Protocol Inference from program executable using symbolic execution and automata learning

Ron Marcovich

Protocol Inference from program executable using symbolic execution and automata learning

Research Thesis

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science

Ron Marcovich

Submitted to the Senate
of the Technion — Israel Institute of Technology
Tamuz 5782 Haifa July 2022

This research was carried out under the supervision of Prof. Orna Grumberg and Dr. Gabi Nakibly, in the Faculty of Computer Science.

Some results in this thesis have been published as articles by the author and research collaborators in conferences and journals during the course of the author's Master's research period, the most up-to-date versions of which being:

Ron Marcovich and Gabi Nakibly. Automatic protocol reverse engineering. In *Black Hat USA*, 2022.

Acknowledgements

I would like to thank my advisors, Orna and Gabi, for their time, support and their very professional and helpful advises.

I would also like to thank my dear family for having my back during my whole studies at the Technion, and particularly during my master degree.

Moreover, I wish to thank my dear friends, who are always by my side to support, consult and guide.

I wish to send my love and appreciation to my love, Liron.

The generous financial help of the Technion is gratefully acknowledged.

Contents

List of Figures

Abstract **1**

1 Introduction **3**

1.1 Problem Setting . . . 3

1.2 Motivation . . . 4

1.3 Previous Works . . . 6

2 Preliminaries **9**

2.1 Deterministic Finite Automaton (DFA) . . . 9

2.2 Automata Learning . . . 10

2.2.1 Growing Alphabet . . . 10

2.2.2 Equivalence query approximations . . . 10

2.3 Model Checking . . . 11

2.4 Symbolic Execution . . . 11

2.4.1 Example . . . 12

3 Problem Definition **15**

3.1 Definitions and Notations . . . 15

3.2 Assumptions . . . 16

4 Learning a DFA of a Protocol (Exact Version) **19**

4.1 Modified Membership Queries . . . 19

4.2 Modified Equivalence Queries . . . 19

4.3 Handling a growing alphabet . . . 20

4.4 Initialization and Output . . . 20

4.5 Correctness . . . 20

5 Learning a DFA of a Protocol (Approximation) **21**

5.1 Characterizing Message Types . . . 21

5.1.1 Extracting predicate to represent a set of messages . . . 22

5.1.2 Example . . . 23

5.2 Handling Alphabet Changes . . . 23
5.2.1 Insertion of message type candidate . . . 23
5.2.2 Insertion of multiple message type candidates . . . 24
5.2.3 Example of alphabet changes . . . 25
5.3 Equivalence Oracle . . . 25
5.4 Modified Symbolic Execution . . . 26
5.4.1 Implementing assumptions and assertions in symbolic execution . 27
5.4.2 Hooking calls to send and receive procedures . . . 27
5.4.3 Extending symbolic states to track query state . . . 28
5.5 Membership Oracle . . . 29
5.5.1 Monitoring phase . . . 30
5.5.2 Probing phase . . . 32

6 Optimizations 37
6.1 Prefix Closed Property . . . 37
6.2 Fast Equivalence Queries . . . 37
6.3 Execution Cache . . . 38
6.3.1 Example . . . 38

7 Implementation and Results 41
7.1 Implementation . . . 41
7.1.1 LearnLib . . . 41
7.1.2 angr . . . 42
7.1.3 Learning Client (Learner) . . . 42
7.1.4 Symbolic Execution Server (Teacher) . . . 43
7.2 SMTP Client Experiment . . . 43
7.3 Gh0st RAT Experiment . . . 44

8 Conclusions 49
8.1 Method Validation . . . 49
8.2 Future Works . . . 50

Hebrew Abstract i

List of Figures

1.1 An example of a C&C session between a malware and a server. The attacker sends through the server commands that the malware follows. . . 5

1.2 Demonstration of vulnerable behaviours of a protocol. Such behaviours can be exploited by attackers to remote code execution (RCE). 6

2.1 A DFA is often shown as a diagram, describing its states (nodes), initial state, transitions (edges) and accepting states (double circled nodes). . . 9

5.1 Illustration of our algorithm 21

5.2 Colliding predicates 25

5.3 Illustration of symbolic execution during membership query 29

7.1 Illustration of our implementation and the interactions between its components 42

7.2 Illustration of synchronous vs. asynchronous communication 45

7.3 The branch in Gh0st RAT C&C protocol that handles webcam streaming. The letter in the square brackets indicates whether the message is sent or received. 46

7.4 Gh0st RAT full state machine learnt by our method 48

Abstract

Many computer programs these days make use of the internet network in order to send data between machines around the world. Such programs make use of network protocols: a set of rules defining how a communicating peer should or could respond while communicating with other machines. In the common case, a protocol is being transformed into a programming language, and later to a binary code, by a programmer. Not all protocols are publicly documented for commercial or security reasons.

Protocol Inference is the process of gaining information about a protocol from a binary code that implements it. This process is useful in cases such as extraction of the command and control (C&C) protocol of a malware, uncovering security vulnerabilities in a network protocol implementation or verifying conformance to the protocol's standard. Protocol inference usually involves time-consuming work to manually reverse engineer the binary code.

This work introduces a novel method to automatically infer state machine (i.e, a diagram) of a network protocol and its message types directly from a binary code that implements it. To the best of our knowledge, this is the first method to achieve this solely based on a binary code of a single side of the protocol. We assume that: the binary code that implements the protocol is available, the protocol being learned can be described as a DFA and that the length of a message is limited. In contrast, we do not assume any of the following: access to another side of the protocol, access to captures of the protocol's traffic, and prior knowledge of message formats.

We present a modified automata learning algorithm that is suitable for learning a DFA of a protocol. In particular, this algorithm handles the fact that the alphabet (i.e, the message types of the protocol) is not known in advance and is uncovered during the learning. The modified algorithm assumes an oracle that is capable of answering queries about the protocol to be learned. Because it is infeasible to develop such oracle in our setting, we propose an approximation for our modified version. The approximation leverages symbolic execution to uncover the message types of the protocol and to test if a sequence of messages is valid according to the protocol.

We validate the proposed method by inferring real-world protocols including the C&C protocol of Gh0st RAT, a well-known malware. Our experiments show that the method correctly infers the state machine of the protocol, as long as the symbolic execution engine it relies on is able to execute the given binary.

Chapter 1

Introduction

1.1 Problem Setting

Many computer programs these days make use of the internet network in order to send data between machines around the world. The basic unit of information exchanged between programs is called a *Message*. Messages are exchanged between parties in a predefined and application-specific manner called *Protocol*: a set of rules defining how a communicating peer should or could respond upon receiving a message and how messages are formatted. When two parties choose to communicate, they form a *Session* and agree on the protocol they will follow.

Not all protocols are publicly documented for security and/or commercial reasons. Software developers may choose not to share a specification of a protocol in order to protect their intellectual property and prevent others from developing competing software. They may also choose to hide the specification to prevent malicious activities. With access to a specification of a protocol, it is easier to exploit and abuse a software that uses it. A protocol is usually defined in a formal document that acts as an ”agreement” between all the programs that use the protocol. This document describes the messages a peer should accept, as well as the expected behaviours and responses that programs running the protocol should follow in order to ensure correct communication with other machines running the same protocol. In the absence of publicly available such document, the protocol is either documented and kept confidential, or not documented at all - in case it is used by a single developer.

A computer program, in particular a computer program that communicates over the network, is a sequence of machine instructions that the computer may execute upon demand. These instructions implement the logic of the program, and their execution fulfills the purpose of the program. Each instruction is encoded in binary form, and therefore the set of machine instructions is often called *binary code*. It is considered difficult for humans to read and understand binary code.

A developer of a computer program that communicates with the network should *implement the protocol*, i.e make the logic of the program follow the protocol. The

programmer develops the program using a human-readable programming language, that is later translated to a binary code. Once the program is published, its binary code is distributed to the users, causing their computers to run the logic of the protocol. It is possible and somewhat common that multiple implementations of a protocol exist and are developed by different developers. In such case, the protocol ensures that computers running different implementations can communicate with each other correctly.

A common communication pattern that network protocols use is *the client-server model*. In this model, multiple computers called *clients*, communicate with a single computer called *server*. During a session, the server provides services or content to a client. A forgone conclusion is that the server runs a different program than the clients, even though they all communicate using the same protocol. According to this definition, it is possible for a client A to send (by mistake) a message to another client B, but their session will fail because client A will not receive from client B the messages that it expects according to the protocol. In reality, the implementation of the network makes it impossible to form a session between two clients.

As comes up from this setting, the common case is that a protocol is being transformed into a programming language, and later to a machine code, by the programmer. The opposite process, of gaining information about the protocol from the machine code that implements it, is called *Protocol Inference*. Our goal in this work is to infer a state machine of a network protocol (i.e, a diagram that describes it) given a binary code that implements the protocol. Our method is fully automatic except one requirement: we require the user of our method to provide addresses of methods that send or receive data from the network. We later discuss this requirement and explain why it is fair. We introduce a novel method to classify protocol messages into types by inferring predicates of message groups. We also developed a technique to infer formats of incoming messages while we do not assume to have the binary code of other implementations, like a server, or access to network traffic recordings. We infer the protocol just as it is reflected in the given binary. In the case of a client-server protocol, the protocol is mostly entirely reflected in the binary code of the client and we are going to exhaust that in our work.

1.2 Motivation

Protocol inference is mostly relevant in the field of cyber-security. Researchers that focus on defensive cyber activities, often discover malicious programs they want to investigate during their work. A *Malware* is an example for a malicious computer program created to harm the computer that it runs on, as well as to spy on the user of the computer. That is the reason malwares run secretly and without the consent of the user. Advanced malwares implement a *command and control (C&C) protocol* and form a session with the attacker's server. During that session, they receive commands from the server. An example for a C&C session between a server and a malware is shown in

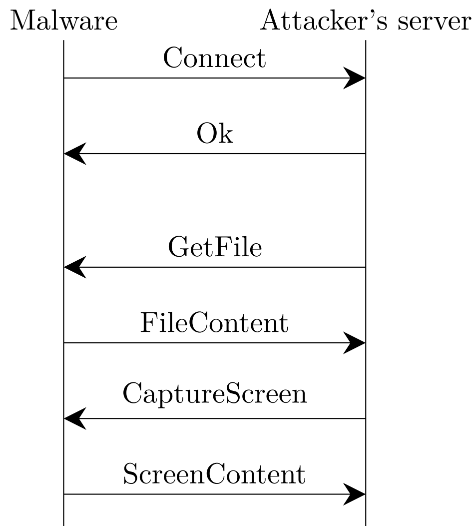


Figure 1.1: An example of a C&C session between a malware and a server. The attacker sends through the server commands that the malware follows.

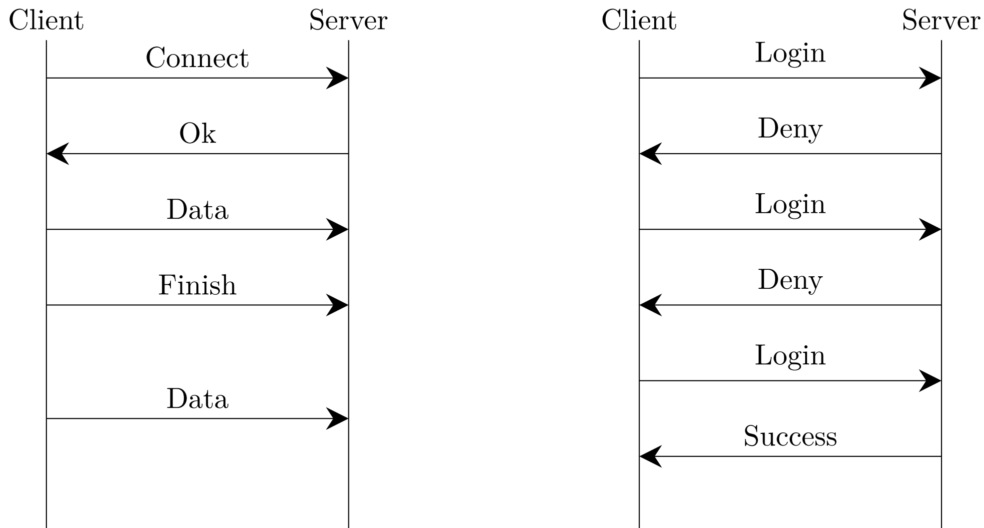
Figure 1.1.

Some commands instruct the malware to collect personal information and send it back to the server, some may write a file to the disk of the victim and others may change the configuration of his machine. It is also common for a malware to accept a suicide command, to which it responds by terminating itself and removing its traces.

Classical methods to infer a C&C protocol of a malware involves reverse engineering of its binary code. This task, however, is quite hard: reverse engineering using static methods is difficult and time consuming. This is, in part, because the malware code contains (possibly a lot of) logic that is not related to their C&C protocol and might interfere with the protocol inference task. An automated protocol inference method can ease the difficulty of this process by obviating the need to dive into the binary code of the malware.

