if $\delta_1(\iota_1, \overline{w})$ is defined, since $\delta_2(\iota_2, \overline{w})$ is a terminal state, $\delta_1(\iota_1, \overline{w})$ must be also a terminal state (because if $\delta_1(\iota_1, \overline{w} \cdot c)$ is defined then so is $\delta_2(\iota_2, \overline{w} \cdot c)$ for any c).

In both cases, there is a unique prefix $\widetilde{w}$ of w (it will be $\overline{\overline{w}}$ for the former and $\overline{w}$ for the latter) so that $\delta_1(\iota_1, \widetilde{w})$ is a terminal state. $\square$

The following proposition gives a stronger result than Proposition 3.2.5 in a more strict case where the observational prediction is derived from a π-equivalence.

Proposition 3.2.6. *Given a LTS $\mathscr{A}$, let $\sim_1$ and $\sim_2$ be respectively a π-equivalence in $\mathscr{A}$ and in $\mathscr{A}/\!\sim_1$, then there is a π-equivalence $\sim$ in $\mathscr{A}$ so that $\mathscr{A}/\!\sim \,=\, \mathscr{A}/\!\sim_1/\!\sim_2$.*

Proof. Let $h_{\sim_1} : \mathscr{A} \to \mathscr{A}_1$ and $h_{\sim_2} : \mathscr{A}/\!\sim_1 \to \mathscr{A}/\!\sim_1/\!\sim_2$ be respectively the surjective complete morphism derived from $\sim_1$ and $\sim_2$ (cf. Proposition 3.2.2 and Remark 3.2.1).

$$\begin{array}{ccc} & \mathscr{A}/\!\sim_1 & \\ {}^{h_{\sim_1}}\nearrow & & \searrow^{h_{\sim_2}} \\ \mathscr{A} & \xrightarrow{\quad h_{\sim_2}\circ h_{\sim_1} \quad} & \mathscr{A}/\!\sim_1/\!\sim_2 \end{array}$$

Let $h = h_{\sim_2} \circ h_{\sim_1}$, by Proposition 3.1.1, h is a surjective complete morphism. Consequently, by Theorem 1, there is a π-equivalence $\sim$ in $\mathscr{A}$ so that its derived morphism $h_\sim$ satisfying $h_\sim = h$, and $\mathscr{A}/\!\sim \,=\, \mathscr{A}/\!\sim_1/\!\sim_2$. $\square$

Note 3.2.2. We use the operator $\otimes$ to denote $\sim = \sim_2 \otimes \sim_1$, this operator corresponds to the function composition $\circ$ in the construction of derived morphism $h_\sim = h_{\sim_2} \circ h_{\sim_1}$. Immediately, the composition by $\otimes$ is associative:

$$(\sim_3 \otimes \sim_2) \otimes \sim_1 = \sim_3 \otimes (\sim_2 \otimes \sim_1)$$

but not commutative in general. It is worth noting that the composition by $\otimes$ of π-equivalences is different from the normal composition of equivalence relations.

Remark 3.2.7. The π-equivalence $\sim$ in the proof above can be constructed directly from $\sim_1$ and $\sim_2$. Indeed, consider the relation $\sim$ in $\mathscr{A}$ being defined by

$$q \sim q' \iff [q]_{\sim_1} \sim_2 [q']_{\sim_1}$$

Because $\sim_1$ and $\sim_2$ are equivalences, $\sim$ is immediately an equivalence. We now verify the co-inductive property of $\sim$. Since if q or q' is a terminal state then nothing needs to be verified, it remains to verify the case where both are not terminal states.

For any transition $q \xrightarrow{c} q_1$ in $\mathscr{A}$, there is a corresponding transition $[q]_{\sim_1} \xrightarrow{c} [q_1]_{\sim_1}$ in $\mathscr{A}/\sim_1$. Because q' is not a terminal state and $[q]_{\sim_1} \sim_2 [q']_{\sim_1}$ by the definition above of the relation $\sim$, then

1. q' itself has a transition $q' \xrightarrow{c} q'_1$ in $\mathscr{A}$, and then
2. there is a corresponding transition $[q']_{\sim_1} \xrightarrow{c} [q'_1]_{\sim_1}$ in $\mathscr{A}_2$ so that $[q_1]_{\sim_1} \sim_2 [q'_1]_{\sim_1}$, and then $q_1 \sim q'_1$ from the definition of $\sim$.

In summary, we have proved that for any transition $q \xrightarrow{c} q_1$, there is $q' \xrightarrow{c} q'_1$ so that $q_1 \sim q'_1$. Similarly, we can obtain the same conclusion for any transition $q' \xrightarrow{c} q'_1$. Thus $\sim$ is an π-equivalence.

Remark 3.2.8. The explicit definition of $\sim$ and its proof in Remark 3.2.7 use the co-inductive technique. That is derived naturally from the fact that the π-equivalence is co-inductively defined (cf. Definition 3.2.2). In such a definition (and proof), we do not see how the relation is constructed, instead we see how it is destroyed. We refer to [140] for discussions about the naturalness of the co-induction.

Reduction sequence

By Proposition 3.2.4 and Theorem 1, if $\sim$ is a π-equivalence in a LTS $\mathscr{A}$ then $\mathscr{A} \sqsubseteq_{op} \mathscr{A}/\sim$. Moreover if $\sim$ is nontrivial[1], then $\mathscr{A}/\sim$ has strictly fewer state than $\mathscr{A}$. A best compact possible observational prediction of $\mathscr{A}$ is obtained through a sequence of observational prediction LTS(s).

Definition 3.2.5 (Reduction sequence)**.** Given a LTS $\mathscr{A}$, a reduction sequence derived from $\mathscr{A}$ consists of observational prediction LTS(s)

$$\mathscr{A}_1 \sqsubseteq_{op} \mathscr{A}_2 \sqsubseteq_{op} \cdots \sqsubseteq_{op} \mathscr{A}_n$$

where $\mathscr{A}_1 = \mathscr{A}$, and $\mathscr{A}_{i+1} = \mathscr{A}_i/\sim_i$ with $\sim_i$ is a nontrivial π-equivalence in $\mathscr{A}_i$ for $i = 1, 2, \ldots, n-1$, moreover there is no nontrivial π-equivalence in $\mathscr{A}_n$.

[1]There is a unique trivial π-equivalence in any LTS, that is the identity relation.

Normal forms The last LTS of a reduction sequence derived from $\mathscr{A}$ is called a *normal form* of $\mathscr{A}$ (e.g. the LTS $\mathscr{A}_n$ above is a normal form of $\mathscr{A}$), other LTS(s) (except $\mathscr{A}$) of the sequence are called intermediate LTS(s).

The following result about the normal form of a LTS is a direct consequence of Proposition 3.2.6, it says that for any reduction consequence derived from a LTS $\mathscr{A}$ which has the last LTS $\mathscr{A}_n$ and consists of several intermediate LTS(s), then there is a reduction consequence derived from $\mathscr{A}$ having also the last LTS $\mathscr{A}_n$ but without any intermediate LTS(s).

Proposition 3.2.7. *Given a reduction sequence $\mathscr{A}_1 \sqsubseteq_{op} \mathscr{A}_2 \sqsubseteq_{op} \cdots \sqsubseteq_{op} \mathscr{A}_n$ derived from the LTS $\mathscr{A} = \mathscr{A}_1$, then there is a direct reduction sequence $\mathscr{A}_1 \sqsubseteq_{op} \mathscr{A}_n$.*

Proof. By Proposition 3.2.6, using the notion of composition $\otimes$ (cf. Note 3.2.2), let

$$\sim = \sim_{n-1} \otimes \sim_{n-2} \otimes \cdots \otimes \sim_1$$

then $\mathscr{A}_n = \mathscr{A} / \sim$ and that gives the result. □

Example 3.2.9. We reconsider in Figure 3.12 the LTS $\mathscr{A}$ in Example 3.2.2. Some reduction sequences can be derived from $\mathscr{A}$ are as follows

- $\mathscr{A}_1 \sqsubseteq_{op} \mathscr{A}_2 \sqsubseteq_{op} \mathscr{A}_3 \sqsubseteq_{op} \mathscr{A}_4$ (cf. Figure 3.13) where $\mathscr{A}_1 = \mathscr{A}$ and
  - $\sim_1 = \{\{q_1\}, \{q_2, \bot_1\}, \{\bot_2, \bot_3, \bot_4\}, \{\bot_5, \bot_6\}\}$ in $\mathscr{A}_1$ (cf. Figure 3.12),
  - $\sim_2 = \{\{q_1\}, \{\bot_{234}, \bot_{56}\}\}$ in $\mathscr{A}_2$ (cf. Figure 3.13a),
  - $\sim_3 = \{\{q_1\}, \{q_2 \bot_1\}, \{\bot_{23456}, q_3\}\}$ in $\mathscr{A}_3$ (cf. Figure 3.13b).
- $\mathscr{A}'_1 \sqsubseteq_{op} \mathscr{A}'_2 \sqsubseteq_{op} \mathscr{A}'_3$ (cf. Figure 3.14) where $\mathscr{A}'_1 = \mathscr{A}$ and
  - $\sim'_1 = \{\{\bot_1, q_1\}, \{\bot_2, \bot_3, \bot_4, \bot_5, \bot_6\}\}$ in $\mathscr{A}'_1$ (cf. Figure 3.12),
  - $\sim'_1 = \{\{q_2\}, \{q_3\}, \{q_1, \bot_{23456}\}\}$ in $\mathscr{A}'_1$ (cf. Figure 3.14a).
- $\mathscr{A}''_1 \sqsubseteq_{op} \mathscr{A}''_2$ (cf. Figure 3.15) where $\mathscr{A}''_1 = \mathscr{A}$ and
  - $\sim''_1 = \{\{q_1, \bot_1\}, \{q_2, \bot_2, \bot_3, \bot_4\}, \{q_3, \bot_5, \bot_6\}\}$ in $\mathscr{A}''_1$ (cf. Figure 3.12)

We can observe that each reduction sequence has its own last LTS, then the LTS $\mathscr{A}$ have several normal forms, for example $\mathscr{A}_4$ (cf. Figure 3.13c), $\mathscr{A}'_3$ (cf. Figure 3.14b) and $\mathscr{A}''_2$ (cf. Figure 3.15a) are different normal forms of $\mathscr{A}$.

Remark 3.2.9. Hereafter, we use the cardinality notation $|\mathscr{A}|$ for the size (i.e. the number of states) of a LTS $\mathscr{A}$. This size can be also denoted by $|Q(\mathscr{A})|$.

Proposition 3.2.8. *For a LTS $\mathscr{A}$, any reduction sequence derived from $\mathscr{A}$ is finite.*

Proof. The number of states of each LTS in a reduction sequence decreases strictly, because $|\mathscr{A}|$ is finite, the result follows immediately. □

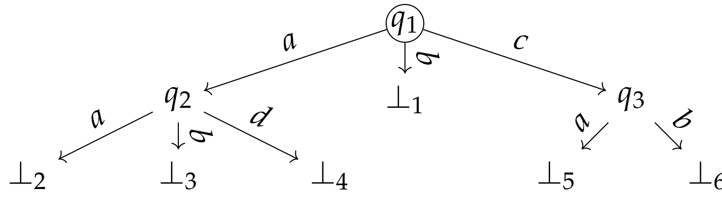
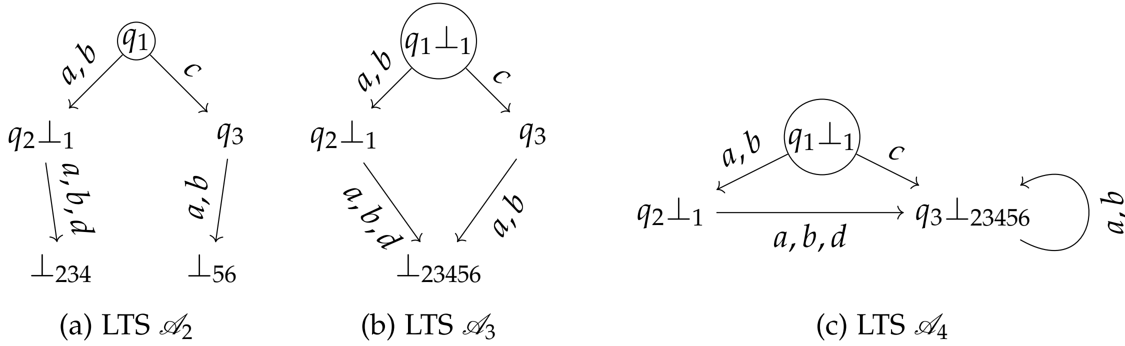


Figure 3.12: Original LTS $\mathscr{A} = \mathscr{A}_1 = \mathscr{A}_1' = \mathscr{A}_1''$



(a) LTS $\mathscr{A}_2$ (b) LTS $\mathscr{A}_3$ (c) LTS $\mathscr{A}_4$

Figure 3.13: Reduction sequence $\mathscr{A}_1 \sqsubseteq_{op} \mathscr{A}_2 \sqsubseteq_{op} \mathscr{A}_3 \sqsubseteq_{op} \mathscr{A}_4$

Variety of normal forms We have observed in Example 3.2.9 that a LTS can have different normal forms, but have the same number of states. But actually we have even no hope that the normal forms have the same number of states. Since giving a counter-example is somehow not trivial, we give first a proposition that specifies a necessary and sufficient condition for a LTS which is irreducible by a reduction sequence, namely this LTS is a normal form.

Proposition 3.2.9. *Given a LTS $\mathcal{A}$ and a π-equivalence $\sim$, if there exist two equivalence classes Q_i and Q_j of the quotient set $Q(\mathcal{A})/\sim = \{Q_1, Q_2, \ldots, Q_n\}$ satisfying: for any state $q_i \in Q_i$ and $q_j \in Q_j$, there exists always a π-equivalence $\sim'$ in $\mathcal{A}$ so that*

$$q_1 \sim' q_2$$

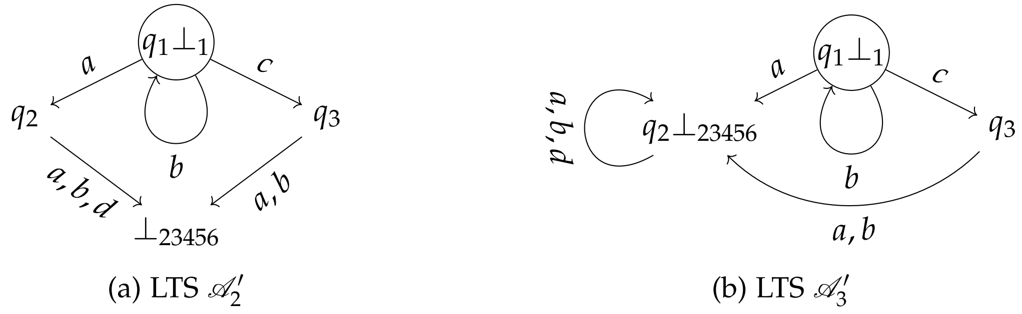
then $\mathcal{A}/\sim$ is reducible by a reduction sequence. Otherwise $\mathcal{A}/\sim$ is irreducible.

Proof. By Propositions 3.2.2 and 3.2.7, each state of the quotient LTS $\mathscr{A}/\sim$ corresponds to a unique and distinguished class equivalence of the quotient set $Q(\mathcal{A})/\sim$. We will use the equivalence classes $Q_1, Q_2, \ldots, Q_n$ to denote the state of $\mathscr{A}/\sim$.

Suppose that there are Q_i and Q_j satisfying the condition in the proposition. We construct a symmetric-transitive relation R as follows

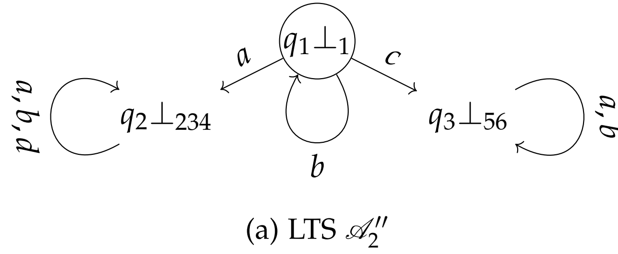
- $R_1 = \{(Q_i, Q_j), (Q_j, Q_i)\}$,
- calculate
  - $R_{i+1}^1 = \{(Q_k', Q_l') \mid \exists c, (Q_k, Q_l) \in R_i : \delta_\sim(Q_k, c) = Q_k',\ \delta_\sim(Q_l, c) = Q_l'\}$[1],

[1] $\delta_\sim$ is the transition function of the quotient LTS $\mathcal{A}/\sim$.



(a) LTS $\mathscr{A}_2'$ (b) LTS $\mathscr{A}_3'$

Figure 3.14: Reduction sequence $\mathscr{A}_1' \sqsubseteq_{op} \mathscr{A}_2' \sqsubseteq_{op} \mathscr{A}_3'$



(a) LTS $\mathscr{A}_2''$

Figure 3.15: Reduction sequence $\mathscr{A}_1'' \sqsubseteq_{op} \mathscr{A}_2''$

– $R^2_{i+1} = R_i \cup R^1_{i+1}$, then
– let R_{i+1} be the transitive closure of R^2_{i+1}.

for $i = 1, 2 \ldots$ until a value n where $R_n = R_{n+1}$, then let $R = R_n$.

By the construction, R is an equivalence relation. We can also verify the conditions of π-equivalence along construction steps of R to see that R is a π-equivalence. □

Remark 3.2.10. In Definition 3.2.5, a LTS is a normal form if and only if it does not contain nontrivial π-equivalence. In Proposition 3.2.9, we gives another condition to verify whether a LTS is a normal form or not. It is worth noting that the condition works on the initial LTS $\mathcal{A}$ and not $\mathcal{A}/\sim$, then it can be verified directly in the initial LTS of a reduction sequence.

By Proposition 3.2.9, we have an immediate corollary that will support some reasoning about counter-examples for normal forms of different number of states.

Corollary 3.2.4. *Given a LTS $\mathcal{A}$ and a π-equivalence $\sim$ in $\mathscr{A}$. If for any pair of equivalence classes Q_i and Q_j of $\sim$, we can always find states $q_i \in Q_i$ and $q_j \in Q_j$ so that $q_i \not\sim' q_j$ for all π-equivalence $\sim'$ in $\mathscr{A}$, then $\mathcal{A}/\sim$ is a normal form.*

The corollary means that whenever we construct a π-equivalence $\sim$ satisfying: for any pair of equivalence classes of $\sim$, there is a state in one and there is another state in the other so that these two states are not equivalent for any π-equivalence. Then we can get a normal form as the quotient of the original LTS by $\sim$.

Moreover, we know that the number states of the normal form is also the number of equivalence classes of $\sim$. So a trick is to find different partitions of Q satisfying Corollary 3.2.4 so that they have different number of classes. For example if we have a set of states $Q = \{q_a, q_b, q_c, q_d\}$ satisfying the following configuration[1]

$$\begin{array}{ll} q_c \sim_1 q_a & q_a \neq q_b \\ q_c \sim_1 q_b & q_d \neq q_c \\ q_d \sim_2 q_a & q_d \neq q_b \end{array}$$

then we can find two π-equivalences of different number (one has 4 and the other has 3) of equivalence classes

$$\{\{q_A, q_C\}, \{q_B\}, \{q_D\}\} \text{ and } \{\{q_A, q_D\}, \{q_B, q_C\}\}$$

Remark 3.2.11. The configuration above is not possible if $\sim_1$ and $\sim_2$ are bisimulations, for example the following situation cannot happen

$$q_c \sim_1 q_a \quad q_c \sim_1 q_b \quad q_a \neq q_b$$

because the bisimulation is transitive. Concretely, from $q_a \sim_1 q_c$ and $q_c \sim_1 q_b$, we have $q_a \sim_1' q_b$ where $\sim_1'$ is the relational composition

$$\sim_1' = \{(q, q') \mid \exists q'' : q \sim_1 q'' \text{ and } q'' \sim_1 q'\}$$

which is also a bisimulation [114].

Example 3.2.10. The LTS in Figure 3.16 consists of 4 sub-LTS(s) Q_a, Q_b, Q_c, Q_d given in Figures 3.17a to 3.17d, respectively. There are two normal forms corresponding to two π-equivalences

$$\begin{aligned} \sim_1 = \quad & \{\{q_a, q_c\}, \{q_b\}, \{q_d\}, \{q_e^a, q_e^c, q_e^d\}, \{q_f^a, q_f^c\}, \{q_g^a, q_g^c, q_g^d \bot_2^a, \bot_2^c\}, \{q_h^c, \bot_1^a, \bot_1^c\} \\ & \{q_e^b\}, \{q_f^b, q_f^d\}, \{q_g^b\}, \{q_h^d, \bot_1^b, \bot_1^d\}, \{q_1, \bot_3^a, \bot_3^b, \bot_3^c, \bot_3^d, \bot_4^d\}\} \\ \sim_2 = \quad & \{\{q_a, q_d\}, \{q_b, q_c\}, \{q_a^e, q_d^e\}, \{q_a^f, q_d^f\}, \{q_a^g, q_d^g\}, \{\bot_1^a, q_h^d, \bot_1^d\}, \\ & \{q_e^b, q_e^c\}, \{q_f^b, q_f^c\}, \{q_g^b, \bot_2^c, \bot_2^b\}, \{q_h^c, \bot_1^c, \bot_1^b\}, \{q_1, \bot_2^a, \bot_2^d, \bot_4^d, \bot_3^b, \bot_c^3\}\} \end{aligned}$$

where $\sim_1$ has 12 equivalence classes and $\sim_2$ has 11. Consequently, these normal forms has respectively 12 and 11 states.

However, in practice we may observe that the numbers of states of normal forms are quite stable. That is because a π-equivalence is very easy to become a bisimulation. Indeed, for a LTS having no terminal states then, by Definition 3.2.2, π-equivalence is actually a bisimulation, moreover it is an equivalence relation. We call such a relation a *bisimulation-equivalence*. If we consider reduction sequences beginning from a LTS

[1]Here, the notation $\neq$ means that two states are not equivalent for any π-equivalence.

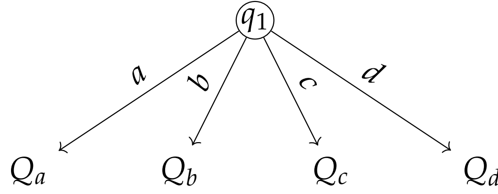


Figure 3.16: LTS having normal forms with different number of states

having no terminal states, then the normal forms of this LTS always have the same number of states.

Proposition 3.2.10. *Given a LTS $\mathscr{A}$ having no terminal states, then all normal forms of $\mathscr{A}$ have the same number of states. Concretely, if $\mathscr{A}_{n_1}$ and $\mathscr{A}_{n_2}$ are some normal forms of $\mathscr{A}$, then*

$$|\mathscr{A}_{n_1}| = |\mathscr{A}_{n_2}|$$

By Definition 3.2.1, the bisimilarity is defined as the union of all bisimulation relations in a labeled transition system. The bisimilarity is an equivalence relation, and also is a bisimulation itself. It is the unique largest bisimulation [114, 140]. Similarly, we can define the unique largest bisimulation-equivalence as the transitive closure of the union of all bisimulation-equivalences. In Lemmas 3.2.3 and 3.2.4 below, we assume that $\mathcal{A}$ has no terminal states.

Lemma 3.2.3. *The union of all bisimulation-equivalences in $\mathcal{A}$ is a bisimulation-equivalence.*

Proof. Let $\sim_u$ denote the union of all bisimulation-equivalence in $\mathcal{A}$. To prove $\sim_u$ is a bisimulation-equivalence, let us consider two bisimulation-equivalences $\sim_1$ and $\sim_2$ and its relational composition

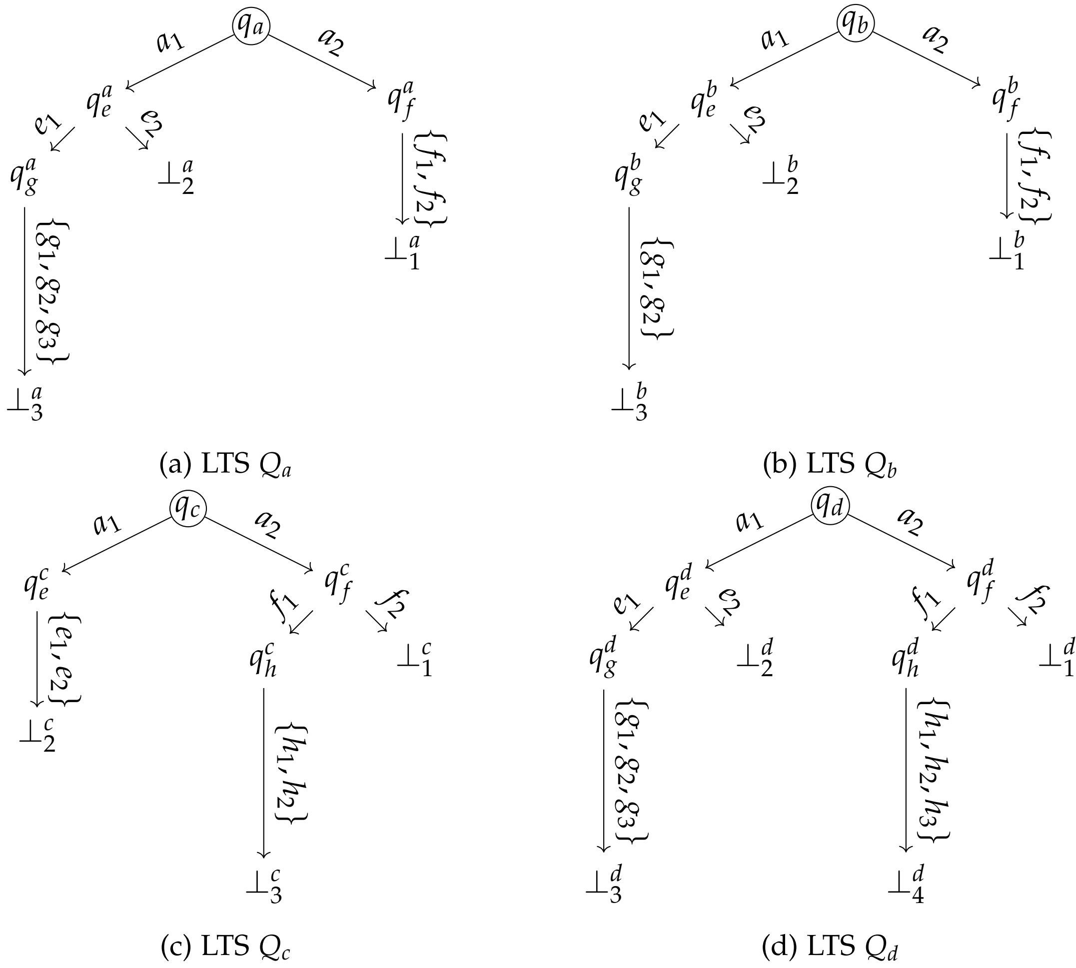
$$\sim_1 \circ \sim_2 = \{(q, q') \mid \exists q'' : q \sim_1 q'' \text{ and } q'' \sim_2 q'\}$$

Let $\sim_1 \odot \sim_2$ denote the reflexive-symmetric-transitive of $\sim_1 \circ \sim_2$, then $\sim_1 \odot \sim_2$ is immediately a bisimulation-equivalence, consequently $\sim_1 \odot \sim_2 \subseteq \sim_u$. And that gives the result. □

Remark 3.2.12. By definition, the bisimilarity is a bisimulation and an equivalence relation [114, 140]. In other words, the bisimilarity is a bisimulation-equivalence, and is also the unique largest bisimulation-equivalence. This argument gives a direct proof for Lemma 3.2.3.

The technical idea of the proof of Proposition 3.2.10 is to prove that the number of states of a normal form is the number equivalence classes of $Q(\mathcal{A})$ by the unique largest bisimulation-equivalence $\sim_u$.

Lemma 3.2.4. *Given a LTS $\mathcal{A}$ and a π-equivalence $\sim$ in $\mathscr{A}$, if $\sim \neq \sim_u$ then there is nontrivial π-equivalence in the quotient LTS $\mathscr{A}/\sim$. Namely, $\mathscr{A}/\sim$ can be reduced again by a reduction sequence.*



(a) LTS Q_a (b) LTS Q_b (c) LTS Q_c (d) LTS Q_d

Figure 3.17: Sub LTS(s)

Proof. We notice that each state in $\mathscr{A}/\sim$ corresponds to a classes equivalence of $\sim$ in the set of all states Q of $\mathscr{A}$. Consider the following relation in $Q/\sim$

$$[q]_\sim \sim' [q']_\sim \iff \text{there are } q_1 \in [q]_\sim \text{ and } q_1' \in [q']_\sim \text{ so that } q_1 \approx_u q_1'$$

By Lemma 3.2.3 and the supposition $\sim \neq \sim_u$ of Lemma 3.2.4, we have $\sim$ is strictly more rough than $\sim_u$, then $\sim'$ is a nontrivial π-equivalence. Namely, $\mathcal{A}/\sim$ can be reduced again by a reduction sequence. □

Proof. (of Proposition 3.2.10) By Proposition 3.2.7, any LTS in a reduction sequence of $\mathcal{A}$ is the quotient LTS $\mathcal{A}/\sim$ for some π-equivalence $\sim$. By Lemma 3.2.4, if $\sim \neq \sim_u$ then $\mathcal{A}/\sim$ can be reduced again. By Proposition 3.2.8, any reduction sequence is finite, then any normal form must be $\mathcal{A}/\sim_u$ since it is a irreducible LTS. Moreover $\sim_u$ is unique, and that gives the result. □

Now let us back to the general case where the considered LTS(s) have terminal states, the following result is a direct consequence of Proposition 3.2.10. It explains

why the number of states of a normal form is "mostly" stable.

Proposition 3.2.11. *Given reduction sequences Seq_1 and Seq_2, not necessarily beginning from the same LTS. If there is a LTS $\mathcal{A}$ having no terminal states in both Seq_1 and Seq_2, then the last LTS(s) of Seq_1 and of Seq_2 have the same number of states.*

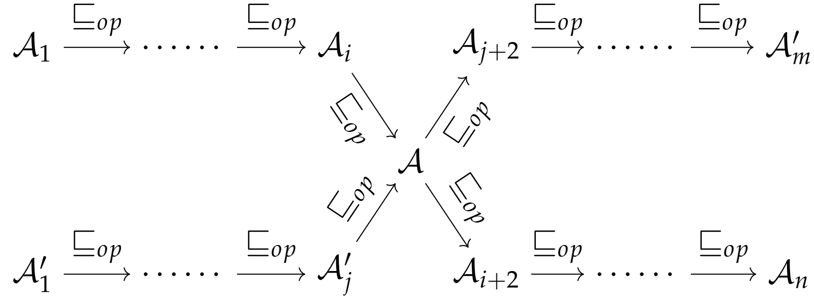


Figure 3.18: Intersected reduction sequences

Proof. The situation of the proposition is illustrated in Figure 3.18. Two reduction sequences beginning from different LTS(s) $\mathcal{A}_1$ and $\mathcal{A}'_1$, they have a common LTS $\mathcal{A}$ which has no terminal states.

$$Seq_1 \triangleq \mathcal{A}_1 \sqsubseteq_{op} \cdots \sqsubseteq_{op} \mathcal{A}_i \sqsubseteq_{op} \mathcal{A} \sqsubseteq_{op} \mathcal{A}_{i+2} \sqsubseteq_{op} \cdots \sqsubseteq_{op} \mathcal{A}_n$$
$$Seq_2 \triangleq \mathcal{A}'_1 \sqsubseteq_{op} \cdots \sqsubseteq_{op} \mathcal{A}'_j \sqsubseteq_{op} \mathcal{A} \sqsubseteq_{op} \mathcal{A}'_{j+2} \sqsubseteq_{op} \cdots \sqsubseteq_{op} \mathcal{A}'_m$$

By Proposition 3.2.10, the normal forms $\mathcal{A}_n$ and $\mathcal{A}'_m$ have the same number of states, that is the number of equivalence classes of $Q(\mathcal{A})/\sim$ where $\sim$ is the unique largest bisimulation-equivalence in $Q(\mathcal{A})$. □

Best compact possible labeled transition system

The following result says that the best compact possible observational prediction will be some normal form.

Proposition 3.2.12. *Given a LTS $\mathscr{A}$, let $\mathscr{A}_{min}$ be an observational prediction of $\mathscr{A}$ of the minimal number of states, then there exists a normal form $\mathcal{A}_n$ of $\mathcal{A}$ so that*

$$|\mathscr{A}_{min}| = |\mathscr{A}_n|$$

Proof. Since $\mathscr{A} \sqsubseteq_{op} \mathscr{A}_{min}$, by Theorem 2, there is a LTS $\mathscr{A}'$ having no more state than $\mathscr{A}_{min}$ and a surjective complete morphism $h\colon \mathscr{A} \to \mathscr{A}'$. Moreover $\mathscr{A}_{min}$ is already of the minimal number of state then

$$|\mathscr{A}'| = |\mathscr{A}_{min}|$$

and then there is no nontrivial π-equivalence in $\mathscr{A}'$. Since h is a surjective complete morphism, by Theorem 1, there is a π-equivalence $\sim$ in $\mathscr{A}$ so that $h_{\sim} = h$, that leads to a direct reduction sequence

$$\mathscr{A} \sqsubseteq_{op} \mathscr{A}'$$

That means $\mathscr{A}'$ is also a normal form of $\mathscr{A}$, let $\mathcal{A}_n = \mathcal{A}'$ and that gives the result. □

In fact, if we do not count the number of states as the criteria to define which normal form is better, then it will be hard to define which normal form is better in the sense of observational prediction. That is because of one normal form is may not an observational prediction of another normal form (cf. also Proposition 3.2.10 and example 3.2.9), in other words the normal forms are not comparable in the sense of observational prediction. If we consider the set of all observational predictions of a LTS with $\sqsubseteq_{op}$ as the order in this set, then we obtain a poset where normal forms are *maximal elements*. In general, we have the following result.

Proposition 3.2.13. *Given a LTS $\mathcal{A}$ and two normal forms $\mathcal{A}_{n_1}$ and $\mathcal{A}_{n_2}$ of $\mathcal{A}$ satisfying*

$$|\mathcal{A}_{n_1}| > |\mathcal{A}_{n_2}|$$

then $\mathcal{A}_{n_1} \not\sqsubseteq_{op} \mathcal{A}_{n_2}$.

Proof. Suppose that $\mathcal{A}_{n_1} \sqsubseteq_{op} \mathcal{A}_{n_2}$, by Theorem 2, there is a surjective complete morphism $h\colon \mathcal{A}_{n_1} \to \mathcal{A}'_{n_2}$ where $\mathcal{A}'_{n_2}$ is some LTS having no more states than $\mathcal{A}_{n_2}$. Consequently, we have $|\mathcal{A}_{n_1}| > |\mathcal{A}'_{n_2}|$. By Proposition 3.2.3, there is a nontrivial π-equivalence $\sim$ in $\mathcal{A}_{n_1}$ so that $\mathcal{A}_{n_1}/\sim\, = \mathcal{A}'_{n_2}$, that means $\mathcal{A}_{n_1}$ can be reduced again by a reduction sequence. This contradiction gives the result. □

Summary In Section 3.2.3, we have introduced reduction sequences and the normal forms of a LTS $\mathcal{A}$. The results obtained assert that the best compact possible observational prediction of $\mathcal{A}$ must be one of the normal forms of $\mathcal{A}$.

3.3 Algorithms

This section can be considered as the calculation part, based on results developed in Section 3.2. We will give algorithms to calculate a normal form of a given labeled transition system. Let us first recall some obtained theoretical results.

1. by Definition 3.2.4, to simulate correctly a given LTS $\mathscr{A}$, a LTS must be an observational prediction of $\mathscr{A}$,

2. by Propositions 3.2.2 and 3.2.4, such an observational prediction can be obtained as a quotient $\mathscr{A}/\sim$ of $\mathscr{A}$ by some nontrivial π-equivalence $\sim$ (recall that a π-equivalence is trivial if and only if it is the identity relation), and

3. by Proposition 3.2.12, the best compact possible observational prediction is always some normal form of $\mathcal{A}$, that can be obtained from a reduction sequence. Though we do not know exactly which reduction sequence gives the best compact prediction, by Proposition 3.2.11 we know that we still have a high chance to obtain it.

3.3.1 Compact labeled transition system construction

Straightforwardly from Definition 3.2.5, we have the algorithm in Listing 3.3 producing a normal form (i.e. a best compact possible observational prediction) of a labeled transition system.

```
LTSReduction(LTS 𝒜) {
  𝒜' ← 𝒜;
  while (true) {
    ∼ ← StateEquivCons(𝒜');
    if (∼ is trivial) break; // no more nontrivial π-equivalence
    else 𝒜' ← 𝒜'/∼;
  }
  return 𝒜';
}
```

Listing 3.3: Label transition system reduction algorithm

Let us start from the initial LTS $\mathscr{A}$, then look for a nontrivial π-equivalence $\sim$ in $\mathscr{A}$, we obtain next the quotient LTS $\mathscr{A}/\sim$ which has strictly fewer states than $\mathscr{A}$. This procedure is repeated with $\mathscr{A}$ is replaced by $\mathscr{A}/\sim$ until no more nontrivial π-equivalence is found.

Remark 3.3.1. Because the π-equivalences can be selected arbitrarily at each step, the returned labeled transition systems of Algorithm 3.3 are not unique (cf. Example 3.2.9), but their numbers of states are "mostly" stable as given by Proposition 3.2.11.

3.3.2 Π-equivalence verification and construction

The crucial point of the LTS reduction algorithm is the procedure `StateEquivCons` which returns a nontrivial π-equivalence in a LTS $\mathscr{A}$ whenever such a relation exists in $\mathscr{A}$. The co-inductive definition of the π-equivalence (cf. Definition 3.2.2) is not constructive: essentially it does not say about how to construct such a relation, instead it say about how to verify whether a given relation is a π-equivalence or not.

We may note that the number of states in $\mathscr{A}$ is finite, then the number of binary relations is also finite. Consequently, a verification algorithm, by checking each subset of $Q(\mathscr{A}) \times Q(\mathscr{A})$, gives indirectly a construction algorithm. However, this direct approach is practically infeasible because there are $2^{Q(\mathscr{A}) \times Q(\mathscr{A})}$ subsets.

In Listing 3.4, we use a slightly different approach by first introduce a verification algorithm and then describe how to modify it to get a construction algorithm.

```
StateEquivVerif(equivalence R) {
  R_unv ← {(q,q'),(q,q') ∈ R | q ≠ q',q ≠ ⊥,q' ≠ ⊥}; // the set of unverified pairs
  R_ved ← R \ R_unv; // the set of verified pairs

  while (R_unv ≠ ∅) {
    (q,q') ← an element in R_unv;
    if (for each q -c-> q_1 there exists q' -c-> q'_1 and vice-versa) {
      R_ved ← R_ved ∪ {(q,q'),(q',q)}; R_unv ← R_unv \ {(q,q'),(q',q)};

      R^l ← {(q_1,q'_1),(q'_1,q_1) | ∃c: q -c-> q_1, q' -c-> q'_1};
      R^l_ved ← {(q,q') ∈ R_l | either p = q, or p = ⊥ or p' = ⊥};
      R_ved ← R_ved ∪ R^l_ved; R_unv ← R_unv ∪ (R^l \ R_ved);

      if (R_ved ∪ R_unv ≠ R) return false;
    }
    else return false;
  }
  return true;
}
```

Listing 3.4: π-equivalence verification algorithm

Remark 3.3.2. The input R of the algorithm in Listing 3.4 is an equivalence relation. To make the algorithm work with any relation, we can simply add a pre-processing procedure to verify whether R is an equivalence relation or not before passing R to the procedure `StateEquivVerif`. Also in the presentation of the algorithm, we write $q = \perp$ or $q \neq \perp$ to mean that the state q is or is not a terminal state, correspondingly.