Besides investigating malwares, the field of cyber security mainly revolves around *security vulnerabilities*. Those are bugs, mistakes in programs, that may allow a skilled attacker to alter the behaviour of the program. Security vulnerabilities, when they exist in programs distributed to users, expose computers to cyber attacks and make it possible for attackers to take control over a computer. Security vulnerabilities may appear in an implementation of a network protocol. In such a case, they may allow attackers to remotely abuse the protocol and trigger unexpected behaviour in the target machine. For example, a vulnerability in an implementation of a protocol in a server program may allow unauthorized user to access confidential data of other clients served by the same server. Therefore, it is important to detect and fix these vulnerabilities. An example for a vulnerable behaviour of a protocol is shown in Figure 1.2a.

By inferring the protocol of a program, a researcher may compare the implementation of the protocol to other implementations and find anomalies in the investigated



(a) An example for a vulnerable behaviour in a protocol; The server accepts messages after the session terminates.

(b) An example for a backdoor in a protocol. This backdoor bypasses the authentication after three continuous attempts.

Figure 1.2: Demonstration of vulnerable behaviours of a protocol. Such behaviours can be exploited by attackers to remote code execution (RCE).

program's protocol that can lead to security issues. A researcher would like to do that even if he only has access to the binary of the program, in order to rule out the possibility that the program was tampered on purpose during its development or distribution. By tampering the program's protocol prior to its distribution, an attacker can inject a "backdoor" - a security vulnerability created by the attacker that he will use later for his malicious activities. That is why it can be useful for security experts to verify the behaviour of network programs before they are incorporated in servers and critical systems. An example for a vulnerable behaviour inserted by an attacker is shown in Figure 1.2b.

Deploying automated protocol inference methods can assist security researches by automating a very technical and difficult process and help them in their work to reveal malicious activities and understand illegal activities.

1.3 Previous Works

There are several published works that deal with the problem of learning information about a network protocol. The approach presented in [CKW07] uses recorded network traffic as input (and therefore assuming the user is able to record real sessions of the protocol). This method analyzes the given traffic and uses heuristics to extract different protocol fields.

The approach of [CKW07], however, is not always able to extract all the required information about the protocol using the recorded traffic only. Therefore, a number of methods [CYLS07] [CS13] [WCKK08] [CPC+08] were introduced that combine the

recorded traffic with execution traces of the server. This allows them to learn more details about the structure of messages, and even gain some insights about the semantics of messages and message fields.

All of these works do not deal with the problem of learning the protocol state machine. This was the motivation for the work of Comparetti et al. which introduced Prospex [CWKK09]. Their work integrates methods from previous works to gain information about the messages in the protocol but extended them in two directions: First, they developed a mechanism to identify messages of the same type. They use this information to partition messages with similar role in the protocol into clusters. The second extension is a method to infer the state machine of the protocol.

Prospex method, like all the mentioned methods, requires captures of traffic sessions between a client and a server. Moreover, the result of these methods depends on the amount and the variety of the recorded sessions. In some cases it is not possible to record real sessions, and even when it is, we are not guaranteed that the capturing exposes the protocol's full state machine. This can occur with malwares, for example, where the malware is controlled by a C&C server and we are not able to manually control its interaction, and therefore unable to record a variety of sessions. Also, we have no information about the C&C server or the abilty to track its execution, making the task of inferring the C&C protocol more difficult. Retrieving execution traces of the client can also be difficult due to anti debugging techniques that might prevent controlled execution.

The work by Lim et al. [LRL06] suggests a method to extract output formats of executable, including output network formats. They construct a Hierarchical Finite State Machine (HFSM) that over-approximates the output data format. They use Value-Set Analysis (VSA) and Aggregate Structure Identification (ASI) to annotate HFSMs with information that partially characterizes some of the output data values.

Another important contribution is the work of Cho et al. [CBSS10] that introduced a method for on-line inference of botnet C&C protocol, using active instances of it. They chose to represent a protocol as a Mealy machine and used L* extension by Shahbaz et al. [SG09] for learning Mealy Machines. They actively query the control server (which they call master) responses for a sequences of messages. They also introduce caching and prediction optimizations to L* [Ang87] in order to reduce the amount of membership queries sent to the server. Their work, as an on-line method, assumes the server is available and answers appropriately. They also assume known message formats by using previous work [CYLS07].

The work of Peled et al. [PVY02] describes several methods to check specification on a black box machine with unknown internal structure. One of their methods uses learning with L* [Ang87] to learn the internal implementation of the black box.

In our case, we cannot establish a connection with the server since it is either offline by the time of the analysis or we simply do not want to reveal the investigation activity. Therefore, there is a need for a method to infer the protocol of an executable without

having to record real sessions and without any information about the server. Such a method has to leverage all the information available in the executable itself (the binary code) in order to infer the protocol state machine.

Another related work is the work of Alur et al. [ACMN05] introducing a method to infer a Java class specification (order of method calls) using model checking and L*, which is similar to what we apply in our work. In their case, however, the alphabet (the set of methods) is known in advance.

A series of previous works [EGP08] [PGB+08] [CGP03] utilize L* for the purpose of model checking and suggest learning-based algorithms to automate assumption generation in assume-guarantee verification. Gheorghiu et al. [GGP07] and Chaki et al. [CS07] are the first works to introduce alphabet refinement to this process.

Recent work by Jeppu et al. [JMK22] presents a new approach to generate abstractions of systems from execution traces. They use model checking to verify their abstraction. In case of mismatch, the abstraction is refined according to traces found by model checking of the target system and that contradict the current abstraction.

Another work by Cho et al. called MACE [CBP+11] presents a method to learn a state machine of a server using L* and symbolic execution. They also use L* extension for inferring Mealy state machines [SG09]. They use symbolic execution to uncover messages that the client may send, and then send them to the server in order to learn its response to different message sequences. This work is similar to ours: we also use L* and symbolic execution. There are, however, two main differences: First, they assume a known abstraction function is available that can extract the message type out of the server's response. Second, they assume a running server is available, that can answer the client's requests. We do not have these assumptions.

Chapter 2

Preliminaries

2.1 Deterministic Finite Automaton (DFA)

A deterministic finite automaton (DFA) M is a five-tuple, $(Q, \Sigma, \delta, q_0, F)$, all of them nonempty, whereas: Q is a finite set of states, Σ is a finite set of input symbols (alphabet), $\delta : Q \times \Sigma \to Q$ is a transition function, $q_0 \in Q$ is an initial state and $F \subseteq Q$ is a set of accepting states.

Let $\Sigma^* = \{\sigma_1...\sigma_n \mid \sigma_i \in \Sigma, n \geq 0\}$ the set of all finite strings over the alphabet Σ. Given $w \in \Sigma^*$, we say that M accepts w if and only if there exist $r_0, ..., r_n$ such that:

1. $\forall 0 \leq i \leq n, \quad r_i \in Q$
2. $r_0 = q_0$
3. $\forall 0 \leq i \leq n-1, \quad r_{i+1} = \delta(r_i, \sigma_{i+1})$
4. $r_n \in F$

We define $L(M) = \{w \in \Sigma^* \mid M \text{ accepts } w\}$ as the language of M. A set of words $L \subseteq \Sigma^*$ is a **regular language** iff there exists a DFA M such that $L = L(M)$.

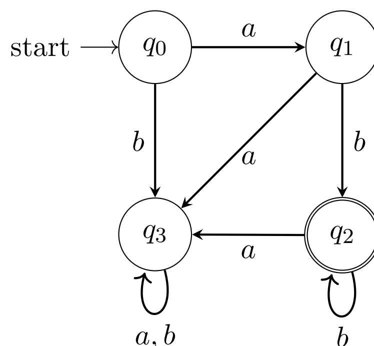


Figure 2.1: A DFA is often shown as a diagram, describing its states (nodes), initial state, transitions (edges) and accepting states (double circled nodes).

2.2 Automata Learning

Automata learning relates to the general problem of identifying an unknown regular language L by learning its DFA. In 1987 Angluin introduced the L* algorithm [Ang87]. The algorithm assumes a *minimally adequate Teacher*, which is an oracle that can answer two types of queries about the regular language L: ***Membership query*** and ***Equivalence query***. In a Membership query, the Teacher should indicate whether a given word w is in L or not. In an Equivalence query, the Teacher should indicate, given a conjectured DFA M', whether $L(M') = L$, and provide a counterexample otherwise (a word in the symmetric difference of L and $L(M')$).

In its internal data, the L* algorithm saves a description of the currently learned DFA in a structure called observation table. The observation table is updated during the learning process according to the answers of the Teacher.

2.2.1 Growing Alphabet

Some use cases of L* require the capability of adding new alphabet symbols during the algorithm. Previous works [GGP07] [CS07] have already introduced the need to modify the alphabet of an L* instance. When a counterexample is returned by the teacher and reveal new alphabet symbols, they refine the alphabet and then L* is restarted from the beginning with the refined alphabet. However, it is not always required to restart L* and its internal implementation may support growing alphabets already. In the implementation of L* that we use [IHS15] the need to restart L* is obviated by updating the necessary parts of the observation table on the fly whenever a new alphabet symbol is discovered. Then, a set of membership queries is sent in order to learn the behaviour that the new alphabet symbol introduces and the currently learned DFA is updated accordingly.

2.2.2 Equivalence query approximations

Equivalence queries are difficult to answer in real world black-box scenarios where the DFA for L does not exist at all. Therefore, Angluin proposed a method to test equivalence queries using random sampling oracle, assuring probabilistic approximation of the real DFA. The sampling oracle tries to generate words w in the symmetric difference of L and $L(M')$ and sends them as membership queries. In case of disagreement between the membership answer for w and whether $w \in L(M')$, w is considered as a counterexample of the approximated equivalence query. This method is probabilistic and highly depends on the algorithm of the sampling oracle. Various test suites generation methods like W-Method [Cho78] and Wp-Method [FvBK+91] have been suggested, and may also be used to approximate equivalence queries by running test suites against the suggested DFA.

2.3 Model Checking

One of the most common methods for formal verification is model checking [CGK+18]. Model checking is a procedure that given a program P and a specification φ checks whether P satisfies φ. If the answer is negative, then the model checking provides a counterexample for a program run that does not satisfy the specification.

Bounded Model Checking (BMC) [BCCZ99] is a model checking technique based on SAT solvers. The basic idea of BMC is to consider counterexamples of a particular length k and generate a propositional formula that is satisfiable iff such a counterexample exists. In particular, it means that counterexamples longer than k will not be discovered.

CBMC [CKL04] is a tool for the formal verification of ANSI-C programs using Bounded Model Checking (BMC). The tool supports almost all ANSI-C language features. CBMC translates C programs to a set of constraints in the form of Single Static Assignment (SSA). CBMC uses assertions in the code as a mean of specification and allows to insert assumptions in order to allow ignoring some irrelevant execution paths.

Due to lack of tools to apply model checking on binary code, we chose to simulate methods of model checking with symbolic execution. Therefore we do not use model checking in our work but apply concepts of model checking in our algorithm. In case that a model checking tool for binary code is presented by a future work, it can be easily integrated in our algorithm.

2.4 Symbolic Execution

Symbolic execution is a static method of analyzing a program. During the analysis it determines what constraints the program's input must satisfy in order to execute each execution path in the program. This is done by following the program's code assuming symbolic values for inputs rather than getting concrete values for the inputs from the user (or the network, in our case). The symbolic execution engine develops expressions in terms of symbolic variables for values that occur in the program, as well as constraints in terms of those symbolic variables for each possible outcome of each conditional branch. To perform symbolic execution over a set of symbolic variables V, a symbolic state is defined to contain the current symbolic values for each variable in the program, as well as constraints C in terms of V that should hold in order to reach that state in a concrete execution. A symbolic state contains a list of its predecessors as well, so it is not possible for a symbolic state to represent two different execution paths. Moreover, a symbolic state can represent an execution during which the program failed and terminated. We call such a symbolic state an abort state.

Symbolic execution begins with a single initial state located at the entry point of the program, with initial symbolic or concrete values for the variables. The execution happens by stepping the set of states and generating new descendant states. Stepping a

single state may result with multiple new descendant states, if, for example, the parent state corresponds to a conditional branch *cond*. In such a case, two descendant states must be created to represent *cond* evaluation to either true or false. This is necessary because *cond* determines the next instruction that executes after the branch. The successor along the true branch will include a new constraint, obtained by replacing the variables of *cond* with symbolic expressions representing the current values of these variables. Similarly, the successor along the false branch will include a new constraint obtained by replacing the variables of $\neg cond$. Before stepping a state, the symbolic execution may verify that the current constraints of the state are satisfiable, meaning that for at least one input value, the state is reachable. This verification is done in order to discard states that represent infeasible paths, representing impossible executions. An abort state will also be discarded and will not be further stepped.

As mentioned in Section 2.3, we use symbolic execution to simulate assumptions and assertions of model checking, due to the lack of tools to apply model checking on binary code. We further explain how we implement assumptions and assertions in Section 5.4.

2.4.1 Example

Consider the pseudo-code in Listing 2.1, and let $V = \{X, Y\}$ be the set of symbolic variables. Let the initial values for the variables be $x = X, y = Y$.

Listing 2.1: Example of symbolic execution

```
1:  x = y
2:  y = y + 1
3:  if (x > y) {
4:        ...
5:  } else {
6:        ...
7:  }
```

Symbolic execution of this code snippet begins with entry state s_1 at line 1, with variable values $\{x = X, y = Y\}$ and $C_1 = \emptyset$. Stepping s_1 results with a new state s_2 at line 2, with $\{x = Y, y = Y\}$ and $C_2 = \emptyset$. Stepping s_2 results with s_3 at line 3 with $\{x = Y, y = Y + 1\}$ and $C_2 = \emptyset$. Stepping s_3 results with two descendant states s_4 and s_5 with $\{x = Y, y = Y + 1\}$ for both of them. Since s_3 is a conditional branch, s_4 represents the true branch, at line 4, and has $C_4 = \{Y > Y + 1\}$, while s_5 represents the false branch, at line 6, and has $C_5 = \{Y \leq Y + 1\}$. At this stage, the set of states contains s_4 and s_5. When symbolic execution tries to step s_4, it finds that C_4 is unsatisfiable, since there is no Y such that $Y > Y + 1$. Therefore, s_4 is discarded and will not be further stepped since it represents an execution path that is not possible with concrete values. On the other hand, the symbolic execution finds that C_5 for s_5

is satisfiable (for example, with the assignment $Y = 1, X = 1$) and therefore steps s_5 to the next instruction.

Chapter 3

Problem Definition

3.1 Definitions and Notations

Our goal, given a binary code of a program, is finding a DFA that accepts a language L, where each word in L matches a sequence of message types received and sent by the program. We say that this DFA is the state machine of the protocol implemented by the program.

We refer to the binary code of the program as "the program". That is, every time we use the term "program" we refer to the binary code supplied as input to our method. We say that a concrete run of the program is valid if and only if the program does not terminate due to an error during the run.

We denote by S and R the finite set of messages that may be sent and received by the program, respectively. S and R are disjoint. S and R are finite because messages are limited in length (See Assumption 1 below).

A session is a sequence of messages $m_1 \dots m_k$ such that $\forall 1 \leq i \leq k, m_i \in S \cup R$. We say that a session $m_1 \dots m_k$ is valid for the program if and only if there exists a valid run of the program along which the same sequence of messages are sent or received in exactly the same order as in $m_1 \dots m_k$. Formally, a session $m_1 \dots m_k$ is valid for the program if and only if:

- For $k = 0$, an empty session is always valid.
- For $k = 1$:
  - If $m_1 \in S$, there exists a valid run in which m_1 is sent by the program and no message is sent or received prior to m_1.
  - If $m_1 \in R$, there exists a valid run in which m_1 is received by the program and no message is sent or received prior to m_1.
- For $k > 1$:
  - $m_1 \dots m_{k-1}$ is a valid session

– If $m_k \in S$, there exists a valid run of the session $m_1 \dots m_{k-1}$ in which the program sends m_k after sending or receiving m_{k-1} and no message is sent or received between m_{k-1} and m_k.

– If $m_k \in R$, there exists a valid run of the session $m_1 \dots m_{k-1}$ in which the program receives m_k after sending or receiving m_{k-1} and no message is sent or received between m_{k-1} and m_k.

Valid sessions are prefix-closed, meaning that if $m_1 \dots m_k$ is a valid session then $\forall 1 \leq l \leq k-1$, $m_1 \dots m_l$ is also a valid session.

Messages are partitioned into subsets of message types according to their semantics in the protocol and their effect on the protocol state machine. Let T_D be a partition of messages of D where $D \in \{R, S\}$. Given a message $m \in D$, we denote by $type(m)$ the pair $\langle D, t \rangle$ such that $t \in T_D$ and $m \in t$. We call such a pair $\langle D, t \rangle$ a **Message Type**. t is finite because both S and R are finite.

The concept of message types derives from real-world protocols, where messages are divided to types. Every message type in the protocol has different semantics. For example, different message types have different effect on the state of the protocol and trigger a different response by the other side of the communication. Messages in real-world protocols usually begin with a header - a fixed length sequence of bytes specifying meta-data about the message, including its type, timestamps, message length and more. The receiving side parses the header in early stages of the processing and executes the appropriate procedure to handle the specific type of message.

We define the alphabet of the protocol as a finite set of pairs:

$$\Sigma_L = \{\langle D, t \rangle \mid t \in T_D, D \in \{R, S\}\}$$

Given a session $m_1 \dots m_k$ such that $\forall 1 \leq i \leq k, m_i \in S \cup R$, we define $\Theta\colon (S \cup R)^* \to \Sigma_L^*$:

$$\Theta(m_1 \dots m_k) = type(m_1) \dots type(m_k)$$

We abstract all valid sessions to a regular language L over the alphabet Σ_L:

$$L = \{\Theta(m_1 \dots m_k) \mid m_1 \dots m_k \text{ is a valid session}\}$$

Note that L is prefix closed because valid sessions are prefix closed. Note that R and S, as well as their partitions T_S and T_R, are unknown in advance, hence the alphabet Σ of the DFA in unknown in advance. It is the task of our method to uncover Σ_L as it determines the DFA.

3.2 Assumptions

We assume the following about the input to our method:

1. **Binary code of the program is available:** a binary code that implements the protocol to be inferred is available.

2. **Message Length is limited:** there exists N such that no message in the protocol to be inferred is longer than N bytes. This is a reasonable assumption since concrete messages must be finite. In practice, our method allow a message to be longer than N bytes, as long as the first N bytes may allow to infer its message type. This assumption is required since symbolic lengths are computationally difficult to infer.

3. **Protocol Regularity:** the protocol can be modeled as a DFA (See Section 2.1) over Σ_L in terms of message types allowed from each state. Formally, L is regular language. If L is not regular, our algorithm will fail or will never complete the inference.

In contrast, we do not assume:

1. Access to the binary code of the program that implements the other side of the protocol. (For example, a server)

2. Access to network traffic recordings that contain valid sessions.

3. Access to an online instance of the server, and we assume it is impossible to run the binary code and form a real session. Our method entirely relies on static analysis of the binary code.

4. Prior knowledge about sent or received messages' formats and the partition to message types. For example, most protocols include a header in messages, with information regarding the type of the message or its length. This information must be located at a fixed offsets in the message because every implementation of the protocol must be able to extract this information from a message at a very early stage of the processing. We do not assume to know these offsets and our method intends to detect them automatically.

Chapter 4

Learning a DFA of a Protocol (Exact Version)

In this chapter we propose a suitable method for learning a DFA of a protocol. The learning of the protocol's DFA is based on a modified version of the L* algorithm [Ang87]. We modify the algorithm's queries in order to uncover the alphabet Σ, that is, the message types of the protocol. Initially, Σ is $\emptyset$.

For ease of exposition, in this chapter we assume a Teacher that is capable of answering the queries we present. However, this assumption is unrealistic when both the state machine and the alphabet Σ_L are unknown. In Chapter 5 we propose suitable approximations for the Learner and the Teacher.

4.1 Modified Membership Queries

The classical membership query returns $True$ or $False$, indicating if a given w is in L or not. We modify it as follows: If $w \in L$, $True$ is returned together with a set $Cont(w)$ of message types $\langle D, t \rangle$ such that $w \cdot \langle D, t \rangle \in L$. If $w \notin L$, $False$ is returned. The set $Cont(w)$ may reveal new alphabet symbols.

4.2 Modified Equivalence Queries

In a classical equivalence query the Learner provides a conjectured DFA M over alphabet $\Sigma_M = \Sigma$. $True$ is returned if $L = L(M)$. Otherwise, $False$ is returned and a counterexample w in the symmetric difference of L and $L(M)$ is returned as well. We modify it as follows: $False$ is returned if there exist $w \in \Sigma_M^*$ for which one of the following hold: (1) w is in the symmetric difference of L and $L(M)$; (2) A set $Miss(M) \neq \emptyset$ exists such that for all $\sigma \in Miss(M)$, $\sigma \notin \Sigma_M$ but $w \cdot \sigma \in L$. In the former case, w is returned as a counterexample. In the latter case, every $w \cdot \sigma$ is considered a counterexample. $True$ is returned if for all $w \in \Sigma_M^*$, neither (1) nor (2) hold.

4.3 Handling a growing alphabet

Given a set of message types $C = Cont(w)$ or $C = Miss(M)$ output by a query, Σ is set to $\Sigma \cup C$. If Σ changes during this assignment, then we say that the alphabet grows. To handle a growing alphabet we use a modified L* algorithm presented in [IHS15]. In a nutshell, the modified algorithm updates the observation table to handle the new alphabet symbols while the general learning cycle is kept similar to the classical L* algorithm.

4.4 Initialization and Output

The Learner starts with $\Sigma = \emptyset$. The first query of the Learner is $w = \varepsilon$. Note that the answer to this query is $True$ since an empty session is valid. $Cont(\varepsilon)$ is then added to Σ. The Learner continues to utilize queries according to the L* algorithm and extends Σ and the learnt DFA according to the queries' answers. The algorithm terminates when an equivalence query returns $True$. The algorithm outputs the learnt DFA that represents the protocol's state machine and Σ that represents the protocol's message types.

4.5 Correctness

The correctness of the algorithm is based on the modified definition of equivalence queries and correctness of the classical L* algorithm.

Theorem 4.1. *The modified Learner terminates with $L(M) = L$ and $\Sigma_M = \Sigma_L$.*

Proof. The Learner terminates when it gets $True$ as an answer from the Teacher on an equivalence query. In this case, there is no $w \in \Sigma_M^*$, which is in the symmetric difference of L and $L(M)$. Thus, $L = L(M)$. Also, there is no $w \in \Sigma_M^*$, such that $\sigma \notin \Sigma_M$ but $w \cdot \sigma \in L$. This means that there is no (reachable) message type $\sigma \in \Sigma_L$ that has not been revealed already by our modified Learner. Consequently, $\Sigma = \Sigma_L$, as required. ∎

Chapter 5

Learning a DFA of a Protocol (Approximation)

The learning algorithm presented in Chapter 4 assumes a Teacher capable of answering the queries as we described. This assumption, however, is not feasible in real world analysis, where both the language of the protocol and its message types are unknown. Therefore, we opt to propose an approximated Teacher. The Teacher approximates the queries described in Chapter 4. As a result, some modifications are required in the Learner, in order to cope with the fact that the Teacher is approximated.

This chapter details how the answers to the queries of the modified L* algorithm presented in Chapter 4 are approximated. The components of the method and their interactions are presented in Figure 5.1.

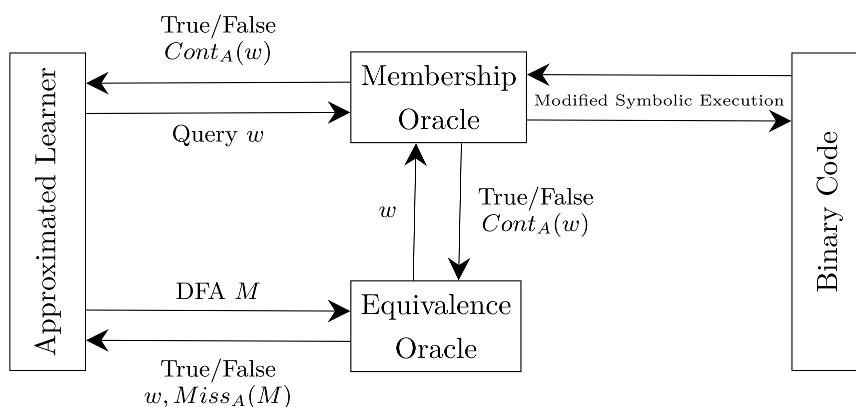


Figure 5.1: Illustration of our algorithm

5.1 Characterizing Message Types

Recall that a message type is a pair $\langle D, t\rangle$ where t is a set of finitely many messages. Our algorithm requires a method to identify message types without having to store the entire set t. In most real-world protocols, message types are distinguishable by

one or more fields in the message (a sequence of bytes) that identify the type of the message. We do not assume to know in advance how to identify the type of a message and therefore we present a method to describe a set of messages based on their common format. We represent a set t using a predicate $\mathcal{P}$ describing the format of that message type. This predicate represents all messages of the same type. Hence, we represent a message type by a pair $\langle D, \mathcal{P} \rangle$. We emphasize that the sets R and S are unknown and are never identified by our algorithm. Instead, our algorithm infers predicates that describe the format of message types.

Given a set $x \subseteq D$ where $D = R$ or $D = S$, we associate x with a predicate $\mathcal{P}_x$. $\mathcal{P}_x$ is a predicate over variables $\{B_0, \ldots, B_{N-1}\}$, where B_i represents the ith byte of the message for $0 \leq i \leq N-1$. Recall that N is the maximal length of a message (See Section 3.2). $m[i]$ denotes the value of the i-th byte of m, such that $0 \leq m[i] < 256$. We denote by $\mathcal{P}_x(m)$ the assignment of $m[i]$ for B_i in $\mathcal{P}_x$. When $\mathcal{P}_x(m) = True$ we say that $\mathcal{P}_x$ matches m. We define $\mathcal{M}(D, \mathcal{P}_x)$ to be the set of messages from D that is matched by the predicate $\mathcal{P}_x$:

$$\mathcal{M}(D, \mathcal{P}_x) = \{m \in D \mid \mathcal{P}_x(m) = True\}$$

According to our representation, $type(m)$ is redefined such that $type(m)$ denotes the tuple $\langle D, \mathcal{P} \rangle$ such that $m \in \mathcal{M}(D, \mathcal{P})$. The approximation we present in the following sections ensures that for $D = R$ and $D = S$, the sets $\mathcal{M}(D, \mathcal{P})$ for any $\langle D, \mathcal{P} \rangle \in \Sigma$ are disjoint.