Algorithm 3.4 verifies whether the input R (as an equivalence relation) is a π-equivalence or not. The sets R_{unv} and R_{ved} contains respectively the unverified and verified pairs of states of R. Since the verification of reflexive pairs or pair containing some terminal states is not necessary, these sets are initialized by

$$R_{unv} \leftarrow \{(q,q'),(q,q') \in R \mid q \neq q', q \neq \perp, q' \neq \perp\} \text{ and } R_{ved} \leftarrow R \setminus R_{unv}$$

Whenever there are still some pairs need to be verified (this is determined by checking whether R_{unv} is empty or not), then a pair (q,q') is taken out from R_{unv}.

Next, the states q and q' are verified whether they satisfy the conditions of the π-equivalence or not. Since $q \neq q'$ and both are not terminal states, the verification means checking:

- for any transition $q \xrightarrow{c} q_1$, there exists also $q' \xrightarrow{c} q'_1$ for some states q_1 and q'_1, and
- that is also true for any transition from q'.

If not then we can conclude immediately that R is not a π-equivalence. Otherwise, the sets R_{unv} and R_{ved} are updated to reflect that the pairs (p,p') and (p',p) are verified:

$$R_{unv} \leftarrow R_{unv} \setminus \{(q,q'),(p',p)\} \text{ and } R_{ved} \leftarrow R_{ved} \cup \{(q,q'),(p',p)\}$$

We construct the local set R^l of corresponding targets from q and q':

$$R^l \leftarrow \{(q_1, q'_1), (q'_1, q_1) \mid \exists c\colon\ q \xrightarrow{c} q_1, q' \xrightarrow{c} q'_1\}$$

Some pairs in R^l do not need to be verified (such pairs are reflexive or contain some $\perp$ states), they are selected into a local set

$$R^l_{ved} \leftarrow \{(q, q') \in R_l \mid \text{either } p = q, \text{ or } p = \perp \text{ or } p' = \perp\}$$

and the set R_{ved} is updated: $R_{ved} \leftarrow R_{ved} \cup R^l_{ved}$. The local set of unverified pairs is determined by $R^l \setminus R_{ved}$, then R_{unv} is updated:

$$R_{unv} \leftarrow R_{unv} \cup \left(R^l \setminus R_{ved}\right)$$

At this point, both R_{ved} and R_{unv} are updated with new "local information", that is the set R^l of corresponding targets from q and q', then we verify whether the invariant

$$R_{ved} \cup R_{unv} = R$$

is preserved or not. If not, there are some unverified pairs which should belong to R but they are actually not, then R cannot be a π-equivalence. If the verification can reach to the point where R_{unv} is empty, namely all pairs are verified and passed, then R is a π-equivalence.

Lemma 3.3.1. *Given an equivalence relation R, the procedure `StateEquivVerif` of the input R halts always, and it returns `true` if and only if R is a π-equivalence.*

Proof. Let R^i_{unv} and R^i_{ved} denote values of the set R_{unv} and R_{ved} at the beginning of i-th while-step. We prove the invariants

$$R = R^i_{unv} \cup R^i_{ved} \text{ and } |R^{i+1}_{ved}| = |R^i_{ved}| - 2 \text{ for all } i$$

Indeed, this is already true for $i = 1$. This is also true at the beginning of the $(i+1)$-th while-step, because the commands in the if-condition of the i-th while-step have been fully executed. The invariants are proved

Since $|R^{i+1}_{ved}| = |R^i_{ved}| - 2$, `StateEquivVerif` (R) halts after no more than $|R|/2$ while-step. We observe that the commands inside each while-step verify the conditions of the π-equivalence (cf. Definition 3.2.2). Moreover, `StateEquivVerif` (R) returns true if and only if it halts at $R \setminus R_{ved} = R_{unv} = \emptyset$, that means all pairs in R are verified and passed. □

Corollary 3.3.1. *The worst-case (time) complexity of Algorithm 3.4 is $\mathcal{O}(|R|)$.*

In Algorithm 3.4, if the verification of the invariant $R = R_{ved} \cup R_{unv}$ (as the last command of each while-step) is always executed and passed then R is eventually a

π-equivalence. Consequently, instead of keeping the relation R as a fixed input, if we update it by

$$R \leftarrow R_{ved} \cup R_{unv}$$

then Algorithm 3.4 becomes automatically a construction algorithm, but to make it return an equivalence relation, the local set R_l need to be set appropriately.

```
StateEquivCons(LTS 𝒜) {
  R ← {(q,q) | q ∈ Q(𝒜)}; // the trivial π-equivalence

  foreach ((q,q') ∈ Q(𝒜) × Q(𝒜), q ≠ q') {
    R_unv ← {(q,q'), (q',q)}; // the set of unverified pairs
    R_ved ← {(q,q) | q ∈ Q(𝒜)}; // the set of verified pairs
    R ← R_ved;

    while (R_unv ≠ ∅) {
      (q,q') ← an element of R_unv;
      if (for each q -c-> q_1 there exists q' -c-> q' and vice-versa) {
        R_unv ← R_unv \ {(q,q'), (q',q)}; R_ved ← R_ved ∪ {(q,q'), (q,q')};

        R_l ← {(q_1,q'_1), (q'_1,q_1) | ∃c : q -c-> q_1, q' -c-> q'_1};
        R_l ← R_l \ R_ved; R_unv ← R_unv ∪ R_l;

        R ← R_ved ∪ R_unv; R ← transitive-symmetric closure of R;
        R_unv ← R \ R_ved;
      }
      else break;
    } // end of while

    if (R_unv = ∅) return R;
  } // end of foreach

  return R;
}
```

Listing 3.5: π-equivalence construction algorithm

The algorithm in Listing 3.5 returns a nontrivial π-equivalence whenever such a relation exists in the input LTS $\mathscr{A}$, otherwise it returns the trivial π-equivalence. In fact, it does what one in Listing 3.4 does but in the opposite direction: the verification

$$\text{if } (R_{ved} \cup R_{unv} \neq R) \text{ return false}$$

is replaced by the construction

$$R \leftarrow R_{ved} \cup R_{unv};\ R \leftarrow \text{transitive-symmetric closure of } R$$

Proposition 3.3.1. *The procedure* `StateEquivCons` *halts always. It returns a nontrivial π-equivalence whenever such a relation exists, otherwise it returns the trivial π-equivalence.*

Proof. One may recognize that the while-loop in `StateEquivCons` is a traditional tech-

nique to calculate the relation R as a fixed-point. To make it clear, for each pair

$$(q, q') \in Q(\mathscr{A}) \times Q(\mathscr{A}) \text{ where } q \neq q'$$

let R^i, R^i_{unv} and R^i_{ved} denote respectively value of the sets R, R_{unv} and R_{ved} just after the last command of the "if ..." block (i.e. after the update $R_{unv} \leftarrow R \setminus R_{ved}$ of the i-th while-step. We prove that:

1. R^i is an equivalence relation, $R^i \subseteq R^{i+1}$
2. $R^i = R^i_{unv} \cup R^i_{ved}$, and
3. $R^i_{ved} \subsetneq R^{i+1}_{ved}$

for all i.

Indeed, we have $\{(q,q) \mid q \in Q(\mathscr{A})\} \subseteq R^i$ for all i, then R^i is reflexive. Moreover R^i is also transitive and symmetric at the end of each while-step, so R^i is an equivalence relation. We have also R^{i+1} is the transitive-symmetric closure of $R^i \cup R_l$, then $R^i \subseteq R^{i+1}$. From the last command

$$R_{unv} \leftarrow R \setminus R_{ved}$$

in the "if ..." block, we have $R^i = R^i_{unv} \cup R^i_{ved}$. Also in each while-step, we have

$$R^{i+1}_{ved} = R^i_{ved} \cup \{(q,q'), (q,q')\}$$

then $R^i_{ved} \subsetneq R^{i+1}_{ved}$.

Since $R^i_{ved} \subsetneq R^{i+1}_{ved} \subseteq Q(\mathscr{A}) \times Q(\mathscr{A})$, the while-loop must halt at some value n of i, consequently `StateEquivCons` halts always and returns

1. either $R = R^n$ if there is some pair $(q,q') \in Q(\mathscr{A}) \times Q(\mathscr{A})$ and $q \neq q'$ satisfying $q \sim q'$ for any π-equivalence $\sim$ (in this case we have also $R \subseteq \sim$),
2. or the trivial π-equivalence.

and that gives the results. □

The following result gives an evaluation for the time complexity of Algorithm 3.5. This evaluation is still very rough, but it shows the advantage of the algorithm in comparison with the direct approach which verifies $2^{|Q(\mathscr{A})| \times |Q(\mathscr{A})|}$ subsets.

Proposition 3.3.2. *The worst-case (time) complexity of Algorithm 3.5 is* $|Q(\mathscr{A})|^4$.

Proof. Given any pair $(p,q) \in Q(\mathscr{A}) \times Q(\mathscr{A})$ and $p \neq q$, the algorithm constructs the minimal π-equivalence $\sim^{min}_{pq}$ containing (p,q) after no more than $|\sim^{min}_{pq}|$ computation steps, then time complexity to verify such a pair is no more than $|Q(\mathscr{A})|^2$.

If $\sim_{pq}^{min}$ is the trivial π-equivalence[1], then the algorithm continues to verify another pair, otherwise the algorithm terminates and gives out $\sim_{pq}^{min}$. The worst-case is that there is no nontrivial π-equivalence and the algorithm has to verify all pairs possible, namely the algorithm has to verify no more than $|Q(\mathscr{A})|^2$ pairs. Thus the worst-case time complexity of the algorithm is no more than $|Q(\mathscr{A})|^4$. □

Summary Algorithms 3.5 and 3.3 in this section are constructive consequences of the theoretical framework given in Section 3.2. These algorithms will be used in the next chapter under a concrete context. In the one hand, they allows us constructing needed objects (i.e. normal forms of labeled transition systems). In the other hand, we will see in Section 4.3.2 of the next chapter that these objects allow us verifying the theoretically predicted results in this concrete context.

[1]We note that there is a unique trivial π-equivalence in any LTS $\mathcal{A}$, this is the identity relation.

Chapter 4

Concrete interpretation

In Chapter 3, we have introduced labeled transition systems as a mathematical model of the program, focused in modeling how the program processes its inputs. In some conceptual discussions about technical details of this model, we have presented also our original motivations that explain why these details should occur in this model. These motivations come actually from real-world programs, and they can be considered as a sketchy interpretation only.

The goal of this chapter is to present in detail a concrete interpretation that helps recovering the input message format of a program. As discussed at the end of Section 2.3, our approach is to categorize the input messages according to their corresponding execution traces - as a concrete interpretation of observable behaviors. Consequently, the concrete interpretation of the program is a set of execution traces. Abstracting this set by a labeled transition system, we obtained a mathematical model where the theoretical results developed in Section 3.2 can be applied.

This abstraction, from execution traces to a labeled transition system, is realized through a *stepwise abstraction* procedure that consists of several supporting objects, each of them and the abstraction from one to the other need to be carefully formalized. Then in the first part of this chapter, we present the technical notions and terminology about the syntax and the semantics of binary codes. That helps describing clearly objects and their construction, which is presented in the stepwise abstraction of the second part.

The results of this chapter have been presented partially in [17, 18, 19]. This is joint work with Guillaume Bonfante and Jean-Yves Marion.

4.1 Notions and terminology

Since malwares are mostly available under the compiled binary codes, working with this kind of codes is compulsory in malwares mitigation researches. The technical notions and terminology about binary codes introduced below can be found in textbooks [25], technical manuals [80, 81, 108], or handbooks about binary codes reverse

engineering [51, 147].

Instruction sets and reasoning First, different from a generic mathematical model, the binary codes are machine dependent where each processor architecture has its own instruction set, and this instruction set may evolve over time. In this thesis, we limit our interest within the `x86` and `x86-64` instruction sets, that are also the most popular platform where the malicious codes operate. However, our approach remains general enough and other instruction sets could be used.

Second, because the machine instructions are sophisticated enough, the current trend is to lift them into an intermediate representation (abbr. IR codes) which has more formal semantics and then easier for reasoning, some popular instances are REIL [56] and BIL [23]. However, the current proposed IR codes still have several technical limits.

The lifting method in both REIL and BIL is instruction-to-instructions, namely a machine instruction will be translated into several IR instructions, but because the machine instruction set is large enough, the translation is not yet complete. Concretely, the code lifting of REIL works only for `x86` instruction set. Until just recently (version 0.8), BAP works for `x86-64` instruction set but the translation is not complete, a report of unsupported instructions and translation failures through daily testing can be referenced at [5]. Additionally, BAP's tools work only on Linux™platform and that requires considerable efforts to implement a system analyzing Windows™programs.

Because of the limits discussed above, and we in fact do not need to take care of every semantics aspects of machine instructions. In this thesis, we reason directly on the machine instruction set together with a proper abstraction, which serves only our interests.

Third, the machine instructions constitute also a low-level programming language taking care of low-level properties (e.g. heap, stack, register usages, memory accesses, execution flow, function calls, argument passing conventions, etc). Similar to the reason above, we do not need to take care of all low-level properties to present the objects of the stepwise abstraction procedure: only needed properties are presented.

Formalization We give also in this section a lightweight formalization for the technical notations. It is worth noting that there is currently no standard formalization in malware binary codes analysis (and generally, in program analysis). This is due to the fact that each work focuses on a different aspect of binary codes, and reasoning in each aspect uses a particular logic (e.g. predicate logic, Hoare logic, etc.). Consequently, the corresponding formal model of binary codes is normally chosen to be convenient with the used logic.

The formalization given here does not aim at giving a standard. It gives instead notations allowing to specify what we mean in various real-world situations without repeating analogous details. For the logic reasoning aspect, it allows us representing

in Section 4.2.2 the condition of any execution trace under a first-order logic conjunction form of local conditions.

4.1.1 Low-level representation

Virtual memory and program A program is mapped into the virtual memory as binary data, each byte is indexed by a memory address. Starting from a special address called the *entry point* (abbr. EP) of the program, the CPU fetches, decodes, and executes gradually each time a part of this data; this part is called a *machine instruction*. The address where the CPU fetches each machine instruction is stored in a special register, called the *instruction pointer*.

First, saying the mapped binary data of a program, we mean to the binary code loaded into the *virtual memory space* reserved for the program by the operating system. Aside from the data of the program itself on some physical storage, it consists of also data of supplemental libraries that the program uses in execution. This data is loaded automatically by the *loader* of the operating system, or explicitly by the program.

Second, strictly speaking, not all mapped binary data can be executed. The modern CPUs are equipped with memory protection mechanisms, which allow executing only data of certain mapped regions (or *segments*), other data is not executable. We do not take interest in this detail and transparently consider only the executable regions of the program.

Formalization The *state of a program* is characterized by the state of (i.e. the value stored in) CPU's registers and of the virtual memory space of the program. Operating on the registers and the memory space, the execution of instructions by the CPU changes the state of the program.

Remark 4.1.1. The no-operation `nop` instruction (which is indeed `xchg eax,eax` [80]) changes nothing except increases the value of the `eip` register.

The virtual memory space where a program located is modeled as a sequence of bytes, the range of values that a byte can received is $\texttt{Bytes} = [0, \ldots, 255]$. The position (i.e. address) of a byte in the memory is specified by a value in the range `Addrs`, practically `Addrs` is $[0, \ldots, 2^x - 1]$ for x-bit memory space where x is 32 or 64. The *state of the memory* is then determined by a function:

$$\mu \colon \texttt{Addrs} \to \texttt{Bytes}$$

which specifies the byte value at each address in the address space `Addrs`. Let `Registers` denote the set of registers, the *instruction pointer* $\texttt{IP} \in \texttt{Registers}$, the *state of registers* is determined by a tuple:

$$\nu = \{r \leftarrow v \mid r \in \texttt{Registers},\ v \in [0, \ldots, 2^{|v|} - 1]\}$$

where $|v|$ is the length of the register v. The *state of the program* is then modeled by a couple $\xi = (\mu, \nu)$.

We can distinguish registers from memory addresses $\texttt{Registers} \cap \texttt{Addrs} = \emptyset$, we can distinguish also different registers, and different memory addresses. Then given a register or a memory address $x \in \texttt{Registers} \cup \texttt{Addrs}$, we can use the function notation $\xi(x)$ or the projection notation $\xi|_x$ to specify the state of x. In general, given a sub-set $xs \subseteq \texttt{Registers} \cup \texttt{Addrs}$, we will write

$$\xi(xs) \text{ or } \xi|_{xs}$$

to specify the *sub-state* on xs of the program state ξ.

We use the assignment notation $\xi[xs \leftarrow v]$ to specify a new state obtained from ξ by updating the state of xs by v (where v must be well-typed with respect to xs in the assignment) whereas keeping the state of all registers and memory addresses in $\texttt{Registers} \cup \texttt{Addrs} \setminus xs$. The notation formalizes the updates of (sub-)states.

4.1.2 Assembly code representation

Instruction syntax A machine instruction can represented by the syntax of the assembly language, here we use the Intel™syntax. An instruction has form

$$\texttt{operator} \quad \texttt{operand}_1, \texttt{operand}_2, \ldots$$

The *operator* (or mnemonic), as its name, determines the operation performed on the operands. Most of operands occur explicitly in the assembly syntax of the instruction, others (e.g. the `eflags` register) are hidden. An operand can be some register or a memory address of some types, it can be a target (i.e. where the results of the instruction is stored) or a source (i.e. the input of the instruction), or both. The types of operands and the hidden operands of an instruction are specified in the manual reference [80].

Remark 4.1.2. There are differences between `x86` and `x86-64` instruction sets, together with accommodated implementation of the operating systems. Those do not lead to any serious effect in our context (e.g. new 64 bit instructions `rax`, `rbx`, `rip` etc. are introduced in `x86-64` sets) and they will be noticed when needed.

Syntax of binary programs and the binary code disassembly problem Unfortunately, there are no decidable method to disassemble statically the binary data into separated machine instructions [79]. More precisely, such a separation does not statically exist (but dynamically exists), there is no strict difference between data and executable codes [143]. A reason for this undecidability comes from indirect branching instructions, which make the CPU fetching data from a more or less arbitrary memory address; another reason comes from self-modifying codes where the existing mapped codes can generate themselves other executable codes.

Example 4.1.1. The pieces of codes in Listings 4.1 and 4.2 are extracted from a `UnpackMe`. This program uses the Windows NT API `NtContinue` (cf. Figure 4.1) to divert its control flow: the address of the instruction executed after `NtContinue` returned is specified in a `CONTEXT` structure, located at the address determined by a `dword` at the address `[ebp+8]` (see the instruction at `0x77b49efb`.

In this case, the next fetched instruction will be `pushad` at `0x4078c0`, and before it should be data (e.g. at `0x4078bf` in Listing 4.2). The correct disassembly is shown in Listing 4.2 then. However, if the disassembler does not take care of the effect of `NtContinue`, it can incorrectly recognize that there is an instruction at `0x4078bf`, then is led to a "domino effect" of incorrect disassembly as shown in Listing 4.1.

```
...
0x77b49ef9 push 1
0x77b49efb push dword [ebp+8]
0x77b49efe call NtContinue
...
0x4078bf   add [eax-0x42],ah
0x4078c2   adc eax,0x8d004070
0x4078c7   mov esi,0xffff9feb
...
```

Listing 4.1: Incorrect disassembly

```
...
0x77b49ef9 push 1
0x77b49efb push dword [ebp+8]
0x77b49efe call NtContinue
...
0x4078bf   db 00
0x4078c0   pushad
0x4078c1   mov esi,0x407015
...
```

Listing 4.2: Correct disassembly

```
770A9EF1   FF 75 08         push dword ptr ss:[ebp+8]
770A9EF4   E8 01 01 00 00   call <ntdll._LdrpInitialize@8>
770A9EF9   6A 01            push 1
770A9EFB   FF 75 08         push dword ptr ss:[ebp+8]
770A9EFE   E8 ED 5F FE FF   call <ntdll._NtContinue@8>
770A9F03   50               push eax
770A9F04   E8 7C 19 02 00   call <ntdll._RtlRaiseStatus@4>
770A9F09   CC               int3
```

Figure 4.1: Control flow diversion using `NtContinue`

Note 4.1.1. The binary code disassembly problem is not just a technical challenge. In theoretical models of computation (e.g. Turing machine, λ-calculus, etc.), there is no difference between executable codes and data, both of them are considered simply as data [83, 92]. In practice, well-decoded data can be mapped and executed as instructions using some OS-specific APIs (e.g. `mprotect` in Linux™or `VirtualProtect` in Windows™). Such a technique is popular in self-modifying codes. To uniform codes as data, it may be better to think of that the codes does not execute themselves, this is indeed the CPU fetches codes (as data) and changes the state of the program depending on the fetched data (i.e. codes).

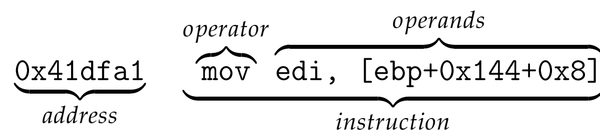
Avoiding self-modifying codes Practically, we examine only a part of the program (mostly its input parsers) and we assume that the undecidability above does not occur in the examined codes. Consequently, we go to the view that *programs as a sequences of machine instructions*. This is not true in general, for example the self-modifying program given in Example 4.1.1, and handling self-modifying codes is still a very challenging problem [30, 130]. Here, we make this view first to avoid awful technical details, and this is also a limit of our work.

Example 4.1.2. The following piece of codes extracted from `Trojan.Zbot`, describes partly how the malware searches for some files in the infected host. It demonstrates the binary level representation of the program as a sequence of machine instructions

```
...
0x41dfa1    mov edi,[ebp+0x144+0x8]
0x41dfa7    mov eax,[ebp+0x144+0xc]
0x41dfad    mov ebx,[ebp+0x144+0x10]
0x41dfb3    xor ecx,ecx
0x41dfb5    test edi,edi
0x41dfb7    setnz cl
0x41dfba    mov [ebp+0x144-0x158],eax
0x41dfbd    test ecx,ecx
0x41dfbf    jnz 0x41dfc6
0x41dfc1    call 0x402969
0x41dfc6    xor ecx,ecx
...
```

Listing 4.3: `Trojan.Zbot`

The structure of an instruction is illustrated as follows



This instruction is located at the address `0x41dfa1`, its operator is `mov` and its operands are (the constants are omitted) registers `edi`, `ebp` and the memory of type `dword` at the address calculated by $v_{\texttt{ebp}} + \texttt{0x144} + \texttt{0x8}$ where $v_{\texttt{ebp}}$ is the current value stored in the register `ebp`; also in this case the operands are shown completely within the syntax of the instruction.

For some instructions, the operands are not shown completely in the assembly syntax, for example the instruction located at `0x41dfb3`

```
xor ecx, ecx
```

Aside from the register `ecx`, the register `eflags` is also an operand: the technical description of `xor` in [80] says that `SF`, `PF`, `ZF` flags in `eflags` are set depending on the result of the operator `xor`, while `CF`, `OF` flags (also in `eflags`) are cleared.

Instruction and program formalization An instruction's opcode ins at the memory address addr is characterized by a pair

$$\epsilon = (\texttt{ins}, \texttt{addr})$$

and hereafter we will use this notation to specify an instruction, namely we specify the instruction not only by its opcode but also by its location.

Then the program, or more precisely the examined codes of the program, is statically represented by a sequence

$$P = [(\texttt{ins}_1, \texttt{addr}_1), (\texttt{ins}_2, \texttt{addr}_2), \ldots, (\texttt{ins}_n, \texttt{addr}_n)]$$

4.1.3 Execution of programs

Control-flow and execution trace The *control-flow* is the *order* on instructions, which follows the CPU's executed instructions. As discussed above, the CPU fetches, decodes, and executes gradually instructions of the program. The address where the CPU fetches an instruction is stored in the instruction pointer (abbr. IP), which is the register eip or rip on x86 or x86-64 instruction set, respectively. The control-flow then determines a sequence of executed instructions, that is called an *execution trace*.

Formalization Given a program P being currently in a state ξ, the address addr of the current executed instruction is well determined by $\xi(\texttt{IP})$ (i.e. the state of the instruction pointer IP). This instruction (cf. Remark 4.1.3) is then well determined, and denoted by

$$\mathcal{E}(\xi)$$

The instruction's opcode in the pair $\epsilon = \mathcal{E}(\xi)$ is called the *underlying instruction* of ξ and denoted by

$$\mathcal{I}(\xi)$$

Remark 4.1.3. We note that an instruction $\epsilon = \mathcal{E}(\xi)$ is determined by a pair $(\texttt{ins}, \texttt{addr})$ where ins is the opcode and addr is the location. So the underlying instruction $\mathcal{I}(\xi)$ is nothing but the opcode ins of ϵ. The notion "underlying instruction" just emphasizes the current executed opcode, given the current state ξ of the program.

The state ξ is updated depending on the execution of the underlying instruction $\mathcal{I}(\xi)$. Suppose that we examined the program from a state ξ_1 then the execution of the program is a sequence $P(\xi)$ of state transitions

$$P(\xi) = \xi_1 \to \xi_2 \to \ldots$$

where the transition $\xi_i \to \xi_{i+1}$ depending on the execution of the underlying instruc-

tion $\mathcal{I}(\xi_i)$. The corresponding sequence of instructions

$$\epsilon_1 \to \epsilon_2 \to \dots$$

where $\epsilon_i = \mathcal{E}(\xi_i)$, is called the *execution trace* starting from ξ_1, denoted it by $T(\xi_1)$. Sometimes, for convenience, we write a trace as a string

$$T(\xi_1) = \epsilon_1 \cdot \epsilon_2 \cdots$$

Changing of control-flow

Normally, the instruction pointer `IP` is set automatically by the address of the positionally next instruction, but if the executed instruction is a *control-flow instruction* (abbr. CFI) or an *exception instruction* then it can modify explicitly the value of `IP`.

The *conditional (direct) jumps* are traditionally considered as CFIs. For the *indirect jumps/calls* (e.g. `call edx`) and the *return* (i.e. `ret`) instruction, the address executed next is stored respectively in the target and in the top of the stack (i.e. the memory address specified by `esp`), then they are also CFIs.

Example 4.1.3. The piece of codes in Listing 4.4 is extracted from one in Example 4.1.2, the instruction executed after `jnz 0x41dfc6` depending on its effect: it verifies the flag `ZF` of the register `eflags`, and if this flag is not zero then the next executed instruction will be one located at the address `0x41dfc6` (i.e. `xor ecx,ecx`) otherwise one located at `0x41dfc1` (i.e. `call 0x402969`) will be executed.

```
...
0x41dfbf    jnz 0x41dfc6
0x41dfc1    call 0x402969
0x41dfc6    xor ecx,ecx
...
```

Listing 4.4: Effect of the control flow instruction

`Rep`**-prefixed instructions** There is a class of instructions, though it consists indeed of CFIs in accordance with the generic definition "ones can modify the value of `eip`" but as fas as we know, is not considered anywhere. This class consists of `rep`-*prefixed* instructions, such an instruction will repeat the execution of its suffix (that is another instruction) until some condition is satisfied. Consequently, the instruction executed next may be itself or the positionally consecutive instruction, depending on whether the condition has been satisfied or not.

Remark 4.1.4. In the `x86` and `x86-64` instruction sets, the conditional jumps are always direct. The `loop/loop(xx)` are also control flow instructions, but they are considered deprecated and hardly generated by compilers because their semantics can be done simply by conditional jumps, moreover their latency is high in modern CPUs [81].

Example 4.1.4. The piece of codes in Listing 4.5 is extracted from `wget`. It describes how the program verifies whether the first 4 bytes of a received message are ‘‘`HTTP`’’ or not. The number of verified bytes is stored in `ecx` (by `mov ecx,0x4`), the address of the constant pattern ‘‘`HTTP`’’ is stored in `edi` (by `mov edi,0x49ded0`), the address of the received message buffer is stored in `esi` (by `mov esi,eax`). The `rep`-prefixed instruction compares the pattern with the received message (by `rep cmpsb [esi],[edi]`).

```
...
0x404f75    mov ecx,0x4
...
0x404f80    mov edi,0x49ded0
0x404f85    mov esi,eax
0x404f87    xor edx,edx
0x404f89    rep cmpsb [esi],[edi]   /* string matching by rep */
0x404f8b    pop edi
...
```

Listing 4.5: “HTTP” string matching

Now depending on the value of first 4 bytes in the received message, the `rep`-prefixed instruction can be re-executed multiple times. For example if the first 4 bytes are ‘‘`HTaB`’’ (i.e. the first 2-bytes are matched but the third is not) then the instruction will be executed 3 times. In other words, if we execute the codes with this message and log each executed instruction, then the following sequence (of executed instructions) will be received:

```
...
0x404f75    mov ecx,0x4
...
0x404f80    mov edi,0x49ded0
0x404f85    mov esi,eax
0x404f87    xor edx,edx
0x404f89    rep cmpsb [esi],[edi]  /* first comparison (matched H = H) */
0x404f89    rep cmpsb [esi],[edi]  /* second (matched T = T) */
0x404f89    rep cmpsb [esi],[edi]  /* third (unmatched T ≠ a)) */
0x404f8b    pop edi
...
```

Listing 4.6: Sequence of executed instructions with input ‘‘`HTa...`’’

The instruction `rep cmpsb [esi],[edi]` is executed at the first time to compare the byte at `[esi]` with one at `edi`, here there is a match since both bytes are `h`, then both `esi` and `edi` are increased by 1, and the comparison is repeated. Consequently, the instruction executed next is still `rep cmpsb [esi],[edi]` instead of the positionally consecutive instruction `pop edi`.

Exception instructions We have considered the state and the control-flow of a program. While the control-flow specifies the execution order of instructions, the state is changed by this execution. However, the state of a program can be modified also by

the execution of instructions which do not belong to the program. This execution is activated by *exception instructions*.

There are several classes of exceptions (interrupts, traps, etc), their detail descriptions can be referenced in [25]. We consider in the following example the employment of *traps* in implementing the system call invocation mechanism of modern operating systems.

Example 4.1.5. The following piece of codes is extracted from the `x86-64` binary form of `curl`, it describes the invocation of the system call `sys_recvfrom`. The detail of the invocation mechanism can be referenced in [108], in this case the identity (here it is `0x2d`) of the system call is loaded into the register `eax`. Then the instruction `syscall`, as a general instruction invoking all kinds of system calls, is executed. The result is stored in the register `rax`, next compared to verify whether it is smaller than -4095 (which is the decimal form of the hex value `0xfffffffffffff001`) or not.

```
...
0x7fc1224e26ac  mov eax,0x2d                 /* id of sys_recvfrom   */
0x7fc1224e26b1  syscall                      /* invoke sys_recvfrom */
0x7fc1224e26b3  cmp rax,0xfffffffffffff001 /* compare with -4095   */
0x7fc1224e26b9  jnb 0x7fc1224e26ef
...
```

Listing 4.7: Exception `syscall`

The difference from the normal control-flow is that the execution of `sys_recvfrom` is performed actually in the *kernel-space*. This execution is transparent from the viewpoint of the program: it is seen as a single instruction `syscall` in the code[1], but it affects the state of the program: the value of the register `rax` is modified.

Remark 4.1.5. We should notice an important difference: *the exceptions cannot be abstracted as the normal control-flow*, concretely the system call invocation using traps in Example 4.1.5 cannot be implemented using the procedure calls. The technical reason is that they are implemented differently (e.g. one operate in kernel-mode with full access rights, the other operates in user-mode with restricted rights, etc).

A more essential reason is that *the results of an exception cannot be determined by the logic of the program*, they are actually external stimulus from the environment that the program must be able to handle, if it does not then there will be errors.

So in the low-level detail, we know that the CPU executes actually the instructions of the exception, however under the program viewpoint this execution is abstracted as (the execution of) a single instruction, and after that, the positionally consecutive instruction is executed as normal. We keep this proper abstraction in this thesis, consequently the instructions invoking exceptions will be not considered as the control-flow instructions.

[1] We suppose that the binary codes must be stored in the virtual memory space of the program, and normally located in the *user-space*, except for kernel-mode malwares.

Hereafter, we consider the execution of program where the exceptions do not occur. Obviously, this is not true in general, most programs need exceptions to communicate with the environment, for example to receive their inputs. But if we consider only the processing of the input, and omit the mechanism up on that the input is updated, then our consideration is rational.

In this context, the results of the execution of a underlying instruction are deterministic. Consequently, the execution trace starting from a given program state ξ is deterministic. The following result is simple, but important.

Proposition 4.1.1. *The execution trace $T(\xi)$ is uniquely defined by the program state ξ.*

Initial state and input buffer

We have seen in Examples 4.1.3 and 4.1.4 that the CFIs can make the instructions of the program executed in different orders, depending on the initial state of the program. So by executing the program under different initial states, we can observe different execution traces. We will characterize the changing the program's state by an update on the subset of memory space, next called the *input message buffer*.

Formalization The input message buffer is a subset $\texttt{Buf} \subsetneq \texttt{Addrs}$ so that by update the state ξ by any value v of Buf (namely the updated state is $\xi[\texttt{Buf} \leftarrow v]$), then we can obtain different execution traces

$$T(\xi[\texttt{Buf} \leftarrow v])$$

by changing the value v. By Proposition 4.1.1, the trace $T(\xi[\texttt{Buf} \leftarrow v])$ depends only on the state $\xi[\texttt{Buf} \leftarrow v]$.

In this thesis, we are interested in the update of the initial state by the environment. Concretely, *this update should be made by an exception*. However, we note also that this update must be taken before the observation of execution traces, and not within.

Example 4.1.6. We extract again from Example 4.1.4 a piece of codes and mark its instructions as in Listing 4.8. This piece is relevant with changes in the received message buffer whose address is stored in `eax`: bytes of the received message will be compared with the constant string ‘‘HTTP’’.

```
...
0x404f85    mov esi,eax             /* ins1 */
0x404f87    xor edx,edx             /* ins2 */
0x404f89    rep cmpsb [esi],[edi]   /* ins3 */
0x404f8b    pop edi                 /* ins4 */
...
```

Listing 4.8: Control-flow depends on the initial state

In executing this piece from the initial state where the instruction pointer points always to $\texttt{ins}_1$ but the buffer (specified by `eax` and then also by `esi`) may contain

different values, we can observe the following execution traces (note that the character a used below is not important, for example it is can be replaced by other character different from {H,T,T,P}):

received message	corresponding execution trace
‘‘a...’’	$\texttt{ins}_1, \texttt{ins}_2, \texttt{ins}_3, \texttt{ins}_4$
‘‘Ha...’’	$\texttt{ins}_1, \texttt{ins}_2, \texttt{ins}_3, \texttt{ins}_3, \texttt{ins}_4$
‘‘HTa...’’	$\texttt{ins}_1, \texttt{ins}_2, \texttt{ins}_3, \texttt{ins}_3, \texttt{ins}_3, \texttt{ins}_4$
‘‘HTTa...’’	$\texttt{ins}_1, \texttt{ins}_2, \texttt{ins}_3, \texttt{ins}_3, \texttt{ins}_3, \texttt{ins}_3, \texttt{ins}_4$
‘‘HTTP...’’	$\texttt{ins}_1, \texttt{ins}_2, \texttt{ins}_3, \texttt{ins}_3, \texttt{ins}_3, \texttt{ins}_3, \texttt{ins}_4$

We can also observe that these traces constitute a tree (cf. Figure 4.2) where each path from the root to a leaf corresponds to an execution trace. The general form of this tree, which will be defined later, is called the *execution tree*.

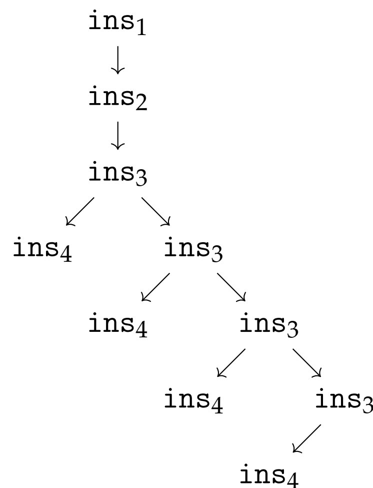


Figure 4.2: Simple execution tree

Example 4.1.7. We can observe that the CFIs are also *trace-relative*, that means it can change the control flow or not depending on its position in a trace. For example, in the piece of codes in Listing 4.8, if we consider a trace of form

$$\texttt{ins}_1, \texttt{ins}_2, \texttt{ins}_3, \texttt{ins}_3, \texttt{ins}_3, \texttt{ins}_3, \ldots$$

which corresponds to inputs of form “HTT...”. Then the 4-th $\texttt{ins}_3$ is indeed not a CFI because the next executed instruction is always $\texttt{ins}_4$.

4.2 Stepwise abstraction

Given a program P at the original state ξ and an input buffer `Buf`, the abstraction from the execution traces to a labeled transition system consists of supporting objects: *execution tree*, *message tree*, *initial labeled transition system*. The construction steps are described as follows

First, by fixing some limit length n, we explore execution traces of P starting from states $\xi[\texttt{Buf} \leftarrow v]$ for all states v of `Buf`, so that the length of each traces is n (cf. Remark 4.2.1).

Second, the set of execution traces forms naturally a tree, where the branching occurs only at CFIs[1], then this tree can be simplified to contain just such instructions. This tree is called the *execution tree* where each path from the root to a leaf corresponds to an execution trace.

Third, on the execution tree, each edge (i.e. branch) from a node describes implicitly also the input-dependence condition for that the corresponding control flow instruction takes the branch. We take explicitly this condition, then we obtain a new tree from the execution tree where the edges are labeled by their corresponding conditions, and each path from the root to a leaf corresponds to an execution trace. We continue merging some vertices on this tree, so that the conditions on any path (from the root to a leaf) are mutually independent, the result is called the *message tree*.

Fourth, because for each path in the message tree, the input-dependence conditions are mutually independent, then this message tree can be converted naturally into a labeled transition system, called the *initial LTS*, where

- the vertices are converted into the states, and the root becomes the initial state whereas the leaves become the terminal states,
- the outgoing edges of any vertex become the transitions from the corresponding state (of this vertex) where the edges conditions are letter of the LTS's alphabet.

The initial LTS has form of a tree where each path from the initial state to a terminal state corresponds to a class of input messages. And *finally*, the final LTS is simply the normal form of the initial LTS.

Remark 4.2.1. Since we consider the program as a static sequence of assembly instructions (cf. Section 4.1.2), there are cases where the executed instruction is not in the static sequence. In such a case, we stop the trace just before the outside instruction is executed, and then the length of the trace may shorted than the limit n.

[1]We recall that CFI stands for control-flow instruction.

Why the final LTS is needed?

Since the the original LTS is abstracted directly from the execution traces of finite length, there are some problems in using it directly to classify the input messages.

First, the original LTS is still very sophisticated, its number of states increases exponentially with the chosen limit length n of execution traces. The initial LTS has about $2^{\mathcal{O}(n)}$ states where n is the limit length (cf. Example 4.2.1). Whereas the full execution trace of the input parser of a program, for example `wget`, has more than 1000 instructions, but working on a labeled transition system of $2^{\mathcal{O}(1000)}$ states is impractical[1]. The final LTS, which is a normal form of the initial LTS, as we will see, is much more smaller.

Second, even though the initial LTS is sophisticated, it represents very few information about how program parses the input message, concretely it gives a very rough messages classification (cf. Example 4.2.2). Whereas the final LTS, because it is a normal form of the initial LTS, is an observational prediction of the initial LTS. That means it can predict the unobservable behaviors given observable behaviors in the initial LTS.

Example 4.2.1. We can see in Figure 4.3 the initial labeled transition systems of `wget` abstracted from the set of execution traces with the limit length is just 60.

Example 4.2.2. The LTS in Example 4.2.1 represents the information about the set of execution traces where each of them is limited at the length of just 60 instructions whereas the full trace is about 1000 instructions.

4.2.1 From execution traces to the execution tree

Let P be a program having the static form (notice that P is assumed to not contain exception instructions)

$$P = [(\mathtt{ins}_1, \mathtt{addr}_1), (\mathtt{ins}_2, \mathtt{addr}_2), \ldots, (\mathtt{ins}_n, \mathtt{addr}_n)]$$

and $\mathtt{Buf} \subsetneq \mathtt{Addrs}$ be an input message buffer of P. The program P starts at a state ξ where $\xi(\mathtt{IP}) = \mathtt{addr}_1$, any execution trace $\mathcal{T}(\xi[\mathtt{Buf} \leftarrow v])$ is limited by a limit length n, namely only the prefix of length n is considered.

Proposition 4.2.1. *Given two different execution traces*

$$T_1 = \mathcal{T}(\xi[\mathtt{Buf} \leftarrow v_1]) \text{ and } T_2 = \mathcal{T}(\xi[\mathtt{Buf} \leftarrow v_2])$$

obtained by changing value of `Buf`*, then they have a unique common prefix t of form $t = \bar{t} \cdot \epsilon$*

[1] We should distinguish the static size of the parser and the dynamic size of an execution trace, the static size is much smaller than the dynamic size. For example, the execution traces unroll the loops in the static code, then a same piece of codes will be repeated multiple times.

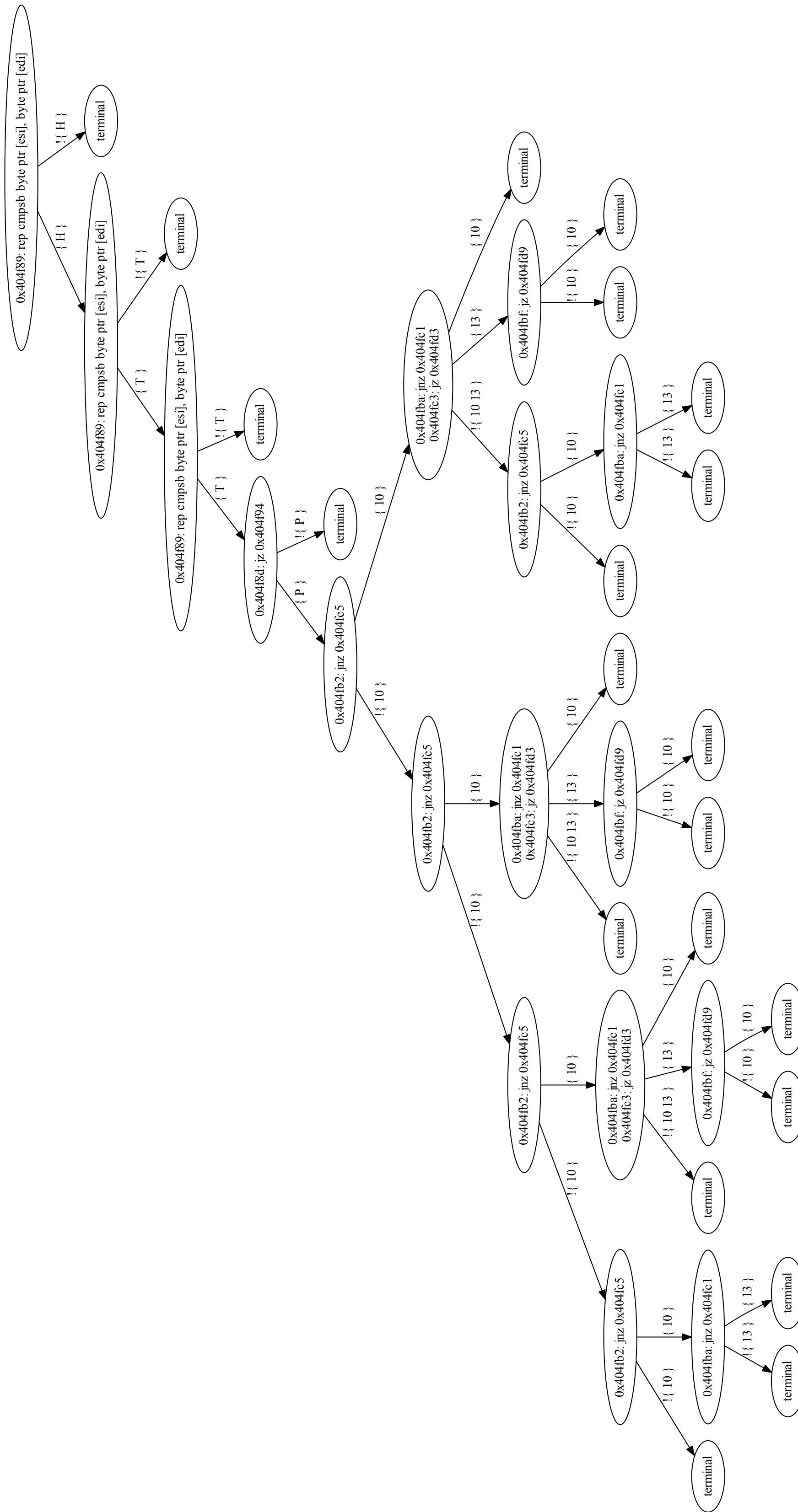


Figure 4.3: Initial labeled transition system of wget with chosen finite length of 60

where ϵ is a control flow instruction, in others word we can write

$$T_1 = \bar{t} \cdot \epsilon \cdot \epsilon_1 \cdots \text{ and } T_2 = \bar{t} \cdot \epsilon \cdot \epsilon_2 \cdots$$

where $\epsilon_1 \neq \epsilon_2$.

Proof. Because there is no exception instruction in P, then the difference between T_1 and T_2 occurs only after a control flow instruction. That gives the result. □

Remark 4.2.2. The assertion of Proposition 4.2.1 is not true if P has not a static form, i.e. P contains self-modifying codes. For example, the instructions in the prefix t can modify the next executed instruction, that means we can obtain $\xi_1(\texttt{IP}) \neq \xi_2(\texttt{IP})$ where `ins` is not necessarily a control flow instruction.

The algorithm in Listing 4.9 constructs an *execution tree* from a set of execution traces. The following proposition is a direct corollary of Proposition 4.2.1.

```
ExecTreeConstruction(set of execution traces Ts) {
  T ← a trace of Ts;
  ExecTree ← T;  Ts ← Ts \ {T};
  foreach (T ∈ Ts) {
    ϵ_tree ← root(ExecTree);  ϵ_trace ← head(T);
    while (ϵ_tree = ϵ_trace) {
      ϵ'_trace ← next_ins(T,ϵ);  /* return the instruction after ϵ_trace in T */
      if (ϵ'_trace ≠ NULL) {
        if (edge ϵ_tree → ϵ'_trace ∉ ExecTree) {
          add a vertex of label ϵ'_trace into ExecTree;
          add the edge ϵ_tree → ϵ'_trace into ExecTree;
        }
        ϵ_tree ← ϵ'_trace;  ϵ_trace ← ϵ'_trace
      }
      else break;
    }
  }
}
```

Listing 4.9: Execution tree construction algorithm

Proposition 4.2.2. *The ExecTree constructed by the algorithm in Listing 4.9 in a tree where the branching occurs only at CFI labeled vertices, and each path from the root to a leaf is an execution trace.*

Example 4.2.3. Consider the program P having static form $[\texttt{ins}_1, \texttt{ins}_2, \texttt{ins}_3, \texttt{ins}_4]$ as given in Listing 4.8 (cf. Example 4.1.7). The input message buffer `Buf` beginning from some address given by $\xi(\texttt{esi})$ (cf. also Example 4.1.4). Then it can be verified that, by changing the value of `Buf`, there are only following 4 execution traces

$$\texttt{ins}_1 \to \texttt{ins}_2 \to \texttt{ins}_3 \to \texttt{ins}_4$$

$$\texttt{ins}_1 \to \texttt{ins}_2 \to \texttt{ins}_3 \to \texttt{ins}_3 \to \texttt{ins}_4$$

$$\texttt{ins}_1 \rightarrow \texttt{ins}_2 \rightarrow \texttt{ins}_3 \rightarrow \texttt{ins}_3 \rightarrow \texttt{ins}_3 \rightarrow \texttt{ins}_4$$
$$\texttt{ins}_1 \rightarrow \texttt{ins}_2 \rightarrow \texttt{ins}_3 \rightarrow \texttt{ins}_3 \rightarrow \texttt{ins}_3 \rightarrow \texttt{ins}_3 \rightarrow \texttt{ins}_4$$

and the execution tree constructed by the algorithm in Listing 4.9 from these traces is given in Figure 4.2.

Example 4.2.4. A real-world execution tree constructed from all execution traces of `wget`, by modifying the value of the input message, is given in Figure 4.4. The limit length n of traces is chosen as 25.

Execution trace and tree simplification

We have observed in Example 4.1.7 that some control flow instructions are *trace-relative*, the definition of this notion is given in the following proposition.

Example 4.2.5. In Example 4.1.7, the CFI $\texttt{ins}_3$ is branchable or not depending on its position on a trace. For example, in a trace of form

$$\texttt{ins}_1 \cdot \texttt{ins}_2 \cdot \texttt{ins}_3 \cdot \texttt{ins}_3 \cdot \texttt{ins}_3 \cdot \texttt{ins}_3 \cdots$$

then the first 3 occurrences of $\texttt{ins}_3$ are branchable but the fourth is not because the executed instruction after this occurrence is always $\texttt{ins}_4$.

Definition 4.2.1 (Branchable CFI)**.** Given a trace $T = \mathcal{T}\left(\xi[\texttt{Buf} \leftarrow v[\right)$, a control flow instruction `cfi` in T is called branchable if there exists another trace $T' = \mathcal{T}\left(\xi[\texttt{Buf} \leftarrow v]\right)$ so that T and T' has a common prefix $t = \bar{t} \cdot \texttt{cfi}$ as

$$T = \bar{t} \cdot \texttt{cfi} \cdot \epsilon \cdots \text{ and } T' = \bar{t} \cdot \texttt{cfi} \cdot \epsilon' \cdots$$

where $\epsilon \neq \epsilon'$.

Example 4.2.6. The traces from the root to the leaves in Figure 4.5 are traces of `wget` (where the chosen limit length is 50) but only input-dependent CFI(s) are kept. The ones in orange are branchable, whereas ones in blue are not.

From the definition of branchable control flow instructions, we have the notion of simplified trace.

Definition 4.2.2 (Simplified trace)**.** Given trace T, a trace constructed from T by keeping only branchable control flow instructions in T and the last instruction, is called a simplified trace derived from T, and denoted by $\mathit{Simple}\left(T\right)$.

Because the branching occurs only at the branchable instructions, we have immediately Proposition 4.2.3. Moreover, the algorithm in Listing 4.9 can be applied directly for the corresponding set of simplified traces; consequently it gives a tree, called the *simplified execution tree*.

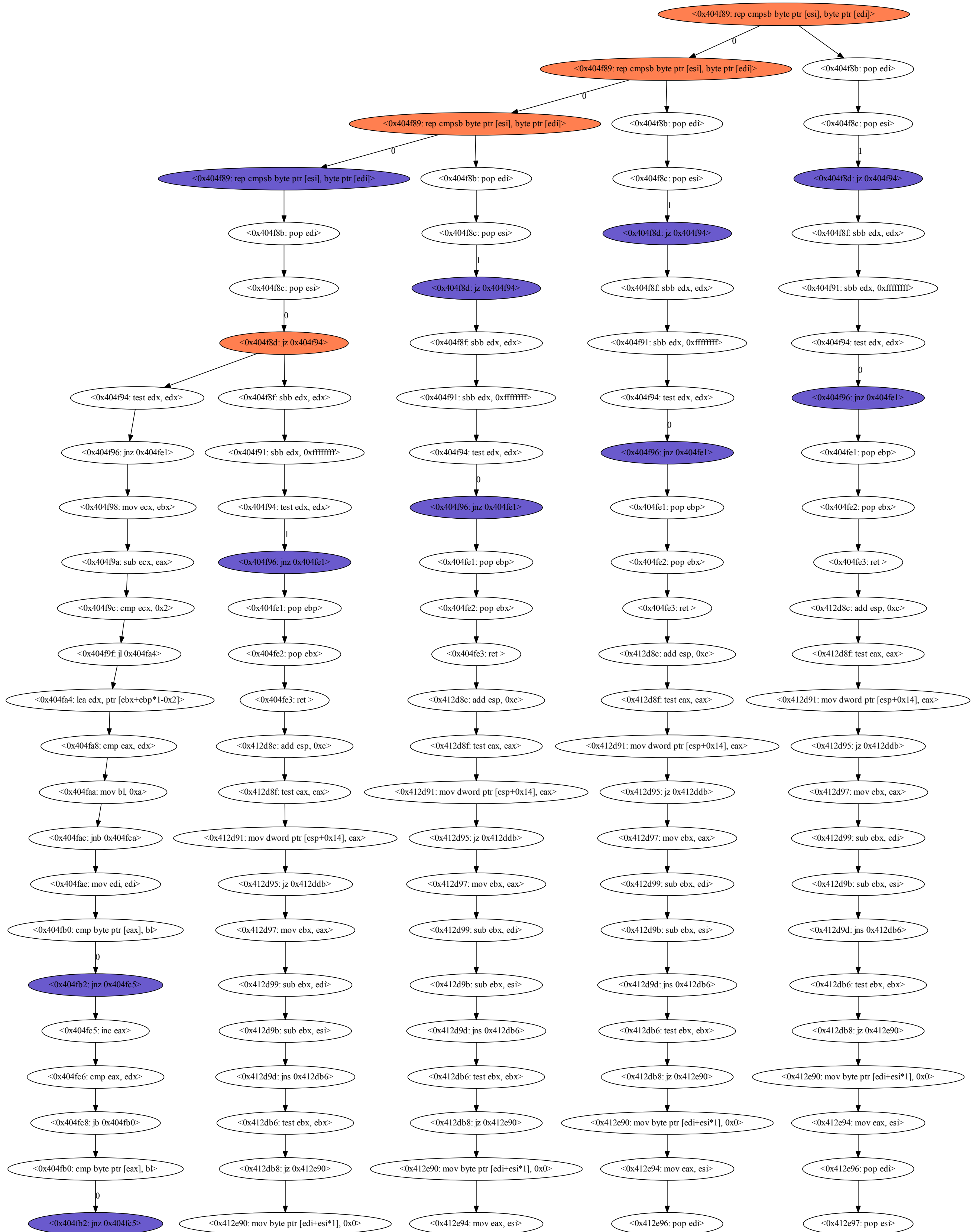


Figure 4.4: Execution tree of `wget` with the finite length of 25

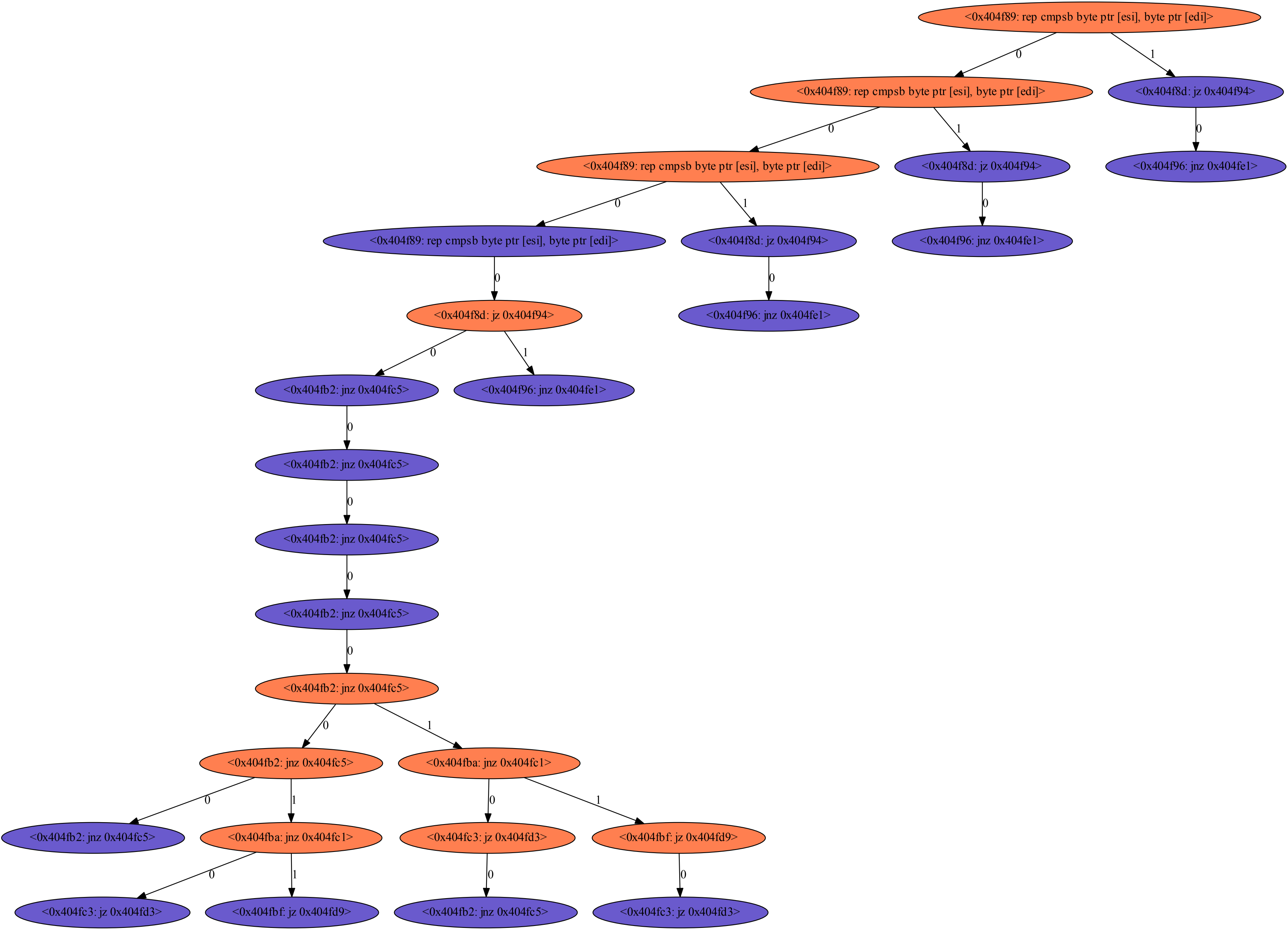


Figure 4.5: Execution tree where only CFI(s) in traces are kept

Proposition 4.2.3. *Given a simplified trace s, there is a unique execution trace T so that $Simple\,(T) = s$.*

The simplified execution tree can be obtained directly from the execution tree by reducing *maximal unbranched paths* on the tree (cf. Remark 4.2.3). Concretely, for each unbranched path

- add a direct edge from the head to the tail if the head is a branchable CFI,
- if such an edge is added then remove all vertices in the path which are not adjacent with the newly added edge, otherwise remove all vertices (in the path) except the tail.

Remark 4.2.3. A maximal unbranched path is a longest path so that except the head or the tail, all vertices in the path are unbranchable CFI(s), so it can be a path either

1. between two consecutive branchable CFI(s), or
2. between the root and an branchable CFI (but there is no other branchable CFI between), or
3. between a branchable CFI and a leaf (but there is no other branchable CFI between).

Example 4.2.7. The tree given in Figure 4.6 is the simplified execution tree of P. It can be constructed from simplified traces

$$\begin{aligned}
&\mathtt{ins}_3 \to \mathtt{ins}_4\\
&\mathtt{ins}_3 \to \mathtt{ins}_3 \to \mathtt{ins}_4\\
&\mathtt{ins}_3 \to \mathtt{ins}_3 \to \mathtt{ins}_3 \to \mathtt{ins}_4\\
&\mathtt{ins}_3 \to \mathtt{ins}_3 \to \mathtt{ins}_3 \to \mathtt{ins}_3 \to \mathtt{ins}_4
\end{aligned}$$

Or directly from the execution tree of P (cf. Figure 4.2) by observing that there is only a reducible maximal unbranched path

$$\mathtt{ins}_1 \to \mathtt{ins}_2 \to \mathtt{ins}_3$$

because the head $\mathtt{ins}_1$ is not branchable, then there is no newly added edge, and all vertices in the path (except the tail) are removed.

Result Since the branching occurs only at branchable CFI(s), by Proposition 4.2.2, the simplified execution tree has a similar property as the execution tree: each path from the root to the leaf corresponds to a unique execution trace, then each path corresponds to a unique class of messages. The result of this first step is the *simplified execution tree*, which is called shortly as the execution tree for the next step, and denoted by $\mathcal{E}\,(P)$.

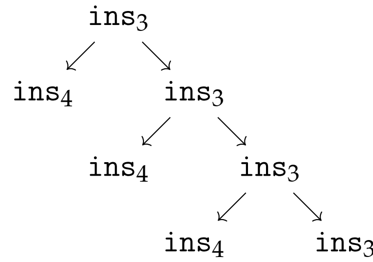


Figure 4.6: Simplified execution tree of P

4.2.2 From the execution tree to the message tree

In the execution tree, an edge starts always from a branchable CFI (which is called the source CFI of the edge). By Proposition 4.1.1, the branching depends only on the value of the input buffer `Buf`. Then we construct the message tree directly from the execution tree $\mathcal{E}(P)$ by labeling each edge with the input-dependence conditions making the source branchable CFI following the trace containing this edge. The message tree is denoted by $\mathcal{M}(P)$.

Example 4.2.8. The tree in Figure 4.7 is the message tree constructed from the (simplified) execution tree in Figure 4.6. We use the subscript index ($\texttt{ins}_3^i$ and $\texttt{ins}_4^j$) to distinguish different occurrences of the same instruction $\texttt{ins}_3$ and $\texttt{ins}_4$.

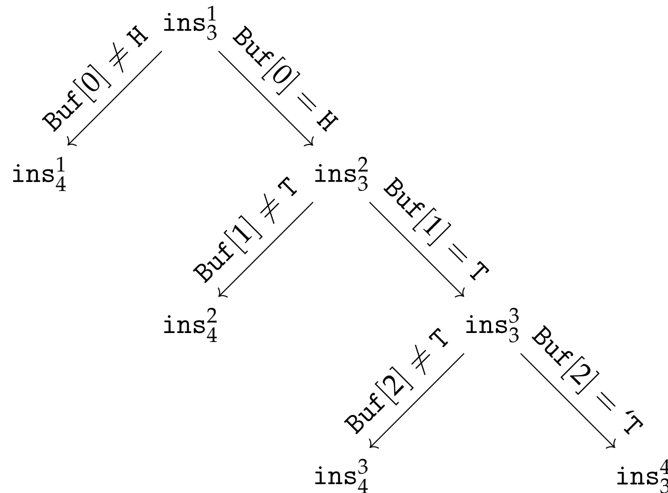


Figure 4.7: Message tree $\mathcal{M}(P)$ of P

Remark 4.2.4. Hereafter, when we say a path in the message tree, we mean that this is a path from the root to some leaf.

Local and path conditions

We observe that each occurrence of (branchable) CFI on the execution tree has a branching condition which depends on only a subset of the input message buffer

Buf. For example the branching condition of the occurrence $\texttt{ins}_3^1$ depends only the value of $\{\texttt{Buf}[0]\} \subsetneq \texttt{Buf}$. We declare a hypothesis that the latter reasoning bases on.

Hypothesis 1 (Static dependency)**.** Given a (not leaf) node in the message tree, then

- the branching conditions of the corresponding CFI depend on a fixed subset of the input message buffer Buf, and
- each branching condition (of this CFI) is also fixed.

In the general cases, the assumption that the each occurrence of a CFI in the message tree $\mathcal{M}(P)$ must depend on a fixed subset of Buf is not true (cf. Example 4.2.9). But practically, it is true for many parsers, for example all finite state machine parsers satisfy this assumption.

Example 4.2.9. Let us consider the piece of codes in Listing 4.10, the input message buffer is at [ds:m], the first CFI depends on only the first byte of the buffer (and then it validates the hypothesis). But the second CFI is not because the byte of the input buffer that it depends on, is determined from the value of the first byte of the buffer. Namely, if the value of the first byte is 3 then the CFI depends on the 3-th byte of the input buffer, if the value is 4 then the CFI depends on the 5-th byte, and so on.

```
0x400500    movzx rcx, byte [ds:m]
0x400508    xor eax, eax
0x40050a    cmp rcx, 0x1
0x40050e    je 0x400526                    /* first CFI  */
0x400510    movzx ecx, byte [ds:rcx+m]
0x400517    mov eax, 0x3
0x40051c    cmp ecx, 0x2
0x40051f    je 0x400530                    /* second CFI */
```

Listing 4.10: Indirect dependency

Given the validity of Hypothesis 1, for each vertex i (i.e occurrence of a CFI) which is not a leaf, let $\texttt{Buf}(\texttt{i})$ denote the subset of Buf that the branching conditions of the corresponding CFI depends on. The condition labeled on each outgoing edge of $\texttt{i} \to \texttt{i}'$ is a subset of

$$\texttt{Bytes}^{\texttt{Buf}(\texttt{i})}$$

and is called a *local condition* of v, denoted by $\mathcal{C}(\texttt{i} \to \texttt{i}')$.

Remark 4.2.5. Such a condition $\mathcal{C}$ is a subset of $\texttt{Bytes}^B$ for some subset $B \subseteq \texttt{Buf}$, we use the notation $\mathcal{D}(\mathcal{C})$ to denote the domain of any element of $\mathcal{C}$, and call also $\mathcal{D}(\mathcal{C})$ the domain of $\mathcal{C}$. For example, $\mathcal{D}(\mathcal{C}(\texttt{i} \to \texttt{i}')) = \texttt{Buf}_i$, or $\texttt{Buf}_i$ is the domain of $\mathcal{C}(\texttt{i} \to \texttt{i}')$.

Proposition 4.2.4. *Given a (not leaf) vertex* i *in the message tree, let* $\texttt{i} \to \texttt{i}_1, \dots, \texttt{i} \to \texttt{i}_n$ *be outgoing edges of* i*, then*

$$\cup_{i=1}^{n} \mathcal{C}(\texttt{i} \to \texttt{i}_i) = \texttt{Bytes}^{\texttt{Buf}(i)}$$

and $\mathcal{C}(\texttt{i} \to \texttt{i}_i) \cap \mathcal{C}(\texttt{i} \to \texttt{i}_j) = \emptyset$ *for all* $i \neq j$.

Proof. Any element of $\texttt{Bytes}^{\texttt{Buf}(\texttt{i})}$ makes the corresponding CFI of $\texttt{i}$ branches into a outgoing edges, that gives immediately the result. □

For any path $T = \texttt{i}_1 \to \texttt{i}_2 \to \cdots \to \texttt{i}_{n-1} \to \texttt{i}_n$, the conjunction

$$\mathcal{C}\left(\texttt{i}_1 \to \texttt{i}_2\right) \wedge \mathcal{C}\left(\texttt{i}_2 \to \texttt{i}_3\right) \wedge \cdots \wedge \mathcal{C}\left(\texttt{i}_{n-1} \to \texttt{i}_n\right)$$

is called the *path condition* [88] of T, and denoted by $\mathcal{C}(T)$. The following proposition is a direct corollary of Hypothesis 1.

Proposition 4.2.5. *The path condition $\mathcal{C}(T)$ of a path T is unique.*

Example 4.2.10. Consider the message tree in Figure 4.7 and let $\cdots \to \texttt{i}$ denote the path from the root to the leaf $\texttt{i}$. We have some path conditions

$$\begin{aligned}
\mathcal{C}\left(\cdots \to \texttt{ins}_3^4\right) &= \{\texttt{Buf}[0] \mapsto \texttt{h}\} \wedge \{\texttt{Buf}[1] \mapsto \texttt{t}\} \wedge \{\texttt{Buf}[2] \mapsto \texttt{t}\} \\
\mathcal{C}\left(\cdots \to \texttt{ins}_4^3\right) &= \{\texttt{Buf}[0] \mapsto \texttt{h}\} \wedge \{\texttt{Buf}[1] \mapsto \texttt{t}\} \wedge (\texttt{Buf}[2] \neq t\})
\end{aligned}$$

where $(\texttt{Buf}[2] \neq t\})$ stands for the set $\texttt{Bytes}^{\texttt{Buf}[0]} \setminus \{\texttt{Buf}[0] \mapsto \texttt{h}\}$.

Since each local condition $\mathcal{C}(\texttt{i}_i \to \texttt{i}_{i+1})$ is a subset of $\texttt{Bytes}^{\texttt{Buf}(\texttt{i}_i)}$ then the path condition $\mathcal{C}(T)$ is also a subset of

$$\texttt{Bytes}^{\texttt{Buf}(\texttt{i}_1) \cup \texttt{Buf}(\texttt{i}_2) \cup \cdots \cup \texttt{Buf}(\texttt{i}_{n-1})}$$

We have the following proposition

Proposition 4.2.6. *Given a path $T = \texttt{i}_1 \to \texttt{i}_2 \to \cdots \to \texttt{i}_{n-1} \to \texttt{i}_n$ then the product*

$$\mathcal{C}(T) \times \texttt{Bytes}^{\texttt{Buf} \setminus \cup_{k=1}^{n-1} \texttt{Buf}(\texttt{i}_k)}$$

is the set of all values v of Buf *so that $T = Simple\left(\mathcal{T}\left(\xi[\texttt{Buf} \leftarrow v]\right)\right)$, and denoted $\mathcal{C}[T]$.*

Proof. The path T depends only the values of bytes at $\cup_{i=1}^{n-1} \texttt{Buf}(\texttt{i}_i)$, consequently the values of bytes at addresses in $\texttt{Buf} \setminus \cup_{i=1}^{n-1} \texttt{Buf}(\texttt{i}_i)$ are independent from the path condition $\mathcal{C}(T)$. And that gives the result. □

Each path T determines a subset $\mathcal{C}[T]$ of the set of all input messages $\texttt{Bytes}^{\texttt{Buf}}$. The following proposition asserts that all paths in the message tree determine actually a partition in $\texttt{Bytes}^{\texttt{Buf}}$.

Proposition 4.2.7. *Let $T_1, \ldots, T_n$ be all paths of the message tree, then*

$$\cup_{i=1}^{n} \mathcal{C}[T] = \texttt{Bytes}^{\texttt{Buf}}$$

moreover $\mathcal{C}[T_i] \cap \mathcal{C}[T_j] = \emptyset$ for all $T_i \neq T_j$.

Proof. The result follows directly from Proposition 4.2.4. □

Path condition normalization

Given a path $T = \mathtt{i}_1 \to \cdots \to \mathtt{i}_n$, we cannot represent directly its condition $\mathcal{C}(T)$ under the conjunction form

$$\mathcal{C}(T) = \mathcal{C}(\mathtt{i}_1 \to \mathtt{i}_2) \wedge \mathcal{C}(\mathtt{i}_2 \to \mathtt{i}_3) \wedge \cdots \wedge \mathcal{C}(\mathtt{i}_{n-1} \to \mathtt{i}_n)$$

into the cartesian product

$$\mathcal{C}(\mathtt{i}_1 \to \mathtt{i}_2) \times \mathcal{C}(\mathtt{i}_2 \to \mathtt{i}_3) \times \cdots \times \mathcal{C}(\mathtt{i}_{n-1} \to \mathtt{i}_n)$$

because there may exists vertices $\mathtt{i}_i, \mathtt{i}_j$ where $\mathtt{Buf}(\mathtt{i}_i) \cap \mathtt{Buf}(\mathtt{i}_j) \neq \emptyset$ (cf Example 4.2.11). Consequently, the condition $\mathcal{C}[T]$ cannot be represented directly as a cartesian product of local conditions.

Example 4.2.11. In observing the message tree of `wget` with limit length to 60, we can observe a path

$$\mathtt{i}_1 \xrightarrow{\mathtt{Buf}[0]=\mathtt{H}} \ldots \xrightarrow{\mathtt{Buf}[3]=\mathtt{P}} \mathtt{i}_5 \xrightarrow{\mathtt{Buf}[4]=\backslash\mathtt{n}} \mathtt{i}_6 \xrightarrow{\mathtt{Buf}[5]\neq\backslash\mathtt{n}} \mathtt{i}_7 \xrightarrow{\mathtt{Buf}[5]=\backslash\mathtt{r}} \mathtt{i}_8$$

where

$$\begin{aligned}
\mathtt{i}_1 = \mathtt{i}_2 = \mathtt{i}_3 &= (\mathtt{rep\ cmpsb\ [esi],[edi],\ 0x40489}) \\
\mathtt{i}_4 &= (\mathtt{jz\ 0x404f94,\ 0x404f8d}) \\
\mathtt{i}_5 &= (\mathtt{jnz\ 0x404fc5,\ 0x404fb2}) \\
\mathtt{i}_6 &= (\mathtt{jnz\ 0x404fc1,\ 0x404fba}) \\
\mathtt{i}_7 &= (\mathtt{jz\ 0x404fd3,\ 0x404fc3}) \\
\mathtt{i}_8 &= (\mathtt{cmp\ [eax+0x2],\ bl,\ 0x404fbc})
\end{aligned}$$

The condition of this path is well represented under the conjunction form

$$(\mathtt{Buf}[0] = \mathtt{H}) \wedge \cdots \wedge (\mathtt{Buf}[3] = \mathtt{P}) \wedge (\mathtt{Buf}[4] = \backslash\mathtt{n}) \wedge (\mathtt{Buf}[5] \neq \backslash\mathtt{n}) \wedge (\mathtt{Buf}[5] = \backslash\mathtt{r})$$

but not under cartesian product form because the following product is nonsense

$$(\mathtt{Buf}[5] \neq \backslash\mathtt{n}) \times (\mathtt{Buf}[5] = \backslash\mathtt{r})$$

Our goal is to represent the condition $\mathcal{C}[T]$ for any path T in the message tree under a catersian product, because such a representation shows the independent effects of subsets of `Buf` to the form of T. Let $\mathtt{Buf}(\mathcal{M})$ be the union of all $\mathtt{Buf}(\mathtt{i})$ where `i` is a vertex of the message tree $\mathcal{M}(P)$.

Definition 4.2.3 (Address closure)**.** Let $\mathcal{R}_{\mathcal{M}} \subseteq \mathtt{Buf}(\mathcal{M}) \times \mathtt{Buf}(\mathcal{M})$ be the relation

defined by the reflexive-symmetric-transitive closure of the relation R

$$(a, a') \in R \Longleftrightarrow \exists \text{ vertex } \mathtt{i}\colon a, a' \in \mathtt{Buf}(\mathtt{i})$$

then $\mathcal{R}_{\mathcal{M}}$ is called the address closure relation of the message tree $\mathcal{M}(P)$.

The address closure relation $\mathcal{R}_{\mathcal{M}}$ is obviously an equivalence relation. Thus $\mathcal{R}_{\mathcal{M}}$ creates a partition in $\mathtt{Buf}(\mathcal{M})$. The following result is derived directly from the definition of $\mathcal{R}_{\mathcal{M}}$.

Lemma 4.2.1. *Given a vertex* $\mathtt{i}$ *of the message tree, then* $\mathtt{Buf}(\mathtt{i})$ *is included in a unique class of the partition created by* $\mathcal{R}_{\mathcal{M}}$ *in* $\mathtt{Buf}(\mathcal{M})$*.*

The address closure $\mathcal{R}$ then gives an important property about the representation of a path condition under a cartesian product.

Proposition 4.2.8. *Let* $\{\mathtt{Buf}_1, \mathtt{Buf}_2, \ldots, \mathtt{Buf}_n\}$ *be the partition of* $\mathtt{Buf}(\mathcal{M})$ *created by the address closure relation* $\mathcal{R}_{\mathcal{M}}$*. For any condition* $\mathcal{C}[T]$*, let* $\mathcal{C}[T]|_{\mathtt{Buf}_i}$ *denote the natural restriction of* $\mathcal{C}[T]$ *on the equivalence class* $\mathtt{Buf}_i$*, then*

$$\mathcal{C}[T] = \mathcal{C}[T]|_{\mathtt{Buf}_1} \times \mathcal{C}[T]|_{\mathtt{Buf}_2} \times \cdots \times \mathcal{C}[T]|_{\mathtt{Buf}_n} \times \mathtt{Bytes}^{\mathtt{Buf} \setminus \mathtt{Buf}(\mathcal{M})}$$

Proof. Consider the path $T = \mathtt{i}_1 \to \mathtt{i}_2 \to \cdots \to \mathtt{i}_k$, the path condition $\mathcal{C}(T)$ has its conjunction form

$$\mathcal{C}(T) = \mathcal{C}(\mathtt{i}_1 \to \mathtt{i}_2) \wedge \mathcal{C}(\mathtt{i}_2 \to \mathtt{i}_3) \wedge \cdots \wedge \mathcal{C}(\mathtt{i}_{k-1} \to \mathtt{i}_k)$$

and $\mathcal{C}[T] = \mathcal{C}(T) \times \mathtt{Bytes}^{\mathtt{Buf} \setminus \cup_{l=1}^{k-1} \mathtt{Buf}(\mathtt{i}_l)}$. From the conjunction form of $\mathcal{C}(T)$, we observe that if there is a partition of $\mathtt{Buf}$ so that any $\mathtt{Buf}(\mathtt{i}_l)$ is included completely in a unique class of this partition, then $\mathcal{C}[T]$ is the cartesian product of its restrictions on the classes of the partition. Since the partition $\{\mathtt{Buf}_1, \mathtt{Buf}_2, \ldots, \mathtt{Buf}_n\}$ is created from the address closure relation $\mathcal{R}$, the result follows Lemma 4.2.1. □

Example 4.2.12. We reconsider the path T in Example 4.2.11, its conjunction form is

$$(\mathtt{Buf}[0] = \mathtt{H}) \wedge \cdots \wedge (\mathtt{Buf}[3] = \mathtt{P}) \wedge (\mathtt{Buf}[4] = \mathtt{\backslash n}) \wedge (\mathtt{Buf}[5] \neq \mathtt{\backslash n}) \wedge (\mathtt{Buf}[5] = \mathtt{\backslash r})$$

The restriction of $(\mathtt{Buf}[5] \neq \mathtt{\backslash n}) \wedge (\mathtt{Buf}[5] = \mathtt{\backslash r})$ in $\mathtt{Buf}[5]$ is simply $(\mathtt{Buf}[5] = \mathtt{\backslash r})$, then we have the product form of $\mathcal{C}[T]$ as

$$(\mathtt{Buf}[0] = \mathtt{H}) \times \cdots \times (\mathtt{Buf}[3] = \mathtt{P}) \times (\mathtt{Buf}[4] = \mathtt{\backslash n}) \times (\mathtt{Buf}[5] = \mathtt{\backslash r}) \times \mathtt{Bytes}^{\mathtt{Buf} \setminus \{0,1,2,3,4,5\}}$$

Message tree normalization

We obtain now an abstract representation any path T in the message tree $\mathcal{M}(P)$ as the cartesian product of the condition $\mathcal{C}[T]$

$$\mathcal{C}[T] = \mathcal{C}[T]|_{\mathtt{Buf}_1} \times \mathcal{C}[T]|_{\mathtt{Buf}_2} \times \cdots \times \mathcal{C}[T]|_{\mathtt{Buf}_n} \times \mathtt{Bytes}^{\mathtt{Buf} \setminus \mathtt{Buf}(\mathcal{M})}$$

where $\texttt{Bytes}^{\texttt{Buf}\setminus\texttt{Buf}(\mathcal{M})}$ is common to all paths and then we will implicitly omit it. Our next goal is to construct an abstract message tree, from the abstract representation above of paths. Some coming constructions rely on the following hypothesis.

Hypothesis 2 (Static order)**.** For any path $\texttt{i}_1 \to \texttt{i}_2 \to \cdots \to \texttt{i}_k$ in the message tree $\mathcal{M}(P)$, let $\texttt{Buf}_{o(k)}$ denote the equivalence class of $\mathcal{R}(\mathcal{M})$ containing $\texttt{Buf}(\texttt{i}_k)$, then for any $l \leq l'$ we have $o(l) \leq o(l')$.

Intuitively, Hypothesis 2 assumes that the order of parsed addresses in the message buffer Buf does not change over different execution traces. In other words, the message is read from left to right in whatever execution trace. In general cases, this hypothesis is not true (cf. Example 4.2.13 below and also Example 4.2.9), but practically many input message parsers satisfy this hypothesis.