5.1.1 Extracting predicate to represent a set of messages

Given a set of messages $x \subseteq D$, $\mathcal{P}_x$ is extracted using the following simple definition: we hold constraints on message bytes that have the same value for all the messages in x. Formally, let $m \in x$, we define for all $0 \leq i \leq N-1$:

$$\varphi_i = \left\{ \begin{array}{ll} B_i = m[i], & \text{if } \forall m' \in x, m'[i] = m[i] \\ True, & \text{Otherwise} \end{array} \right\}$$

$$\mathcal{P}_x = \bigwedge_{i=0}^{N-1} \varphi_i$$

We emphasize that the above definition may be replaced with a more elaborate one, if needed. We choose this definition because it is simple and is sufficiently useful for many real world protocols.

Note that $x \subseteq \mathcal{M}(D, \mathcal{P}_x)$. That is, $\mathcal{P}_x$ can match messages from D, which are not necessarily in x, but have the same format as the messages of x. This characteristic allows our algorithm to find a predicate for large sets t using a smaller example set $x \subset t$. This is possible only when x is diverse enough and contains a variety of messages from t. Otherwise, the predicate will be over-fitted to match x. In Section 5.5.2 we explain

how to generate predicates that are sufficiently general to describe message types even though we are given only a small subset of examples for that message type.

5.1.2 Example

As an example, consider the set:

$$t = \{b_0 b_1 b_2 b_3 \mid b_0 = C_0, b_1 = C_1\}$$

of message type $\langle D, t\rangle$, where C_0 and C_1 are constants. That is, t is the set of all messages in which the first and second bytes are C_0 and C_1, respectively, and the third and forth bytes can have any value. Let:

$$x = \{C_0 C_1 b_2 b_3 \mid 56 \leq b_2 \leq 67, 123 \leq b_3 \leq 127\}$$

The predicate for x according to the definition above is $\mathcal{P}_x = (B_0 = C_1 \wedge B_1 = C_2)$. The predicate for t is also $\mathcal{P}_t = (B_0 = C_1 \wedge B_1 = C_2)$ although $x \subsetneq t$. This example demonstrates that a set x may be sufficient for inferring the predicate for t.

5.2 Handling Alphabet Changes

Recall that the exact version of our method is infeasible in our setting. To remedy this, we replace the sets of message types, $Cont(w)$ and $Miss(M)$, with their approximations, denoted $Cont_A(w)$ and $Miss_A(M)$, of **message type candidates**.

As we use an approximation to generate the new message type candidates, they may intersect with previously found message types currently in Σ. This breaks the assumption that sets of message types are pairwise disjoint. Therefore, we present here an algorithm that, given $C = Cont_A(w)$ or $C = Miss_A(M)$, incorporates C into Σ while making sure the elements of Σ remain pairwise disjoint.

5.2.1 Insertion of message type candidate

Let $c = \langle D, \mathcal{P}\rangle \in C$ be a message type candidate. We say that $\langle D, \mathcal{P}\rangle$ collides with $\sigma = \langle D_\sigma, \mathcal{P}_\sigma\rangle \in \Sigma$ if $D_\sigma = D$ and $\mathcal{M}(D, \mathcal{P}) \cap \mathcal{M}(D, \mathcal{P}_\sigma) \neq \emptyset$. In order to detect if c collides with σ, we check the satisfiability of $\mathcal{P} \wedge \mathcal{P}_\sigma$. We denote by Σ the current alphabet and by Σ' the updated alphabet after the changes. The algorithm initializes $\Sigma' = \Sigma$.

The procedure to handle collisions of a message type candidate c is presented in Algorithm 5.1. We initialize $\mathcal{P}'$ with $\mathcal{P}$ and Σ' with Σ (line 2). In order to maintain Σ' as a partition of D, every message type $\langle D_\sigma, \mathcal{P}_\sigma\rangle \in \Sigma'$ that collides with $\langle D, \mathcal{P}'\rangle$ is removed (line 5). A new message type to match the non-intersecting parts of $\mathcal{M}(D_\sigma, \mathcal{P}_\sigma)$ is inserted to Σ' (line 7) and a separate new message type is inserted to match the intersecting parts (line 9). $\mathcal{P}'$ is updated to contain only the non-intersecting parts

Algorithm 5.1 The procedure to handle message type candidate

```
1: function HANDLE_CANDIDATE(⟨D, 𝒫⟩ ∈ C, Σ)
2:     𝒫′ ← 𝒫, Σ′ ← Σ
3:     for all ⟨D_σ, 𝒫_σ⟩ ∈ Σ such that D = D_σ do
4:         if 𝒫_σ ∧ 𝒫′ is satisfiable then
5:             Σ′ ← Σ′ \ {⟨D, 𝒫_σ⟩}
6:             if 𝒫_σ ∧ ¬𝒫′ is satisfiable then
7:                 Σ′ ← Σ′ ∪ {⟨D, 𝒫_σ ∧ ¬𝒫′⟩}
8:             end if
9:             Σ′ ← Σ′ ∪ {⟨D, 𝒫_σ ∧ 𝒫′⟩}
10:            𝒫′ ← 𝒫′ ∧ ¬𝒫_σ
11:        end if
12:    end for
13:    if 𝒫′ is satisfiable then
14:        Σ′ ← Σ′ ∪ {⟨D, 𝒫′⟩}
15:    end if
16:    Σ ← Σ′
17: end function
```

from $\mathcal{M}(D, \mathcal{P}')$ (line 10). After checking for collisions against all message types of Σ', $\mathcal{P}'$ contains the non-intersecting parts of $\mathcal{P}$. The resulting $\langle D, \mathcal{P}' \rangle$ does not collide with symbols in Σ' and can be inserted to Σ' (line 14) while Σ' remains a partition of R and S. Finally, Σ is assigned with the resulting Σ'.

We note that Σ' remains a partition during the algorithm. Σ' is initially a partition because Σ is a partition. Consider Figure 5.2. When a collision is found, only $\langle D, \mathcal{P}' \wedge \neg\mathcal{P}_\sigma \rangle$ can collide with another $\langle D_{\sigma_i}, \mathcal{P}_{\sigma_i} \rangle \in \Sigma$. This is because if $\mathcal{P}_{\sigma_i} \wedge \mathcal{P}_\sigma \wedge \mathcal{P}'$ or $\mathcal{P}_{\sigma_i} \wedge \mathcal{P}_\sigma \wedge \neg\mathcal{P}'$ are satisfiable by $m' \in D$, then $\mathcal{P}_{\sigma_i} \wedge \mathcal{P}_\sigma$ is also satisfiable by m' and therefore Σ did not form a partition even before handling c. Therefore it is only required to check for collisions of other alphabet symbols with the non-intersecting parts of $\mathcal{P}'$ (the updated $\mathcal{P}'$ from line 10).

During the procedure we must discard unsatisfiable predicates. A predicate may become unsatisfiable in two special cases of collision: If $\mathcal{M}(D, \mathcal{P}_\sigma) \subset \mathcal{M}(D, \mathcal{P}')$ then $\mathcal{P}_\sigma \wedge \neg\mathcal{P}'$ is not satisfiable and should be discarded (line 6). If $\mathcal{M}(D, \mathcal{P}_\sigma) \supset \mathcal{M}(D, \mathcal{P}')$ then $\mathcal{P}' \wedge \neg\mathcal{P}_\sigma$ is not satisfiable and should not be inserted to Σ (line 13).

5.2.2 Insertion of multiple message type candidates

The algorithm to handle the entire set C of message type candidates is presented in Algorithm 5.2. We run Algorithm 5.1 for every $c \in C$ (line 4). After running this procedure for all message type candidates, the elements of the resulting Σ are pairwise disjoint and are set as the new alphabet. If, during the above procedure, message types are removed from Σ then the L* algorithm must be restarted with the updated Σ since the learning was done with inaccurate alphabet (line 9). If message types are only added to Σ (and not removed) then we say that Σ grows. In the latter case the method

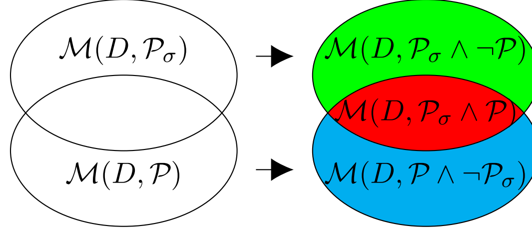


Figure 5.2: Colliding predicates

from the exact algorithm (Section 4.3) is used without having to restart L* (line 7).

Algorithm 5.2 The procedure to handle a set of message type candidates C

```
1:  function HANDLE_CANDIDATES(C)
2:      Σ_old ← Σ
3:      for all c ∈ C do
4:          HANDLE_CANDIDATE(c, Σ)
5:      end for
6:      if Σ_old ⊆ Σ then
7:          SET_ALPHABET(Σ)
8:      else
9:          RESTART_L*_WITH_ALPHABET(Σ)
10:     end if
11: end function
```

5.2.3 Example of alphabet changes

Consider an alphabet candidate $c = \langle D, \mathcal{P} \rangle$ that collides with $\langle D_\sigma, \mathcal{P}_\sigma \rangle$ where $D = D_\sigma$. The modification process of the involved predicates is illustrated using the sets represented by $\mathcal{P}$ and $\mathcal{P}_\sigma$ in Figure 5.2. When we find that c collides with $\langle D_\sigma, \mathcal{P}_\sigma \rangle$: $\langle D_\sigma, \mathcal{P}_\sigma \rangle$ is removed from Σ'; $\langle D, \mathcal{P}_\sigma \wedge \neg\mathcal{P} \rangle$ to match the non-intersecting parts of $\langle D_\sigma, \mathcal{P}_\sigma \rangle$ (green) is added to Σ'; $\langle D, \mathcal{P}_\sigma \wedge \mathcal{P} \rangle$ to match the intersecting parts (red) is added to Σ'; and $\mathcal{P}'$ is set to $\mathcal{P} \wedge \neg\mathcal{P}_\sigma$ (cyan) to contain the non-intersecting parts of c.

Finally, $\langle D, \mathcal{P}' \rangle$ where $\mathcal{P}' = \mathcal{P} \wedge \neg\mathcal{P}_\sigma$ will be added to Σ'. One alphabet symbol is removed and three alphabet symbols are created during the addition of c, including the c.

5.3 Equivalence Oracle

Answering equivalence queries in real world for black box systems is generally infeasible [Ang87]. Therefore, we define here an oracle to approximate equivalence queries. We take advantage of a commonly used approach in which an equivalence query is approximated using a sampling oracle, as we explained in Section 2.2.2. We use the Wp-Method [FvBK+91] to generate a test suite $T \subset \Sigma_M^*$ of queries w. In this method, T is generated using M and the alphabet Σ_M. Every $w \in T$ is sent as a membership

query. We use $Cont_A(w)$ when $w \in L$ to find $Miss_A(M)$, containing message type candidates.

The procedure to run a test suite T against a conjectured DFA M is shown in Algorithm 5.3. Each $w \in T$ is tested using a membership query. If $Cont_A(w)$ contains symbols that are not in Σ_M, then $False$ is returned with w and $Miss_A(M) = Cont_A(w) \setminus \Sigma_M$. If w is in the symmetric difference of L and $L(M)$, then $False$ is returned with w as a counterexample. If missing message types are not found and a counterexample w is not found in the entire test suite T, $True$ is returned and the learning terminates. Recall that, if $Miss_A(M)$ is returned, then every $w \cdot \sigma$, such that $\sigma \in Miss_A(M)$, is handled as a counterexample.

Algorithm 5.3 The procedure to approximate equivalence queries

```
1:  function RUN__TEST__SUITE(M, T ⊂ Σ*_M)
2:      for all w ∈ T do
3:          ⟨w ∈ L, Cont_A(w)⟩ ← membership(w)
4:          if w ∈ L then
5:              if Cont_A(w) \ Σ_M ≠ ∅ then
6:                  return ⟨False, w, Cont_A(w) \ Σ_M⟩
7:              end if
8:              if w ∉ L(M) then
9:                  return ⟨False, w⟩
10:             end if
11:         else if w ∈ L(M) then                    ▷ It holds that w ∉ L
12:             return ⟨False, w⟩
13:         end if
14:     end for
15:     return True
16: end function
```

5.4 Modified Symbolic Execution

Our algorithm answers membership queries by symbolic execution of the binary code. During the symbolic execution we need to find feasible executions in which the binary code follows a given sequence of alphabet symbols. We call this process *monitoring* the sequence. To monitor a sequence, we introduce assertions and assumptions into symbolic execution. In order to link between a sequence of alphabet symbols and the binary code, we propose the modifications we apply on top of classic symbolic execution. Some other modifications are required in order to implement the extension of membership queries to probe message type candidates as we defined in Section 5.5.2.