Example 4.2.13. The piece of codes in Listing 4.11 demonstrates that the order of parsed addresses can change over different execution traces. Indeed, let Buf be the input message buffer at the address 0x601040 then we observe two traces, starting from $\texttt{i}_1$

$$\texttt{i}_1 \xrightarrow{\texttt{Buf}[0]=1} \texttt{i}_2 \xrightarrow{\texttt{Buf}[1]=2} \texttt{i}_3 \xrightarrow{\texttt{Buf}[2]=3} \ldots$$
$$\texttt{i}_1 \xrightarrow{\texttt{Buf}[0]\neq 1} \texttt{i}'_2 \xrightarrow{\texttt{Buf}[2]=3} \texttt{i}'_3 \xrightarrow{\texttt{Buf}[1]=4} \ldots$$

where the orders of parsed addresses are different. It is $\texttt{Buf}[0], \texttt{Buf}[1], \texttt{Buf}[2]$ in the first trace, and is $\texttt{Buf}[0], \texttt{Buf}[2], \texttt{Buf}[1]$ in the second one.

```
0x400512    movsx edi, byte [ds:0x601040]  /* first byte */
0x40051a    cmp   edi, 0x1
0x400520    jne   0x400570                 /* i1         */
0x400526    movsx eax, byte [ds:0x601041]  /* second     */
0x40052e    cmp   eax, 0x2
0x400533    jne   0x400564                 /* i2         */
0x400539    movsx eax, byte [ds:0x601042]  /* third      */
0x400541    cmp   eax, 0x3
0x400546    jne   0x400558                 /* i3         */
...
0x400570    movsx eax, byte [ds:0x601042]  /* third      */
0x400578    cmp   eax, 0x3
0x40057d    jne   0x4005ae                 /* i'2        */
0x400583    movsx eax, byte [ds:0x601041]  /* second     */
0x40058b    cmp   eax, 0x4
0x400590    jne   0x4005a2                 /* i'3        */
```

Listing 4.11: Dynamic order parsing

Given the validity of Hypothesis 2, the construction of the abstract message bases on the following result.

Proposition 4.2.9. *Given two different paths T_1 and T_2 with conditions*

$$\mathcal{C}[T_1] = \mathcal{C}[T_1]|_{\texttt{Buf}_1} \times \mathcal{C}[T_1]|_{\texttt{Buf}_2} \times \cdots \times \mathcal{C}[T_1]|_{\texttt{Buf}_n}$$

$$\mathcal{C}[T_2] = \mathcal{C}[T_2]|_{\texttt{Buf}_1} \times \mathcal{C}[T_2]|_{\texttt{Buf}_2} \times \cdots \times \mathcal{C}[T_2]|_{\texttt{Buf}_n}$$

then the two conditions have a unique common prefix, in other words they have form

$$\mathcal{C}[T_1] = \mathcal{C}[T]|_{\texttt{Buf}_1} \times \cdots \mathcal{C}[T]|_{\texttt{Buf}_i} \times \mathcal{C}[T_1]|_{\texttt{Buf}_{i+1}} \times \cdots \times \mathcal{C}[T_1]|_{\texttt{Buf}_n}$$
$$\mathcal{C}[T_2] = \mathcal{C}[T]|_{\texttt{Buf}_1} \times \cdots \mathcal{C}[T]|_{\texttt{Buf}_i} \times \mathcal{C}[T_2]|_{\texttt{Buf}_{i+1}} \times \cdots \times \mathcal{C}[T_2]|_{\texttt{Buf}_n}$$

where $\mathcal{C}[T_1]|_{\texttt{Buf}_{i+1}} \cap \mathcal{C}[T_2]|_{\texttt{Buf}_{i+1}} = \emptyset$.

Proof. Since T_1 and T_2 are different path in the message tree $\mathcal{M}(P)$, they have a unique common prefix and we can write them as

$$T_1 = \texttt{i}_1 \to \cdots \to \texttt{i}_j \to \texttt{i}_{j+1} \to \ldots$$
$$T_2 = \texttt{i}_1 \to \cdots \to \texttt{i}_j \to \texttt{i}'_{j+1} \to \ldots$$

where $\texttt{i}_1 \to \cdots \to \texttt{i}_j$ is the common prefix and $\texttt{i}_{j+1} \neq \texttt{i}'_{j+1}$. By Hypothesis 2, there will be equivalence classes $\texttt{Buf}_1, \texttt{Buf}_2, \ldots, \texttt{Buf}_k$ (for some k), so that they contains $\texttt{Buf}(\texttt{i}_1), \ldots, \texttt{Buf}(\texttt{i}_j)$. Because $\texttt{i}_1 \to \cdots \to \texttt{i}_j$ is the common prefix, then $\mathcal{C}(T_1)$ and $\mathcal{C}(T_2)$ have the same conjunction

$$\mathcal{C}(\texttt{i}_1 \to \texttt{i}_2) \wedge \cdots \wedge \mathcal{C}(\texttt{i}_{j-1} \to \texttt{i}_j)$$

but $\mathcal{C}(\texttt{i}_j \to \texttt{i}_{j+1}) \cap \mathcal{C}\left(\texttt{i}_j \to \texttt{i}'_{j+1}\right) = \emptyset$. Hence, there is some $0 \leq i \leq k-1$ so that

$$\mathcal{C}[T_1]|_{\texttt{Buf}_l} = \mathcal{C}[T_2]|_{\texttt{Buf}_l}$$

for $l = 0, 1, \ldots, i$ but $\mathcal{C}[T_1]|_{\texttt{Buf}_{i+1}} \cap \mathcal{C}[T_2]|_{\texttt{Buf}_{i+1}} = \emptyset$. □

The algorithm in Listing 4.12 constructs a new tree *MsgTree* (called the normalization message tree) from the set cartesian products of conditions $\mathcal{C}[T]$ for all path T of the message tree $\mathcal{M}(P)$. The correctness of the algorithm is given by Proposition 4.2.9, also from the construction of *MsgTree*, we have the following result.

Proposition 4.2.10. *Let* $T_{\mathcal{N}}$ *be a path (from the root* r *to a leaf) in MsgTree*

$$T_{\mathcal{N}} = r \xrightarrow{\mathcal{C}_1} s_2 \xrightarrow{\mathcal{C}_2} \ldots \xrightarrow{\mathcal{C}_n} s_{n+1}$$

then there is a unique path T *in the message* $\mathcal{M}(P)$ *so that*

$$\mathcal{C}[T] = \mathcal{C}_1 \times \mathcal{C}_2 \times \cdots \times \mathcal{C}_n$$

and vice-versa.

`MsgTreeConstruction(`$\mathcal{C}s = \{\mathcal{C}[T] = \mathcal{C}[T]|_{\texttt{Buf}_1} \times \cdots \times \mathcal{C}[T]|_{\texttt{Buf}_n} \mid T \text{ is a path in } \mathcal{M}(P)\}$`) {`
  MsgTree $\leftarrow$ a single vertex r;

```
  foreach (C[T]|_{Buf_1} × ··· × C[T]|_{Buf_n} ∈ Cs) {
    i ← 1; s ← r;
    while (s --C[T]|_{Buf_i}--> s' is an edge of MsgTree) {
      s ← s'; i ← i+1;
    }
    while (i ≠ n) {
      s' ← new vertex;
      add s' and edge s --C[T]|_{Buf_i}--> into MsgTree;
      i ← i+1; s ← s';
    }
  }
}
```

Listing 4.12: Normalized message tree construction algorithm

The following proposition restates a property about the local conditions at vertices applied to the message tree (cf. Proposition 4.2.4), but now applied to the normalized message tree.

Proposition 4.2.11. *Given a (not leaf) vertex s in MsgTree, let $s \xrightarrow{\mathcal{C}_1} s_1, \ldots, s \xrightarrow{\mathcal{C}_n} s_n$ be outgoing edges of s, then the conditions $\mathcal{C}_i$ have the same domain (cf. Remark 4.2.5)*

$$\mathcal{D}(\mathcal{C}_1) = \cdots = \mathcal{D}(\mathcal{C}_n) = \mathcal{D}_s$$

Moreover $\mathcal{C}_i \cap \mathcal{C}_j = \emptyset$ for all $i \neq j$, and $\cup_{i=1}^{n} \mathcal{C}_i = \mathtt{Bytes}^{\mathcal{D}_s}$.

Example 4.2.14. The normalized message tree of `wget` with limit length 60 is given in Figure 4.8. We can observe that if the branchable CFI(s) are direct (i.e. the target are specified in the instruction) then the execution tree is always a binary tree, but even in this case the message tree is not a binary tree in general.

Result By Proposition 4.2.10, the normalized message tree *MsgTree* gives the same partition of the set $\mathtt{Bytes}^{\mathtt{Buf}}$ as the message tree $\mathcal{M}(P)$ (cf. Proposition 4.2.6): *each path corresponds to a unique class of messages*. But *MsgTree* has an important property that *the local conditions in any path are independent*, the tree is the result of this second step and is denoted also by $\mathcal{M}(P)$ where the meaning is clear from the context. It is worth noting that the normalized tree is abstract, it does not keep the notation of control flow instructions in the vertices.

4.2.3 From the message tree to the initial LTS

In the message tree, we now know that each edge is labeled by a condition, which is a subset of $\mathtt{Bytes}^{\mathtt{B}}$ for some subset of parsed addresses $\mathtt{B} \subseteq \mathtt{Buf}$. By Hypothesis 2, we have assumed that the order of parsed addresses is fixed, in other words this order is included implicitly in the message tree. So we can omit the addresses in each subset

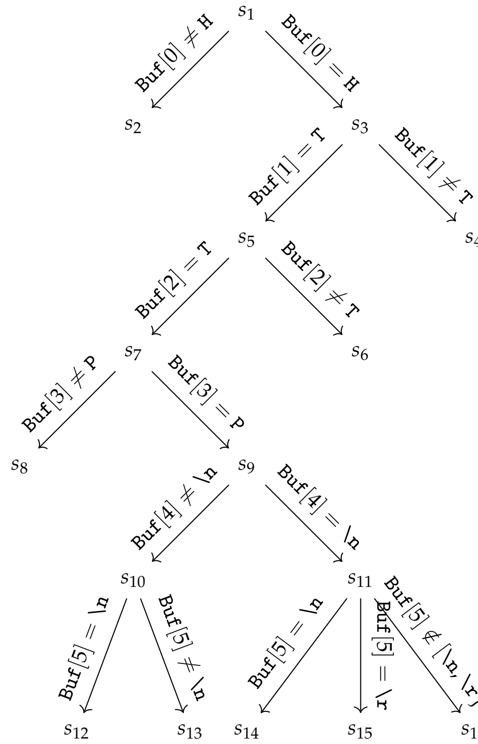


Figure 4.8: Normalized message tree of wget with limit length 60

of parsed addresses B and keep only their values, so the condition on each edge is now a subset of values of $\texttt{Bytes}^{|B|}$. By this way, we transform the message tree into a labeled transition system $\mathcal{A}\langle Q, \iota, \Sigma, \delta \rangle$ where

- Q consists of all vertices of the message tree,
- ι is the root of the tree,
- Σ consists of all conditions on edges of the message tree, and
- $\delta(s, c) = s'$ if and only if there is an edge $s \xrightarrow{c} s'$ in the message tree.

and we can also observe that the terminal states correspond to the leaves of the message tree. Since each path of the messages tree corresponds to a unique class of messages, *each path from the initial state to a terminal state in the initial LTS corresponds to a unique class of messages*. This transformed LTS is the result of this abstraction step, and we call it the initial LTS, and denote it $\mathcal{A}_i(P)$.

Example 4.2.15. The LTS in Figure 4.9 is transformed from the message tree in Figure 4.8. The initial state is s_1, the terminal states are $\perp$-labeled.

4.2.4 From the initial to the final LTS

In this step, we simply apply the algorithms in Section 3.3 to search for π-equivalences in the initial LTS and construct a normal form. This normal form is called the final

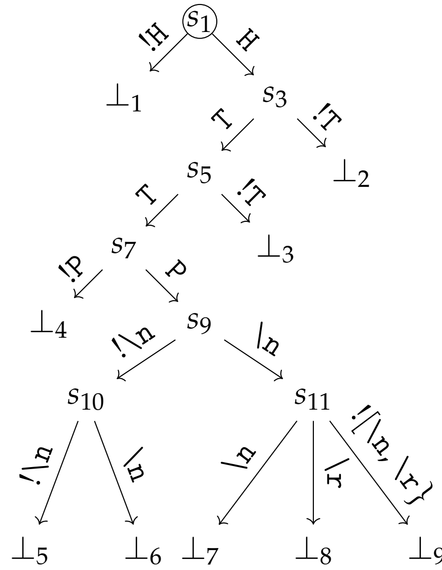


Figure 4.9: Initial LTS of `wget` with limit length 60

LTS, denoted by $\mathcal{A}_n(P)$ and it is also the result of this abstraction step.

Example 4.2.16. By the stepwise abstraction, the initial LTS in Figure 4.10a is constructed from the execution tree of `wget` where the execution traces are limited to length 65. We can verify that the following equivalence

$$\begin{aligned}\sim = \{&\{s_1\}, \{\perp_1\}, \{s_2\}, \{\perp_2\}, \{s_3\}, \{\perp_3\}, \{s_4\}, \{\perp_4\}, \{s_5, s_6, \perp_5\},\\ &\{s_7, s_8\}, \{\perp_7, \perp_{10}\}, \{\perp_8, \perp_{11}\}, \{\perp_9, \perp_{12}\}\}\end{aligned}$$

is a π-equivalence. The quotient of the initial LTS by $\sim$ is the quotient LTS in Figure 4.10b. The final LTS (i.e. the normal form of the initial LTS) is shown in Figure 4.10c (for illustrative purpose, we leave a terminal state in this final LTS, theoretically this state can be merged into any other state (cf. Definition 3.2.2)).

4.2.5 Message classification by the final LTS

We recall that each (finite) path from the initial state to a terminal state in the initial LTS $\mathcal{A}_i(P)$ corresponds to a distinguished class of messages, namely to a distinguished subset of $\texttt{Bytes}^{\texttt{Buf}}$.

In the final LTS $\mathcal{A}_n(P)$, which is an observational prediction of $\mathcal{A}_i(P)$, because there may exists loop-back vertices (cf. Figure 4.10c) then the paths from the initial state are infinite in general. Each path in $\mathcal{A}_n(P)$ still corresponds to a unique class of messages but now we can get an *arbitrary refined classification* depending the length of the path. More precisely, a (finite) prefix of a path corresponds to a unique class of messages, but the extensions of this prefix will continue classify this class: each extension corresponds to a unique subclass of messages.

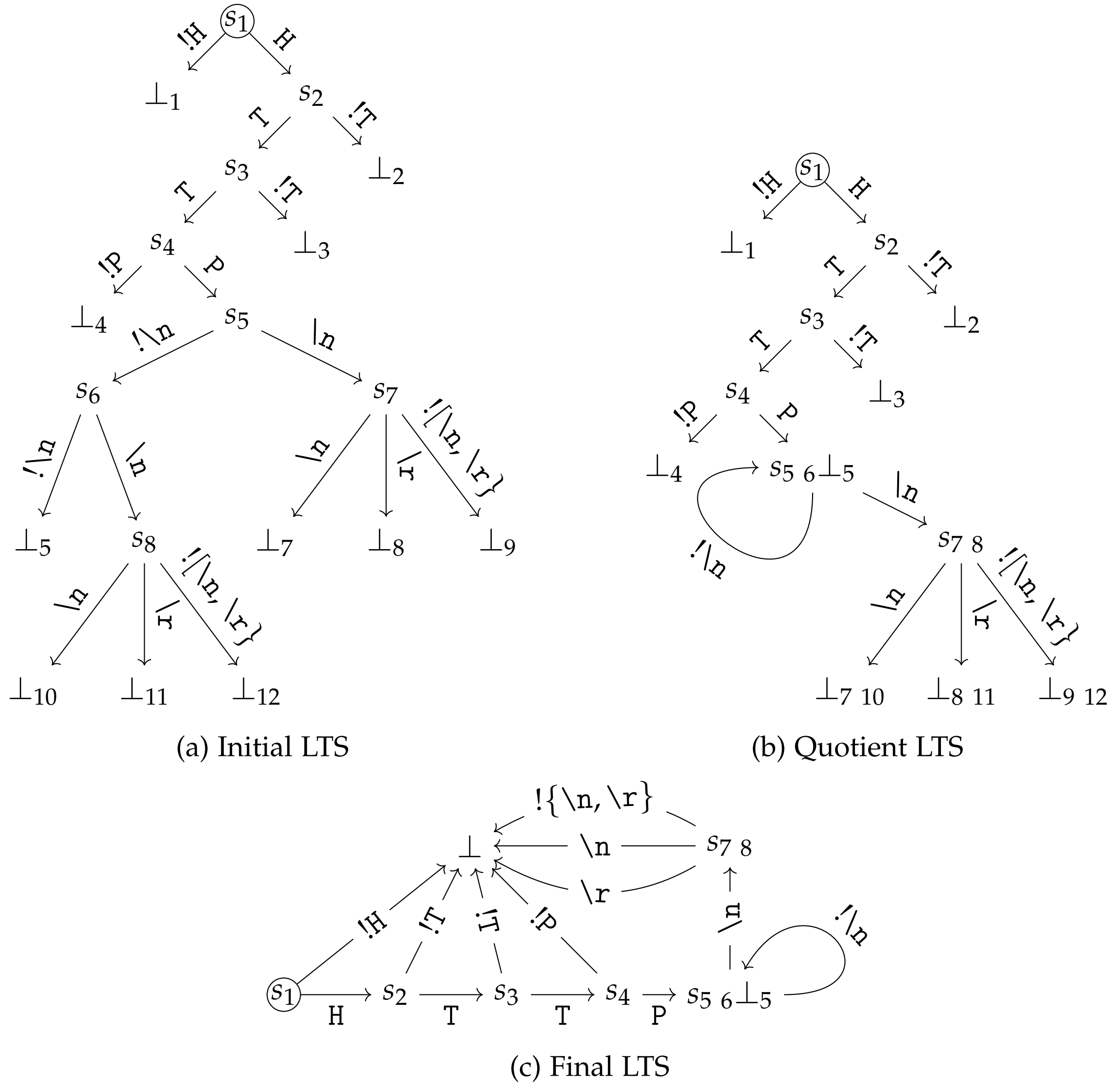


Figure 4.10: LTS(s) of `wget` with limit length 65

The reasoning about the format of messages on $\mathcal{A}_n(P)$, as illustrated in Example 4.2.17 and remark 4.2.6 below, does not look like the reasoning on the previous work in the message format extraction [26, 28, 41, 45]. But in many cases we will get a similar or even more precise result.

Example 4.2.17. We observe that the paths (from the initial state to the terminal state) in this final LTS distinguish the following classes of messages

$\texttt{!H} \cdot \texttt{Bytes}^{n-1}$ $\quad\quad$ $\texttt{H}\cdot\texttt{!T} \cdot \texttt{Bytes}^{n-2}$

$\texttt{H} \cdot \texttt{T}\cdot\texttt{!T} \cdot \texttt{Bytes}^{n-3}$ $\quad\quad$ $\texttt{H} \cdot \texttt{T} \cdot \texttt{T}\cdot\texttt{!P} \cdot \texttt{Bytes}^{n-4}$

$\texttt{H}\dot{\texttt{T}} \cdot \texttt{T} \cdot \texttt{P} \cdot (\texttt{!\textbackslash n})^k \cdot \texttt{\textbackslash n} \cdot \texttt{\textbackslash n} \cdot \texttt{Bytes}^{n-k-6}$ $\quad\quad$ $\texttt{H} \cdot \texttt{T} \cdot \texttt{T} \cdot \texttt{P} \cdot (\texttt{!\textbackslash n})^k \dot{\texttt{\textbackslash n}} \cdot \texttt{\textbackslash r} \cdot \texttt{Bytes}^{n-k-6}$

$\texttt{H} \cdot \texttt{T} \cdot \texttt{T} \cdot \texttt{P} \cdot (\texttt{!\textbackslash n})^k \cdot \texttt{\textbackslash n}\cdot\texttt{!}\{\texttt{\textbackslash n}, \texttt{\textbackslash r}\} \cdot \texttt{Bytes}^{n-k-6}$

for $k = 0, 1, \ldots$ and n is the length of `Buf`. So one can understand what is a well-formatted message, as decided by `wget`, such a message should at least have it first 4 bytes as $\texttt{H} \cdot \texttt{T} \cdot \texttt{T} \cdot \texttt{P}$ (cf. Remark 4.2.6). Moreover, by considering the class

$$\texttt{H} \cdot \texttt{T} \cdot \texttt{T} \cdot \texttt{P} \cdot (\texttt{!\textbackslash n})^k \cdot \texttt{\textbackslash n} \cdot \texttt{\textbackslash n} \cdot \texttt{Bytes}^{n-k-6}$$

we can see that how the program process its input messages it matches the first 4 bytes with $\texttt{H} \cdot \texttt{T} \cdot \texttt{T} \cdot \texttt{P}$, and continue accept non `\n` characters, etc. This information is obtained by the stepwise abstraction just described above, *without manual efforts in reverse engineering the program*.

Remark 4.2.6. Obviously, this is only an incomplete information about the message. As discussed in Section 3.2.3, the prediction capability of the final LTS depends also on the observed information (i.e. the initial LTS), here because the limited length is quite short then the final LTS cannot give complete information about the message.

Example 4.2.18. The final LTS considered in Example 4.2.17 gives incomplete information about the format of messages because of the limited length of execution traces is short. The final LTS given in Figure 4.11 whose the initial state is visualized by cornflower blue, is the result of the stepwise abstraction where the limited length of traces is 90. It indeed gives a complete information which can be reverified by comparing with the result of the manual reverse engineering.

Summary The stepwise abstraction procedure with different construction steps, from execution traces to a final LTS, is the practical guide for our approach in message format classification. In the next section, we will present an implementation following this guide and experimental results.

4.3 Implementation and experimentation

We have implemented a prototype following the *stepwise abstraction procedure* described in Section 4.2. This prototype consists of about 16.500 lines of C/C++ code, and it is available online at [16]. In this section, we first present some new technical solutions implemented in the prototype. Next, we gives some experimentations on real-world programs which test the theoretical results of the thesis.

4.3.1 Implementation

The practical realization of the stepwise abstraction procedure starts with the exploration of all execution traces. This is known as the *code coverage problem* [22, 69, 118] as discussed in Section 2.2.2. We do not use the IR code lifting (cf. Remark 4.3.1), instead we reason directly on the semantics of `x86/x86-64` instructions with help of Pin Dynamic Binary Instrumentation (abbr. DBI) framework [104].

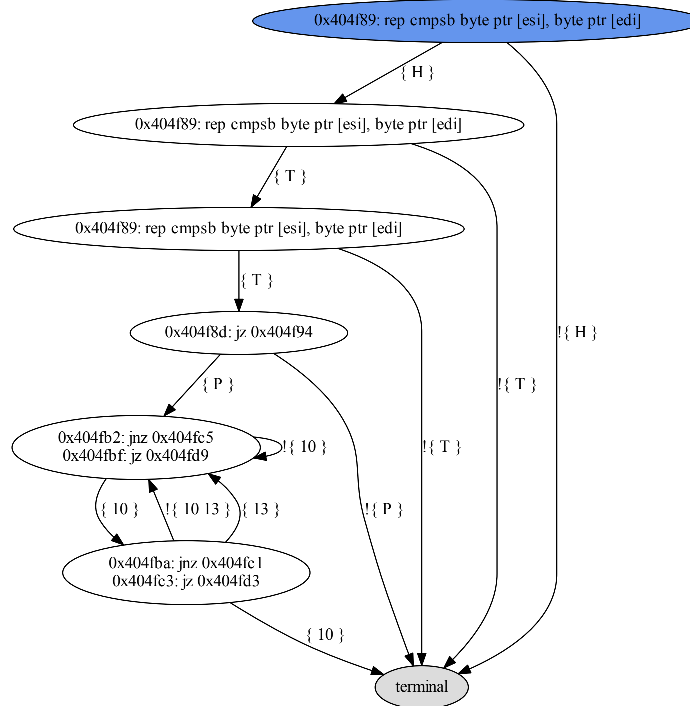


Figure 4.11: Final LTS of `wget` with limit length 90

Remark 4.3.1. The current approach [22, 118] uses the dynamic analysis with the help of constraint solvers [69]. This is a modern trend and scalable whenever the constraints, which are constructed by lifting the binary codes to IR codes, are well established. The disadvantage is mostly technical, the lifting cannot be always obtained because the immaturity of current supported platforms.

For example, the authors in [22] have used their own open source BAP platform, there are still several technical difficulties in working with BAP: its lifting tool `iltrans` does not allow translating coherently an arbitrary piece of binary codes to its BIL code (whereas reasoning on a complete translation of a program is very sophisticated), BAP works only on Linux platform, and until the currently newest version (0.8) the work on the `x86-64` instruction set is not yet complete. The authors in [118] use their private platform, while the theoretical approach [69] is clear, the technical detail of lifting is unknown.

Pin DBI framework and Pintool implementation Roughly speaking, a DBI framework has similar principles with a just-in-time (abbr. JIT) compiler. Here the binary codes of the original instrumented program will be modified (i.e. recompiled) by instrumenting codes (written by users), and the DBI framework manage this modification on-the-fly, in both running time and loading time [121]. There are currently sev-

eral DBI implementations, for example, DynInst [11], Pin [76, 104], DynamoRIO [21], Valgrind [122], etc with different approaches in code instrumentation. Among them Pin is one the most popular framework.

Concretely, Pin fetches the original binary codes of the examined program and modifies them depending on the instrumenting codes. The modified codes, stored in a code cache managed by Pin, will be executed instead of the original codes. The instrumenting codes are written by users under form of Pintools. The Pintools can be arranged to get both loading (i.e. initializing) and running time information about the original codes[1]. More importantly, this instrumentation is transparent from the original codes, namely the original codes do not recognize the existence of the instrumenting codes[2].

Using the Pin DBI framework, we have implemented a Pintool [16] which realizes the following functions:

- a *dynamic tainting analysis* [123, 142] (abbr. DTA) to detect, for each control flow instruction, the nearest execution point from which the branching decision of the CFI is affected,
- a *reverse execution* [13] engine allowing the program rollbacks from any execution point.

While the former technique is quite popular, some other authors have actually given their own implementation prototype of the DTA based on Pin (e.g. [37, 86]). The latter, about the implementation of a rollback mechanism, does not capture yet much interest.

The current implementation of rollback mechanism is to use some system emulator for saving/restoring program states. For example, the authors in [118] have implemented a QEMU [10] component to snapshot-ing completely the virtual memory space of program. Such implementation is indeed heavyweight.

Pin can be considered as a process-level virtual machine [150], our implementation using Pin is lightweight since it tracks only the original values of the overwritten memory addresses, and employs the mechanism of saving/restoring registers states of Pin (with API(s) `Pin_SaveContext`, `Pin_ExecuteAt`, etc). The source codes of this implementation is obviously available at [16].

4.3.2 Experimentation

We have tested the prototype Pintool in several programs, all of them are 32 bit binary codes in Windows™platform, the experimentations are realized on a regular machine (Intel®Core i5 − 3230M CPU with 4G RAM). The following experimental results will test some theoretical details discussed in the previous chapters.

[1] The current code injection mechanism of Pin does not allow Pintools getting loading information of `ntdll.dll`, `kernel32.dll` and `kernelbase.dll` libraries [150].

[2] Strictly speaking, the code instrumentation is not completely transparent, some authors have proposed methods to detect whether a program running under Pin or not [59].

Code coverage

The first class of experimentation concerns the first step of the stepwise abstraction: we need to collect the set of different execution traces resulted from changing the value of the input message buffer.

Number of explored control flow instructions As discussed in Section 4.2.1, the execution traces form the execution tree and the number of control flow instructions increases exponentially with the limit length of execution traces.

Moreover in the execution tree there may exist *unbranchable* CFI(s) (cf. also Definition 4.2.1). Here, we aim at a special class of CFI(s) which keep their branching decision for all values of the input message. This class consists of CFI(s) which depend on the input, but they keep always their branching decision. That distinguishes from the CFI(s) which are independent from the input message, namely they may change their branching conditions because of some other stimulus, not the input message.

The existence of this class of CFI(s), though have been predicted in the theoretical part, but seems to be illogical or at least showing some code redundancies. We will give a detail discussion about this phenomenon at the last part of this section.

In Figure 4.13, we can observe the exponential growth of CFI(s) in real-world programs. Interestingly enough, except the case of the HTTP parsers of `wget` and `links`; and of the FTP parser of `ncftpget` (cf. also detail information in Tables 4.1 to 4.3), other parsers do not contain unbranchable CFI(s). It is worth recalling that, the number of CFI(s) is about $2^{\mathcal{O}(n)}$ where n is the limit length. Here the CFI(s) are distinguished also by their proposition in distinguished traces, in fact the number of real CFI(s) (which are distinguished only by their addresses) in the program's parser is much more smaller.

Covering time of control flow instructions The running time of our Pintool in covering all branchable CFI(s), evidently, is linear with the number of CFI(s) and vary with different programs. Since the number of CFI(s) increases exponentially with the limit length n of traces, the running time increases also exponentially with n (cf. Figure 4.12 and also detail information in Tables 4.1 and 4.2).

Thanks of the lightweight reverse execution engine, the running time does not exceed 500 seconds in all realized tests. Such a modest amount of time is hardly obtained if a traditional system-wide state saving/restoring is deployed.

Labeled transition systems, message classification and semantics

In Section 4.2.5, we have discussed the classification of messages on the final LTS and given an exemplified classification for the case of `wget` (cf. also Figure 4.11), we will present other examples below. Because our approach is automatic and generic, the presentations would be similar for different programs. Then, we first present in detail the program `links` [96] and its processing for the first input message (but

trace length	branchable	unbranchable	total	covering time(s)
30	4	10	14	6
35	4	11	15	7
40	4	12	16	7
45	6	12	18	7
50	10	13	23	8
55	14	16	30	8
60	20	20	40	8
65	31	25	56	9
70	50	34	84	10
75	79	50	129	12
80	124	75	199	14
85	198	114	312	20
90	319	177	496	30
95	512	279	791	47
100	821	442	1263	70
115	3412	1783	5195	269
120	5483	2845	8328	402
125	8809	4541	13350	559
130	14146	7248	21394	982

Table 4.1: Number of CFI(s) in the execution tree of (the `HTTP` parser of) `wget`

with different values). This program is an open source text-based web browser with console interface. But here we are not interested in its sources, we instead grasp its 32-bit compiled binary codes for Windows at [96].

For the program, we first give its execution tree, initial and final LTS(s)[1]. We explain also some imprints of the stepwise abstraction procedure in these objects. Next, we present the message classification reasoning based on the final LTS. Finally and the most importantly, we give a clear comparison for different message semantics given by different program programs (cf. also Section 2.3.2), what is implicitly discussed in previous researches about message format extraction [26, 28, 41, 45], but has not yet any explicit discussion. At the end, we will give briefly results for other programs.

Execution tree, initial and final labeled transition systems The *final LTS* is given in Figure 4.15a, whereas several much more sophisticated *initial LTS(s)* are given in Figure 4.19. These LTS(s) are constructed (by the stepwise abstraction procedure discussed in Section 4.2) from the sets of execution traces, limited at length of 145 (cf. Figure 4.19a), of 160 (cf. Figure 4.19b) and of 175 instructions (cf. Figure 4.19c), respectively. We can verify the validity of the construction by checking the condition

[1] The message tree is not given because it is graphically almost identical with the initial LTS. We need only to remember that the initial LTS is the message tree where the order of processed bytes of input messages are considered to be static and known (cf Hypothesis 2).

trace length	branchable	unbranchable	total	covering time(s)
130	31	15	46	7
135	42	19	61	9
140	50	19	69	10
145	56	23	76	10
150	74	30	104	14
155	89	31	120	16
160	102	34	136	20
165	129	48	177	28
170	158	53	211	32
175	186	54	240	39
180	230	79	309	56
185	281	95	376	60
190	341	100	441	86
195	415	130	545	90
200	496	167	663	103
210	755	223	978	105
215	877	292	1169	113
220	1134	361	1495	152
225	1372	395	1767	191
230	1571	497	2068	231
235	2035	670	2705	271
240	2488	722	3188	493

Table 4.2: Number of CFI(s) in the execution tree of `links`

given in Proposition 4.2.7 for these LTS.

The *execution trees* are even much more sophisticated, we given in Figure 4.20 an execution tree where execution traces are limited at length of just 100 instructions.

All of these objects are constructed automatically, without any manual intervention, by our Pintool. Whereas the manual reverse code engineering to understand the parser of `links` will touch some codes as given in Figure 4.22.

Control flow instructions, nodes and states The execution tree is a binary tree whose each branching node corresponds to exactly an CFI, but the inversion is not true. By our implementation of the tainting analysis, we can detect that some CFI(s) are independent actually from the input message. In Figure 4.21, we give the simplified execution tree of one in Figure 4.20 where only input dependent CFI(s) are kept. More interestingly, we see that some CFI(s) are input dependent but *they does not branch by any values of the input message* (cf. CFI(s) in blue). We will discuss in more detail about this phenomenon later.

The initial (and then the final) is not a binary tree in general, moreover each of their states can correspond to several CFI(s) (cf. Figures 4.19a to 4.19c). That indeeds shows

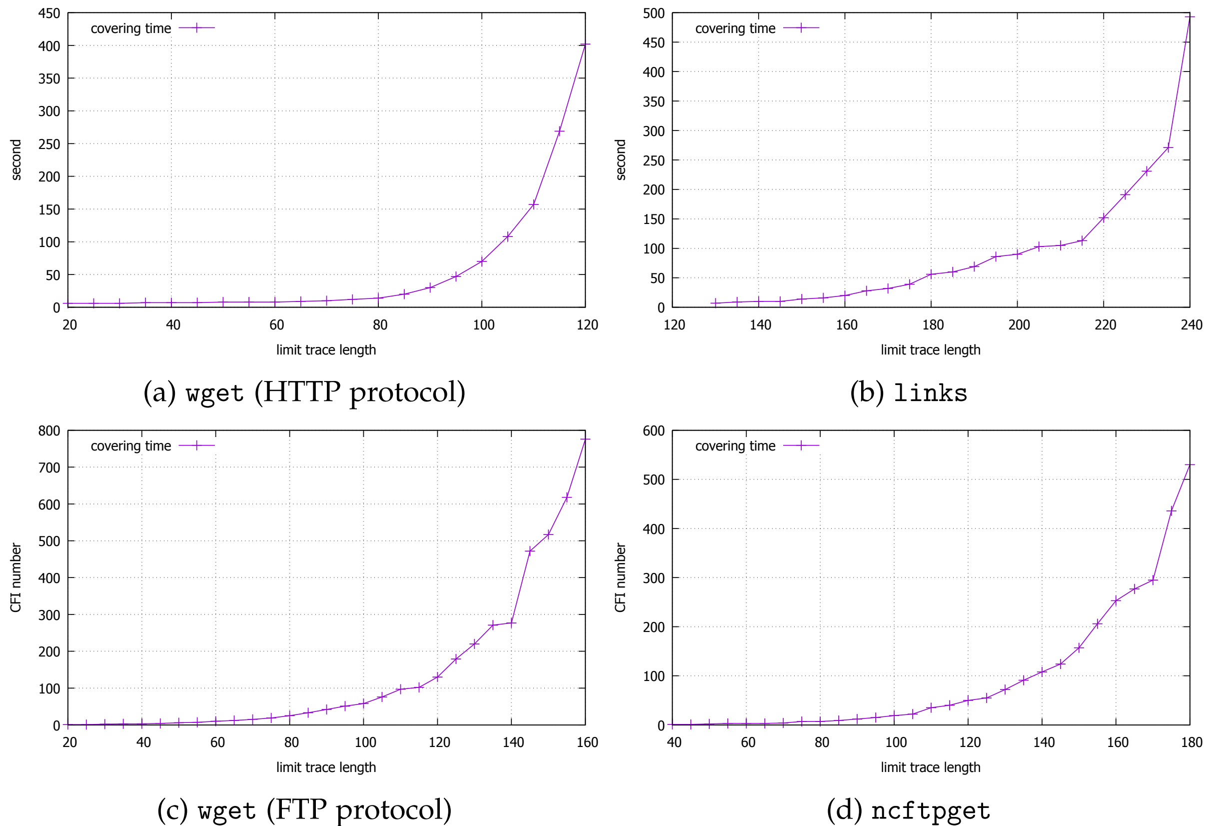
trace length	branchable	unbranchable	total	covering time(s)
40	12	3	15	1
45	15	4	19	1
50	17	5	22	2
55	23	7	30	3
60	29	9	38	3
65	33	12	45	3
70	45	15	60	4
75	56	21	77	7
80	65	24	89	7
85	84	30	114	9
90	104	41	145	12
95	126	50	176	15
100	159	62	221	19
105	193	80	273	22
110	244	96	340	35
115	299	121	420	40
120	358	155	513	50
125	467	187	654	55
130	563	239	802	72
135	670	295	965	91
140	886	362	1248	108
145	1064	463	1527	124
150	1264	565	1829	157
155	1663	698	2361	206
160	2012	899	2911	253
165	2396	1084	3480	277
170	3112	1347	4459	295
175	3805	1723	5528	436
180	4558	2085	6643	530

Table 4.3: Number of CFI(s) in the execution tree of `ncftpget`

the imprint of the message tree normalization (cf. Section 4.2.2) where we will group several CFI(s) using the address closure relation (cf. Definition 4.2.3). For example, we can observe that 3 CFI(s) respectively at `0x4680f`, `0x468136` and `0x468143` are grouped into a node in the initial LTS.

Example 4.3.1. To understand why these CFI(s) have been grouped into a single node, we consider in Figure 4.14 (which are obtained by disassembling the binary codes of `links`) how these CFI(s) are used.

Indeed, these CFI(s) will divert the control flow depending on the value of the register `dl`, which stores in fact the value of a byte of the input message buffer. We can observe that this value is compared with `0x0d`, `0x0a` and `0x00`. That fits also with their corresponding state: we can observe that this state have 4 branches depending



(a) `wget` (HTTP protocol) (b) `links`

(c) `wget` (FTP protocol) (d) `ncftpget`

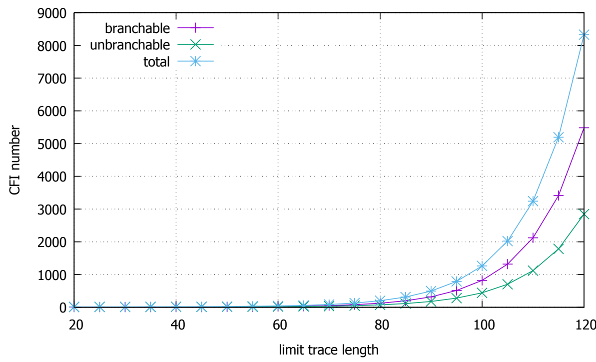
Figure 4.12: Code coverage running time

on whether the value of the examined byte is `0x0d`, `0x0a`, `0x00` or none of them.

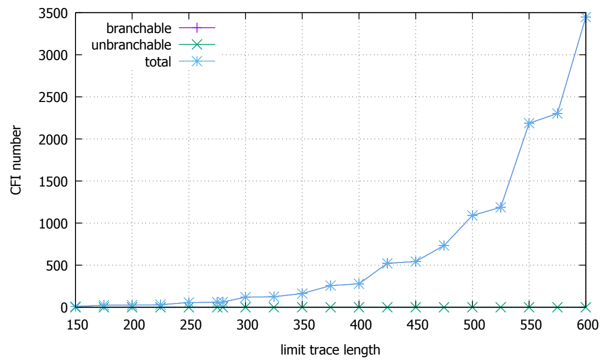
Remark 4.3.2. It might be worth noting that the analysis given in Example 4.3.1 is just a demonstration for what happens inside our Pintool. We do not analyze the program manually, indeed the Pintool detects automatically these situations and constructs the initial LTS.

Message classification When we increase the limit length, the initial LTS(s) become more and more sophisticated (cf. Figures 4.19a to 4.19c) but the final LTS (cf. Figure 4.15a), which is a minimal *observational prediction* of the initial LTS(s) (cf. Section 3.2.2), is much more smaller. More importantly, the final LTS contains circles, that means it can predict the behaviors of the program for arbitrarily long messages. These properties justify our approach to the message classification based on the final LTS.

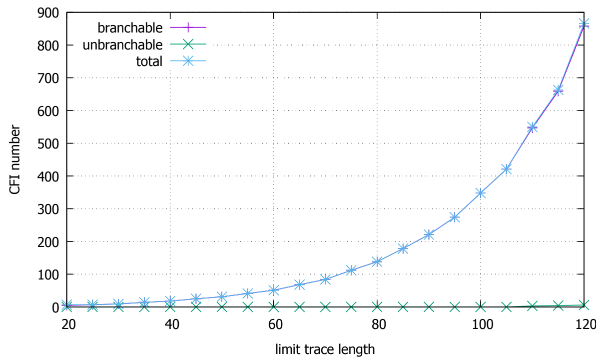
In Figures 4.15a to 4.15c, we note that 0, 10 and 13 are respectively ASCII codes of `NULL`, `line-feed` (abbr. LF or `\n`) and `carriage-return` (abbr. CR or `\r`) characters. We keep also a terminal states for a comprehensive explanation, theoretically this state can be merged into any other state (cf. Definition 3.2.2). We first observe that the following classes of messages, characterized by the paths from the initial state to the



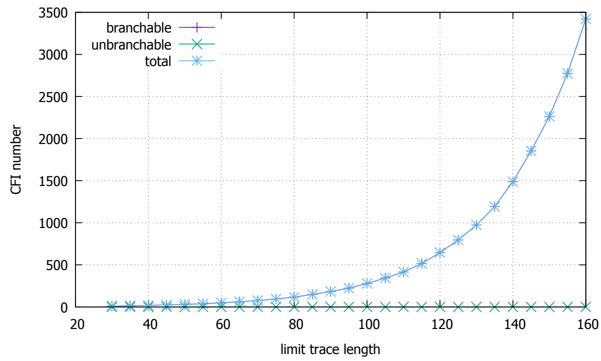
(a) `wget` (HTTP protocol)



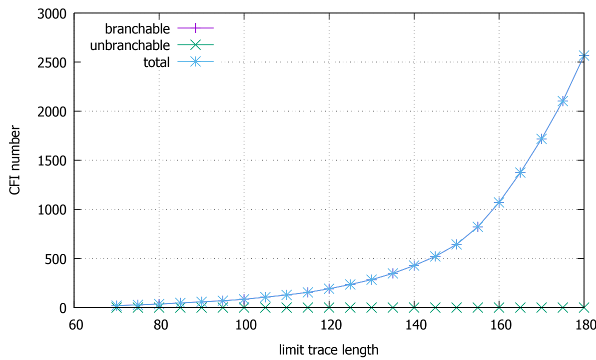
(b) `nginx`



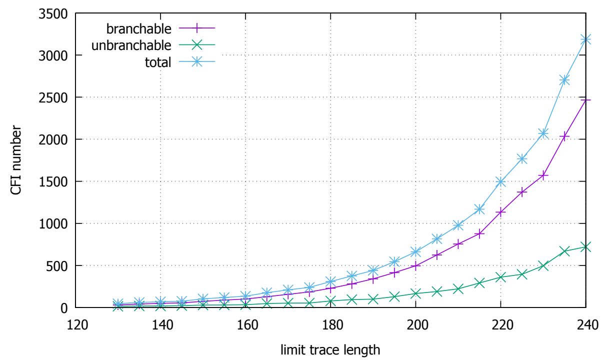
(c) `curl`



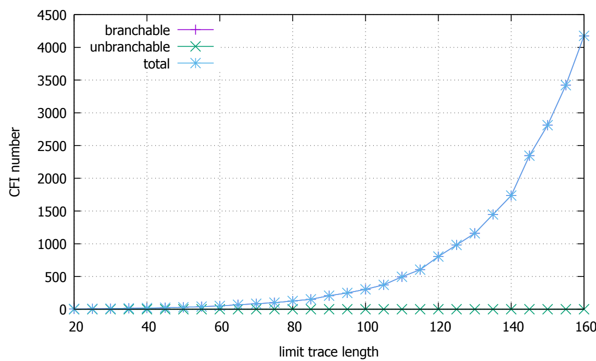
(d) `jwhois`



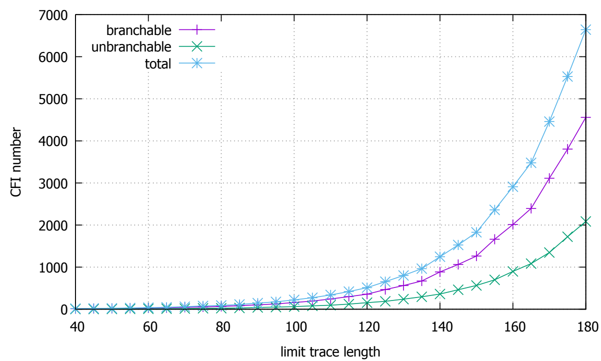
(e) `netcat`



(f) `links`



(g) `wget` (FTP protocol)



(h) `ncftpget`

Figure 4.13: Exponential growth of CFI(s) in execution trees

```
0046810C  80 FA 0D             cmp dl,D
0046810F  0F 84 80 0D 00 00    je links.468E95
00468115  39 44 24 48          cmp dword ptr ss:[esp+48],eax
00468119  0F 8E 5D 01 00 00    jng links.46827C
0046811F  8B 9C 24 E4 00 00 00 mov ebx,dword ptr ss:[esp+E4]
00468126  83 C6 01             add esi,1
00468129  83 C1 01             add ecx,1
0046812C  0F B6 54 03 10       movzx edx,byte ptr ds:[ebx+eax+10]
00468131  83 C0 01             add eax,1
00468134  84 D2                test dl,dl
00468136  0F 84 18 01 00 00    je links.468254
0046813C  39 FE                cmp esi,edi
0046813E  7D C8                jge links.468108
00468140  80 FA 0A             cmp dl,A
00468143  75 C3                jnz links.468108
```

Figure 4.14: Different CFI(s) process the same group of bytes

terminal state, are distinguished by the program

$$\texttt{!H} \cdot \texttt{Bytes}^{n-1} \qquad \texttt{H}\cdot\texttt{!T} \cdot \texttt{Bytes}^{n-2}$$
$$\texttt{H} \cdot \texttt{T}\cdot\texttt{!T} \cdot \texttt{Bytes}^{n-3} \qquad \texttt{H} \cdot \texttt{T} \cdot \texttt{T} \cdot \texttt{T}\cdot\texttt{!P} \cdot \texttt{Bytes}^{n-4}$$

From these classes, because the number of meaningful values in these classes is quite small, we can make a strong hypothesis that the program may reject the received message if the first 4 bytes are not $\texttt{H} \cdot \texttt{T} \cdot \texttt{T} \cdot \texttt{P}$. More precisely, it will check byte-by-byte, and may reject the message immediately if it detects any unmatch with $\texttt{H} \cdot \texttt{T} \cdot \texttt{T} \cdot \texttt{P} \cdot \texttt{Bytes}^{n-4}$. Next, the following class of message

$$\texttt{H} \cdot \texttt{T} \cdot \texttt{T} \cdot \texttt{P} \cdot \texttt{NULL} \cdot \texttt{Bytes}^{n-5}$$

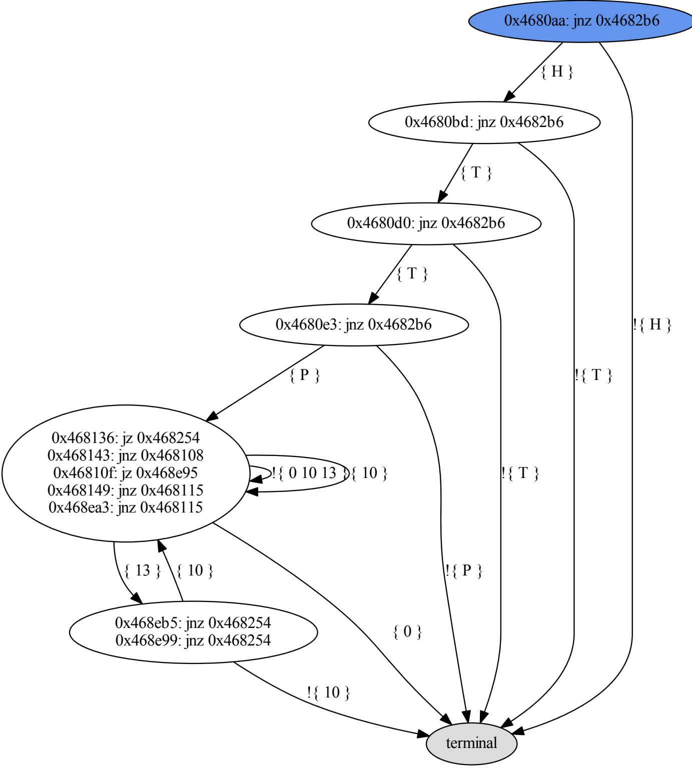
also gives a strong hypothesis that the program may reject the message if if detects that the 5-th byte is NULL, given the first 4 bytes are $\texttt{H} \cdot \texttt{T} \cdot \texttt{T} \cdot \texttt{P}$.

More interestingly, the following classes of messages (categorized into two categories) are distinguished by the program

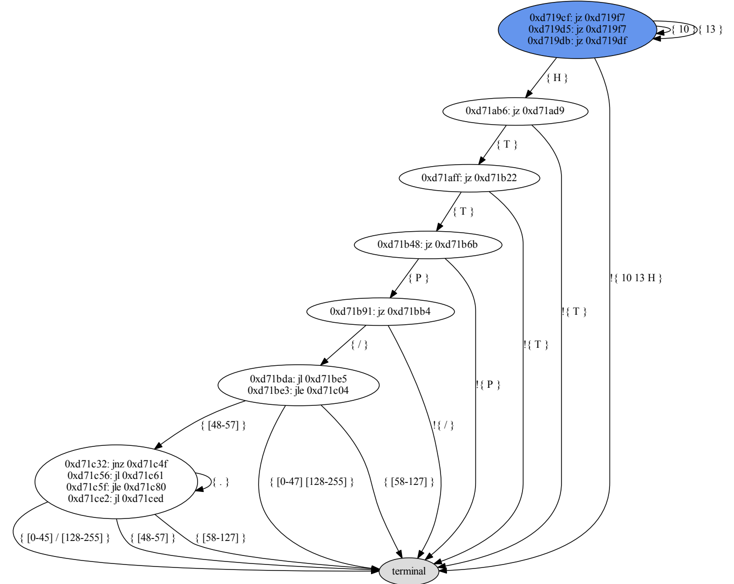
$$\texttt{H} \cdot \texttt{T} \cdot \texttt{T} \cdot \texttt{P} \cdot !\{\texttt{\textbackslash r}, \texttt{\textbackslash n}\}^k \cdot \texttt{NULL} \cdot \texttt{Bytes}^{n-5-k}$$
$$\texttt{H} \cdot \texttt{T} \cdot \texttt{T} \cdot \texttt{P} \cdot (\texttt{\textbackslash r} \cdot \texttt{\textbackslash n})^l \cdot \texttt{\textbackslash r}\cdot!\texttt{\textbackslash n} \cdot \texttt{Bytes}^{n-6-2l}$$

for $k = 1, 2 \ldots$. The first category means that, after first 4 bytes $\texttt{H} \cdot \texttt{T} \cdot \texttt{T} \cdot \texttt{P}$ are matched, the program may continuously search for a character \r or \n, but whenever it meets a character NULL, it may reject or stop parsing the message.

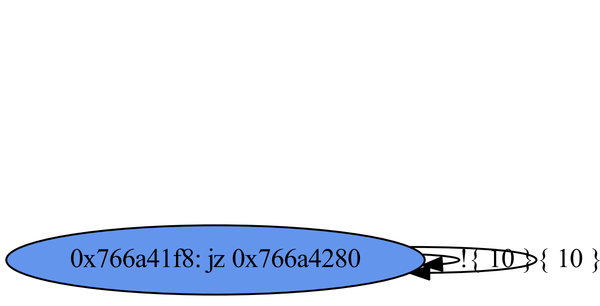
The second category means that the programs may separate the message into *fields* separated by the pattern (or *separator*) $\texttt{\textbackslash r} \cdot \texttt{\textbackslash n}$, but whenever it finds a pattern $\texttt{\textbackslash r} \cdot \texttt{\textbackslash n}$, it will reject or stop parsing the message. Similarly, one can extract other classes of messages to understand how the program treat an input message.



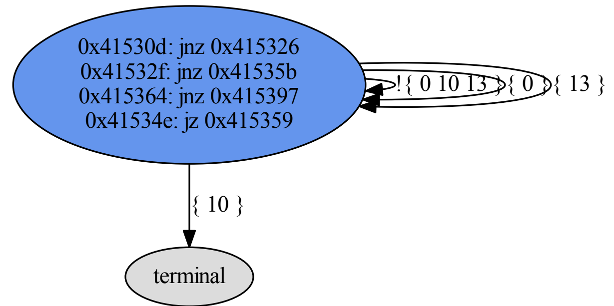
(a) `links`



(b) `nginx`



(c) `netcat`



(d) `ncftpget`

Figure 4.15: Final labeled transition systems

Message classification and message format In the interpretation above about path-based message classification, one can see that the notations of *field*, *separator* can occur naturally. But different from other researches [26, 28, 41, 45], in constructing the final LTS by the stepwise abstraction procedure, we do not need give any special assumption about the existence of such notations.

Message semantics given by a program Practically, we know that the program `links` should consume messages of the HTTP protocol, and so do the programs `nginx` and `wget`. Now considering the final LTS(s) of these programs, given in Figures 4.11, 4.15a and 4.15b respectively, one can observe that these LTS(s) gives different message classifications. Or using a classic notation, one can say that there are 3 different message formats, for the same HTTP protocol.

Actually, we can see clearly here the argument discussed implicitly in other researches [26, 28, 41, 45] that *the program gives semantics for the message*. Given messages of the same protocol (here it is HTPP), different programs give different semantics for these messages.

For an extreme case, one can consider the LTS of given in Figure 4.15c of the program `netcat`[1], and then conclude that the message semantics given by this program is very simple. It fits with the actual operation of `netcat`: the program only establishes connection to a target and send/receiving messages, but do not take any special interest in the message formats.

Relative prediction In Section 3.2.1, we have stated that the final LTS is a predictive model, constructed from observable behaviors (here they are execution traces), of the real program. But the prediction given by the final LTS is also *relative* (cf. also Example 3.2.1). Practically, that means a final LTS constructed from execution traces where the limit length is too short, may give a wrong prediction about the program. The LTS in Figure 4.16 is constructed from execution traces of `links` where the limit length is just 100.

This final LTS gives indeed a wrong prediction, for example one can first observe the following class of messages

$$\mathtt{H} \cdot \mathtt{T} \cdot \mathtt{T} \cdot \mathtt{P} \cdot \mathtt{\backslash r} \cdot \mathtt{\backslash n} \cdot \mathtt{\backslash r} \cdot !\{\mathtt{H}\} \cdot \mathtt{Bytes}^{n-8}$$

and then make a hypothesis that the program will reject or stop parsing the message if the first 7 bytes are not $\mathtt{H} \cdot \mathtt{T} \cdot \mathtt{T} \cdot \mathtt{P} \cdot \mathtt{\backslash r} \cdot \mathtt{\backslash n} \cdot \mathtt{\backslash r}$; and even the first 7 bytes are matched, the program still rejects (or stops parsing) the 8-th byte is not `H`. This hypothesis is actually denied by the final LTS in Figure 4.15a, constructed from execution traces with longer limit length.

Remark 4.3.3. The visual representation of LTS(s) are drawn automatically by our Pintool thanks to the Graphviz visualization software [72]. Because of the optimal layout algorithm for drawing directed graph of this software [67], the initial state (characterized by the unique node in cornflower blue) in some final LTS(s) (e.g. in Figure 4.16) is not drawn by the node on the top.

[1]The input message is obtained by executing netcat with a parameter specifying the target HTTP server and a parameter specify the sending command (here it is `GET....` The program establishes a TCP connection to the server, and send the command over this connection, and finally receives the response message of the server.

But one can verify easily which is the initial state by checking the top node of an initial LTS and find its corresponding node in the final LTS. Because an initial LTS is always a tree, its initial state is always the node on the top (cf. Figures 4.19a to 4.19c).

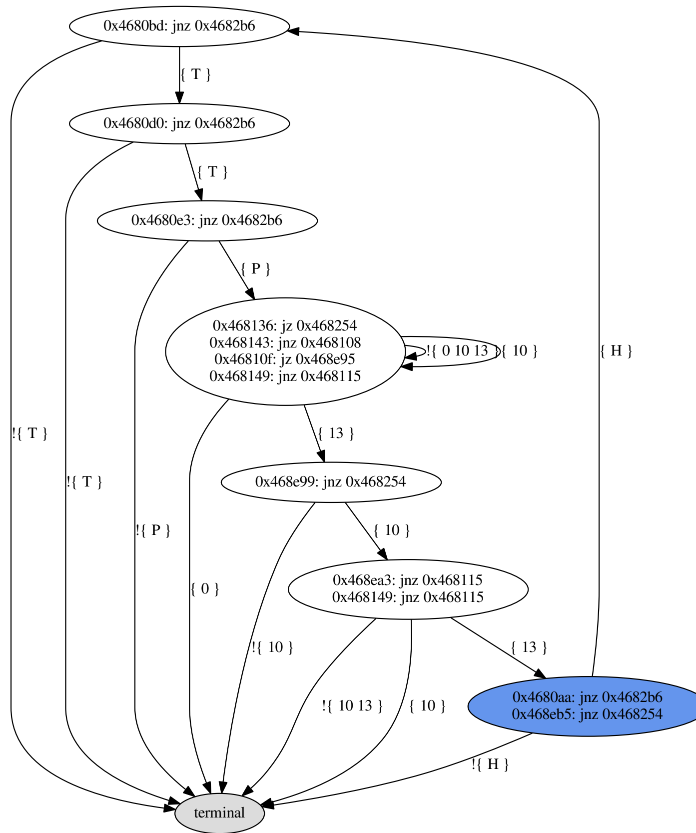


Figure 4.16: Invalid message classification given by a LTS

Unbranchable control flow instructions

In experimentations realized in Section 4.3.2, we have observed the existence of input-dependent but unbranchable CFI(s). They are blue CFI(s) visualized in Figure 4.21 for the case of `links`[1], and in Figure 4.5 for the case of `wget` (cf. also Example 4.2.6).

First, some of them are input-dependent but their decisions depend also on other factors. For example, in checking whether the first 4 bytes of the input message are `H·T·T·P` or not, `wget` has compare the input message buffer (stored at the address determined by the value of `esi`) with a constant string `H·T·T·P` (stored at the address

[1]The orange CFI(s) are branchable (cf. Definition 4.2.1) but we do not draw their other branches because, in this specific case, this branch does not lead to any other branchable CFI(s).

```
00404F72   83 FD 04          cmp ebp,4
00404F75   B9 04 00 00 00    mov ecx,4
00404F7A   7F 02             jg wget_a_unpacked.404F7E
00404F7C   8B CD             mov ecx,ebp
00404F7E   56                push esi
00404F7F   57                push edi
00404F80   BF D0 DE 49 00    mov edi,wget_a_unpacked.49DED0
00404F85   8B F0             mov esi,eax
00404F87   33 D2             xor edx,edx
00404F89   F3 A6             rep cmps byte ptr ds:[esi],byte ptr es:[edi]
00404F8B   5F                pop edi
00404F8C   5E                pop esi
```

Figure 4.17: Unbranchable CFI because of supplemental factors

determined by the value of `edi`), by the instruction

```
rep cmpsb [esi],[edi]
```

Consequently, the 4-th occurrence of `rep cmpsb [esi],[edi]` in execution traces of `wget` (cf. Figure 4.5) is input-dependent but unbranchable (whereas the first 3 occurrences are branchable). That is because the maximal repeat number of this instruction is determined by the value stored in `ecx`, here it is 4 (cf. Figure 4.17). That means after the 4-th `rep` instruction is executed, the next executed instruction is always `pop edi` at the address `0x404f88`, regardless of the result of the comparison in `rep`.

Second, some of them are actually redundant. For example, consider the CFI at the address `0x468136` of `links`

```
jz 0x468254
```

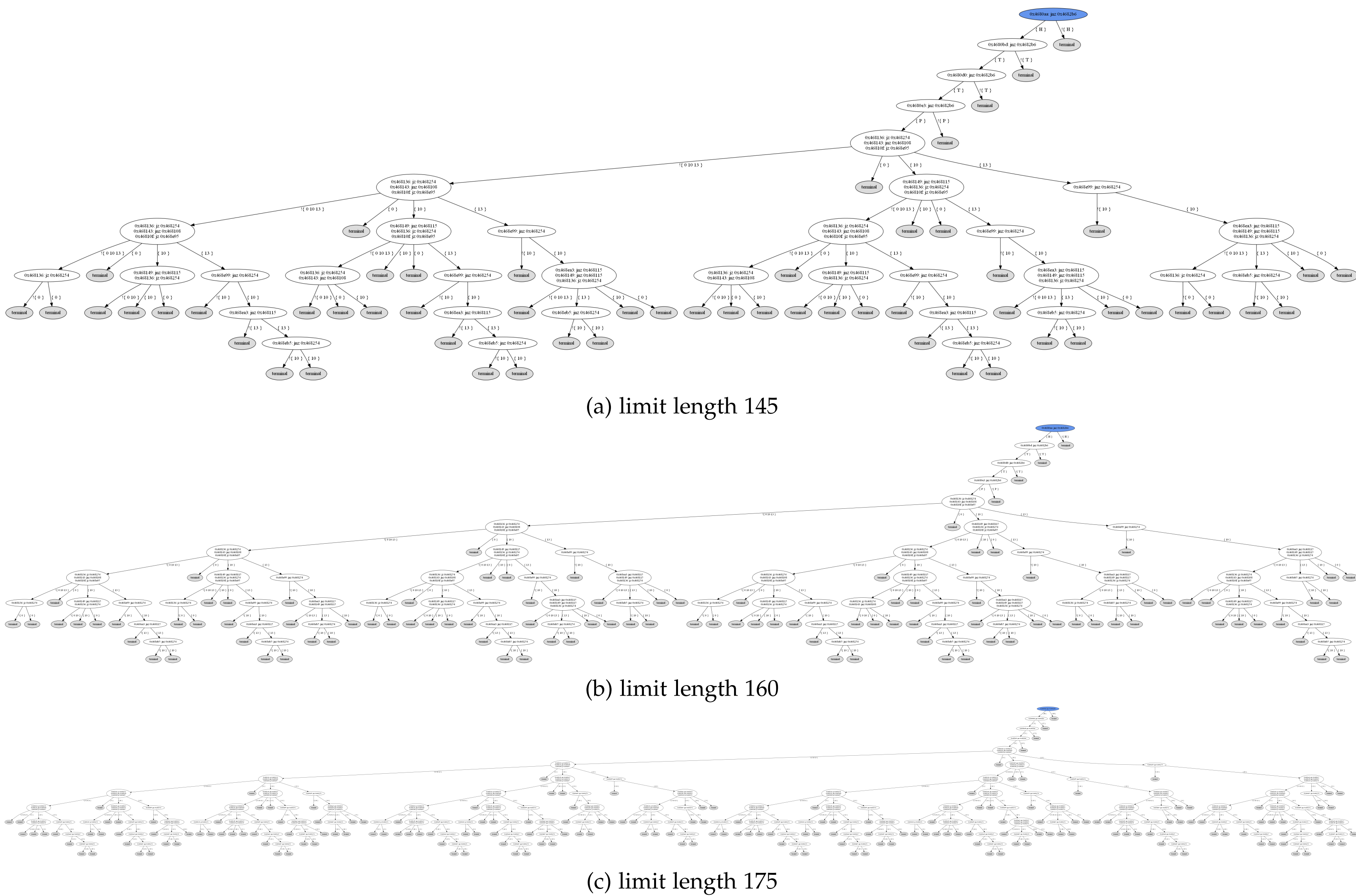
and its first occurrence (cf. Figure 4.21). Its decisions depending on the result of the comparison just before it, namely depending on the value of `dl` is zero or not (cf. Figure 4.18). But this value is never zero at this occurrence of the CFI, because to reaching to this occurrence, the first 4-bytes must be `H·T·T·P` already, then the value of `dl` at the first occurrence of this CFI must be always `H`.

```
00468129   83 C1 01            add ecx,1
0046812C   0F B6 54 03 10      movzx edx,byte ptr ds:[ebx+eax+10]
00468131   83 C0 01            add eax,1
00468134   84 D2               test dl,dl
00468136   0F 84 18 01 00 00   je links.468254
0046813C   39 FE               cmp esi,edi
```

Figure 4.18: Unbranchable CFI because of redundancy

Summary

We have given a detail discussion for the case of `links` and give also results for other programs. The reason why we have chosen `links`, first, is obviously because our method work well in this case. But more importantly, it gives different use cases where we can test our theoretical results. In these use cases, we can see concretely how the program gives semantics for messages, by comparing the final LTS of `links` with ones of other programs.



(a) limit length 145

(b) limit length 160

(c) limit length 175

Figure 4.19: Initial labeled transition system of `links` abstracted from execution trees of different trace's limit lengths

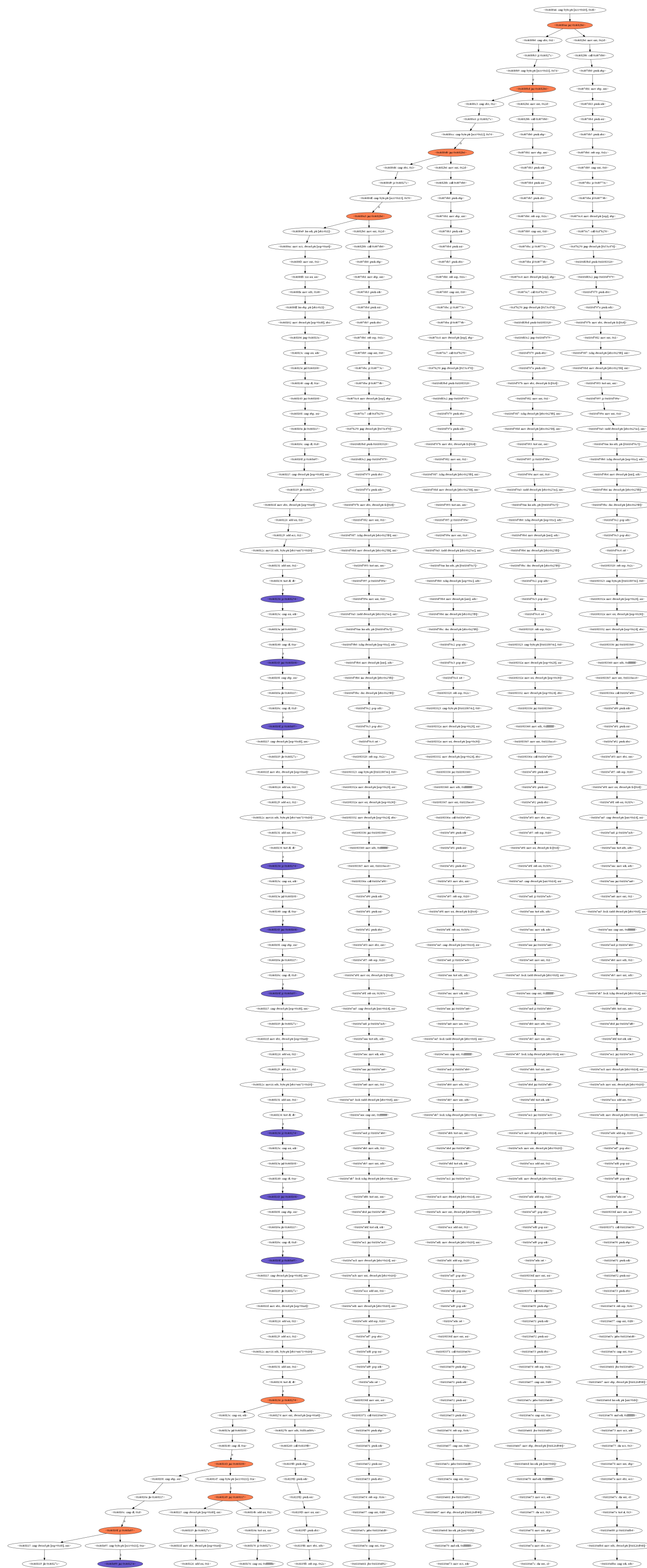


Figure 4.20: Execution tree of `links` of limit length 100

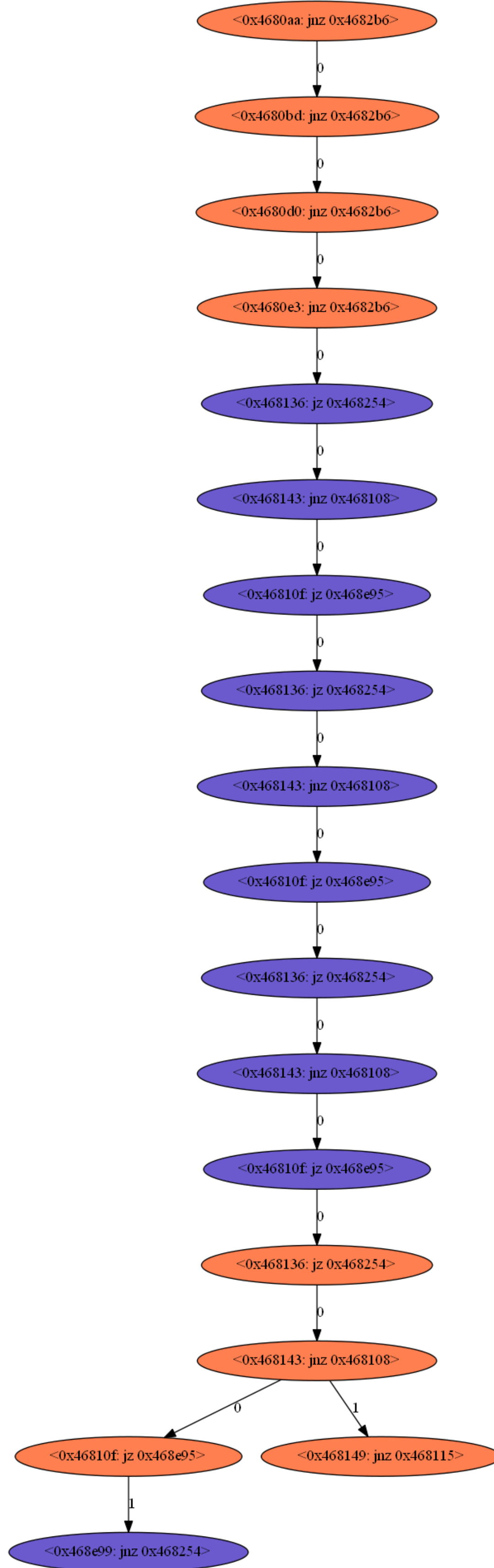


Figure 4.21: Input message dependent CFI(s) of `links`

```
0046809E  85 DB               test ebx,ebx
004680A0  0F 8E D6 01 00 00   jng links.46827C
004680A6  80 79 10 48         cmp byte ptr ds:[ecx+10],48
004680AA  0F 85 06 02 00 00   jnz links.4682B6
004680B0  83 FB 01            cmp ebx,1
004680B3  0F 84 C3 01 00 00   je links.46827C
004680B9  80 79 11 54         cmp byte ptr ds:[ecx+11],54
004680BD  0F 85 F3 01 00 00   jnz links.4682B6
004680C3  83 FB 02            cmp ebx,2
004680C6  0F 84 B0 01 00 00   je links.46827C
004680CC  80 79 12 54         cmp byte ptr ds:[ecx+12],54
004680D0  0F 85 E0 01 00 00   jnz links.4682B6
004680D6  83 FB 03            cmp ebx,3
004680D9  0F 84 9D 01 00 00   je links.46827C
004680DF  80 79 13 50         cmp byte ptr ds:[ecx+13],50
004680E3  0F 85 CD 01 00 00   jnz links.4682B6
004680E9  8D 7B FF            lea edi,dword ptr ds:[ebx-1]
004680EC  8B 8C 24 E4 00 00 00 mov ecx,dword ptr ss:[esp+E4]
004680F3  B8 01 00 00 00      mov eax,1
004680F8  31 F6               xor esi,esi
004680FA  BA 48 00 00 00      mov edx,48
004680FF  8D 6B FD            lea ebp,dword ptr ds:[ebx-3]
00468102  89 5C 24 48         mov dword ptr ss:[esp+48],ebx
00468106  EB 34               jmp links.46813C
00468108  39 F5               cmp ebp,esi
0046810A  7E 09               jle links.468115
0046810C  80 FA 0D            cmp dl,D
0046810F  0F 84 80 0D 00 00   je links.468E95
00468115  39 44 24 48         cmp dword ptr ss:[esp+48],eax
00468119  0F 8E 5D 01 00 00   jng links.46827C
0046811F  8B 9C 24 E4 00 00 00 mov ebx,dword ptr ss:[esp+E4]
00468126  83 C6 01            add esi,1
```

Figure 4.22: Manual reverse engineering of the input message parser of `links`

In all realized tests, our implementation (under the form of a Pintool) works for all cases where the examined program uses some character-based parser. There are obviously cases where it does not work, and we will discuss about them below.

4.3.3 Limits

In Section 2.3.1, we have given critiques to current methods in message format extraction. Our approach has also limits, it is better than other methods in the cases discussed above, and is worse than them in the cases discussed below. These limits come both from our technical implementation and our theoretical hypothesis, and we will discuss about them in this section.

Theoretical limits

The stepwise abstraction procedure uses Hypotheses 1 and 2. The former requires that the set of bytes affecting each CFI is static and the branching conditions of each CFI are fixed. As previously discussed, these conditions are not satisfied in general. For example, they makes our methods cannot work with message formats having *direction fields*. In this case, the value of a field can be used to determine the set of affecting bytes for a CFI, and then this set is not static, consequently the branching conditions of this CFI are not fixed. Hence, our method is worse than other approaches, e.g. [26,

28, 41, 101], that can handle direction fields.

The latter requires that positions of parsed bytes of the input buffer follows a unique order in different execution traces, namely this order is maintained over different traces. Because a direction field will specify the positions of next interested bytes, then it can change the order or parsed by over different execution traces. Consequently, the existence of *direction fields* can again invalidate this hypothesis.

There is another limit coming from the byte regrouping using the address closure (cf. Definition 4.2.3). If the partition, generated by the address closure relation on the set of input buffer's bytes, is too rough, then the result is actually nonsense. For example, if this partition consists of only one class that is the input buffer itself, then the result is nonsense: the initial labeled transition system consists of a single state, and all transitions are loop-backs. Practically, this is also the case of *encrypted messages*, in this case, the branching conditions of a CFI can depend on the values of all bytes of the input buffer.

Finally, as previously discussed, the number of CFI in the execution tree is about $2^{\mathcal{O}(n)}$ where n is the limit length of execution traces. Then the number of states in the initial LTS is also about $2^{\mathcal{O}(n)}$. The algorithm in Listing 3.5 about searching for a nontrivial π-equivalence is polynomial with number of states in the LTS (cf. Proposition 3.3.2). Hence the running time of the stepwise abstraction procedure is about $2^{\mathcal{O}(n)}$. That makes our method infeasible with long enough execution traces.

Technical limits

Suppose that the theoretical limits discussed above does not exist, there are still limits coming from our practical implementation.

The essential point in the stepwise abstraction procedure is to *explore different execution traces* of the program that can obtained by changing the value of the input buffer. Additionally, for each branchable control flow instruction, to *determine precisely its branching conditions*. Our technical approach uses the dynamic tainting analysis to determine the set of bytes of the input buffer that affects to each control flow instruction, then uses the reverse execution to test all value possibles of this affecting set.

This approach cannot handle the case where the affecting set consists of several bytes. For example if a set consists of 4 bytes, then the number of re-executions is 2^{32}, that makes our implementation infeasible. In fact, in the current implementation, to be able to determine precisely the branching condition of each CFI, we have required that each set consists of exactly 1 bytes. That makes it can handle character-based parsers only.

There is other limits concerned directly with the implementation. First, the Pintool have been designed to explore the execution traces of the passed program whenever the main thread of this program receives some input message. This is implemented by intercepting several Windows API(s) (`WSARecv`, `WSARecvFrom`, `recv`, `recvfrom`, `InternetReadFile`, `InternetReadFileEx`) and Linux system calls (`recv`, `recvfrom`) at

loading time[1]. This implementation does not allow tracing programs which

- receive their inputs without using the functions above,
- inject themselves into the existing memory space of another program, because in this case
  - the current implementation of the Pintool does not know exactly which thread will receive input messages,
  - the injected program has been loaded before (i.e. the program loading stage have finished) and addresses of the imported Windows API(s) have been determined, the Pintool cannot insert its appropriate interfering functions for imported Windows API(s),
  - the imported Windows API(s) can be modified or diverted also by other hooking mechanisms.

Though these drawbacks can be considered just as a technical problem, but it makes difficult in using directly this current implementation to analyze modern malwares. We will present how we have managed to analyze the malware ZeuS in Section 4.3.4.

Second, the reverse execution inherits some limits of Pin. Indeed, Pin can be thought of as a process-level virtual machine [150], then it cannot instrument kernel-level executions. The reverse execution is implemented using Pin API(s), it cannot control the writing into the program memory space taken by kernel-executions, consequently it can work safely only when there are no exceptions (e.g. system calls) in the execution of the program.

4.3.4 Case of ZeuS bot

We reserve this part to present experiments on ZeuS bot, a very well-know malware. In fact, ZeuS is a common name for a family of malwares, consisting of several versions. The first versions of ZeuS appeared at the first time in 2007 [60], and the malware is still actively developed today. The source codes of this malware are for sale in the underground markets, and source codes of some early versions are even leaked online (e.g. [166]).

Because of technical limits discussed in Section 4.3.3, our experimental Pintool cannot detect the execution point where the malware hooked some Windows API(s) to steal credential information, then cannot construct directly the execution tree and the final LTS describing how the malware parses the inputs. We can only test the Pintool on the parser extracted and recompiled from the available source codes.

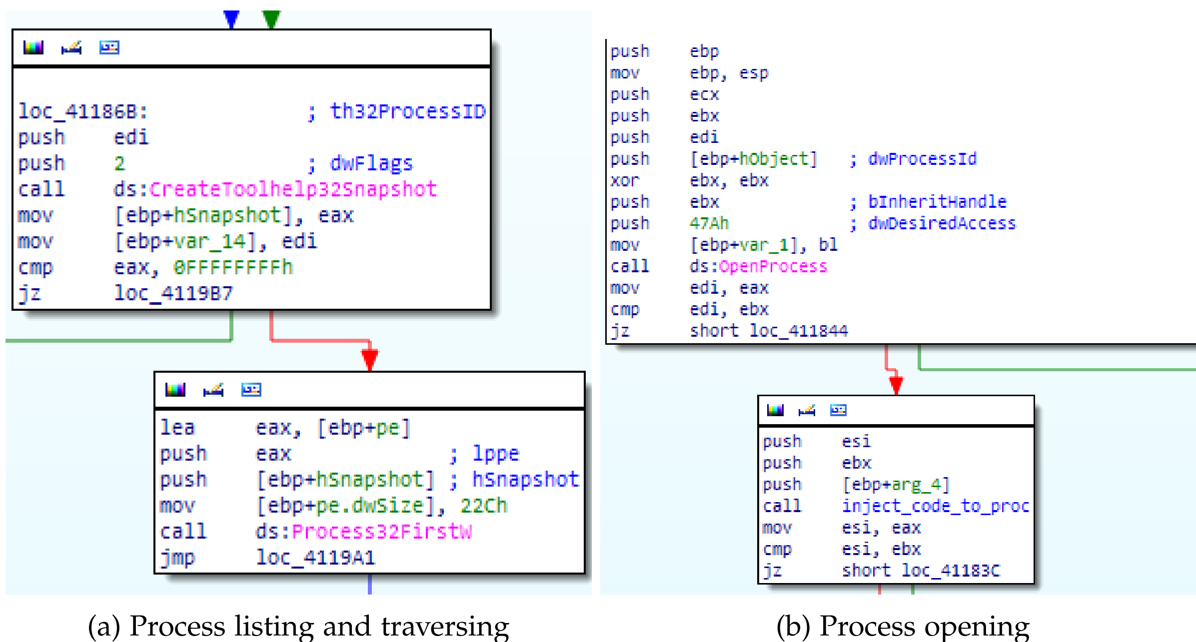
However, the obtained results are promising. It suggests that if the technical difficulties about detecting the execution point can be handled, then our approach can

[1] One can refer the detail implementation in the source codes at [16].

be used to deal with real-world malwares. The analysis given following aims at discussing these technical difficulties, in the hope that gives some perspectives for the future work.