We emphasize that, except intercepting the networking activity of the binary, we leave the rest of symbolic execution as it is, without modifications. This is mandatory to correctly simulate the actions that the binary code performs in order to compose or parse messages.

Next we describe the parts we modify in detail.

5.4.1 Implementing assumptions and assertions in symbolic execution

Our algorithm uses symbolic execution of the binary code to answer queries sent by the learner. We do not use model checking in our work due to the lack of tools and methods to apply model checking on binaries. Our method, however, uses concepts similar to those involved in model checking and applies them on symbolic execution of the binary code. We simulate assumptions and assertions using symbolic execution constraints and using the fact that symbolic execution checks the satisfiability of states during the execution and discards unsatisfiable executions. We insert assumptions to the constraints of a symbolic state when we want to limit the considered executions to executions that satisfy the assumptions. Conflicting constraints or concrete values will cause the execution to become unsatisfiable and discarded. Similarly, we insert assertions to the constraints of a symbolic state when we want to limit the considered executions to executions where both the constraints developed by the symbolic execution and all of the assertions are satisfiable.

5.4.2 Hooking calls to send and receive procedures

Hooking calls in symbolic execution allows to replace the behaviour of a binary code in certain functions with customized procedures. When hooks are applied and a state s calls a hooked function, a customized procedure is executed on s instead of the original function. The customized procedure should return a state that represents the program after the call to the function. When a function is not hooked, it is executed according to the symbolic execution engine.

When we apply symbolic execution on the binary code, we hook functions the user provides in order to monitor a sequence and to discover message type candidates. We ask the user of our method to manually identify functions that send or receive data from the network. The user should provide the addresses of the relevant functions and should provide a code snippet that extracts the message buffer and message length from a given symbolic state. The user should also specify for each function whether it is a send function or a receive function. For simplicity of exposition, we assume there is a single send function and a single receive function. However, the presented method can be equally implemented in case there are several such functions.

Identifying the necessary hooks and specifying the required code snippet are most of the times simple and require a small manual effort by the user of our method. Send and receive functions of the binary code share a common property that they call the send or receive functions of the operating system. Therefore, these functions are easy to find with classical reverse engineering and disassemble tools. In case the user cannot find these functions, he can choose to provide the send and receive functions of the operating system. In any case, it is not in the scope of this work to identify send and

receive functions.

In addition to identifying the functions, the user should provide a code snippet that extracts the message length and buffer from a symbolic state s. When such state s calls a hooked function and our customized procedure is executed, it is necessary to get the message length and buffer from the variables of the state. The code snippet that the user provides will most likely get these items from the arguments of the call. Note that inferring this code snippet manually is considered trivial.

5.4.3 Extending symbolic states to track query state

Let $w = \sigma_1 \dots \sigma_n$ be a sequence of alphabet symbols queried in a membership query. Let s be a symbolic state. Recall the definition of symbolic state from Section 2.4. We extend the definition of s to contain additional fields. These fields are used to track which alphabet symbols from the sequence have been sent or received in the execution that s represents, from the initial state and up to s. In addition, we add to s fields that will be used during the probing phase of the query. During symbolic execution, these fields are propagated from parent state to all its descendant states.

Extensions for monitoring

We extend s to store in addition to the original fields:

- A sequence $w_s = \sigma_1 \dots \sigma_n$. The sequence, and especially its predicates, are used to insert assumptions and assertions during the symbolic execution, to limit the considered executions to executions that match the query.

- An index i_s such that $0 \leq i_s \leq n$. This index marks the index of the last symbol monitored in the execution when reaching s. That is, the sequence $\sigma_1 \dots \sigma_{i_s}$ was sent or received in the execution up to s. i_s is incremented by one every time a send or receive that match σ_{i_s} occurs in the execution. If $i_s = n$, the entire sequence w_s was monitored in the execution represented by s.

When the symbolic execution is initialized to answer a membership query w, the initial state s is defined with $w_s = w$ and $i_s = 0$.

Extensions for probing

When a state s is executed during the probing phase, we use two additional fields that we add to s:

- A symbolic value msg_s. msg_s is initialized during the first send or receive call that occurs in the probing phase. It is initialized with the symbolic value sent or received. msg_s will be used to generate a message type candidate for s as we explain in Section 5.5.2.

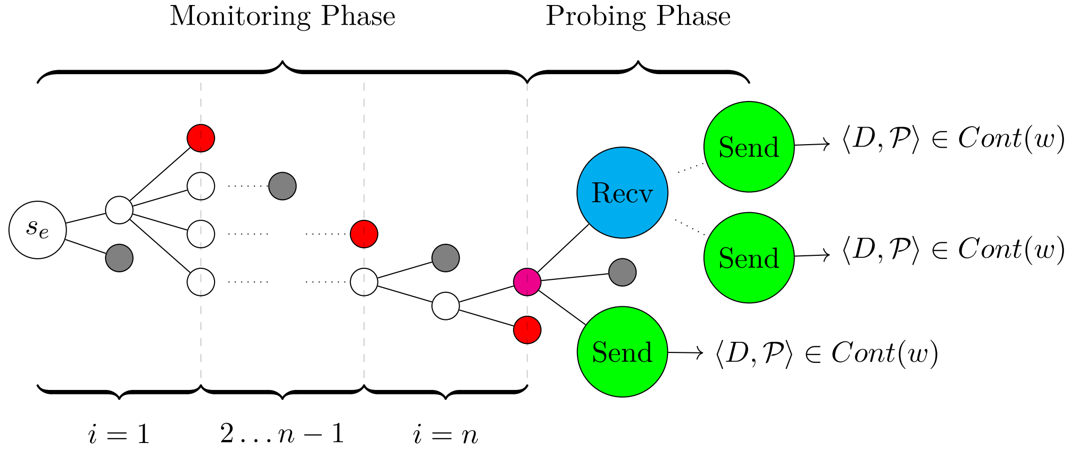


Figure 5.3: Illustration of symbolic execution during membership query

- A flag $D_s \in \{R, S, \emptyset\}$, which specifies whether the first communication in the probing phase was a send or a receive. It is initialized with $D_s = \emptyset$ and set only during the first send or receive in the probing phase.

5.5 Membership Oracle

Let $w \in \Sigma^*$ be a sequence of message types sent as a membership query. The algorithm should answer whether $w \in L$ and if $w \in L$ it should also provide $Cont_A(w)$ – a set of message type candidates that may follow w. By definition, w is a sequence of message types. Recall that such a sequence corresponds to sessions of the protocol. We answer membership queries using symbolic execution of the binary code. With symbolic execution we are able to answer whether $w \in L$ or not. However, we return a set $Cont_A(w)$ that approximates $Cont(w)$.

A symbolic execution begins with a single active initial state, located at the entry point of the binary code. By stepping forward from active states iteratively, a set of new active states is generated, representing multiple different execution paths of the binary code. The properties defined in Section 5.4 are propagated to generated descendant states. We divide the symbolic execution into two phases: monitoring phase, which answers whether w is a valid session of the protocol, and the probing phase, which results in possible continuations of w. The latter phase is executed only if w is a valid session. During the monitoring phase we guide the symbolic execution to consider only execution paths that follow the given sequence w. During the probing phase, however, we take into account all feasible executions that are developed as continuations to the executions that we found during the monitoring phase.

An illustration of both the monitoring and the probing phase is shown in Figure 5.3. We will further explain this figure in the upcoming subsections.

5.5.1 Monitoring phase

Let $w = \langle D_{\sigma_1}, \mathcal{P}_{\sigma_1}\rangle \ldots \langle D_{\sigma_n}, \mathcal{P}_{\sigma_n}\rangle$ be the queried sequence. We hook the functions in the binary code that send and receive messages. The procedures inserted in the hooks are presented in the following subsections. We perform the monitoring phase in n stages: We start with a single initial state s_e located at the binary code's entry point with $i_{s_e} = 0$ and $w_{s_e} = w$. For each stage $1 \leq i \leq n$ we add constraints of the predicate of the message type $\langle D_{\sigma_i}, \mathcal{P}_{\sigma_i}\rangle$. These constraints restrict the symbolic execution to execution paths that send or receive a message of type $\langle D_{\sigma_i}, \mathcal{P}_{\sigma_i}\rangle$ at the current stage. We then resume the symbolic execution of all active states, until all active states are restricted to the i-th message type. In other words, we begin a stage i with states that have $i_s = i - 1$ and finish it with a set of new states with $i_s = i$. Since we monitor the network activity, we discard executions that send or receive messages that do not match the queried sequence. In the next subsections we explain in detail how we eliminate execution paths that do not match w. Recall that states with unsatisfiable constraints as well as abort states, are discarded automatically.

If we successfully finish the execution of the last stage (for $i = n$) with at least one active state, then there is at least one valid session of the binary code that matches the sequence of message types of w. In such a case the answer to the query is $True$. If, however, during one of the stages there are no active states left, then w represents invalid session for the binary code, and therefore the oracle returns $False$ as the query's result.

In Figure 5.3, the monitoring phase is illustrated in the left part of the figure. States that represent infeasible executions (infeasible constraints) are discarded (gray). States that represent feasible executions that do not match the query w are discarded as well (red). The stages of the monitoring are illustrated as well. The figure represents a membership query that is answered with $True$, as a single active state (magenta) is found at the end of the monitoring phase.

Monitoring incoming messages

The receive function is hooked during the monitoring phase with the following procedure. The purpose of this procedure is to advance the monitoring of the queried sequence when the binary code receives a message. Let s be a symbolic state during the i-th stage in which the binary code calls the receive function, and let $\sigma_i = \langle D_{\sigma_i}, \mathcal{P}_{\sigma_i}\rangle$ where $i = i_s + 1$ be the next expected alphabet symbol in the query.

If σ_i represents outgoing messages (i.e $D_{\sigma_i} = S$), then s is not an execution path that can match the queried sequence w: the execution path represented by s receives a message in the i-th stage whereas the query w represents sessions that send a message from $\mathcal{M}(S, \mathcal{P}_{\sigma_i})$ in the i-th stage. Therefore we discard s.

On the other hand, if σ_i represents incoming message type (i.e $D_{\sigma_i} = R$), we move s to the $i + 1$-th stage with s' as its successor and set $i_{s'} = i = i_s + 1$. We attach

to s' an assumption that a message from $\mathcal{M}(R, \mathcal{P}_{\sigma_i})$ is read from the network. We implement this assumption by inserting the predicate $\mathcal{P}_{\sigma_i}(msg)$ to the constraints of s', where msg is the received message buffer. We acquire msg using the code snippet that the user provides in order to extract it from s. s represents the program state upon calling receive and msg must be part of this state (See Section 5.4.2).

Once s' resumes execution, it will continue as if the received message satisfies $\mathcal{P}_{\sigma_i}$ and thus "forcing" descendant states to follow only execution paths that represent the reception of messages from $\mathcal{M}(R, \mathcal{P}_{\sigma_i})$ during the i-th stage.

Algorithm 5.4 The monitor procedure

```
 1: function MONITOR(s, D ∈ {R, S})
 2:     ⟨D_σi, P_σi⟩ = w_s[i_s + 1]
 3:     msg ← user_code_snippet(s)
 4:     if D_σi = D then
 5:         s' ← s.copy()
 6:         if D = R then
 7:             s'.add_assumption(P_σi(msg))
 8:         else if D = S then
 9:             s'.add_assertion(P_σi(msg))
10:         end if
11:         i_s' ← i_s + 1
12:         return s'
13:     else
14:         return null
15:     end if
16: end function
```

The pseudo-code in Algorithm 5.4 for $D = R$ describes the monitoring process that occurs when the receive function is called from a state s in order to receive a message msg. We check the properties we added to assist the monitoring phase and get the next expected alphabet symbol by the query (line 2). That is, the $i_s + 1$ symbol in w_s. Recall that w_{s_e} was initialized with the queried sequence w, so this is the $i_s + 1$ symbol of w. In case $D_{\sigma_i} = D = R$, we duplicate s (line 5) and attach the constraint $\mathcal{P}_{\sigma_i}(msg)$ to the new copy (line 7), resulting in s'. We also set $i_{s'}$ to $i_s + 1$ (line 11) to indicate that the $i_s + 1$-th symbol of $w_s = w_{s'}$ was monitored. If $D_{\sigma_i} = S$, we terminate the execution path of s. In the first case, the procedure returns the state s' to continue the execution path of s during the $i + 1$-th stage (line 12). In the second case, the procedure returns $null$ to discontinue the execution path of s (line 14).

Monitoring outgoing messages

The procedure to hook a send function is similar to the one used above for incoming messages. The purpose of the procedure is to perform the monitoring of the queried sequence when the binary code sends a message. Let s be a symbolic state in which the binary code calls the send function, and let $\sigma_i = \langle D_{\sigma_i}, \mathcal{P}_{\sigma_i} \rangle$ where $i = i_s + 1$ be the

next expected alphabet in the query.

In case σ_i represents incoming messages (i.e $D_{\sigma_i} = R$), it means that the execution path of s does not match the sequence query w and we discard s. On the other hand, if σ_i represents an outgoing message type (i.e $D_{\sigma_i} = S$), we move s to the $i+1$-th stage with s' as its successor and set $i_{s'} = i = i_s + 1$. We attach to s' an assertion that a message from $\mathcal{M}(S, \mathcal{P}_{\sigma_i})$ is sent to the network. We implement this assertion by inserting the predicate $\mathcal{P}_{\sigma_i}(msg)$ to the constraints of s', where msg is the sent message buffer. We acquire a reference to msg using the code snippet the user provides (See Section 5.4.2).

If s represents an execution path that does not send a message from $\mathcal{M}(S, \mathcal{P}_{\sigma_i})$ in the i-th stage, s' will be found unsatisfiable. Otherwise, if s' is found satisfiable, it means that the binary code sends a message from $\mathcal{M}(S, \mathcal{P}_{\sigma_i})$ in the execution path that s represents, and s' resumes this path in the $i+1$-th stage.

The procedure is presented in Algorithm 5.4 for $D = S$.

5.5.2 Probing phase

The purpose of the probing phase is to generate $Cont_A(w)$ for w for which the monitoring phase returned $True$. As above, we hook the send and receive functions of the binary code but insert different procedures. We describe them in the upcoming subsections. The aim of the probing procedure is to uncover all symbolic states that represent execution paths in which a message is sent or received following w. For each such state s the constraints on the message buffer received or sent, denoted as msg_s, are collected.

Recall the definition of msg_s and D_s from Section 5.4.3. Given a state s in the probing phase, the purpose of these hooks is to store msg_s and D_s and trigger the generation of message type candidates. We assume that all concrete values for msg_s in the context of state s belong to the same message type and we generate a message type candidate to represent it.

In Figure 5.3, the probing phase is shown in the right part of the figure. Since the purpose of the probing phase is to discover all message type candidates that can follow the sequence w, we continue the execution from active states matching the query w at the end of the monitoring phase (magenta). That is, states with $i_s = n$. Infeasible executions are discarded (gray). A message type candidate c is generated from every state s (green), and added to $Cont_A(w)$. After a message type candidate is generated from s, s is removed from the set of active states. The probing phase is executed until there are no active states left. In the following subsections we explain how and when a message type candidate c is generated during the probing phase. We present a novel approach to generate a message type candidate for executions that receive a message (cyan) in the probing phase. This is necessary because incoming messages are inputs and are not created by the binary code.

Probing outgoing messages

The hooking procedure used in the send function is straightforward. Here msg_s is the sent message's symbolic buffer. We assume that the symbolic buffer has enough constraints under the current state s that sufficiently represent the sent message type. Therefore, no further symbolic execution is needed for s in the probing phase and the symbolic state s is passed to the procedure to generate a message type candidate. This procedure is described in an upcoming subsection.

Let s be a state in which the binary code calls a send function during the probing phase. In this case, we simply clone s to new state s' with $D_{s'} = S$ and $msg_{s'} = m$, where m is a reference to the sent message. Then we pass s' to the procedure to generate a message type candidate. When a message type candidate is successfully generated from s', s' is removed from the set of active states to discontinue the execution path of s. This execution path has already produced a message type that follows the queried sequence and there is no need to resume its execution.

Probing incoming messages

Let s be a state in which the binary code calls a receive function during the probing phase. Let msg be the symbolic received message. Upon calling receive, the content of msg is an unconstrained symbolic value as it is received as an input by the binary code. Hence, one cannot extract information on the format of the message type that is expected to be received in state s. To solve this, we present the following novel approach to uncover information regarding the expected received message type: we clone s to s' in order to add the fields $msg_{s'} = msg$ and $D_{s'} = R$. Then we resume symbolic execution of s'. During the execution of s' we assume that the binary code will parse the received message, hence constraints will be developed on $msg_{s'}$ that will reveal the format expected by the binary code. We choose to resume the execution until the binary code sends or receives another message, or until the code terminates. We assume that until that point the code completes parsing the received message and acts upon its content, hence sufficient constraints are accumulated on the message buffer to identify the expected message type to be received. During these instructions s' is developed into possibly multiple descendant states. These states are then passed to the procedure to generate a message type candidate. This procedure is described in the upcoming subsection.

Let $\widetilde{s}$ be a descendant state of s' in which either send or receive function are called for the first time after s'. Alternatively, $\widetilde{s}$ is a descendant state of s' in which the binary code terminates. $\widetilde{s}$ represents a point of the binary code in which the incoming message is already analyzed and its type is determined. Therefore it has passed through conditional branches that append constraints on $msg_{s'}$. These constraints will give us information on the structure of the message. $\widetilde{s}$ is then passed to the procedure to generate message type candidate. When $\widetilde{s}$ has successfully generated a message type

candidate, it is removed from the set of active states to discontinue the execution path represented by $\widetilde{s}$. This execution path has already produced a message type that follows the queried sequence and there is no need to resume its execution.

The procedure to generate message type candidates

Let s be a state that successfully probed either sent or received message – msg_s. The purpose of the procedure described here is to generate a message type candidate from s. Note that concrete values satisfying the constraints of s on msg_s represent valid messages in the protocol. We assume that these concrete messages form a message type t. We ask the symbolic execution engine to solve msg_s and generate NUM_SOL[1] possible concrete values for msg_s.

Let x be the set of generated concrete messages. We extract $\mathcal{P}_x$ as described in Section 5.1. Then, we iteratively refine $\mathcal{P}_x$ by trying to find concrete values m' of msg_s, that contradict $\mathcal{P}_x$ in a sense that $\neg\mathcal{P}_x(m')$ is $True$. Such m' are concrete values that can appear in msg_s in a real execution (since they were solution to the msg_s). Nevertheless, they are not represented by $\mathcal{P}_x$. In case we find such m', we add them to x and regenerate $\mathcal{P}_x$. We repeat this process until the solver is unable to find additional m' that contradict $\mathcal{P}_x$. This procedure allows us to find a set $x \subset t$ that represents the variety of t.

Recall our statement from Section 5.1 regarding the diversity of the set x when generating $\mathcal{P}_x$. The procedure described here uses a smaller, diverse enough set x to describe a larger set $\mathcal{M}(D_s, \mathcal{P}_x)$. $\mathcal{M}(D_s, \mathcal{P}_x)$ approximates the set of all concrete values of msg_s.

Let $concretize(s, buffer, n, C)$ be a function of the symbolic execution engine that returns up to n concrete values for $buffer$ that corresponds to the constraints of s and to additional constraints C. In case there are no possible values that follow the constraints, the function returns $\emptyset$. The pseudo-code of the message generation process is shown in Algorithm 5.5.

Algorithm 5.5 The procedure to generate message type candidate

```
1: function GENERATE_ALPHABET_CANDIDATE(s)
2:     x ← ∅
3:     contradictions ← ∅
4:     do
5:         P_x ← extract_predicate(x)
6:         contradictions ← concretize(s, msg_s, NUM_SOL, ¬P_x)
7:         x ← x ∪ contradictions
8:     while contradictions ≠ ∅
9:     Cont_A(w) ← Cont_A(w) ∪ {⟨D_s, P_x⟩}
10: end function
```

[1] In our implementation $NUM_SOL = 10$

Example of probing message type candidates

To illustrate how a conditional branch reveals information on a symbolic value, consider the pseudo-code in Listing 5.1 of a binary code:

Listing 5.1: Pseudo-code to demonstrate the probing phase

```
1:  Send(Connect);
2:  msg = Receive();
3:  if (msg == "HelloV1") {
4:        Send("InitV1");
5:        ...
6:  } else if (msg == "HelloV2") {
7:        Send("InitV2");
8:        ...
9:  } else {
10:       abort();
11: }
```

Assume a query $w = \langle S, \mathcal{P} \rangle$ where $\mathcal{P} = (B_0B_1B_2B_3B_4B_5B_6 = \text{"Connect"})$. The monitoring phase for this query and this binary code is done with a single satisfiable state in line 2. The probing phase resumes symbolic execution from this state. In line 2 the binary code receives msg in a state s. msg refers to a symbolic value with no constraints, as it is an input from the network. A state $s' = s$ is resumed with $msg_{s'} = msg$. The execution splits according to the conditional branches: a state $\widetilde{s_1}$ represents execution at line 4 with a constraint that $msg = \text{"HelloV1"}$, a state $\widetilde{s_2}$ represents execution at line 7 with a constraint that $msg = \text{"HelloV2"}$ and a state s_3 represents execution at line 10 which aborts and is discarded. Both $\widetilde{s_1}$ and $\widetilde{s_2}$ represent a call to send, which triggers the generation of message type candidate from $msg_{\widetilde{s_1}} = msg_{\widetilde{s_2}} = msg$. In the context of $\widetilde{s_1}$, the analysis is tied to the constraint $msg = \text{"HelloV1"}$ and generates a message type candidate $\langle R, \mathcal{P}_1 \rangle$ where:

$$\mathcal{P}_1 = (B_0B_1B_2B_3B_4B_5B_6 = \text{"HelloV1"})$$

In the context of $\widetilde{s_2}$ the analysis is tied to the constraint $msg = \text{"HelloV2"}$ and generates an alphabet symbol $\langle R, \mathcal{P}_2 \rangle$ where:

$$\mathcal{P}_2 = (B_0B_1B_2B_3B_4B_5B_6 = \text{"HelloV2"})$$

The set $Cont_A(w) = \{\langle R, \mathcal{P}_1 \rangle, \langle R, \mathcal{P}_2 \rangle\}$ is returned with the answer that $w \in L$.

Chapter 6

Optimizations

We develop several optimizations to reduce the running time of the method and allow it to scale to real-world protocol implementations. These optimizations take advantage of the characteristics of network protocols and the algorithm itself. Since symbolic execution is the most time consuming part of the algorithm, the developed optimizations focus on reducing the number of needed symbolic executions, as well as reducing the running time of symbolic executions.

6.1 Prefix Closed Property

This optimization leverages the fact that the protocol's regular language L is a prefix-closed set (See Section 3.1). It is based on a similar technique, employed in [IHS15]. The optimization allows to answer some membership queries immediately by the Learner without having to resort to symbolic execution. Every membership query w that was answered with $False$ is stored by the membership oracle in a cache. For every membership query w sent to the oracle, it is first checked whether there exists $x, y \in \Sigma^*$ such that $xy = w$ and x is in the cache. In such a case, the query immediately returns $False$. In other words, if a prefix of w is not in L, then by definition of prefix-closed set it must hold that $w \notin L$. Thus, we avoid unnecessary applications of symbolic execution.

When, during the discovery of new message types, an alphabet symbol is removed and L* is restarted, all queries w in the cache that contain a removed symbol are removed from the cache. These queries are invalid with the new alphabet and cannot be a prefix of a query over the new alphabet.

6.2 Fast Equivalence Queries

Let $w \in \Sigma^*$ be a query for which the membership oracle answered $True$, and let $Cont_A(w)$ be the returned set of alphabet candidates. We store w and its associated $Cont_A(w)$ in a cache called *continuations cache*. The equivalence oracle answers an

equivalence query for DFA M by utilizing this cache. The oracle checks consistency of M with the continuations cache: for every w in the cache and for every $\sigma \in Cont_A(w)$, it checks whether M accepts $w \cdot \sigma$. If M rejects $w \cdot \sigma$, the equivalence oracle returns $False$ and returns $w \cdot \sigma$ as a counterexample. Thus it alleviates the need to run symbolic execution to answer the query. Note that, the cache stores alphabet symbols after resolving collisions, and not message type candidates. This is necessary so that the cache can return counterexamples over the current alphabet. When alphabet symbol is removed, all cache entries containing the removed symbol are erased.

The correctness of this optimization follows from the definition of $Cont_A(w)$. According to the definition, every state machine M that claims $L(M) = L$ should satisfy $w \cdot \sigma \in L(M)$.

Checking consistency with the continuations cache is considered fast compared to symbolic execution of the program. Because we begin an equivalence query with checking consistency with the continuations cache, we make sure that if a counterexample can be provided faster than symbolic executions of a test suite (See Section 5.3), it is provided, obviating the need to generate and check a test suite.

6.3 Execution Cache

This optimization uses symbolic states s resulting at the end of the monitoring phase for w as initial states for query wx for any $x \in \Sigma^*$. All queries w for which the teacher returns $True$ are stored in a cache called "Execution Cache" with all active symbolic states resulting at the end of the monitoring phase for w. Then, whenever a query w' is sent, the teacher finds decomposition $w' = p \cdot s,\ p = p_1 \dots p_k$ such that p is the longest word in the cache. Then, the monitoring phase for w' begins with the states saved for p, in the $i = k + 1$ stage of the monitoring. We skip the first k stages because the states saved for p contains exactly all execution paths for sessions $p_1 \dots p_k$. The rest of the query remains the same as described in Section 5.5.

Without this optimization, our algorithm would apply symbolic execution multiple times for the same sequence $p_1 \dots p_k$. This optimization assures that we use states obtained during previous symbolic executions instead of developing them again from the initial state of the program.