Stealth techniques

There exist already several analysis on ZeuS [1, 2, 60, 82], the following analysis focuses on the stealth techniques used by ZeuS[1]. The malware does not execute as a explicit and dependent process, it instead injects itself into the memory space of other processes, then execute as a thread of the injected process. The codes in Figure 4.23a demonstrates how the malware select a process to inject itself into. It calls the API `CreateToolhelp32Snapshot` to get a list of currently executed processes, then uses the API `Process32FirstW/Process32NextW` to traverse this list. For each traversed process, the malware tries to open this process by calling `OpenProcess` to inject codes (cf. Figure 4.23b).



(a) Process listing and traversing (b) Process opening

Figure 4.23: Selecting process for injecting codes

The malware calls the API `VirtualAllocEx` to allocate new memory inside the memory space of the opened process (cf. Figure 4.24a). Next, it calls `WriteProcessMemory` to copy itself into the allocated memory (cf. Figure 4.24b). Finally, the malware calls the API `CreateRemoteThread` to activate the injected codes as a new execution thread of the injected process (cf. Figure 4.25).

[1]The analyzed sample has MD5 hash `c84b88f6c567b2da651ad683b39ceb2d` and is obtained from https://malwr.com.

```
push    40h               ; flProtect
push    3000h             ; flAllocationType
push    ebx               ; dwSize
push    0                 ; lpAddress
push    [ebp+hProcess]    ; hProcess
call    ds:VirtualAllocEx
mov     [ebp+lpAddress], eax
test    eax, eax
```

(a) Memory allocation

```
jnz     short loc_4188E3
push    ecx               ; lpNumberOfBytesWritten
push    [ebp+nSize]       ; nSize
push    ebx               ; lpBuffer
push    [ebp+lpAddress]   ; lpBaseAddress
push    [ebp+hProcess]    ; hProcess
call    ds:WriteProcessMemory
test    eax, eax
```

(b) Writing codes

Figure 4.24: Injecting codes into the opened process

```
push    ebx               ; lpThreadId
push    ebx               ; dwCreationFlags
push    ebx               ; lpParameter
add     eax, offset sub_416A44
push    eax               ; lpStartAddress
push    ebx               ; dwStackSize
push    ebx               ; lpThreadAttributes
push    edi               ; hProcess
call    ds:CreateRemoteThread
```

Figure 4.25: Activating codes

Technical difficulties The steal techniques discussed above, where ZeuS is a concrete instance, neutralize the function of our Pintool. Concretely, in tracing the execution such a program, the Pintool loses control where the program inserts itself as a (or several) new thread in another process: the Pintool does not know exactly which thread should be traced.

More importantly, the API interception mechanism of the Pintool functions only at the loading time. First, it scans (by calling Pin's API `RTN_FindByName`) the loaded dlls to determine the addresses of intercepted API(s). Then it inserts instrumented functions to intercept two execution points where the API is called and returns (by calling Pin's API `RTN_InsertCall`).

This interception mechanism does not work for programs using steal techniques discussed above because it does not make sense to intercept API(s) at the loading time of the instrumented program. The program may not call these API(s), instead it injects its codes in another process, and use the Windows API `GetProcAddress` to get directly addresses of these API(s) but in the context of the injected process.

One may think that the Pintool should instrument the injected process instead of the program. But in this case, the injected process may have already loaded all necessary dlls. Consequently, the loading time interception mechanism of the Pintool does not work again.

Remark 4.3.4. It may worth noting that the technical difficulties above concern with the input capturing only. It does not concern with the limits of our *stepwise abstraction* procedure and with the limits of message classification using the final LTS. In other words, it does not concern with our approach in message format extraction.

The general and automatic treatment for these technical difficulties is a must for a real-world malwares analysis tool. Our implemented Pintool is only an experimental prototype, it is currently at the first step toward such a tool.

Credential information parser

ZeuS hooks also the Windows API(s) `send` and `WSASend` to collect credential information of users in the infected host. To do that, the hooked raw information is parsed also by the malware. We can test out the Pintool on this parser by extracting and recompiling the relevant codes from the source codes of the malware at [166]. Concretely, in the following test case, we extract the function `socketGrabber` of ZeuS, remove irrelevant system calls inside the function, and change the macro `XOR_COMPARE` into a character based comparison, recompile it, and put it under the examination of our Pintool.

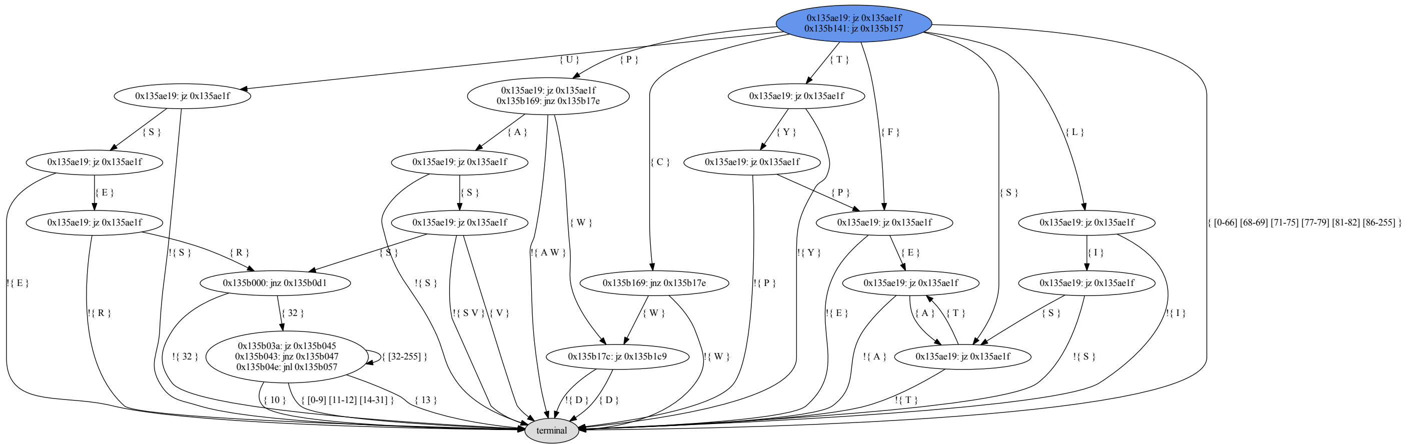


Figure 4.26: Final LTS of ZeuS with limit length 350

The final label transition system in Figure 4.26 is obtained from the stepwise abstraction procedure where execution traces are limited at length of 350 instructions (note that 32 is the ASCII code of the space character). First, by considering paths from the initial state to the terminal state, we receive following classes of messages are distinguished by the parser

$\mathtt{U} \cdot \mathtt{S} \cdot \mathtt{E} \cdot \mathtt{R} \cdot \mathtt{SPACE} \cdot \{32, 33, \ldots, 255\}^{n-5}$ $\quad$ $\mathtt{P} \cdot \mathtt{A} \cdot \mathtt{S} \cdot \mathtt{S} \cdot \mathtt{SPACE} \cdot \{32, 33, \ldots, 255\}^{n-5}$

$\mathtt{P} \cdot \mathtt{A} \cdot \mathtt{S} \cdot \mathtt{V} \cdot \mathtt{Bytes}^{n-4}$ $\quad$ $\mathtt{P} \cdot \mathtt{W} \cdot \mathtt{D} \cdot \mathtt{Bytes}^{n-3}$

$\mathtt{C} \cdot \mathtt{W} \cdot \mathtt{D} \cdot \mathtt{Bytes}^{n-3}$ $\quad$ $\mathtt{T} \cdot \mathtt{Y} \cdot \mathtt{P} \cdot \mathtt{E} \cdot \{!\mathtt{A}\} \cdot \mathtt{Bytes}^{n-5}$

$\mathtt{F} \cdot \mathtt{E} \cdot \mathtt{A} \cdot \mathtt{T} \cdot \{!\mathtt{A}\} \cdot \mathtt{Bytes}^{n-5}$ $\quad$ $\mathtt{S} \cdot \mathtt{T} \cdot \mathtt{A} \cdot \mathtt{T} \cdot \{!\mathtt{A}\} \cdot \mathtt{Byte}^{n-5}$

$\mathtt{L} \cdot \mathtt{I} \cdot \mathtt{S} \cdot \mathtt{T} \cdot \{!\mathtt{A}\} \cdot \mathtt{Byte}^{n-5}$

The words USER, PASS, PASV, PWD, CWD, TYPE, FEAT, STAT, LIST occur in these classes are exactly commands defined in the ftp protocol. That may be interpreted as an insight that ZeuS steals information concerned with ftp communications. Moreover, besides these words there is no more commands of ftp occurring in the final LTS, so we may think that Zeus is interested only in this subset of ftp commands.

Second, we can observe also the following classes of messages in the final LTS (note that the set $\{1, 2, \ldots, 31\}$ contains ASCII codes of all control characters)

$$\mathtt{U} \cdot \mathtt{S} \cdot \mathtt{E} \cdot \mathtt{R} \cdot \{!\mathtt{SPACE}\} \cdot \mathtt{Bytes}^{n-5}$$
$$\mathtt{U} \cdot \mathtt{S} \cdot \mathtt{E} \cdot \mathtt{R} \cdot \mathtt{SPACE} \cdot \{32, 33, \ldots, 255\}^k \cdot \backslash\mathtt{r} \cdot \mathtt{Bytes}^{n-k-6}$$
$$\mathtt{U} \cdot \mathtt{S} \cdot \mathtt{E} \cdot \mathtt{R} \cdot \mathtt{SPACE} \cdot \{32, 33, \ldots, 255\}^k \cdot \backslash\mathtt{n} \cdot \mathtt{Bytes}^{n-k-6}$$
$$\mathtt{U} \cdot \mathtt{S} \cdot \mathtt{E} \cdot \mathtt{R} \cdot \mathtt{SPACE} \cdot \{32, 33, \ldots, 255\}^k \cdot (\{1, 2 \ldots, 31\} \setminus \{\backslash\mathtt{r}, \backslash\mathtt{n}\}) \cdot \mathtt{Bytes}^{n-k-6}$$
$$\mathtt{P} \cdot \mathtt{A} \cdot \mathtt{S} \cdot \mathtt{S} \cdot \{!\mathtt{SPACE}\} \cdot \mathtt{Bytes}^{n-5}$$
$$\mathtt{U} \cdot \mathtt{S} \cdot \mathtt{E} \cdot \mathtt{R} \cdot \mathtt{SPACE} \cdot \{32, 33, \ldots, 255\}^k \cdot \backslash\mathtt{r} \cdot \mathtt{Bytes}^{n-k-6}$$
$$\mathtt{U} \cdot \mathtt{S} \cdot \mathtt{E} \cdot \mathtt{R} \cdot \mathtt{SPACE} \cdot \{32, 33, \ldots, 255\}^k \cdot \backslash\mathtt{n} \cdot \mathtt{Bytes}^{n-k-6}$$
$$\mathtt{U} \cdot \mathtt{S} \cdot \mathtt{E} \cdot \mathtt{R} \cdot \mathtt{SPACE} \cdot \{32, 33, \ldots, 255\}^k \cdot (\{1, 2, \ldots, 31\} \setminus \{\backslash\mathtt{r}, \backslash\mathtt{n}\}) \cdot \mathtt{Bytes}^{n-k-6}$$

These classes show clearly how ZeuS parses a USER or PASS command: it checks whether the next character is space or not, if yes then it collects the bytes until a control character is detected.

Static analysis and automated dynamic analysis We have discussed in Section 2.2.1 the limits of static and dynamic analysis. In Section 2.2.2, we have discussed that the code coverage can be used to improve the incompleteness of the dynamic analysis. The classes of messages in the case of ZeuS (and other classes in the cases of benign programs in Section 4.3.2) are results obtained normally by static analysis because each class contains arbitrarily long messages. But here we have obtained them actually by dynamic analysis with code coverage. One can observe that the incompleteness of the dynamic analysis have been improved to be as good as the static analysis. Additionally, with dynamic analysis, we can bypass lots of obfuscation techniques of malicious codes (cf. Section 2.2.1).

Chapter 5

Behavioral obfuscation

Until recently, program obfuscation was still considered as black-art, as cryptography was more than 30 years ago. Some pioneer and rigorous works published in the last decade have led to a renewal of scientific interest on this topic. Researches on program obfuscation are sensitive since the obtained results are a double-edged sword: they can be used for software protection but also to increase the nuisance of malwares. In order to improve malware detection one should understand how they conceal themselves. The goal of this chapter is to study how behavioral obfuscation happens and can be detected. This study is performed through the use of a new abstract model for process and kernel interactions based on monoidal categories, where program observations are considered to be finite lists of system call (abbr. syscall) invocations. In the first step, we show how malicious behaviors can be obfuscated by simulating the observations of benign programs. In the second step, we show how to generate such malicious behaviors through a technique called path replaying and we extend the class of captured malwares by using some algorithmic transformations on morphisms graphical representation. In the last step, we show that all the obfuscated versions we obtained can be used to detect well-known malwares in practice.

The results of this chapter have been presented in [127]. This is joint work with Romain Péchoux and inspired by many original ideas of Jean-Yves Marion.

5.1 Introduction

A traditional technique used by malware writers to bypass malware detectors is program transformation. Basically, the attacker applies some transformations (e.g. useless code injection, change of function call order, code encryption, etc.) a given malware in order to build a new version having the same malicious behavior, i.e. semantically equivalent relatively to a particular formal semantics. This version may bypass a malware detector succeeding in detecting the original malware if the transformation is cleverly chosen. This risk is emphasized in [48] "*an important requirement of a robust malware detection is to handle obfuscation transformation*".

Currently, the works on *Code obfuscation* have been one of the leading research topic in the field of software protection [39]. By using code obfuscation, malwares can bypass code pattern-based detectors so that the detector's database has to be regularly updated in order to recognize obfuscated variants. As a consequence of Rice's theorem, verifying whether two programs are semantically equivalent is undecidable in general, which annihilates all hopes to write an ideal detector. Consequently, efficient detectors will have to handle code obfuscation in a convenient way while ensuring a good tractability. Most of recent researches have focused on *semantics-based* detection [35, 48], where programs are described by abstractions independent from code transformations. Since the semantics of the abstracted variants remains unchanged, the detection becomes more resilient to obfuscation.

In this chapter, we will focus on *behavior-based* techniques [64, 90], a sub-class of semantics-based ones, where programs are abstracted in terms of *observable behaviors*, that is interactions with the environment. Beside the works on detection (cf. [47] for an up-to-date overview), detection bypassing is also discussed academically in [63, 106, 129, 159] and actively in the underground. To our knowledge, there are only a few theoretical works on *behavioral obfuscation*. The lack of formalism and of general methods leads to some risks from the protection point of view: first, malwares deploying new attacks, that is attacks that were not practically handled before, might be omitted by current detectors. Second, the strength of behavior-based techniques might be overestimated, in particular if they have not a good resilience to code obfuscation.

Our main contribution is to construct a formal framework explaining how semantics-preserving *behavioral transformations* may evolve. The underlying mathematical abstraction is the notion of monoidal category that we use to model system call interactions and internal computations of a given process with respect to the kernel. First, it allows us to formally define the behaviors in term of syscall observations and to use the categorical abstraction to define the obfuscated versions that an *ideal detector* should recognize based on the knowledge of some malicious behaviors. Second, it allows us to define a nontrivial subclass of behaviorally obfuscated programs on which detection becomes decidable. Finally, we show that, apart from purely theoretical results, our model also leads to some encouraging experimental results since the aforementioned decidability result allows us to recognize distinct versions of malwares from the real-world quite efficiently.

Outline In Section 5.2, we first introduce an example motivating the need for an interactive model in order to represent syscall-based behaviors of programs. Next, we introduce a new abstract and simple model based on monoidal categories, that only requires some basic behavioral properties, and introduce the corresponding notions of observable behaviors. We provide several practical examples to illustrate that, though theoretically oriented, this model is very close to practical considerations. In Section 5.3, we present principles of behavioral obfuscation and some semantics-preserving transformations with respect to the introduced model. In Section 5.4, we

introduce a practical implementation of our model and conclude by discussing related works and further developments.

5.2 Behavior modeling

In this section, we introduce a formal model for behaviors of a program. The main idea is to classify behaviors into two sorts: one consists of *internal behaviors*, which the program can proceed freely without affecting the environment (i.e. the operating system); the other consists of *syscall interactions*, where the invocation of each will affect the environment.

5.2.1 Motivating example

We commence with an example showing how malwares can bypass easily behavior-based detection techniques. This example emphasizes the need of a new model and new detection techniques complementary to current behavior-based detection ones in order to avoid detection failures. Here, the analyzed malware sample is a variant of the trojan `Dropper.Win32.Dorgam` [55]. Since this trojan downloads and activates other malicious codes, we restrict the analysis to stages where these downloaded codes have not been activated yet.

The trojan's malicious behavior consists in three consecutive stages. First, it unpacks two PE files (cf. Listing 5.1) whose paths are added into the registry value `AppInit_DLLs` so that they will be automatically loaded by the malicious codes, which are downloaded later.

```
NtCreateFile       (FileHdl=>0x00000734,RootHdl<=0x00000000,File<=\??\C:\WINDOWS\system32\sys.sys)    Ret=>0
NtWriteFile        (FileHdl<=0x00000734,BuffAddr<=0x0043DA2C,ByteNum<=11264)                          Ret=>0
NtFlushBuffersFile (FileHdl<=0x00000734)                                                              Ret=>0
NtClose            (Hdl<=0x00000734)                                                                  Ret=>0
NtCreateFile       (FileHdl=>0x00000734,RootHdl<=0x00000000,File<=\??\C:\WINDOWS\system32\intel.dll)  Ret=>0
NtWriteFile        (FileHdl<=0x00000734,BuffAddr<=0x0041A22C,ByteNum<=145408)                         Ret=>0
NtFlushBuffersFile (FileHdl<=0x00000734)                                                              Ret=>0
NtClose            (Hdl<=0x00000734)                                                                  Ret=>0
```

Listing 5.1: File unpacking

Second, it creates the key `SOFTWARE\AD` and adds some entries as initialized values (cf. Listing 5.2).

```
NtOpenKey      (KeyHdl=>0x00000730,RootHdl<=0x00000784,Key<=SOFTWARE\AD\)                                Ret=>0
NtSetValueKey  (KeyHdl<=0x00000730,ValName<=ID,ValType<=REG_SZ,ValEntry<=2062)                           Ret=>0
NtClose        (Hdl<=0x00000730)                                                                         Ret=>0
NtOpenKey      (KeyHdl=>0x00000730,RootHdl<=0x00000784,Key<=SOFTWARE\AD\)                                Ret=>0
NtSetValueKey  (KeyHdl<=0x00000730,ValName<=URL,ValType<=REG_SZ,ValEntry<=http://ad.***.com:82)          Ret=>0
NtClose        (Hdl<=0x00000730)                                                                         Ret=>0
NtOpenKey      (KeyHdl=>0x00000730,RootHdl<=0x00000784,Key<=SOFTWARE\AD\)                                Ret=>0
NtSetValueKey  (KeyHdl<=0x00000730,ValName<=UPDATA,ValType<=REG_SZ,ValEntry<=http://t.***.com:82/***)    Ret=>0
NtClose        (Hdl<=0x00000730)                                                                         Ret=>0
NtOpenKey      (KeyHdl=>0x00000730,RootHdl<=0x00000784,Key<=SOFTWARE\AD\)                                Ret=>0
NtSetValueKey  (KeyHdl<=0x00000730,ValName<=LOCK,ValType<=REG_SZ,ValEntry<=http://t.***.com?2062)        Ret=>0
NtClose        (Hdl<=0x00000730)                                                                         Ret=>0
......
```

Listing 5.2: Registry initializing

Third, it calls the function `URLDownloadToFile` of Internet Explorer (abbr. MSIE) to downloads other malicious codes from some addresses in the stored values.

Since the file unpacking at the first stage is general and the behaviors at the third stage are the same as those of the benign program MSIE, a reasonable way for a behavior-based detector D to detect the trojan is by examining its behaviors during the second stage. That means D will detect the following system call sequence[1]:

`NtOpenKey,NtSetValueKey,NtClose,NtOpenKey,NtSetValueKey,...`

corresponding to the consecutive syscalls of Listing 5.2 in order to detect this trojan. Since the association of `NtOpenKey` to each `NtSetValueKey` is verbose and can be replaced by a single call, D has to detect also the sequence:

`NtOpenKey,NtSetValueKey,NtSetValueKey,NtSetValueKey,...`

Moreover, the key handler can be obtained by duplicating a key handler located in another process, so the existence of `NtOpenKey` is even not necessary. That means D has also to detect the sequence:

`NtDuplicateObject,NtSetValueKey,NtSetValueKey,NtSetValueKey,...`

In summary, the three syscall lists described above are equivalent behaviors that the trojan could arbitrarily select in order to perform its task. The remainder of the chapter will be devoted to modestly explain how such lists can be both generated and detected for restricted but challenging behaviors.

5.2.2 Basic categorical background

Category theory is an important branch of mathematics which emphasizes on the abstraction of concrete mathematical objects. Under the categorical viewpoint, essential properties of different mathematical notations are revealed and unified. In theoretical computer science, it has been used as a very useful "toolbox" to abstract definitions and to prove theorems, not only in traditional branches (e.g. domain theory, type theory and formal semantics of programming languages), but also as a rigorous framework for new models of computation [115].

In the following, we introduce only basic categorical notions needed for the construction of our formal model. We refer to [4] as a *computer science oriented* introduction to category theory, a very insightful introduction to monoidal categories can be referenced in [116].

Category A category is a collection of *objects* $m, n, \ldots$ and *morphisms* $s, r, \ldots$ mapping a *source* object to a *target* object, including an identity morphism 1_m for each

[1] For readability, we omit the arguments in the syscall lists.

object m, and an associative *composition* operator $\circ$. As usual, a morphism s of source object $source(s) = m$ and target object $target(s) = n$ will be denoted using one of the following notations:

$$m \xrightarrow{s} n \quad \text{or} \quad s : m \longrightarrow n$$

A category with objects as sets and morphisms as functions can properly represent sequential computation models [97]. Concurrent models employ a new operator named *tensor product*, that leads to the use of monoidal categories [111].

Definition 5.2.1 (Monoidal category [144])**.** A monoidal category is a category with a tensor product $\otimes$ operator, defined on morphisms and objects and satisfying the following properties:

- Given morphisms s_1 and s_2, $s_1 \otimes s_2$ is a morphism of the shape:

$$s_1 \otimes s_2 : source(s_1) \otimes source(s_2) \longrightarrow target(s_1) \otimes target(s_2)$$

- There exists a unit object e.

- The tensor and composition operators have the following properties[1]:

$$(m_1 \otimes m_2) \otimes m_3 = m_1 \otimes (m_2 \otimes m_3) \qquad e \otimes m = m = m \otimes e$$
$$(s_1 \otimes s_2) \otimes s_3 = s_1 \otimes (s_2 \otimes s_3) \qquad 1_e \otimes s = s = s \otimes 1_e$$
$$(s'_1 \otimes s'_2) \circ (s_1 \otimes s_2) = (s'_1 \circ s_1) \otimes (s'_2 \circ s_2)$$

Morphism (resp. object) *terms* are terms built from basic morphisms (resp. objects) as variables, composition and tensor product. For example, $(s_1 \circ s_2) \otimes s_3$ is a morphism term and $m_1 \otimes (m_2 \otimes m_3)$ is a object term.

Graphical representation Morphism and object terms can be given by a standard graphical representation using *string diagrams* [84, 144] defined as follows:

- nodes are morphisms (except for identity morphisms) and edges are objects:

m —(s)— n $\qquad$ 1_m $\qquad$ m

- composition $s_j \circ s_i$:

m_i —(s_i)— $n_i = m_j$ —(s_j)— n_j

- tensor product $s_i \otimes s_j$:

m_j —(s_j)— n_j

m_i —(s_i)— n_i

[1]The monoidal category used here is indeed a strict one.

In (planar) monoidal categories, diagrams are *progressive* [144], namely edges are always oriented from left to right. The left-right direction induces a partial order $\preccurlyeq$ on nodes: $\preccurlyeq$ is defined as the reflexive and transitive closure of the relation $\mathcal{R}$ defined by $s_i \mathcal{R} s_j$ holds if there is an edge from s_i to s_j.

5.2.3 Syscall interaction abstract modeling

From a practical viewpoint, the computations and interactions between processes and the kernel can be divided in two main sorts, the *system calls* interactions and the process or kernel *internal computations*.

System calls are implemented by the trap mechanism where there is a mandatory control passing from the process (caller) to the kernel (callee). A syscall affects to and is affected by both process and kernel data in their respective memory spaces.

We distinguish *syscall names* (e.g. `NtCreateFile`) from *syscall invocations* (e.g. `NtCreateFile(h,...)`). The former are just names while the later compute functions and will be the main concern of our study.

Internal computations are operations inside the process or kernel memory spaces. There is no control passing and they only affect to and are affected by data of the caller memory.

Our purpose to to abstract this concrete interpretation by a categorical model where computations and interactions (i.e. both syscalls and internal computations) will be represented by morphisms on the appropriate objects. For that purpose, objects will consist in formal representations of physical memories.

Definition 5.2.2 (Memory space and memory domain)**.** A memory space M^i is a finite set of addresses corresponding to memory bits:

$$M^i = \{addr_1, addr_2, \ldots, addr_n\}$$

where n is the size of M^i and $i \in \{k, p\}$ is an annotation depending on whether M^i is a kernel memory space (i.e. $i = k$) or a process memory space (i.e. $i = p$). The memory domain $dom(M^i)$ is the set of all bit sequences of size n on M^i:

$$dom(M^i) = \{b_{addr_1} b_{addr_2} \ldots b_{addr_n} | b_{addr_k} \in \{0, 1\}\}$$

For a given memory space M^i, $i \in \{k, p\}$, let m^i, n^i,... denote domains of its subsets, that is $m^i = dom(N^i), n^i = dom(L^i)$ for some $N^i, L^i \subseteq M^i$.

We are now ready to introduce the notion of *interaction category* in order to abstract syscall invocations and internal computations.

Definition 5.2.3 (Interaction category)**.** Let M^p, M^k be memory spaces satisfying $M^p \cap M^k = \emptyset$. The interaction category $\mathcal{C}\langle M^p, M^k \rangle$ is a category equipped with a tensor product $\otimes$ and defined as follows:

- The set of objects is freely generated from:
  - process and kernel memory domains: $m^i, n^i, \ldots, i \in \{k, p\}$,
  - cartesian products: $m^p \times m^k, n^p \times n^k, \ldots$,
  - a unit which is also terminal: $e = dom(\emptyset)$.
- The set of morphisms is freely generated from:
  - process and kernel internal computations: $s^i : m^i \to n^i$, $i \in \{k, p\}$
  - syscall interactions between the process and the kernel:

$$s^{p\text{-}k} : m^p \times m^k \to n^p \times n^k,$$

  - cartesian projections: π_i, $i \in \{k, p\}$.
- The tensor product is partially defined on objects and morphisms by:
  - given $i \in \{k, p\}$ and $N^i, L^i \subseteq M^i$, if $N^i \cap L^i = \emptyset$[1] then

$$dom(M^i) \otimes dom(L^i) = dom(M^i \cup N^i),$$

  - if $m^p \otimes n^p$ and $m^k \otimes n^k$ are both defined then

$$(m^p \times m^k) \otimes (n^p \times n^k) = (m^p \otimes n^p) \times (m^k \otimes n^k),$$

  - if $m^p \otimes n^p$ or $m^k \otimes n^k$ is defined then

$$(m^p \times m^k) \otimes n^p = (m^p \otimes n^p) \times m^k \text{ or } (m^p \times m^k) \otimes n^k = m^p \times (m^k \otimes m^k),$$

  - given $s_1 : m_1 \to n_1$ and $s_2 : m_2 \to n_2$, then

$$s_1 \otimes s_2 : m_1 \otimes m_2 \to n_1 \otimes n_2$$

    is defined by $v_1 \otimes v_2 \mapsto s_1(v_1) \otimes s_2(v_2)$ whenever the following commutative diagram is well-formed (i.e. the tensor is well-defined on objects):

$$\begin{array}{ccc} m_1 \otimes m_2 & \xrightarrow{s_1 \otimes 1_{m_2}} & n_1 \otimes m_2 \\ \downarrow{\scriptstyle 1_{m_1} \otimes s_2} & & \downarrow{\scriptstyle 1_{n_1} \otimes s_2} \\ m_1 \otimes n_2 & \xrightarrow{s_1 \otimes 1_{n_1}} & n_1 \otimes n_2 \end{array}$$

[1] The tensor represents the concurrent accesses and modifications performed by both internal computations and syscall interactions on memory domains. A necessary condition for these operations to be well-defined is that they do not interfere, that is they have to operate on disjoint domains.

Remark 5.2.1. To denote a value v of a memory domain m, the set notation $v \in m$ and the categorical one $v\colon e \to m$, so do the composition $s \circ v$ and the application $s\,(v)$ will be interchangeably used depending on the context.

We show that interaction categories enjoy the mathematical abstractions and properties of monoidal categories:

Proposition 5.2.1. *Each interaction category is a monoidal category where the tensor product is partially defined.*

Proof. For $i \in \{p,k\}$, let (the sets N^i, L^i may also be empty set)

$$\begin{aligned} Obj =& \{dom\,(N^p) \times dom\left(N^k\right) | N^i \subseteq M^i\} \\ Mor =& \{s\colon dom\,(N^p) \times dom\left(N^k\right) \to dom\,(L^p) \times dom\left(L^k\right) | N^i, L^i \subseteq M^i\} \end{aligned}$$

Definition 5.2.3 says that Obj and Mor are respectively generator of the set of objects and the set of morphism of $\mathcal{C}\langle M^p, M^k\rangle$. Moreover, they are both closed under tensor product and composition, hence the set of objects and of morphisms of $\mathcal{C}\langle M^p, M^k\rangle$ are well-defined and equal to Obj and Mor.

The properties applied solely to the tensor in Definition 5.2.1 are directly derived from Definition 5.2.3, we prove the property which is applied to the tensor and the composition. Given morphisms $s_1\colon m_1 \to n_1, s_1'\colon n_1 \to l_1, s_2\colon m_2 \to n_2$ and $s_2'\colon n_2 \to l_2$, we need to verify, whenever they are well-formed, that:

$$(s_1' \otimes s_2') \circ (s_1 \otimes s_2) = (s_1' \circ s_1) \otimes (s_2' \circ s_2)$$

Since the morphism tensoring is defined (cf. Definition 5.2.3) by:

$$\begin{aligned} s_1 \otimes s_2\colon m_1 \otimes m_2 &\to n_1 \otimes n_2 \\ v_1 \otimes v_1 &\mapsto s_1(v_1) \otimes s_2(v_2) \end{aligned}$$

It makes the following diagram commute:

$$\begin{array}{ccccc} & & e & & \\ & \swarrow^{v_1} & \downarrow{\scriptstyle v_1 \otimes v_2} & \searrow^{v_2} & \\ m_1 & \xleftarrow[\pi_{m_1}]{} & m_1 \otimes m_2 & \xrightarrow[\pi_{m_2}]{} & m_2 \\ {\scriptstyle s_1}\downarrow & & \downarrow{\scriptstyle s_1 \otimes s_2} & & \downarrow{\scriptstyle s_2} \\ n_1 & \xleftarrow[\pi_{n_1}]{} & n_1 \otimes n_2 & \xrightarrow[\pi_{n_2}]{} & n_2 \\ {\scriptstyle s_1'}\downarrow & & \downarrow{\scriptstyle s_1' \otimes s_2'} & & \downarrow{\scriptstyle s_2'} \\ l_1 & \xleftarrow{\pi_{l_1}} & l_1 \otimes l_2 & \xrightarrow{\pi_{l_2}} & l_2 \end{array}$$

Hence $(s_1' \circ s_1) \otimes (s_2' \circ s_2) = (s_1' \otimes s_2') \circ (s_1 \otimes s_2)$. □

```
char *src = 0x00150500;
char *dst = 0x00150770;
strncpy(dst,src,10);
```

Listing 5.3: Internal computation

```
char *buf = 0x0015C898;
HANDLE hdl = 0x00000730;
NtWriteFile(hdl,.,buf,1024,.);
```

Listing 5.4: Syscall invocation

Example 5.2.1. Listing 5.3 is an example of (process) internal computation. The function `strncpy` is represented by a morphism:

$$strncpy^p\colon dom([src]) \otimes dom([dst]) \longrightarrow dom([src]) \otimes dom([dst])$$

where $[src]$ and $[dst]$ are 10 bytes memory spaces in M^p respectively beginning at the addresses `0x150500` and `0x150770`.

Example 5.2.2. Listing 5.4 is an example of syscall invocation. The invocation of the syscall name `NtWriteFile` is represented by a morphism:

$$NtWriteFile^{p\text{-}k}\colon dom([buf]) \times dom([hdl]) \longrightarrow dom([buf]) \times dom([hdl])$$

where $[buf]$ is a 1024 bytes memory space in M^p beginning at the address `0x15C898`, and $[hdl]$ is a memory space in M^k identified by the handler `0x730`.

We now discuss basic properties for morphisms realized by a process Q running in the memory space M^p of an interaction category $\mathcal{C}\langle M^p, M^k\rangle$.

Process internal computations are **memory modifiers** operating on some previously allocated memory spaces. Consequently, for any internal computation s^p realized by Q, the equality $source(s^p) = target(s^p)$ always holds. In what follows, for simplicity, we will represent each s^p by a morphism of the shape:

$$s^p\colon dom(M^p) \longrightarrow dom(M^p)$$

That means s^p is implicitly tensored with the identity morphism $1_{dom(M^p \setminus N^p)}$ whenever $dom(N^p) = source(s^p)$.

Syscall interactions are multiform. From the process's viewpoint, they operate on some allocated memory spaces in M^p[1]. Consequently, each syscall invocation is represented by a morphism $s^{p\text{-}k}\colon dom\,(N^p) \times dom\,\left(N^k\right) \longrightarrow dom\,(L^p) \times dom\,\left(L^k\right)$ such that the equality $N^p = L^p$ always holds. As for process internal computations, each syscall interaction will be identified with the shape:

$$s^{p\text{-}k}\colon dom\,(M^p) \times dom\left(N^k\right) \longrightarrow dom\,(M^p) \times dom\left(L^k\right)$$

From the kernel's viewpoint, a syscall interaction morphism can be:

[1] We can safely assume that the process freely uses its memory space M^p by implicitly accept effects of memory allocation syscalls like `NtAllocateVirtualMemory`.

- either a **memory modifier** operating on some fixed and allocated memory space in M^k, hence $N^k = L^k$,
- or a **memory constructor** (resp. **destructor**) allocating (resp. freeing) some fresh (resp. allocated) memory space in M^k, hence $N^k \neq L^k$. Note that a syscall invocation can be both a constructor and a destructor.

Example 5.2.3. The syscall invocation `NtWriteFile(hdl,...)` is a memory modifier (see also Example 5.2.2) while `NtOpenKey(ph,...)` is a memory constructor allocating a new memory space identified by `*ph`, and `NtClose(h)` is a memory destructor freeing the memory space identified by `h`.

The following definition gives sound properties for a list of morphisms which is realized by Q in the interaction category $\mathcal{C}\langle M^p, M^k\rangle$.

Definition 5.2.4 (Canonical property)**.** A nonempty (finite or infinite) list X of morphisms consisting of process internal computations and syscall interactions, i.e. $X = [s_1^{j_1}, s_2^{j_2}, \ldots, s_n^{j_n}, \ldots]$ for $n \geq 1$ and $j_i \in \{p, p\text{-}k\}\ \forall i$, is called canonical if for any syscall interaction $s_i^{p\text{-}k} \in X$ of the shape:

$$s_i^{p\text{-}k} \colon dom\left(M^p\right) \times dom\left(N_i^k\right) \longrightarrow dom\left(M^p\right) \times dom\left(L_i^k\right)$$

then the following properties hold:

- there is no memory duplication: if $s_i^{p\text{-}k}$ is a memory constructor then its constructed memory $T_i^k = L_i^k \setminus N_i^k$ is not duplicated, namely for any $s_j^{p\text{-}k} \in X, j \neq i$; if $j > i$ then $T_i^k \cap T_j^k = \emptyset$, else $T_i^k \cap N_j^k = \emptyset$ and $T_i^k \cap L_j^k = \emptyset$.
- there is no memory reuse: if $s_i^{p\text{-}k}$ is a memory destructor then its destructed memory $U_i^k = N_i^k \setminus L_i^k$ is not reused, namely for any $s_j^{p\text{-}k} \in X$; if $j > i$ then $U_i^k \cap N_j^k = \emptyset$ and $U_i^k \cap L_j^k = \emptyset$.

The former prevents Q from reallocating a memory space and from accessing to an unallocated one, while the later prevents it from accessing to a previously freed memory space.

5.2.4 Process behaviors as paths and semantics

We now provide definitions of process behaviors in term of *paths*, namely lists of consecutive morphisms that processes realized in execution. By assuming that processes can only be examined in finite time, the studied paths are finite.

Definition 5.2.5 (Paths)**.** For a given interaction category, a finite canonical list of morphisms X is called an execution path. The list of all syscall interactions in X is called the corresponding observable path O of X.

Let $\mathcal{X}, \mathcal{O}$ denote the set of all execution paths and of all observable paths, respectively; the function $obs\colon \mathcal{X} \rightarrow \mathcal{O}$ returns the observable path of an execution path given as input, e.g. $obs([s_1^{p\text{-}k}, s_2^p, s_3^p, s_4^{p\text{-}k}]) = [s_1^{p\text{-}k}, s_4^{p\text{-}k}]$.

Execution paths will be used to study all the possible computations (internal computations and syscall interactions) at the process level while observable paths only consist in behaviors that can be grasped by an external observer, that is some sequence of syscall invocations, and will be the main concern of our study.

Remark 5.2.2. Except where explicitly stated otherwise, the term *path* will from now on refer to *execution path*.

Example 5.2.4. The paths X_1, X_2 in Listings 5.5 and 5.6 and their corresponding observable paths $O_1 = obs(X_1)$ and $O_2 = obs(X_2)$ are defined by:

$$X_1 = [strncpy_1^p, strncpy_2^p, NtOpenKey_3^{p\text{-}k}, memcpy_4^p] \quad O_1 = [NtOpenKey_3^{p\text{-}k}]$$
$$X_2 = [memcpy_1^p, strncpy_2^p, strncpy_3^p, NtOpenKey_4^{p\text{-}k}] \quad O_2 = [NtOpenKey_4^{p\text{-}k}]$$

```
strncpy(dst,src1,10);
strncpy(dst+10,src1+10,30);
NtOpenKey(h,...{...dst...});
memcpy(src2,src1,1024);
```

Listing 5.5: Path X_1

```
memcpy(src2,src1,1024);
strncpy(dst+2,src2,15);
strncpy(dst+17,src1+15,25);
NtOpenKey(h,...{...dst+2...});
```

Listing 5.6: Path X_2

Proposition 5.2.2 shows that data modification on M^p and M^k caused by a path X can be represented by a morphism term built from the morphisms of X together with identity morphisms. In what follows, the morphism corresponding to these data modifications will be called **path semantics** of X, denoted $s(X)$.

Proposition 5.2.2. *Given a path $X \in \mathcal{X}$, the data modifications caused by X on M^p and M^k is a morphism $s(X)$ of the shape:*

$$s(X)\colon dom\,(M^p) \times dom\left(N^k\right) \longrightarrow dom\,(M^p) \times dom\left(L^k\right),\ \textit{with}\ N^k, L^k \subseteq M^k.$$

that can be represented by a morphism term obtained from the morphisms of X together with identity morphisms.

Proof. We summarize the shape of morphisms in an execution path. Each syscall interaction is of the shape:

$$s^{p\text{-}k}\colon dom(M^p) \times dom(N^k) \longrightarrow dom(M^p) \times dom(L^k)$$

and belongs to one of two kinds: either *memory modifier* ($N^k = L^k$), or *memory constructor/destructor* ($N^k \neq L^k$). Moreover, we can consider each process internal computation s^p as a special case of $s^{p\text{-}k}$ by writing:

$$s^p\colon dom(M^p) \times dom\,(\emptyset) \longrightarrow dom(M^p) \times dom\,(\emptyset)$$

Since an execution path is canonical (cf. Definition 5.2.4), its morphisms satisfy: *no memory duplication*, namely no memory space is reallocated; and *no memory reuse*, namely no freed memory space is reused.

We prove the proposition by induction on the length n of an execution path. The initial step with $n = 1$ is obvious. Suppose that for each execution path X_n of length n, the path semantics $s(X_n)$ defined by:

$$s(X_n)\colon dom(M^p) \times dom(N_n^k) \longrightarrow dom(M^p) \times dom(L_n^k)$$

can be represented as a morphism term M_n whose variables are morphisms of X_n and identity morphisms (induction hypothesis). Let us consider an execution path

$$X_{n+1} = X_n@[s_{n+1}]$$

for some morphism s_{n+1}. The path semantics $s(X_{n+1})$ can be represented as a morphism term:

$$s(X_{n+1}) = \left(s_{n+1} \otimes 1_{dom(L_n^k \setminus N^k)}\right) \circ \left(s(X_n) \otimes 1_{dom(N^k \setminus L_n^k)}\right)$$

of $s(X_n), s_{n+1}$ and identity morphisms (cf. also Figure 5.1). To show this term is well-formed, we need to verify that $N_n^k \cap \left(N^k \setminus L_n^k\right) = \emptyset$ and $\left(L_n^k \setminus N^k\right) \cap L^k = \emptyset$. Both are trivial if s_{n+1} is a process internal computation since $N^k = L^k = \emptyset$. Let us consider the case where s_{n+1} is a syscall interaction.

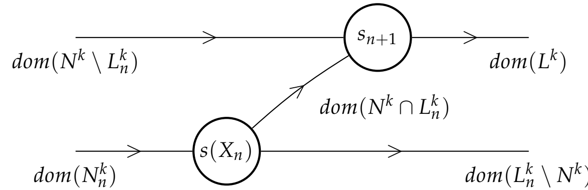


Figure 5.1: Inductive construction

To prove the former, we notice that if $s(X_n)$ frees some memory space then this memory space cannot occurs in source of s_{n+1} anymore (no reuse). That means if N^k and N_n^k share some memory space, then this one must occur in $N^k \cap L_n^k$, in other words $N_n^k \cap \left(N^k \setminus L_n^k\right) = \emptyset$.

To prove the later, we notice that if s_{n+1} allocates some memory space then this memory space cannot occur in target of $s(X_n)$ (no duplication). That means if L^k and L_n^k share some memory space, then this one must occur in $N^k \cap L_n^k$, in other words $\left(L_n^k \setminus N^k\right) \cap L^k = \emptyset$. Hence the inductive step is proved. □

We note that if each morphisms $s \in X$ has form $1_{dom(M^p)} \times s^k$ where s^k is a kernel internal computations, namely s^k is of the shape $s^k\colon dom(N_n^k) \to dom(L_n^k)$ then $s\,(X)$ is equal to:

$$s\,(X) = 1_{dom(M^p)} \times s^k\,(X)$$

where $s^k(X)$ is inductively defined by:

$$s^k(X_{n+1}) = \begin{cases} \left(s^k_{n+1} \otimes 1_{dom(L^k_n \setminus N^k)}\right) \circ \left(s^k(X_n) \otimes 1_{dom(N^k \setminus L^k_n)}\right) & \text{if } N^k \cap L^k_n \neq \emptyset \\ s^k_{n+1} \otimes s^k(X_n) & \text{if } N^k \cap L^k_n = \emptyset \end{cases}$$

So in the general case, the structure of $s(X)$ is *sequential* because the composition through M^p is mandatory. But if we can consider $s(X)$ without considering the data modification on M^p, e.g. in the replay paths, then $s(X)$ has a *concurrent* structure and in this case the string diagram of $s(X)$ can be deformed easily by a coherence theorem (cf. Theorem 4) for monoidal categories.

5.3 Behavioral obfuscation

In this section, we first present a theoretical result which establish sufficient conditions of *behavioral mimic attack*. The intuition behind this result is a malicious program can hide itself by mimic behaviors of a benign program. Then, we introduce some constructive schemata which shows how the mimicry can be obtained, by this way we can really "create" malicious program for a mimic attack. Finally, we present an experimental implementation and experiments on real malwares.

5.3.1 Ideal detectors

We now study behavioral obfuscation from the dual viewpoint: instead of studying program equivalence wrt a given program transformation, we assume the existence of ideal detectors and try to answer the question *which behaviors should the ideal detectors detect based on the knowledge of a given malicious behavior*.

Intuitively, an ideal detector is an oracle that can decide whether the running process is malicious or not by only examining the memory spaces M^p and M^k.

Definition 5.3.1 (Ideal detector)**.** An ideal detector is a boolean target function:

$$D\colon dom\,(M^p) \times dom\left(M^k\right) \to \{T, F\}$$

A verification of D on $X \in \mathcal{X}$ starting at a value $v_0^p \times v_0^k \in source\,(s\,(X))$ is the result of the application $D\left(0_X \otimes s(X)\left(v_0^p \times v_0^k\right)\right)$, where the morphism 0_X is equal to the constant $0 \in dom\left(M^k \setminus target\,(s\,(X))\right)$ (i.e. is equal to 0 on all the kernel addresses that do not belong to the target of $s\,(X)$). If the result of application is T then we say that X is detected by D.

One of the main obfuscation techniques consists in camouflaging behaviors of malwares with those of a benign programs. Such a technique will be illustrated by a ransomware (cf. Listing 5.13) which hides some of its behaviors through the use of Internet Explorer™functionalities.

Formally, consider a path X_1 starting at some value $v_0^p \times v_0^k$ and ending at value $s(X_1)\left(v_0^p \times v_0^k\right)$ detected by the ideal detector D. Consider another path X_2 which also starts at $v_0^p \times v_0^k$ and ends at:

$$s(X_2)\left(v_0^p \times v_0^k\right) = s(X_1)\left(v_0^p \times v_0^k\right) = v_1^p \times v_1^k$$

then X_2 *is also detected by* D. We say that X_2 **obfuscates** X_1 or $obs(X_2)$ **behaviorally obfuscates** $obs(X_1)$ at $\left(v_0^p \times v_0^k, v_1^p \times v_1^k\right)$, denoted by $obs(X_2) \approx obs(X_1)$. Clearly, the relation $\approx$ fixed by a pair of values $\left(v_0^p \times v_0^k, v_1^p \times v_1^k\right)$ is an equivalence relation.

Example 5.3.1. Let us consider the paths X_1', X_2' respectively consisting of the 3 last morphisms of X_1, X_2 in Listings 5.5 and 5.6, namely:

$$X_1' = [strncpy_2^p, NtOpenKey_3^{p\text{-}k}, memcpy_4^p]$$
$$X_2' = [strncpy_2^p, strncpy_3^p, NtOpenKey_4^{p\text{-}k}]$$

In general, $s(X_1') \neq s(X_2')$ but they will end at the same values if we let them start at values so that the data on $[src1] \cup [src2]$[1] are the same. At these particular values, if X_1' is detected by D then X_2' is also detected, and:

$$obs(X_1') = [NtOpenKey_4^{p\text{-}k}] \approx [NtOpenKey_3^{p\text{-}k}] = obs(X_2')$$

It is also clear that these syscall invocations have the same name but actually consist in two different morphisms.

5.3.2 Main obfuscation theorem

We now show a theorem stating that if a benign path has the same effects on the kernel data as a malicious path, then there exist (malicious) paths having the same path semantics as the initial malicious one, and the same observations as the benign one. It seems may be not surprising though, this result has two main advantages. First, it gives a first formal treatment of *camouflage techniques*. Second, the proof of this theorem is *constructive*. This means that the theorem does not only show the existence of such malicious paths but allows us to build them. As we will present later, this plays also a role in a very first step towards an automated way of detecting such malicious paths.

In order to state the theorem, we need to introduce the notions of process and kernel (partial) semantics to distinguish the effects caused by a path on a kernel memory space from the ones caused on a process memory space.

Definition 5.3.2. Given $X \in \mathcal{X}$ and its path semantics $s(X)$ wrt the interaction category $\mathcal{C}\langle M^p, M^k\rangle$, the following morphisms, named:

[1] $[src1]$ and $[src2]$ denote memory spaces as explained in Examples 5.2.1 and 5.2.2.

- kernel semantics $k(X)$, kernel partial semantics $k(X)[v^p]$ at v^p,
- process semantics $p(X)$, process partial semantics $p(X)[v^k]$ at v^k,

are defined so that the following diagram commutes:

$$\begin{array}{ccccc} & & e & & \\ & \swarrow^{v^p} & \downarrow{\scriptstyle v^p \times v^k} & \searrow^{v^k} & \\ dom(M^p) & \xrightarrow{1_{dom(M^p)} \times v^k} & dom(M^p) \times dom(N^k) & \xleftarrow{v^p \times 1_{dom(N^k)}} & dom(N^k) \\ {\scriptstyle p(X)[v^k]}\downarrow & \swarrow_{p(X)} & \downarrow{\scriptstyle s(X)} & \searrow_{k(X)} & \downarrow{\scriptstyle k(X)[v^p]} \\ dom(M^p) & \xleftarrow{\pi_p} & dom(M^p) \times dom(L^k) & \xrightarrow{\pi_k} & dom(L^k) \end{array}$$

Example 5.3.2. The semantics of the path X_1 in Listing 5.5 is a morphism[1]:

$$s(X_1) : dom([src1] \cup [src2] \cup [dst]) \times e \to dom([src1] \cup [src2] \cup [dst]) \times dom([h])$$

and its process and kernel semantics are morphisms:

$$\begin{aligned} p(X_1) &: dom([src1] \cup [src2] \cup [dst]) \times e \longrightarrow dom([src1] \cup [src2] \cup [dst]) \\ k(X_1) &: dom([src1] \cup [src2] \cup [dst]) \times e \longrightarrow dom([h]) \end{aligned}$$

The following theorem shows that if we first find an intermediate path $X_{1\text{-}2}$ having just the same kernel semantics as X_1 (i.e. the same effects on the kernel memory space), then we can later modify $X_{1\text{-}2}$ (while keeping its observable behaviors) to obtain X_2 having the same paths semantics as X_1.

Theorem 3 (Camouflage)**.** *Let $X_1 \in \mathcal{X}$ and $v^p \times v^k \in source(s(X_1))$, for each $X_{1\text{-}2} \in \mathcal{X}$ such that $p(X_{1\text{-}2})[v^k]$ is monic (i.e. injective) and:*

$$k(X_{1\text{-}2})\left(v^p \times v^k\right) = k(X_1)\left(v^p \times v^k\right)$$

there exists $X_2 \in \mathcal{X}$ satisfying $obs(X_2) = obs(X_{1\text{-}2})$ and:

$$s(X_2)\left(v^p \times v^k\right) = s(X_1)\left(v^p \times v^k\right)$$

Proof. We construct X_2 by first letting it imitate $X_{1\text{-}2}$ and then concatenating a process internal computation s^p defined by:

$$s^p = p(X_1)[v^k] \circ p(X_{1\text{-}2})[v^k]^{-1}$$

The morphism s^p is well defined since $p(X_{1\text{-}2})[v^k]$ is monic. The execution path $X_2 = X_{1\text{-}2}@[s^p]$ makes the diagram in Figure 5.2 commute. We deduce from this

[1] $[src1], [src2], [dst]$ and $[h]$ still denote memory spaces.

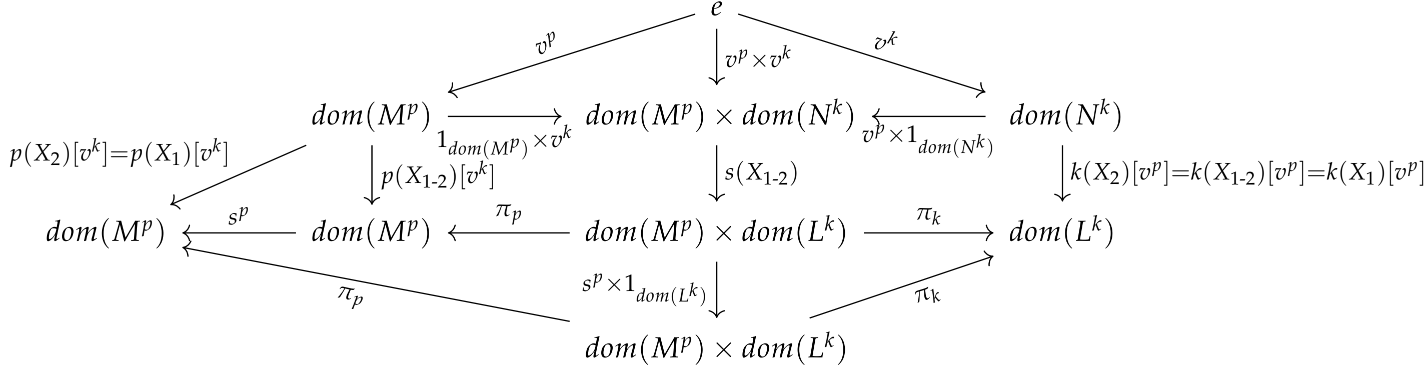


Figure 5.2: Obfuscated path construction

commutative diagram and by Definition 5.3.2 that:

$$p\left(X_2\right) \circ \left(v^p \times v^k\right) = p\left(X_2\right)\left[v^k\right] \circ v^p = p\left(X_1\right)\left[v^k\right] \circ v^p = p\left(X_1\right) \circ \left(v^p \times v^k\right)$$
$$k\left(X_2\right) \circ \left(v^p \times v^k\right) = k\left(X_{1\text{-}2}\right) \circ \left(v^p \times v^k\right) = k\left(X_1\right) \circ \left(v^p \times v^k\right)$$

Consequently, $s\left(X_2\right) \circ \left(v^p \times v^k\right) = s\left(X_1\right) \circ \left(v^p \times v^k\right)$. Moreover, we clearly have $obs\left(X_2\right) = obs\left(X_{1\text{-}2}\right)$ since X_2 is just obtained by concatenation of $X_{1\text{-}2}$ with some (unobservable) process internal computation s^p. And so the result. □

In other words, if X_1 is a malicious path and $X_{1\text{-}2}$ (possibly benign) has the same kernel semantics, then we can build X_2 so that

$$s\left(X_2\right)\left(v^p \times v^k\right) = s\left(X_1\right)\left(v^p \times v^k\right)$$

Then X_2 is also malicious; but $obs\left(X_2\right) = obs\left(X_{1\text{-}2}\right)$, so it looks like a benign path.

Example 5.3.3. The paths X_1 and $X_{1\text{-}2}$ in Listings 5.5 and 5.7 have the same partial kernel semantics at the values `"\SYSTEM\CurrentControlSet\..."` $\cup \ldots \cup \ldots$ of $dom\left([src1] \cup [src2] \cup [dst]\right)$ but the process partial semantics are not (because values on $dom\left([src1] \cup [src2] \cup [dst]\right)$ are set to 0 in $X_{1\text{-}2}$).

```
NtOpenKey(h,...{..."\SYSTEM\CurrentControlSet\..."...});
memset(dst,0,1024);memset(src1,0,1024);memset(src2,0,1024);
```

Listing 5.7: Path $X_{1\text{-}2}$

5.3.3 Obfuscated path generation through path replaying

As previously mentioned, the proof of Theorem 3 will allow to generate paths with camouflaged behaviors. The intuition behind such a procedure (will be presented

below), named *path replaying*, is to transform a path X_1 by specializing some invoked values inside the process memory space M^p. We start by introducing the projection morphisms that allow us to extract the partial values of a given total value.

Definition 5.3.3 (Projection morphisms)**.** Let $v \in m_1 \otimes m_2$, the partial values $v_1 \in m_1$ and $v_2 \in m_2$ are respectively defined by the morphisms π_{m_1} and π_{m_2} making the following diagram commute:

$$\begin{array}{ccccc} & & e & & \\ & \overset{v_1}{\swarrow} & \downarrow v_1 \otimes v_2 & \overset{v_2}{\searrow} & \\ m_1 & \xleftarrow{\pi_{m_1}} & m_1 \otimes m_2 & \xrightarrow{\pi_{m_2}} & m_2 \end{array}$$

Given a path $X = [s_1^{j_1}, s_2^{j_2}, \ldots, s_n^{j_n}]$ and a value $v^p \times v^k \in source\,(s\,(X))$; for $1 \leq i \leq n$, define X_l to be the path containing the first l morphisms of X, i.e. $X_l = [s_1^{j_1}, s_2^{j_2}, \ldots, s_l^{j_l}]$. Consider morphisms $s_l^{j_l} \in obs(X)$ (i.e. $j_l = p\text{-}k$), the source value $v_l^p \in dom\,(M^p)$ invoked by each $s_l^{j_l}$ can be evaluated by:

$$v_l^p = p\,(X_{l-1}) \left(\pi_{source(s(X_{l-1}))} \left(v^p \times v^k\right)\right)$$

Definition 5.3.4 (Replay path)**.** Let $X = [s_1^{j_1}, s_2^{j_2}, \ldots, s_n^{j_n}] \in \mathcal{X}$, $v^p \times v^k \in source\,(s\,(X))$, the *replay* $rep(X) = [r_1, r_2, \ldots, r_n]$ of X at $v^p \times v^k$ is defined by:

$$r_i = \begin{cases} s_i^p & \text{if } j_i = p \\ 1_{dom(M^p)} \times k(s_i^{p\text{-}k})[v_i^p] & \text{if } j_i = p\text{-}k \end{cases}$$

Example 5.3.4. The path in Listing 5.8 is the replay of one in Listing 5.6 at the invoked value `"\SYSTEM\CurrentControlSet\..."` of $dom\,([dst])$.

```
memcpy(src2,src1,1024);
strncpy(dst+2,src2,15);strncpy(dst+17,src1+15,25);
NtOpenKey(h,...{..."\SYSTEM\CurrentControlSet\..."...});
```

Listing 5.8: Replay $rep(X_2)$ of X_2

Proposition 5.3.1. *Let $X_1 \in \mathcal{X}$ and $rep\,(X_1)$ be the replay of X_1 at $v^p \times v^k \in source\,(s\,(X_1))$. For each $X_{1\text{-}2} \in \mathcal{X}$ satisfying $s\,(X_{1\text{-}2}) = s\,(obs\,(rep\,(X_1)))$, the following properties hold:*

- *$p\,(X_{1\text{-}2})\,[v^k]$ is monic and $k(X_{1\text{-}2})\,\left(v^p \times v^k\right) = k(X_1)\,\left(v^p \times v^k\right)$,*
- *if $X_2 = X_{1\text{-}2}@[p(X_1)[v^k]]$, where @ is the usual concatenation operator on lists, then $s\,(X_2)\,\left(v^p \times v^k\right) = s\,(X_1)\,\left(v^p \times v^k\right)$.*

Proof. For readability, let Y denote $obs\,(rep\,(X_1))$. For the first property, since $s\,(X_{1\text{-}2}) = s\,(Y)$ by assumption, we need to show that $p\,(Y)\,[v^k]$ is monic and that:

$$k\,(Y) \left(v^p \times v^k\right) = k\,(X_1) \left(v^p \times v^k\right)$$

By definition of the replay path, Y can be written as follows:

$$Y = [1_{dom(M^p)} \times k(s_i^{p\text{-}k})[v_i^p], \ldots, 1_{dom(M^p)} \times k(s_l^{p\text{-}k})[v_l^p]]$$

To prove the former, note that $s\,(Y)$ has form $1_{dom(M^p)} \times s^k$ where s^k is some morphism of the shape:

$$s^k \colon dom(N^k) \longrightarrow dom(L^k)$$

Hence $p\,(Y)\,[v^k]$ is equal to $1_{dom(M^p)}$ and is obviously monic. The later equality on kernel semantics is trivially deduced from the definition of the replay path.

The second property, $X_2 = X_{1\text{-}2}@[p(X_1)[v^k]]$, is deduced from the proof of Theorem 3. In this proof, the concatenated morphism s^p is defined by:

$$s^p = p\,(X_1)\,[v^k] \circ p\,(X_{1\text{-}2})\,[v^k]^{-1}$$

But $p\,(X_{1\text{-}2})\,[v^k] = p\,(Y)\,[v^k] = 1_{dom(M^p)}$, so $s^p = p\,(X_1)\,[v^k]$. □

The former shows that $X_{1\text{-}2}$ satisfies the assumptions of Theorem 3 while the later explicitly builds an instance of X_2. Hence $obs\,(X_2)$ *behaviorally obfuscates* $obs\,(X_1)$ at $\left(v^p \times v^k, s\,(X_1)\left(v^p \times v^k\right)\right)$, provided that X_1 is detected by D.

5.3.4 Graph-based path transformation

Since $rep\,(X_1)$ is constructed out of X_1 by replacing each $s_i^{p\text{-}k} \in obs\,(X_1)$ by

$$1_{dom(M^p)} \times k(s_i^{p\text{-}k})[v_i^p]$$

the observable paths $obs\,(rep\,(X_1))$ and $obs\,(X_1)$ have the shapes:

$$obs\,(X_1) = [s_i^{p\text{-}k}, \ldots, s_l^{p\text{-}k}]$$
$$obs\,(rep\,(X_1)) = [1_{dom(M^p)} \times k(s_i^{p\text{-}k})[v_i^p], \ldots, 1_{dom(M^p)} \times k(s_l^{p\text{-}k})[v_l^p]]$$

Proposition 5.3.1 provides a straightforward way of generating an obfuscated path X_2 of X_1 by setting:

$$X_2 = X_{1\text{-}2}@[p(X_1)[v^k]]$$

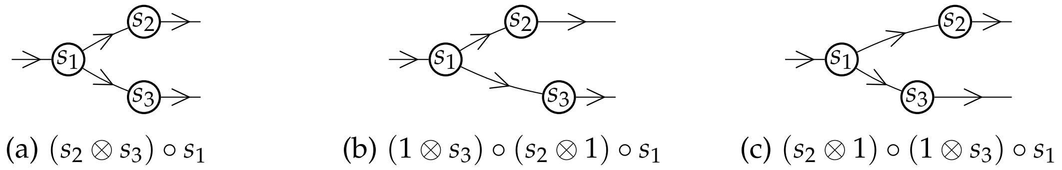
for some $X_{1\text{-}2}$ such that $X_{1\text{-}2} = obs\,(rep\,(X_1))$. The obtained path X_2 complies with $obs\,(X_2) = obs\,(rep\,(X_1))$ and $obs\,(X_2) \neq obs\,(X_1)$.

Though having distinct syscall invocations, these paths are "not that different" in the sense that the lists of the corresponding syscall names are still identical (cf. Example 5.3.4). The general objective of this subsection is to show how to generate paths that are semantically equivalent to $obs\,(rep\,(X_1))$ but with distinct observations. For that purpose, the string diagrams will be used since they allows to consider *semantics-preserving transformations* on the syscall invocations in $obs\,(rep\,(X_1))$.

By Proposition 5.2.2, the path semantics $s(obs(rep(X_1)))$ is represented by a morphism term constructed from the morphisms $1_{dom(M^p)} \times k(s_i^{p\text{-}k})[v_i^p]$. Hence it has a graphical representation as a string diagram, moreover we can safely omit the identity morphism $1_{dom(M^p)}$ in considering these morphisms.

Remark 5.3.1. In what follows, each path $obs(rep(X_1))$ is considered without the identity morphism $1_{dom(M^p)}$ in its elements.

Among string diagrams, we will only consider **path diagrams**, namely those such that the projection of nodes on an horizontal axis is an injective function, so the projection allows us to define a *total order* on nodes. The reason for restricting our graphical representation to path diagrams is that they represent their corresponding paths in an unambiguous way.



(a) $(s_2 \otimes s_3) \circ s_1$ (b) $(1 \otimes s_3) \circ (s_2 \otimes 1) \circ s_1$ (c) $(s_2 \otimes 1) \circ (1 \otimes s_3) \circ s_1$

Figure 5.3: String diagrams

Example 5.3.5. Figures 5.3b and 5.3c are path diagrams respectively representing the paths $[s_1, s_2, s_3]$ and $[s_1, s_3, s_2]$, but Figure 5.3a is not since there is an ambiguity about projections of s_2 and s_3.

The following theorem gives us a sound property, when applied to the corresponding string (or path) diagrams, of the semantics-preserving transformations from one path to another.

Theorem 4 (Coherence of progressive plane diagrams [84, 144])**.** *The equivalence of two morphism terms in monoidal categories is deduced from axioms iff their string diagrams are planar isotopic.*

Following [144], we accept an informal definition of *planar isotopy* between string diagrams as "...one can be transformed to the other by continuously moving around nodes..." (but keep the diagram always progressive), the formal treatment can be referenced in [84].

Example 5.3.6. Figures 5.3a, 5.3b and 5.3c are planar isotopic.

Between path diagrams, planar isotopy can be though of as moving the nodes but keeping the total order compatible with the partial order $\preccurlyeq$ (see also section 5.2.2). Hence, a *linear extension* Y of $obs(rep(X_1))$, namely a permutation where the total order remains to be compatible with $\preccurlyeq$, will preserves the semantics of $obs(rep(X_1))$. That leads to Algorithm 1.

Example 5.3.7. The paths X_3, X_4 in Listings 5.9 and 5.10 can be respectively represented by path diagrams in Figures 5.3b and 5.3c. Consequently, given X_3, Algorithm 1 can generate X_4.

Input: an observable path $obs\,(rep\,(X_1))$
Output: a permutation Y satisfying $s\,(Y) = s\,(obs\,(rep\,(X_1)))$

begin
  $M_1 \leftarrow$ morphism term of $s\,(obs\,(rep\,(X_1)))$;
  $G_1 \leftarrow$ string diagram of M_1;
  $(obs\,(rep\,(X_1))\,, \preccurlyeq) \leftarrow$ poset with order induced from G_1;
  $(Y, \leq) \leftarrow$ a linear extension of $(obs\,(rep\,(X_1))\,, \preccurlyeq)$;
end

Algorithm 1: Obfuscation by diagram deformation

```
NtCreateKey(h,...{..."\SOFTWARE\AD\"...}...);          /*s1*/
NtSetValueKey(h,...{..."DOWNLOAD"...}...,"abc");      /*s2*/
NtSetValueKey(h,...{..."URL"...}...,"xyz");           /*s3*/
```

Listing 5.9: Path X_3

```
NtCreateKey(h,...{..."\SOFTWARE\AD\"...}...);          /*s1*/
NtSetValueKey(h,...{..."URL"...}...,"xyz");           /*s3*/
NtSetValueKey(h,...{..."DOWNLOAD"...}...,"abc");      /*s2*/
```

Listing 5.10: Path X_4

A variable in a morphism term (resp. a node in the string diagram) is also a *placeholder* [144] that can be substituted by another term (resp. another diagram) having the same semantics, so Algorithm 2 is derived from Algorithm 1.

Input: an observable path $obs\,(rep\,(X_1))$
Output: a new path Y satisfying $s\,(Y) = s\,(obs\,(rep\,(X_1)))$

begin
  $M_1 \leftarrow$ morphism term of $obs\,(rep\,(X_1))$;
  $s \leftarrow$ a morphism of $obs\,(rep\,(X_1))$;
  $X \leftarrow$ an execution path satisfying $s(X) = s$;
  $M \leftarrow$ morphism term of X;
  $M_2 \leftarrow$ morphism term obtained by replacing s in M_1 by M;
  $G_2 \leftarrow$ string diagram of M_2;
  $((obs\,(rep\,(X_1)) \setminus s) \cup X, \preccurlyeq) \leftarrow$ poset with order induced from G_2;
  $(Y, \leq) \leftarrow$ a linear extension of $((obs\,(rep\,(X_1)) \setminus s) \cup X, \preccurlyeq)$
end

Algorithm 2: Obfuscation by node replacement

Example 5.3.8. Consider the replacement of s_2 in Listing 5.9 by $X = [s_2', s_2]$ where:

$$s_2' = \texttt{NtSetValueKey}(\texttt{h}, \ldots\ldots\text{"}\texttt{DOWNLOAD}\text{"}\ldots\ldots, \text{"}\texttt{a}'\texttt{b}'\texttt{c}'\text{"})$$

Using this replacement, given X_3 in Listing 5.9, Algorithm 2 can generate X_5.

```
NtCreateKey(h,...{..."\SOFTWARE\AD\"...}...);          /*s1*/
NtSetValueKey(h,...{..."DOWNLOAD"...}...,"a'b'c'");  /*s2'*/
NtSetValueKey(h,...{..."URL"...}...,"xyz");           /*s3*/
NtSetValueKey(h,...{..."DOWNLOAD"...}...,"abc");      /*s2*/
```

Listing 5.11: Path X_5

Proposition 5.3.2. *Algorithms 1 and 2 are sound.*

Proof. Let Y_1 denote $obs\,(rep\,(X_1))$ and $\preccurlyeq$ is the partial order on Y_1 induced from the string diagram G_1 of $s(Y_1)$. Let Y_2 be a permutation of Y_1 whose total order is a linear extension of $\preccurlyeq$, the path diagram of Y_2 is G_2. We prove that G_2 can be deformed to G_1 by moving the nodes but keep the directions of edges.

The trick is to deform G_2 to G_1 through several intermediate diagrams by elementary steps so that the directions of edges are preserved after each step:

$$G_2 \to G_{21} \to G_{22} \to \ldots \to G_1$$

Now let $\leq_1$ be the total order of Y_1, applying a sorting by element exchanging algorithm, e.g. bubble sort, with input Y_2 and total order $\leq_1$. Clearly, the output of sorting will be Y_1, and G_2 is deformed to G_1.

At each step of bubble sort, two adjacent nodes s_i, s_j are swapped. First, neither the edge $s_i \to s_j$ (i.e. $s_i \preccurlyeq s_j$) nor $s_j \to s_j$ (i.e. $s_j \preccurlyeq s_i$) exists since the total orders of Y_1 and Y_2 are both compatible with $\preccurlyeq$. Second, the swapping does not change the direction of other edges since only the relative position between s_i and s_j is changed. Hence the swapping at each step keeps the direction of edges, in other words the deformation keeps the direction of edges. □

5.4 Implementation and experiments

In this section, we apply Algorithms 1 and 2 on several sub-paths extracted from real malwares: `Dropper.Win32.Dorgam` [55] and `Gen:Variant.Barys.159` [8]. The experimental obfuscator, implemented in C++ and Haskell, uses Pin DBI framework [104] for path tracing and Functional Graph Library [58] for path transforming. Its source code is available at [157].

The results are visualized by string diagrams. In each diagram, the numbers appearing as node labels represent the total order in the original path; the fictitious nodes `Input` and `Output` are added as the *minimum* and the *maximum* in such a way that the path can be considered as a lattice. On any line, the number appearing as edge labels represent the handlers which identify the corresponding memory space inside kernel. On different lines, the same numbers may identify different memory spaces. The obfuscated paths generated by Algorithm 1 are linear extensions which are compatible with the order defined in the lattice. Note that their corresponding diagrams are always path diagrams (but they are not illustrated here).

Experiment 1 The trojan `Dropper.Win32.Dorgam` [55] has its registry initializing stage as the trace in Listing 5.2 (cf. Section 5.2.1). Let $[h_i], i \in \{1, 4, 7, 10, 13, 16, 19, 22\}$ denote the memories identified by the handlers of the accessed registry keys, the replay path is formulated by morphisms:

$$NtOpenKey_i^{p\text{-}k} : e \to dom\left([h_i]\right)$$
$$NtSetValueKey_{i+1}^{p\text{-}k} : dom\left([h_i]\right) \to dom\left([h_i]\right)$$
$$NtClose_{i+2}^{p\text{-}k} : dom\left([h_i]\right) \to e$$

Its string diagram is represented in Figure 5.4. The number of possible linear extensions is:

$$e(X) = \binom{24}{3}\binom{21}{3}\binom{18}{3}\binom{15}{3}\binom{12}{3}\binom{9}{3}\binom{6}{3}\binom{3}{3} = \frac{24!}{6^8}$$

namely more than 369 quadrillion extensions.

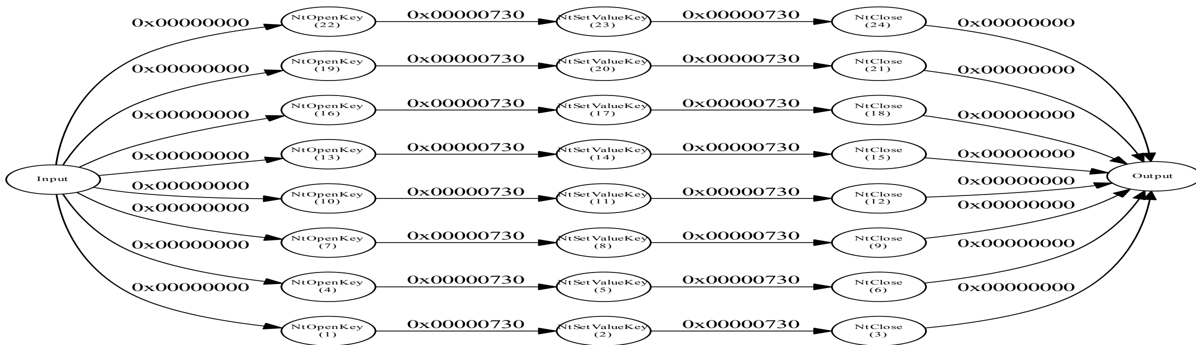


Figure 5.4: Registry initializing string diagram

Exepriment 2 The trojan also uses the trace in Listing 5.12 to create a copy of `iexplore.exe`, its replay path has the string diagram provided in Figure 5.5.

```
NtCreateFile (FileHdl=>0x00000730,File<=\??\C:\Program Files\Internet Explorer\IEXPLORE.EXE)   Ret=>0
NtCreateFile (FileHdl=>0x0000072C,File<=\??\C:\Program Files\iexplore.exe)                     Ret=>0
NtReadFile   (FileHdl<=0x00000730,BuffAddr<=0x0015C898, ByteNum<=65536)                         Ret=>0
NtWriteFile  (FileHdl<=0x0000072C, BuffAddr<=0x0015C898, ByteNum<=65536)                        Ret=>0
......
NtReadFile   (FileHdl<=0x00000730,BuffAddr<=0x0015C898, ByteNum<=65536)                         Ret=>0
NtWriteFile  (FileHdl<=0x0000072C, BuffAddr<=0x0015C898, ByteNum<=48992)                        Ret=>0
NtReadFile   (FileHdl<=0x00000730, BuffAddr<=0x0015C898, ByteNum<=65536)                        Ret=>3221225489
NtClose      (Hdl<=0x00000730)                                                                  Ret=>0
NtClose      (Hdl<=0x0000072C)                                                                  Ret=>0
```

Listing 5.12: File copying

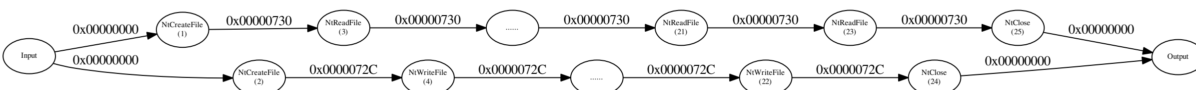


Figure 5.5: File copying string diagram

This path can be considered as an obfuscated version (generated by Algorithm 2) of the path whose string diagram is in Figure 5.6, by considering the equivalences:

$$NtReadFile^{p\text{-}k}_{3(orig)} = [NtReadFile^{p\text{-}k}_{3}, NtReadFile^{p\text{-}k}_{5}, \dots]$$
$$NtWriteFile^{p\text{-}k}_{4(orig)} = [NtWriteFile^{p\text{-}k}_{4}, NtWriteFile^{p\text{-}k}_{6}, \dots]$$

It also means that a *behavior matching* detector can detect an obfuscated path, as-

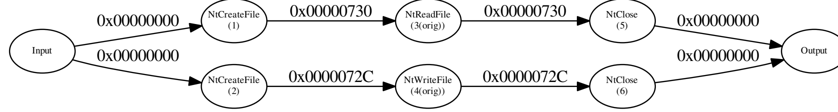


Figure 5.6: File copying original string diagram

suming the prior knowledge of both the original path and the semantics equivalences described above.

Experiment 3 We consider the ransomware `Gen:Variant.Barys.159` [8]. The extracted path in Listing 5.13 explains how the malware conceals itself by injecting code into file explorer process `explorer.exe`.

```
NtOpenProcess        (ProcHdl=>0x00000780,DesiredAccess<=1080,ProcId<=0x00000240)                      Ret=>0
NtCreateSection      (SecHdl=>0x00000778,AllocAttrs<=SEC_COMMIT,FileHdl<=0x00000000)                   Ret=>0
NtMapViewOfSection   (SecHdl<=0x00000778,ProcHdl<=0xFFFFFFFF,BaseAddr<=0x02660000)                     Ret=>0
NtReadVirtualMemory  (ProcHdl<=0x00000780,BaseAddr<=0x7C900000,BuffAddr<=0x02660000,ByteNum<=729088)   Ret=>0
NtMapViewOfSection   (SecHdl<=0x00000778,ProcHdl<=0x00000780,BaseAddr<=0x7C900000)                     Ret=>0
```

Listing 5.13: Code injecting

The malware first obtains the handler `0x780` to the running instance (whose process id is `0x240`) of `explorer.exe` then creates a section object identified by the handler `0x778`. It maps this section to the memory of malware, copies some data of the instance into the mapped memory, modifies data on this memory and maps the section (now contains modified data) back to the instance.

Let $[h_1], [h_2]$ denote the memories identified by handlers of the opened process and of the created section, the replay path is formulated by morphisms:

$$NtOpenProcess^{p\text{-}k}_{1} : e \to dom\left([h_1]\right)$$
$$NtCreateSection^{p\text{-}k}_{2} : e \to dom\left([h_2]\right)$$
$$NtMapViewOfSection^{p\text{-}k}_{3} : dom\left([h_2]\right) \to dom\left([h_2]\right)$$
$$NtReadVirtualMemory^{p\text{-}k}_{4} : dom\left([h_1]\right) \to dom\left([h_1]\right)$$
$$NtMapViewOfSection^{p\text{-}k}_{5} : dom\left([h_1] \cup [h_2]\right) \to dom\left([h_1] \cup [h_2]\right)$$

and the corresponding string diagram is provided in Figure 5.7.

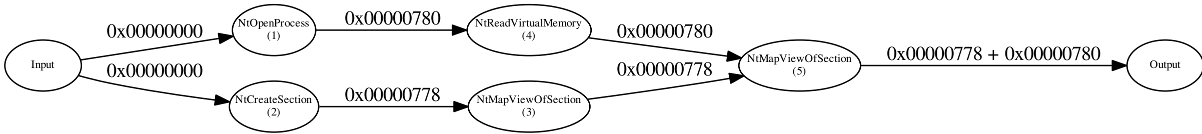


Figure 5.7: Code injecting string diagram

If the morphism $NtReadVirtualMemory_4^{p\text{-}k}$ is replaced by a path $[s_{4\text{-}1}^{p\text{-}k}, s_{4\text{-}2}^{p\text{-}k}]$ corresponding to the syscall invocations in Listing 5.14 then this replacement leads to the string diagram in Figure 5.8.

```
NtReadVirtualMemory(ProcHdl<=0x00000780,BaseAddr<=0x7C900000,BuffAddr<=0x02660000,ByteNum<=9088)
NtReadVirtualMemory(ProcHdl<=0x00000780,BaseAddr<=0x7C909088,BuffAddr<=0x02669088,ByteNum<=72000)
```

Listing 5.14: Replacing NtReadVirtualMemory

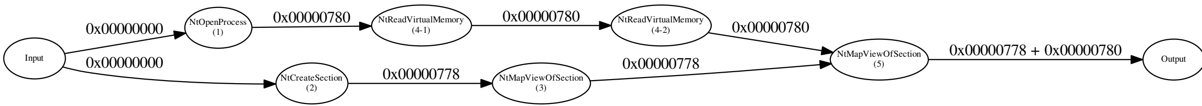


Figure 5.8: Code injecting obfuscated string diagram

The numbers of linear extensions for the original string diagram and the obfuscated one are respectively:

$$e_1(X) = \binom{4}{2}\binom{2}{2} = 6 \qquad e_2(X) = \binom{5}{3}\binom{2}{2} = 10$$

5.5 Obfuscated path detection

We will now discuss the detection by using practical detectors introduced in previous existing works on *behavior matching* [36, 65, 90].