This optimization is implemented in our algorithm. We note that, when alphabet symbols are removed, all entries in the cache that include the removed symbol should also be removed.

6.3.1 Example

Consider again the example shown in Section 5.5.2. The query $w_1 = \langle S, \mathcal{P} \rangle$ has finished its monitoring phase with a single state s in line 2. This state is saved in the execution cache and is associated with w_1. When a query $w_2 = \langle S, \mathcal{P} \rangle.\langle R, \mathcal{P}_1 \rangle.\langle R, \mathcal{P}_2 \rangle$ is received

by the teacher, the teacher finds that w_1 is in the cache with s as active state and that there is no need to start the symbolic execution from the initial state. s is cloned to s', $w_{s'}$ is set to w_2, and the set of active states is initialized with $\{s'\}$. The monitoring phase for this query begins in the second stage ($i = 2$).

Chapter 7

Implementation and Results

In this chapter we present the details of our implementation of the presented method and explore its performance. We evaluated our method against various protocol implementations (including SMTP and other non-standard protocols). Even though our method expects a binary file as input, we preferred to work with binaries for which we have the source code. This enables us to validate our method: having the source code of a binary, it is simpler to infer its protocol manually and compare it against the result of an experiment.

7.1 Implementation

The algorithm was implemented[1] as two independent modules for the Learner and the Teacher. The Learner is implemented as a Java program that communicates with the Teacher using local socket. The Teacher is implemented as a Python program that serves the Learner's queries. An illustration of the implementation components and their interactions is shown in Figure 7.1. We base our implementation on two open source tools:

1. LearnLib [IHS15] – implements the L* algorithm and its variations (for example, [SG09])
2. angr [SWS+16] – a library that provides static analysis and symbolic execution engine for binary codes

7.1.1 LearnLib

[IHS15] An open-source Java tool for automata learning. It provides generic implementations for various learning algorithms, including L* and its variations (for example, [SG09]). LearnLib also provides implementations of membership oracles, equivalence oracles and cache optimizations. These implementations can be configured to use one

[1] https://github.com/ron4548/{InferenceClient,InferenceServer}

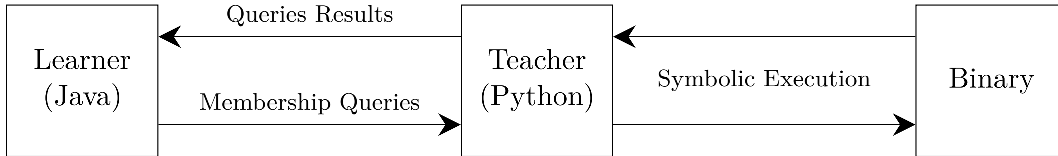


Figure 7.1: Illustration of our implementation and the interactions between its components

another to provide layers of approximations and caching. For example, sampling oracle approximation for equivalence queries is configured to use a membership oracle. The approximation uses this oracle to test its test suite. The implementation of L* in LearnLib also implements L* algorithm that supports growing alphabets and we use this capability in our work.

7.1.2 angr

angr [SWS+16] is Python tool providing static analysis and symbolic execution engine for binary codes. Its execution engine works with program states that comprises of registers and memory data, as well as constraints on symbolic variables. angr includes a solver that can generate concrete values for symbolic variables. angr also supports hooking procedures of a binary code.

7.1.3 Learning Client (Learner)

The Learner begins by initializing a learning process with LearnLib's implementation of L*. Membership queries are first checked with the prefix-closed cache (See Section 6.1). In case of a miss, the query is sent to the Teacher. If the Teacher answers that $w \in L$, then $Cont_A(w)$ is analyzed for new message types which are added to Σ. Intersections between message types are handled as described in Section 5.2.

Conjectured DFA is first checked against the continuations cache as described in Section 6.2. If the conjectured DFA is found to accept all continuations in the cache, an equivalence query approximation is triggered. A test suite is generated using the Wp-Method [FvBK+91] and is tested as explained in Section 5.3. Missing message type are handled as described in Section 5.2. Counterexamples are handled by the internal implementation of L*.

We use the following features of LearnLib:

1. Classical L* implementation
2. Support for growing alphabet in L*
3. Test suite generation with Wp-Method
4. Prefix-closed cache (See Section 6.1)

On the other hand, we implemented the following modifications:

1. Alphabet symbols as tuples $\langle D, \mathcal{P} \rangle$
2. Handling of alphabet changes and collisions (Section 5.2)
3. Continuations cache to support Fast equivalence queries. (Section 6.2)
4. Running tests suites to approximate modified equivalence queries (Section 5.3)

7.1.4 Symbolic Execution Server (Teacher)

Our Teacher runs symbolic execution using angr, and is the only component that interacts with the binary code. The Teacher initializes symbolic execution for the binary code and setups the hooking of the send/receive functions the user provides. The Teacher receives membership queries in a loop, until the Learner finishes the learning. When a membership query is received, we first check the execution cache optimization (See Section 6.3).In case of a miss the monitoring phase executes as described in Section 5.5. If the query results with $True$, the probing phase runs and generates message type candidates. These candidates are collected and sent back to the Learner. The learner then incorporates the candidates in the alphabet.

7.2 SMTP Client Experiment

We applied our method to a basic SMTP client. SMTP (Simple Mail Transfer Protocol) is a common email protocol. The SMTP client was built using an SMTP library [2]. The basic SMTP client we worked with, simply connects to a server and sends an email. The communication with the server is done by the library. The client connects with the server, sends and receives messages according to the client's actions. The source code of the client is available as part of the SMTP library.

SMTP is text-based protocol. That is, messages consist of ASCII[3] encoded characters. In particular, every message in the protocol is terminated by a character that represents a new line. Every message from the server begins with a status code. When a client sends a message to the server, the server may respond with multiple messages. In case the server responds with a sequence of multiple messages, every message except the last message in the response will contain a hyphen (-) after the status code.

The receive function we choose to hook in this example is the function `smtp_getline`. This function reads a line from the network stream (a message). The function has only one argument, which is a structure that represents the connection. It allocates a buffer for the message (the line) and stores it inside that structure. We provide a code snippet to emulate this allocation. We note that the logic of the program handles multiple

[2] https://github.com/somnisoft/smtp-client

[3] A representation of characters in a digital form

messages in the server's response: for each such message, the `smtp_getline` function will be called. The send function we hook is `smtp_write`. This function simply sends a message (writes a line) to the network stream. These two functions are provided to our method as hooks.

Applying symbolic execution on string operations is generally computationally difficult. For example, a symbolic buffer of length N may contain any string of length varying from 0 to $N - 1$ (string is always terminated with a zero byte). The SMTP protocol, for example, parses the status code from received messages by converting the first three ASCII characters of the message to an integer value. String operations, like these two, tend to result with state explosion in the symbolic execution and complex constraints to solve. Therefore, applying our method on the SMTP protocol was challenging.

We applied our method on the binary of the SMTP client. Recall that the client simply sends an email to a remote server. When applying our method, and during the probing phase for one of the queries sent by the learner - the symbolic execution of angr [SWS$^+$16] probably reached an infinite loop or a very heavy computation. Even when we let the symbolic execution run for a few hours, the execution did not terminate. However, from the traces of our method, we could see the various message types correctly discovered by the method until that point, as well as the different queries sent by the learner.

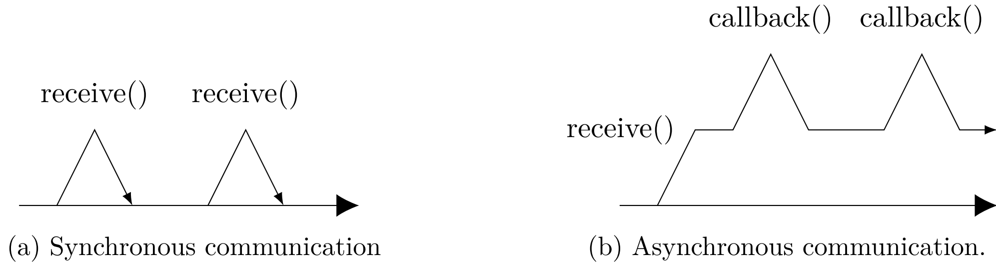
7.3 Gh0st RAT Experiment

Gh0st RAT is a well known malware[4] that runs on Windows machines. Once an instance of Gh0st RAT is run on the victim's computer, the attacker has full control over the system. This includes access to the screen, microphone and camera. The attacker controls the malware using a C&C protocol. The source code of some variants of Gh0st is available on the web. We chose to work with one[5] of these implementations. In this variant, the RAT runs in a multi-threaded process which connects to the attacker's server. When a command is received, a new thread and a new connection are created to handle the command and its further communications.

Our experiment with Gh0st required an adaptation of our method. The Gh0st RAT implementation we worked with receives messages from the network in an asynchronous manner, which is different than the synchronous manner we assumed initially. In our method, we assume that we can hook the receive function of the binary and that the receive function returns the received message (Figure 7.2a). To handle incoming messages in the implementation we worked with, the receive function is run in a loop in a background thread. That is, the receive function is called once and never returns. Instead, once enough data is received to compose the entire received message, a callback

[4] https://attack.mitre.org/software/S0032/

[5] https://github.com/yuanyuanxiang/SimpleRemoter



(a) Synchronous communication

(b) Asynchronous communication.

Figure 7.2: Illustration of synchronous vs. asynchronous communication

function is executed to handle the message (Figure 7.2b). If we hook this function in the same way we described in Chapter 5, the callback will never be executed and the hook will run only once. The adaptation we suggest to our method introduces a slightly different hooking logic. The adapted hooking is identical to the one we present in Section 5.5.2 except that it triggers the callback function after monitoring an incoming message. As a result, we require the user of the adapted hook to identify the callback function and provide its address, in addition to send and receive functions. The rest of the method is not affected by this adaptation.

Initially, we applied our method on this variant. However, angr [SWS+16] is not well-suited for multi-threaded programs. In addition, angr does not fully support Windows API. This lead to difficulties with applying our method on the Gh0st RAT binary directly. To validate that the proposed method can infer a state machine as complex as that of Gh0st RAT, we opted for a different approach. We re-implemented most of the malware's C&C protocol with a simpler architecture that does not involve threads. Consequently, it was not necessary to use the adaptation we developed for the original binary. We applied our method on the re-implementation of the protocol.

We provided our method with two functions that the program uses in order to send and receive messages from the network: `get_message` and `send_message`. Both get a message buffer and its length. The full state machine is complex and contains 27 states and 52 transitions (without rejecting states). We show the full state machine and the discovered alphabet symbols in Figure 7.4. In the protocol a message type is determined by the first byte of the message and some message types provide additional information in the second byte. In Table 7.2 we present the predicate of each type as the prefix common to all the messages in that type.

In Figure 7.3 we show a branch of the state machine, that handles a command to stream the camera of the victim. In this branch, the attacker sends a command to open the camera stream (`[R] WEBCAM`). Then, the client sends information regarding the stream (`[S] BMPINFO`) and waits to receive from the attacker a command to begin streaming (`[R] NEXT`). From now on, the client sends periodically a bitmap of the webcam to the attacker's server (`[S] BMP`). By default, this bitmap is not compressed. The attacker can enable compression of the stream (`[R] COMPR_ON`) and disable it (`[R] COMPR_OFF`). When the compression is on, the bitmap is sent compressed (`COMPR_BMP`).

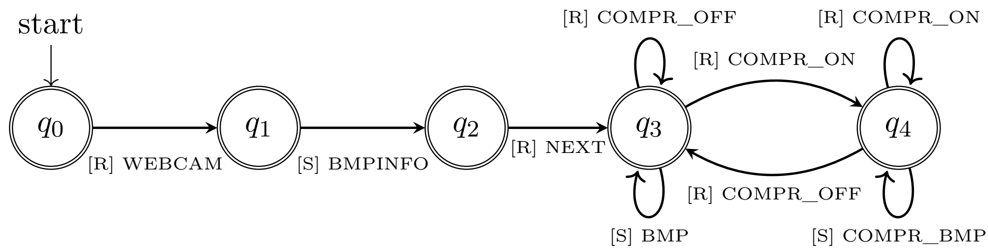


Figure 7.3: The branch in Gh0st RAT C&C protocol that handles webcam streaming. The letter in the square brackets indicates whether the message is sent or received.

Learning time:	142 seconds
Total Membership queries:	45488
Total Equivalence queries:	1
Prefix-Closed cache miss rate:	0.2184
Alphabet size:	45

Table 7.1: Gh0st RAT learning statistics

Statistics of the learning process are shown in Table 7.1. 45 message types were discovered. The learner issued about 45,000 membership queries; more than 78% of them were answered by the prefix-closed cache. Only a single equivalence query was issued. This shows the dramatic effectiveness of the continuations cache to reduce the number of costly equivalence queries. There are no discrepancies between the learnt DFA and the protocol's state machine.