Basically, a behavior matching detector first represents an observable path by a *directed acyclic graph* (abbr. DAG) using the causal dependency between morphisms, a morphism s_j (directly or indirectly) depends on s_i if the sources values of s_j are (directly or indirectly) deduced from the target values of s_i. Then the detector decides a path is malicious or not by verifying whether there exists a malicious pattern occurring as a subgraph of the original DAG. Here the malicious pattern is a (sub-)DAG and it can be obfuscated to another semantics equivalent DAG.

Whereas Algorithm 1 can generate a large amount of paths, the verification of whether an obfuscated path is semantically equivalent to the original path is simple: it is an instance of the *DAG automorphism* problem where every vertex is mapped to itself. The instance can be decided in *P-time* by repeatedly verifying whether two paths have the same set of minimal elements, if they do then remove the set of minimal elements from both paths and repeat; if they do not then decide **No**; if the sets are both empty then decide **Yes** and stop.

The detection of obfuscated paths generated by Algorithm 2 is more challenging. When applied naively, the behavior matching does not work since the algorithm can generate paths of morphisms corresponding to syscall names and invocations distinct from those of the original path. More generally, it may be nonsense to compare an obfuscator and a detector which use different sets of behavioral transformations. In other words, as discussed in [47], a detector which abstracts behaviors by using the transformation set T, will be bypassed by an obfuscator which generates behaviors by using a set T' so that $T \cap T' \neq \emptyset$.

The original *behavior matching* techniques can be strengthened by generating (e.g. by using Algorithm 2) in prior a set of patterns that are semantically equivalent to the original one (see also the discussion in the experiment 2 of Section 5.4). Conversely, that means simplifying obfuscated paths to their original unique form, several simplifying techniques has been studied in some existing works on *semantics rewriting* (e.g. [9]). So we might suggest that a combination of *behavior matching* and *semantics rewriting* will improve the presented analysis. We reserve such an improvement as a future work.

Chapter 6

Conclusions and perspectives

6.1 Conclusions

Our work about input message format extraction bases directly and indirectly on work of many other authors. Concretely, it has been guided by the original ideas of J. Caballero et al [26, 28], who have shown clearly, at the first time, that the semantics of messages should be determined by the program. We have reformulated the problem of message format, proposed another interpretation and developed it using another theoretical framework. That is the theory of reactive systems, pioneered by R. Milner [114].

Formal reverse code engineering The automated approach in reverse code engineering have been capturing many efforts in recent years. In comparison with the classic manual approach, the automated one still seem to be in their infancy. But many researchers in this domains have been, step by step, sharpening their methods. One of the difficulties for the automated methods is to define rigorously what we need in doing reverse engineering.

To see the differences between automated and manual approach, we may think of the manual code reverse engineering is a self-defined process. The reverser takes some codes and try fo find relations between the concrete details in the codes with higher abstract patterns in his expert knowledge. In other words, he tries to fit concrete details with a abstract model existing already in his expert knowledge[1]. The automated approach is similar, except that the expert knowledge needs to be defined formally before reversing codes, and that is the main differences.

In a pilot paper which tries to put the reverse engineering as a rigorous process [33], the authors have given several formal definitions for terminologies of reverse engineering. Concretely, they have proposed that

[1]If such an abstract model does not exist already in his current knowledge, he may acquire the concrete details, with some pre-processing, into his knowledge.

> Reverse engineering is the process of analyzing a subject system to create representation of the system in another form or at a higher level of abstraction.

So the need here is a "representation of the system in another form or at a higher level of abstraction". But such a definition is too general, for example "what is a form?" or "what is an abstraction level?". To be applicable, it must be concretely stated under some clearly defined properties.

These questions are considered by R. Milner in [114], and we will reapply in our context that the reverse code engineering is a *science of artificial* in the sense of H. A. Simon [148]: the reverse code engineers do not study natural objects, they instead study artificial objects, that are programs written by other people. The opinions of the two authors is that *first*, a level of abstraction does not need to give a complete formalization of the examined object, instead it needs only to focus on some specific properties of this object. Second, a form resulted from reverse engineering does not need to be the exact form of the examined object, instead it needs only to simulate this object, at some specific properties.

Code obfuscation as a double-edged sword Since the reverse code engineering is a science of artificial, the efforts in software protection can be employed also to impede the efforts in reverse code engineering. While the researches in software protection have currently lots of solid bases, for example the employment of asymmetric cryptography in malicious programs [165] (see also the challenges discussed in Section 1.1.2). Unfortunately, such a base does not exist yet in reverse code engineering.

Concrete interpretation of message formats In studying the problem of input message format extraction, we have encountered the question of "what is a message format?". And our approach is more or less similar with the proposition above when we state in Section 2.3.3 that "the message format is a mathematical model of the message that fits a concrete implementation". And to practically realize this statement, we go a little further to interpret it concretely, to retrieve "the mathematical model" as a *labeled transition system* that classifies (i.e. "fits") the execution traces (i.e. "a concrete implementation") of the program following the input message values. This interpretation is obviously not a complete formalization of programs, it focuses on a very specific property: under different input values, the program may exhibits different execution traces.

Moreover, the final label transition system is not a "message format" in the sense of authors in [26, 28, 41, 45, 163]. It just simulates how the program parses the input messages so that the specific property observed on it and on the program are the same.

However, we are not sure that this property is essential or not. For example, with the hypothesis given in the stepwise abstraction are invalidated, then the stepwise

abstraction procedure cannot proceed. Another example is that the stepwise abstraction procedure bases on the existence of control flow instructions. But, there exist some theoretical instruction sets where the control flow instructions do not exist, for example the Turing-complete instruction set given in [54] consisting of only `mov` and a `jmp` that needs to be placed at the end of the program. Though we believe that the stepwise abstraction procedure can be fixed to adapt with such a special instruction set, we cannot state here that this property is essential.

In this work, we have used the labeled transition system model that is, more or less, considered as irrelevant with current researches in reverse code engineering. In fact, the theoretical results for our specific purpose are not available in the theory of labeled transition systems, we have constructed some of them in Chapter 3. The essential reason may be the theory of labeled transition systems have been developed to describe mainly the equivalence between processes [68, 128], whereas we need a notation of "prediction" to approximate unobservable behaviors of a process.

However, the linkage between reverse code engineering and the theory of labeled transition systems is not surprising. This theory, like other theories, describe several essential properties of concrete objects. One may, accidentally, find that her studied objects can be described by a theory which is developed from different motivations.

Message semantics The theoretical basis have restricted also the applicable scope of our work, as discussed in Section 4.3.3, but may be for good purposes. They allow us rigorously define what we need in a specific context where all relevant properties are clearly defined. This specific context, in fact, can give clear interpretations for some *implicit discussions* which are not stated clearly before. This situation may be similar with "time" in the following quote

> [141] ... For what is time? Who can easily and briefly explain it?... If no one ask of me, I know; if I wish to explain to him who asks, I know not....

That is the discussions about the semantics, given by the program, of the input message. The authors in [26, 28, 41, 45] have had already strong consciousness that one may get seriously lostness of message semantics if considering only the messages themselves and omitting the program. We just develop concretely their ideas, by showing the differences between labeled transition systems of different programs which parse the messages of the same HTTP protocol.

6.2 Perspectives

Labeled transition system as pattern Aside from the main purpose of the thesis, that means activating dormant codes by understanding the input message format. We can use the final labeled transition system as a "pattern" characterizing the program.

In the one hand, since such a pattern does not contains some low-level properties (e.g. assembly instructions), it is immune from several pattern verification obfuscation

tricks (cf. also the last part of Section 1.1.2) (e.g. dead codes (or `nop`) insertion, registers substitution, instructions transposition). This application of the labeled transition system is indeed very similar with the *template-based pattern* [35] where the authors have proposed the notion of template as a symbolic representation of concrete code patterns.

In the other hand, such a pattern seems to be reliable. Because if we get two programs which gives the same classification of input messages, then we can make a strong hypothesis that they have something in common, though their syntax can be very different. For example, one considers the LTS(s) given in Figures 4.11, 4.15a and 4.15b of `wget`, `links` and `nginx` respectively, can make a strong hypothesis that these programs, are not the same, but must have something in common. Indeed, we know already that these programs parse messages of HTTP protocol, and considering the codes in each program, we can observe also that they are very different. More formally, we can prove that this kind of pattern immune from several code obfuscation techniques, e.g. dead/`nop` instructions insertion, registers substitution, etc.

It may worth noting that in using LTS(s) as patterns, the input messages are not required to be some "concrete messages" received by the program (e.g. messages of HTTP, FTP protocols, etc.). In the first step of a *stepwise abstraction* procedure, we give no assumption about the input messages. They are just values of a buffer that is updated by an *exception*, or generally speaking, by some stimulus of the environment.

Additionally, the labeled transition system is just a concrete model for *relations between execution traces*. So there may exist other sorts of pattern, that are not labeled transition systems, but they are constructed upon some relation between execution traces. In other words, we compare two programs using some relation between execution traces, instead of using directly execution traces.

Better methods for stepwise abstraction procedure In Section 4.3.3, we have discussed about limits of the stepwise abstraction procedure. The technical limits can be improved by some better methods in constructing the execution tree, that requires determining the branching conditions of branchable control flow instructions. For example, the reverse execution plus fuzz testing can be replaced by a method using intermediate representation code lifting and SMT solvers. The theoretical limits (namely Hypotheses 1 and 2) are much harder to fix, they may require a different model than labeled transition systems.

Bibliography

[1] D. Andriesse and H. Bos. *An Analysis of the Zeus Peer-to-Peer Protocol*. Tech. rep. VU University Amsterdam, 2014 (cit. on p. 120).

[2] D. Andriesse, C. Rossow, B. Stone-Gross, D. Plohmann, and H. Bos. "Highly Resilient Peer-to-Peer Botnets Are Here: An Analysis of Gameover Zeus". In: MALWARE. 2013 (cit. on p. 120).

[3] M. Antonakakis, R. Perdisci, Y. Nadji, N. Vasiloglou, S. Abu-Nimeh, W. Lee, and D. Dagon. "From Throw-Away Traffic to Bots: Detecting the Rise of DGA-Based Malware". In: USENIX Security. 2012 (cit. on p. 21).

[4] S. Awodey. *Category Theory*. 2nd ed. Oxford Logic Guides. Oxford University Press, USA, Aug. 2010 (cit. on p. 128).

[5] BAP. *BAP nightly report for x86-64*. 2014. URL: http://bap.ece.cmu.edu/nightly-reports/index-x8664.html (cit. on p. 70).

[6] G. N. Barbosa and R. R. Branco. *Prevalent Characteristics in Modern Malware*. 2014. URL: https://www.blackhat.com/docs/us-14/materials/us-14-Branco-Prevalent-Characteristics-In-Modern-Malware.pdf (cit. on pp. 4, 12).

[7] S. Bardin and P. Herrmann. "OSMOSE: Automatic Structural Testing of Executables". In: (2011) (cit. on p. 16).

[8] Barys Malware. *Gen:Variant.Barys.159*. 2013. URL: http://goo.gl/YDC1o (cit. on pp. 145, 147).

[9] P. Beaucamps, I. Gnaedig, and J.-Y. Marion. "Behavior Abstraction in Malware Analysis". In: RV. 2010 (cit. on pp. 4, 149).

[10] F. Bellard. *QEMU Open source processor emulator*. 2014. URL: qemu.org (cit. on p. 102).

[11] A. R. Bernat and B. P. Miller. "Anywhere, Any Time Binary Instrumentation". In: PASTE. 2011 (cit. on p. 102).

[12] P. Biondi and D. Fabrice. *Silver Needle in the Skype*. 2006 (cit. on p. 18).

[13] B. Biswas and R. Mall. "Reverse Execution of Programs". In: *SIGPLAN Notices* 34.4 (1999), pp. 61–69 (cit. on p. 102).

[14] G. Bonfante, M. Kaczmarek, and J.-Y. Marion. "Morphological Detection of Malware". In: MALWARE. 2008 (cit. on pp. 3, 4).

[15] G. Bonfante, M. Kaczmarek, and J.-Y. Marion. "Architecture of a Morphological Malware Detector". In: *Journal in Computer Virology* 5.3 (2009), pp. 263–270 (cit. on p. 4).

[16] G. Bonfante, J.-Y. Marion, and T. D. Ta. *PathExplorer*. URL: https://github.com/tathanhdinh/PathExplorer (cit. on pp. 100, 102, 119).

[17] G. Bonfante, J.-Y. Marion, and T. Thanh Dinh. "Efficient Program Exploration by Input Fuzzing". In: Botconf. 2013 (cit. on pp. 9, 69).

[18] G. Bonfante, J.-Y. Marion, and T. Thanh Dinh. *Malware Message Analysis through Binary Traces*. Poster at GRSD. 2014 (cit. on pp. 9, 69).

[19] G. Bonfante, J.-Y. Marion, and T. Thanh Dinh. "Malware Message Classification by Dynamic Analysis". In: FPS. 2014 (cit. on pp. 9, 29, 69).

[20] T. Brosch and M. Morgenstern. *Runtime Packers: The Hidden Problem?* 2006. URL: https://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Morgenstern.pdf (cit. on pp. 4, 12).

[21] D. Bruening, E. Duesterwald, and S. Amarasinghe. "Design and Implementation of a Dynamic Optimization Framework for Windows". In: FDDO. 2001 (cit. on p. 102).

[22] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song, and H. Yin. "Automatically Identifying Trigger-based Behavior in Malware". In: *Botnet Analysis and Defense*. 2008, pp. 65–88 (cit. on pp. 7, 12, 16, 100, 101).

[23] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. "BAP: A Binary Analysis Platform". In: CAV. 2011 (cit. on pp. 16, 70).

[24] G. Bruneau. *DNS Sinkhole*. Aug. 2010. URL: http://www.sans.org/reading-room/whitepapers/dns/dns-sinkhole-33523 (cit. on p. 21).

[25] R. E. Bryant and D. R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Prentice-Hall, 2008 (cit. on pp. 69, 78).

[26] J. Caballero, P. Poosankam, C. Kreibich, and D. X. Song. "Dispatcher: Enabling Active Botnet Infiltration Using Automatic Protocol Reverse-Engineering". In: CCS. 2009 (cit. on pp. 7, 11, 16–18, 20, 99, 104, 110, 111, 117, 151–153).

[27] J. Caballero and D. Song. "Automatic protocol reverse-engineering: Message format extraction and field semantics inference". In: *Computer Networks* 57.2 (2013), pp. 451–474 (cit. on p. 18).

[28] J. Caballero, H. Yin, Z. Liang, and D. Song. "Polyglot: Automatic Extraction of Protocol Message Format using Dynamic Binary Analysis". In: CCS. 2007 (cit. on pp. 7, 11, 17, 19–22, 99, 104, 110, 111, 118, 151–153).

[29] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. "EXE: Automatically Generating Inputs of Death". In: CCS. 2006 (cit. on p. 16).

[30] H. Cai, Z. Shao, and A. Vaynberg. "Certified Self-Modifying Code". In: PLDI. 2007 (cit. on p. 74).

[31] J. Calvet, C. R. Davis, and P.-M. Bureau. "Malware authors don't learn, and that's good!" In: MALWARE. 2009 (cit. on p. 1).

[32] J. Calvet, J. M. Fernandez, and J.-Y. Marion. "Aligot: Cryptographic Function Identification in Obfuscated Binary Programs". In: *CCS*. 2012 (cit. on p. 3).

[33] E. J. Chikofsky and J. H. Cross. "Reverse Engineering and Design Recovery: A Taxonomy". In: *IEEE Software* 7.1 (1990), pp. 13–17 (cit. on p. 151).

[34] M. Christodorescu and S. Jha. "Static Analysis of Executables to Detect Malicious Patterns". In: SSYM. 2003 (cit. on p. 6).

[35] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. "Semantics-Aware Malware Detection". In: SSP. 2005 (cit. on pp. 4, 126, 154).

[36] M. Christodorescu, C. Kruegel, and S. Jha. "Mining Specifications of Malicious Behavior". In: SIGSOFT. 2007, pp. 5–14 (cit. on p. 148).

[37] J. Clause, W. Li, and A. Orso. "Dytan: A Generic Dynamic Taint Analysis Framework". In: ISSTA. 2007 (cit. on p. 102).

[38] C. S. Collberg and C. Thomborson. "Watermarking, Tamper-proffing, and Obfuscation: Tools for Software Protection". In: *TSE* 28.8 (2002) (cit. on p. 4).

[39] C. Collberg and J. Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. 1st. 2009 (cit. on p. 126).

[40] C. Collberg, C. Thomborson, and D. Low. *A Taxonomy of Obfuscating Transformations*. Tech. rep. 148. Department of Computer Sciences, The University of Auckland, 1997 (cit. on p. 6).

[41] P. M. Comparetti, G. Wondracek, C. Krügel, and E. Kirda. "Prospex: Protocol Specification Extraction". In: SSP. 2009 (cit. on pp. 11, 17, 99, 104, 110, 111, 118, 152, 153).

[42] K. Coogan, S. K. Debray, T. Kaochar, and G. M. Townsend. "Automatic Static Unpacking of Malware Binaries". In: WCRE. 2009 (cit. on p. 13).

[43] S. A. Cook. "The Complexity of Theorem-Proving Procedures". In: STOC. 1971 (cit. on p. 5).

[44] P. Cousot and R. Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". In: *POPL*. 1977 (cit. on p. 14).

[45] W. Cui, J. Kannan, and H. J. Wang. "Discoverer: Automatic Protocol Reverse Engineering from Network Traces". In: USENIX Security. 2007 (cit. on pp. 7, 11, 17–19, 99, 104, 110, 111, 152, 153).

[46] M. Dalla Preda. "Code Obfuscation and Malware Detection by Abstract Interpretation". PhD thesis. 2007 (cit. on p. 13).

[47] M. Dalla Preda. "The Grand Challenge in Metamorphic Analysis". In: ICISTM. 2012, pp. 439–444 (cit. on pp. 126, 149).

[48] M. Dalla Preda, M. Christodorescu, S. Jha, and S. Debray. "A Semantics-Based Approach to Malware Detection". In: POPL. 2007 (cit. on pp. 125, 126).

[49] M. Dalla Preda and R. Giacobazzi. "Semantics-based Code Obfuscation by Abstract Interpretation". In: *JCS* 17.6 (2009) (cit. on p. 13).

[50] M. Dalla Preda, R. Giacobazzi, S. K. Debray, K. Coogan, and G. M. Townsend. "Modelling Metamorphism by Abstract Interpretation". In: SAS. 2010 (cit. on p. 13).

[51] B. Dang, A. Gazet, and E. Bachaalany. *Practical Reverse Engineering: x86, x64, ARM, Windows Kernel, Reversing Tools, and Obfuscation*. Wiley, 2014 (cit. on p. 70).

[52] E. David. *Kaspersky Security Bulletin 2014. Malware Evolution*. 2014. URL: https://securelist.com/files/2014/12/Kaspersky-Security-Bulletin-2014-Malware-Evolution.pdf (cit. on pp. 1, 18).

[53] S. K. Debray and J. Patel. "Reverse Engineering Self-Modifying Code: Unpacker Extraction". In: WCRE. 2010 (cit. on p. 13).

[54] S. Dolan. *mov is Turing-complete*. Tech. rep. 2013 (cit. on p. 153).

[55] Dorgram Malware. *Trojan-Dropper.Win32.Dorgam.un*. 2013. URL: http://goo.gl/3e1AR (cit. on pp. 127, 145, 146).

[56] T. Dullien and S. Porst. "REIL: A platform-independent intermediate representation of disassembled code for static code analysis". In: CanSecWest. 2009 (cit. on p. 70).

[57] M. Egele, T. Scholte, E. Kirda, and C. Kruegel. "A Survey On Automated Dynamic Malware-Analysis Techniques and Tools". In: *ACM Computing Survey* 44.2 (2012) (cit. on p. 12).

[58] M. Erwig. *FGL/Haskell - A Functional Graph Library for Haskell*. 2008. URL: http://web.engr.oregonstate.edu/~erwig/fgl/haskell/ (cit. on p. 145).

[59] F. Falcon and N. Riva. *Dynamic Binary Instrumentation Frameworks: I know you're there spying on me*. 2012 (cit. on p. 102).

[60] N. Falliere and E. Chien. *Zeus: King of the Bots*. Tech. rep. 2009 (cit. on pp. 1, 119, 120).

[61] N. Falliere, L. O. Murchu, and E. Chien. *W32.Stuxnet Dossier*. Tech. rep. Symantec Security Response, Feb. 2011 (cit. on pp. 1, 5).

[62] P. Ferrie. *The Ultimate Anti-Debugging Reference*. Tech. rep. 2011 (cit. on p. 14).

[63] E. Filiol. “Formalisation and implementation aspects of κ-ary (malicious) codes”. In: *Journal in Computer Virology* 3.2 (2007), pp. 75–86 (cit. on p. 126).

[64] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. “A Sense of Self for Unix Processes”. In: SSP. IEEE, 1996, pp. 120–128 (cit. on p. 126).

[65] M. Fredrikson, S. Jha, M. Christodorescu, R. Sailer, and X. Yan. “Synthesizing Near-Optimal Malware Specifications from Suspicious Behaviors”. In: SSP. 2010 (cit. on pp. 4, 148).

[66] A. Fritzler and S. Werndorfer. *UnOfficial AIM/OSCAR Protocol Specification*. 2000. URL: http://www.oilcan.org/oscar/ (cit. on p. 18).

[67] E. R. Gansner, E. Koutsofios, S. C. North, and P.-K. Vo. “A Technique for Drawing Directed Graphs”. In: *IEEE Transactions on Software Engineering* (1993) (cit. on p. 111).

[68] R. v. Glabbeek. “The Linear Time – Branching Time Spectrum I: The Semantics of Concrete, Sequential Processes”. In: *Handbook of Process Algebra*. Elsevier, 2001. Chap. 1, pp. 3–99 (cit. on pp. 37, 153).

[69] P. Godefroid, N. Klarlund, and K. Sen. “DART: Directed Automated Random Testing”. In: PLDI. 2005 (cit. on pp. 16, 17, 100, 101).

[70] P. Godefroid, M. Y. Levin, and D. Molnar. “Automated Whitebox Fuzz Testing”. In: NDSS. 2008, pp. 151–166 (cit. on p. 16).

[71] C. P. Gomes, H. Kautz, A. Sabharwal, and B. Selman. “Satisfiability Solvers”. In: *Handbook of Knowledge Representation* 3 (2008), pp. 89–134 (cit. on p. 16).

[72] Graphviz. *Graph Visualization Software*. 2014. URL: http://www.graphviz.org/ (cit. on p. 111).

[73] GReAT, Kaspersky Lab. *The Mystery of the Encrypted Gauss Payload*. Aug. 2013. URL: http://goo.gl/uvWUA6 (cit. on pp. 1, 5).

[74] F. Guo, P. Ferrie, and T.-C. Chiueh. “A Study of the Packer Problem and Its Solutions”. In: RAID. 2008 (cit. on pp. 4, 12, 13).

[75] G. Hahn and C. Tardif. “Graph homomorphisms: structure and symmetry”. In: *Graph Symmetry*. Springer, 1997, pp. 107–166 (cit. on p. 29).

[76] K. M. Hazelwood. *Dynamic Binary Modification: Tools, Techniques, and Applications*. Morgan and Claypool Publishers, 2011 (cit. on p. 102).

[77] P. Hell and J. Nesetril. *Graphs and Homomorphisms*. Oxford University Press, 2004 (cit. on p. 29).

[78] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. 2007 (cit. on pp. 32, 36, 52).

[79] R. N. Horspool and N. Marovac. “An Approach to the Problem of Detranslation of Computer Programs”. In: *The Computer Journal* 23.3 (1980), pp. 223–229 (cit. on p. 72).

[80] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Sept. 2014 (cit. on pp. 69, 71, 72, 74).

[81] Intel Corporation. *Intel® 64 and IA-32 Optimization Reference Manual*. Sept. 2014 (cit. on pp. 69, 76).

[82] IOActive. *Reversal and Analysis of Zeus and SpyEye Banking Trojans*. Tech. rep. 2012 (cit. on pp. 1, 120).

[83] N. D. Jones. *Computability and Complexity - from a Programming Perspective*. MIT Press, 1997 (cit. on p. 73).

[84] A. Joyal and R. Street. "The Geometry of Tensor Calculus, I". In: *Advances in Mathematics* 88 (1 July 1991), pp. 55–112 (cit. on pp. 129, 143).

[85] M. G. Kang, P. Poosankam, and H. Yin. "Renovo: A hidden code extractor for packed executables". In: WORM. 2007 (cit. on p. 14).

[86] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis. "libdft: Practical Dynamic Data Flow Tracking for Commodity Systems". In: VEE. 2012 (cit. on p. 102).

[87] D. Kholia and P. Wegrzyn. "Looking Inside the (Drop) Box". In: WOOT. 2013 (cit. on p. 18).

[88] J. C. King. "Symbolic Execution and Program Testing". In: *CACM* 19.7 (1976), pp. 385–394 (cit. on pp. 16, 17, 91).

[89] D. Knuth, H. James, and V. Pratt. "Fast Pattern Matching in Strings". In: *SIAM Journal on Computing* (1977) (cit. on p. 22).

[90] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang. "Effective and Efficient Malware Detection at the End Host". In: USENIX Security. 2009, pp. 351–366 (cit. on pp. 3, 4, 126, 148).

[91] J. Z. Kolter and M. A. Maloof. "Learning to Detect and Classify Malicious Executables in the Wild". In: *Journal of Machine Learning Research* 6 (2006) (cit. on p. 4).

[92] D. Kozen. *Automata and Computability*. Springer, 1997 (cit. on p. 73).

[93] C. Lab. *sKyWIper (a.k.a. Flame a.k.a. Flamer): A complex malware for targeted attacks*. Tech. rep. Budapest University of Technology and Economics, 2012 (cit. on p. 1).

[94] K. Lab. *Gauss: Abnormal Distribution*. Tech. rep. 2012 (cit. on p. 1).

[95] D. Labs. *A New Iteration of the TDSS/TDL4 Malware Using DGA-based Command-and-Control*. Tech. rep. 2012 (cit. on p. 21).

[96] T. Labs. *Links Web Browser*. 2014. URL: http://links.twibright.com/ (cit. on pp. 103, 104).

[97] J. Lambek. “Cartesian Closed Categories and Typed Lambda-calculi”. In: Combinators and Functional Programming Languages. 1985, pp. 136–175 (cit. on p. 129).

[98] T. Lecroq. *Knuth-Morris-Pratt algorithm*. 1997. URL: http://www-igm.univ-mlv.fr/~lecroq/string/node8.html (cit. on p. 22).

[99] H. Lee, J. Alves-Foss, and S. Harrison. “The Use of Encrypted Functions for Mobile Agent Security”. In: *HICSS*. 2004 (cit. on p. 5).

[100] J. Lee, T. Avgerinos, and D. Brumley. “TIE: Principled Reverse Engineering of Types in Binary Programs.” In: *NDSS*. 2011 (cit. on pp. 3, 25).

[101] Z. Lin, X. Jiang, D. Xu, and X. Zhang. “Automatic Protocol Format Reverse Engineering through Context-Aware Monitored Execution”. In: NDSS. 2008 (cit. on pp. 7, 20, 21, 118).

[102] Z. Lin, X. Zhang, and D. Xu. “Automatic Reverse Engineering of Data Structures from Binary Execution.” In: *NDSS*. 2010 (cit. on pp. 3, 25).

[103] C. Linn and S. Debray. “Obfuscation of Executable Code to Improve Resistance to Static Disassembly”. In: CCS. 2003 (cit. on pp. 4, 13).

[104] C.-K. Luk et al. “Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation”. In: PLDI. 2005 (cit. on pp. 100, 102, 145).

[105] N. Lutz. “Towards Revealing Attackers’ Intent by Automatically Decrypting Network Traffic”. MA thesis. ETH Zurich, 2008 (cit. on p. 3).

[106] W. Ma, P. Duan, S. Liu, G. Gu, and J.-C. Liu. “Shadow attacks: automatically evading system-call-behavior based malware detection”. In: *JCV* 8 (1 2012), pp. 1–13 (cit. on p. 126).

[107] A. Martin. *Anti analysis tricks in Trojan-Downloader.Win32.Agent.abti*. 2008. URL: http://www.martinsecurity.net/2008/09/01/anti-analysis-tricks-in-trojan-downloaderwin32agentabti (cit. on p. 15).

[108] M. Matz, J. Hubika, A. Jaeger, and M. Mitchell. *System V Application Binary Interface*. Oct. 2013. URL: http://www.x86-64.org/documentation/abi.pdf (cit. on pp. 69, 78).

[109] McAfee. *McAfee LabsThreats Report*. Aug. 2014. URL: http://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q2-2014.pdf (cit. on pp. 1, 18).

[110] O. Medegan. *Skype Reverse*. 2012. URL: http://www.oklabs.net/category/skype-reverse/ (cit. on p. 18).

[111] J. Meseguer and U. Montanari. “Petri Nets are Monoids”. In: *Information and Computation* 88.2 (Oct. 1990), pp. 105–155 (cit. on p. 129).

[112] E. Metcalf and R. Simon. *Neutralize Advanced Threats*. Tech. rep. 2013 (cit. on p. 2).

[113] R. Milner. "A Theory of Type Polymorphism in Programming". In: *Journal of Computer and System Sciences* 17.3 (1978) (cit. on p. 25).

[114] R. Milner. *Communicating and mobile systems: the π-calculus*. Cambridge University Press, 1999 (cit. on pp. 29, 32, 36, 38, 58, 59, 151, 152).

[115] R. Milner. *Bigraphical reactive systems: basic theory*. Tech. rep. 2001 (cit. on p. 128).

[116] R. Milner. *The Space and Motion of Communicating Agents*. Cambridge University Press, 2009 (cit. on p. 128).

[117] A. Moser, C. Kruegel, and E. Kirda. "Limits of Static Analysis for Malware Detection". In: ACSAC. 2007 (cit. on pp. 4, 13).

[118] A. Moser, C. Krügel, and E. Kirda. "Exploring Multiple Execution Paths for Malware Analysis". In: SSP. 2007 (cit. on pp. 7, 12, 16, 100–102).

[119] C. Nachenberg. "Computer Virus-Antivirus Coevolution". In: *CACM* 40.1 (1997), pp. 46–51 (cit. on p. 6).

[120] S. B. Needleman and C. D. Wunsch. "A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins". In: *Journal of Molecular Biology* 48.3 (Mar. 1970), pp. 443–453 (cit. on p. 19).

[121] N. Nethercote. "Dynamic Binary Analysis and Instrumentation". PhD thesis. University of Cambridge, Nov. 2004 (cit. on p. 101).

[122] N. Nethercote and J. Seward. "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation". In: PLDI. 2007 (cit. on p. 102).

[123] J. Newsome and D. Song. "Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software". In: NDSS. 2005 (cit. on p. 102).

[124] M. Oberhumer, L. Molnar, and J. Reiser. *Ultimate Packer for eXecutables*. Sept. 2013. URL: http://upx.sourceforge.net (cit. on p. 13).

[125] T. Ormandy. *Sophail: A Critical Analysis of Sophos Antivirus*. 2011. URL: https://lock.cmpxchg8b.com/sophail.pdf (cit. on pp. 1, 3).

[126] PandaLabs. *Quarterly Report PandaLabs*. Aug. 2014. URL: http://mediacenter.pandasecurity.com/mediacenter/wp-content/uploads/2014/07/Informe-Trimestral-Q2-2014-EN.pdf (cit. on pp. 1, 18).

[127] R. Péchoux and T. D. Ta. "A Categorical Treatment of Malicious Behavioral Obfuscation". In: *TAMC*. 2014 (cit. on pp. 9, 125).

[128] A. Pnueli. "Linear and Branching Structures in the Semantics and Logics of Reactive Systems". In: ICALP. 1985 (cit. on pp. 37, 153).

[129] M. Ramilli and M. Bishop. "Multi-Stage Delivery of Malware". In: MALWARE. 2010 (cit. on p. 126).

[130] T. Reps, J. Lim, A. Thakur, G. Balakrishnan, and A. Lal. "There's Plenty of Room at the Bottom: Analyzing and Verifying Machine Code". In: CAV. 2010 (cit. on p. 74).

[131] S. S. Response. *W32.Duqu - The precursor to the next Stuxnet*. Nov. 2011. URL: http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_duqu_the_precursor_to_the_next%20_stuxnet.pdf (cit. on p. 1).

[132] S. S. Response. *W32.Stuxnet Dossier*. Feb. 2011. URL: http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf (cit. on p. 1).

[133] S. S. Response. *Regin: Top-tier espionage tool enables stealthy surveillance*. Tech. rep. 2014 (cit. on p. 1).

[134] H. G. Rice. "Classes of Recursively Enumerable Sets and Their Decision Problems". In: *Trans. Amer. Math. Soc.* 74 (1953), pp. 358–366 (cit. on pp. 3, 52).

[135] K. Rieck, P. Trinius, C. Willems, and T. Holz. "Automatic Analysis of Malware Behavior using Machine Learning". In: *Journal of Computer Security* 19.4 (2011) (cit. on p. 4).

[136] K. A. Roundy and B. P. Miller. "Binary-Code Obfuscations in Prevalent Packer Tools". In: *ACM Computing Surveys* 46.1 (2013) (cit. on p. 12).

[137] P. Royal. *Analysis of the Kraken Botnet*. Tech. rep. 2008 (cit. on p. 21).

[138] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee. "PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware". In: ACSAC. 2006 (cit. on p. 14).

[139] T. Sander and C. F. Tschudin. "Protecting Mobile Agents Against Malicious Hosts". In: *Mobile Agents and Security*. 1998 (cit. on p. 5).

[140] D. Sangiorgi. "On the Origins of Bisimulation and Coinduction". In: *TOPLAS* 31.4 (2009) (cit. on pp. 29, 36, 38, 54, 59).

[141] P. Schaff. *Nicene and Post-Nicene Fathers: Series I/Volume I/Confessions/Book XI/Chapter 14*. 2014. URL: http://en.wikisource.org/wiki/Nicene_and_Post-Nicene_Fathers%3a_Series_I/Volume_I/Confessions/Book_XI/Chapter_14 (cit. on p. 153).

[142] E. J. Schwartz, T. Avgerinos, and D. Brumley. "All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution". In: *SSP*. 2010 (cit. on p. 102).

[143] B. Schwarz, S. Debray, and G. Andrews. "Disassembly of Executable Code Revisited". In: WCRE. 2002 (cit. on p. 72).

[144] P. Selinger. *A survey of graphical languages for monoidal categories*. Aug. 2009. URL: http://arxiv.org/abs/0908.3347 (cit. on pp. 129, 130, 143, 144).

[145] M. I. Sharif, A. Lanzi, J. T. Giffin, and W. Lee. "Impeding Malware Analysis Using Conditional Code Obfuscation". In: *NDSS*. 2008 (cit. on p. 5).

[146] T. Shields. *Anti-Debugging - A Developer's View*. Tech. rep. Veracode Inc., 2009 (cit. on p. 14).

[147] M. Sikorski and A. Honig. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. No Starch Press, 2012 (cit. on p. 70).

[148] H. A. Simon. *The Sciences of the Artificial*. 3rd. Cambridge, MA: MIT Press, 1996 (cit. on p. 152).

[149] M. Sipser. *Introduction to the Theory of Computation*. 2nd ed. Cengage Learning, 2005 (cit. on pp. 3, 36).

[150] A. Skaletsky, T. Devor, N. Chachmon, R. Cohn, K. Hazelwood, V. Vladimirov, and M. Bach. "Dynamic Program Analysis of Microsoft Windows Applications". In: ISPASS. 2010 (cit. on pp. 102, 119).

[151] D. Song et al. "BitBlaze: A New Approach to Computer Security via Binary Analysis". In: ICISS. 2008 (cit. on p. 16).

[152] F. Song and T. Touili. "Efficient Malware Detection Using Model-Checking." In: *FM*. 2012 (cit. on p. 4).

[153] M. Sutton, A. Greene, and P. Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley, 2007 (cit. on p. 17).

[154] Symantec. *Symantec Internet Security Threat Report*. 2014. URL: http://www.symantec.com/content/en/us/enterprise/other_resources/b-istr_main_report_v19_21291018.en-us.pdf (cit. on pp. 1, 18).

[155] P. Szor. *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, 2005 (cit. on pp. 3, 14).

[156] P. Ször and P. Ferrie. *Hunting For Metamorphic*. Tech. rep. 2001 (cit. on p. 6).

[157] T. Thanh Dinh. *Trace Transformation Tool*. 2013. URL: http://goo.gl/rqCSQ (cit. on p. 145).

[158] A. Tridgell. *How Samba was written*. Aug. 2003. URL: https://www.samba.org/ftp/tridge/misc/french_cafe.txt (cit. on pp. 11, 18).

[159] D. Wagner and P. Soto. "Mimicry Attacks on Host-Based Intrusion Detection Systems". In: CCS. ACM, 2002, pp. 255–264 (cit. on p. 126).

[160] C. Wang, J. Davidson, J. Hill, and J. Knight. "Protection of Software-based Survivability Mechanisms". In: *DSN*. 2001 (cit. on p. 4).

[161] Z. Wang, X. Jiang, W. Cui, X. Wang, and M. Grace. "ReFormat: Automatic Reverse Engineering of Encrypted Messages". In: ESORICS. 2009 (cit. on pp. 7, 20).

[162] H. Wee. "On Obfuscating Point Functions". In: *STOC*. 2005 (cit. on p. 5).

[163] G. Wondracek, P. M. Comparetti, C. Krügel, and E. Kirda. "Automatic Network Protocol Analysis". In: NDSS. 2008 (cit. on pp. 7, 11, 19, 152).

[164] W. Yan, Z. Zhang, and N. Ansari. "Revealing Packed Malware". In: *IEEE Security and Privacy* 6.5 (2008), pp. 65–69 (cit. on p. 4).

[165] A. Young and M. Yung. *Malicious Cryptography: Exposing Cryptovirology*. John Wiley and Sons, 2004 (cit. on pp. 2, 5, 152).

[166] ZeuS source code. 2014. URL: https://github.com/Visgean/Zeus.git (cit. on pp. 119, 122).

// Résumé

La thèse contients de deux parties principales: la première partie est consacrée à l'extraction du format des messages, la deuxième partie est consacrée à l'obfuscation des comportements des malwares et la détection. Pour la première partie, nous considérons deux problèmes: "la couverture des codes" et "l'extraction du format de messages". Pour la couverture des codes, nous proposons une nouvelle méthode basée sur le "tainting intelligent" et sur l'exécution inversée. Pour l'extraction du format des messages, nous proposons une nouvelle méthode basée sur la classification de messages en utilisant des traces d'exécution.

Pour la deuxième partie, les comportements des codes malveillants sont formalisés par un modèle abstraire pour la communication entre le programme et le système d'exploitation. Dans ce modèle, les comportements du programme sont des appels systèmes. Étant donné les comportements d'un programme bénin, nous montrons de façon constructive qu'il existe plusieurs programmes malveillants ayant également ces comportements. En conséquence, aucun détecteur comportemental n'est capable de détecter ces programmes malveillants.

Mots-clés: détection/obfuscation comportementals, codes malveillants, extraction du format des messages.

// Abstract

The thesis consists in two principal parts: the first one discusses the message format extraction and the second one discusses the behavioral obfuscation of malwares and the detection. In the first part, we study the problem of "binary code coverage" and "input message format extraction". For the first problem, we propose a new technique based on "smart" dynamic tainting analysis and reverse execution. For the second one, we propose a new method using an idea of classifying input message values by the corresponding execution traces received by executing the program with these input values.

In the second part, we propose an abstract model for system calls interactions between malwares and the operating system at a host. We show that, in many cases, the behaviors of a malicious program can imitate ones of a benign program, and in these cases a behavioral detector cannot distinguish between the two programs.

Keywords: behavioral detection/obfuscation, malicious codes, message format extraction.