MSG ID	Name	Prefix	MSG ID	Name	Prefix
[R:0]	SERVER_EXIT	0xcd	[R:1]	CMD_BYE	0xcc
[R:2]	CMD_TALK	0x34	[R:3]	CMD_REGEDIT	0x33
[R:4]	CMD_AUDIO	0x22	[R:5]	CMD_SHELL	0x28
[R:6]	CMD_SERVICES	0x32	[R:7]	CMD_SCREEN_SPY	0x10
[R:8]	CMD_CAM	0x1a	[R:145]	CMD_SCREEN_BLOCK_INPUT	0x15
[R:10]	CMD_SYSTEM	0x23	[S:277]	TOKEN_CLIPBOARD_TEXT	0x76
[S:12]	TOKEN_BITMAPINFO	0x73	[S:13]	TOKEN_AUDIO_START	0x79
[S:14]	TOKEN_SERVERLIST	0x81	[R:140]	CMD_SCREEN_SET_CLIPBOARD	0x19
[S:16]	TOKEN_WSLIST	0x7e	[S:17]	TOKEN_TALK_START	0x84
[S:19]	TOKEN_SHELL_START	0x80	[S:20]	TOKEN_CAM_BITMAPINFO	0x77
[S:21]	CMD_BYE	0xcc	[R:32]	CMD_SVCCFG/START	0x83 0x01
[R:24]	CMD_NEXT	0x1e	[R:30]	CMD_SVCCFG/DEMAND_START	0x83 0x04
[R:29]	CMD_SERVICELIST	0x82	[R:31]	CMD_SVCCFG/AUTO	0x83 0x03
[S:22]	SERVER_EXIT	0xcd	[R:33]	CMD_SVCCFG/STOP	0x83 0x02
[R:34]	CMD_REG_FIND	0xc9	[R:36]	CMD_WINDOW_CLOSE	0x00
[R:37]	CMD_PSLIST	0x24	[S:67]	TOKEN_FIRSTSCREEN	0x74
[S:68]	TOKEN_AUDIO_DATA	0x7a	[S:74]	TOKEN_CAM_DIB	0x78 0x00
[S:73]	TOKEN_TALKCMPLT	0x85	[R:75]	CMD_CAM_ENABLECOMPRESS	0x1b
[S:112]	TOKEN_PSLIST	0x7d	[R:76]	CMD_CAM_DISABLECOMPRESS	0x1c
[S:137]	TOKEN_NEXTSCREEN	0x75	[R:138]	CMD_SCREEN_GET_CLIPBOARD	0x18
[S:15]	TOKEN_REGEDIT	0xc8	[R:144]	CMD_SCREEN_CONTROL	0x14
[R:9]	CMD_WSLIST	0x25	[S:199]	TOKEN_CAM_DIB/COMPRESS	0x78 0x01
[R:11]	CMD_LIST_DRIVE	0x01			

Table 7.2: Learnt message types

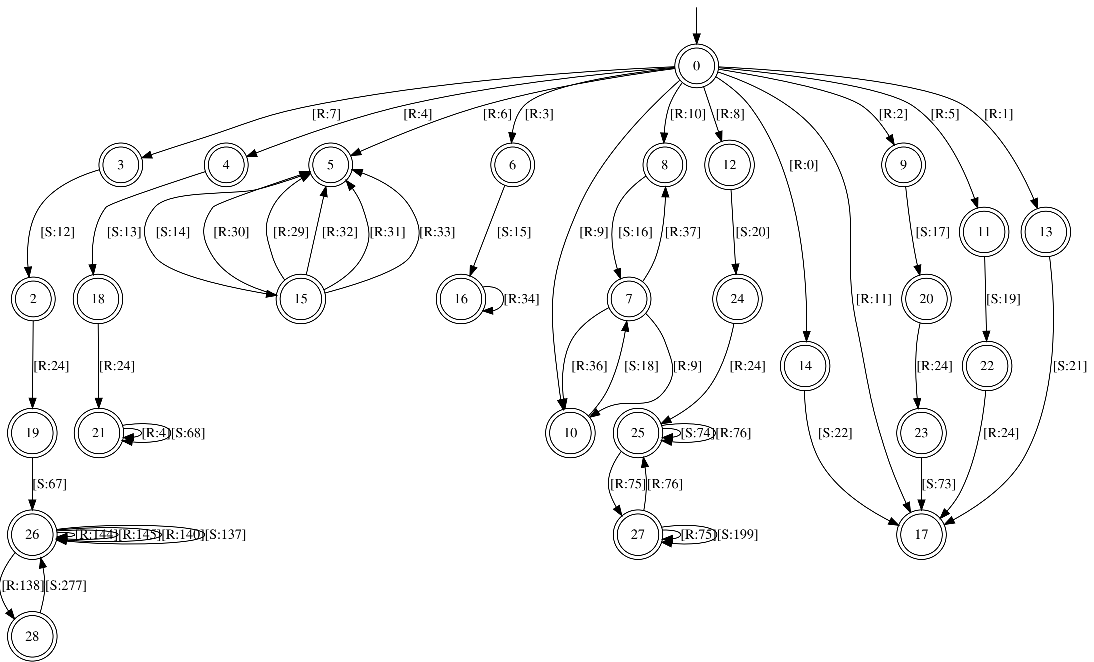


Figure 7.4: Gh0st RAT full state machine learnt by our method

Chapter 8

Conclusions

In this work we present a novel method for inferring the state machine of a protocol implemented by a binary with no a-priori knowledge of the protocol. Our method is based on extended symbolic execution and modified automata learning. The method assumes access to a binary code that implements the protocol.

8.1 Method Validation

We implemented and validated our method on several protocols implementations. When applying the method on SMTP client and Gh0st RAT, we were able to validate the method's techniques to discover message types and to validate sessions (sequences of message types). We faced difficulties with complex program flows due to limitations of the symbolic execution engine. Symbolic execution of complex APIs[1] often results with state explosion or long constraints which take long time to solve. angr tries to solve this problem with *SimProcedures*[2] that emulate complex API operations. However, there are many API calls that are not emulated, or cannot be emulated efficiently. We are certain that further work on symbolic execution of low-level binary code, along with fine tuning of the symbolic execution for a specific program, will improve our method's contribution to practical use cases.

To conclude, we validated our ideas to infer the state machine of a protocol and to uncover its message types. For the validation we used examples from various protocols. Nonetheless, we claim that the method we propose will preform as a good as the symbolic execution engine it relies on. As long as a symbolic execution engine is fully capable of executing the binary code given, our method will be able to extract the protocol and its messages format.

[1] Functions that the program uses and are provided by the operating system or by libraries provided with the operating system

[2] Code snippet that imitates the API functionality

8.2 Future Works

There are several directions in which future works can enhance our method:

1. **Symbolic Execution:** There are some improvements that can be implemented in the symbolic execution part. One of them is merging states that represent the same state of the protocol, in order to relieve the problem of state explosion that is likely to arise when applying our method to complex binaries. As already mentioned, the method may also benefit from the application of different symbolic execution engines for binaries. Further methods for symbolic execution of binaries might be developed in the future.

2. **Message Types Inference:** Several aspects of the message types inference may be improved. First, we introduced a simple approach to extract a predicate to describe a message type. A more sophisticated method can be deployed in a future work and a better representation for message types may be introduced. Second, we introduced a certain algorithm to solve collisions between intersecting message types. A future work may develop a different method to maintain the partition of the messages. For example, a method that is able to merge two different message types may be developed to handle the case where two or more message types are found to serve the same role in the protocol. The method can benefit from such optimization because a smaller set of message types may decrease the running time of the learning process.

3. **Parallelization:** LearnLib [IHS15] learner, which we use in our implementation, generates membership queries in batches. Therefore, it is possible to apply symbolic execution to check every one of these queries in parallel. We did not deal with this optimization, but we do believe it will improve performances.

Bibliography

[ACMN05] Rajeev Alur, Pavol Cerný, P. Madhusudan, and Wonhong Nam. Synthesis of interface specifications for java classes. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 98–109, 2005.

[Ang87] Dana Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In Rance Cleaveland, editor, *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999.

[CBP+11] Chia Yuan Cho, Domagoj Babić, Pongsin Poosankam, Kevin Zhijie Chen, Edward XueJun Wu, and Dawn Song. MACE: Model-inference-Assisted Concolic Exploration for Protocol and Vulnerability Discovery. In *Proceedings of the 20th USENIX Security Symposium*, 8 2011.

[CBSS10] Chia Yuan Cho, Domagoj Babic, Eui Chul Richard Shin, and Dawn Song. Inference and analysis of formal models of botnet command and control protocols. In *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010*, pages 426–439, 2010.

[CGK+18] E.M. Clarke, O. Grumberg, D. Kroening, D. Peled, and H. Veith. *Model Checking – Second Edition.* MIT Press, 2018.

[CGP03] Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Pasareanu. Learning assumptions for compositional verification. In Hubert Garavel and John Hatcliff, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference, TACAS 2003, Held as*

Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings, volume 2619 of *Lecture Notes in Computer Science*, pages 331–346. Springer, 2003.

[Cho78] Tsun S. Chow. Testing software design modeled by finite-state machines. *IEEE Trans. Software Eng.*, 4(3):178–187, 1978.

[CKL04] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[CKW07] Weidong Cui, Jayanthkumar Kannan, and Helen J. Wang. Discoverer: Automatic protocol reverse engineering from network traces. In *Proceedings of the 16th USENIX Security Symposium, Boston, MA, USA, August 6-10, 2007*, 2007.

[CPC+08] Weidong Cui, Marcus Peinado, Karl Chen, Helen J. Wang, and Luis Irún-Briz. Tupni: automatic reverse engineering of input formats. In *Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008*, pages 391–402, 2008.

[CS07] Sagar Chaki and Ofer Strichman. Optimized l*-based assume-guarantee reasoning. In Orna Grumberg and Michael Huth, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007, Proceedings*, volume 4424 of *Lecture Notes in Computer Science*, pages 276–291. Springer, 2007.

[CS13] Juan Caballero and Dawn Song. Automatic protocol reverse-engineering: Message format extraction and field semantics inference. *Computer Networks*, 57(2):451–474, 2013.

[CWKK09] Paolo Milani Comparetti, Gilbert Wondracek, Christopher Krügel, and Engin Kirda. Prospex: Protocol specification extraction. In *30th IEEE Symposium on Security and Privacy (S&P 2009), 17-20 May 2009, Oakland, California, USA*, pages 110–125, 2009.

[CYLS07] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Xiaodong Song. Polyglot: automatic extraction of protocol message format using dynamic binary analysis. In *Proceedings of the 2007 ACM Conference on Computer*

and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007, pages 317–329, 2007.

[EGP08] Michael Emmi, Dimitra Giannakopoulou, and Corina S. Pasareanu. Assume-guarantee verification for interface automata. In *FM 2008: Formal Methods, 15th International Symposium on Formal Methods, Turku, Finland, May 26-30, 2008, Proceedings*, pages 116–131, 2008.

[FvBK+91] Susumu Fujiwara, Gregor von Bochmann, Ferhat Khendek, Mokhtar Amalou, and Abderrazak Ghedamsi. Test selection based on finite state models. *IEEE Trans. Software Eng.*, 17(6):591–603, 1991.

[GGP07] Mihaela Gheorghiu, Dimitra Giannakopoulou, and Corina S. Pasareanu. Refining interface alphabets for compositional verification. In *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007, Proceedings*, pages 292–307, 2007.

[IHS15] Malte Isberner, Falk Howar, and Bernhard Steffen. The open-source learnlib. In Daniel Kroening and Corina S. Păsăreanu, editors, *Computer Aided Verification*, pages 487–495, Cham, 2015. Springer International Publishing.

[JMK22] Natasha Yogananda Jeppu, Tom Melham, and Daniel Kroening. Active learning of abstract system models from traces using model checking. In *Proceedings of the 2022 Conference & Exhibition on Design, Automation & Test in Europe*, DATE '22, page 100–103, Leuven, BEL, 2022. European Design and Automation Association.

[LRL06] Junghee Lim, Thomas W. Reps, and Ben Liblit. Extracting output formats from executables. In *13th Working Conference on Reverse Engineering (WCRE 2006), 23-27 October 2006, Benevento, Italy*, pages 167–178, 2006.

[PGB+08] Corina S. Pasareanu, Dimitra Giannakopoulou, Mihaela Gheorghiu Bobaru, Jamieson M. Cobleigh, and Howard Barringer. Learning to divide and conquer: applying the l* algorithm to automate assume-guarantee reasoning. *Formal Methods Syst. Des.*, 32(3):175–205, 2008.

[PVY02] Doron A. Peled, Moshe Y. Vardi, and Mihalis Yannakakis. Black box checking. *Journal of Automata, Languages and Combinatorics*, 7(2):225–246, 2002.

[SG09] Muzammil Shahbaz and Roland Groz. Inferring mealy machines. In *FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings*, pages 207–222, 2009.

[SWS+16] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Krügel, and Giovanni Vigna. SOK: (state of) the art of war: Offensive techniques in binary analysis. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 138–157. IEEE Computer Society, 2016.

[WCKK08] Gilbert Wondracek, Paolo Milani Comparetti, Christopher Krügel, and Engin Kirda. Automatic network protocol analysis. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008*, 2008.

כיוון שהרצות סימבוליות של תכנית דורשות משאבים ועלולות להימשך זמן ארוך מהרגיל, אנו מציעים שלוש אופטימיזציות במטרה להימנע ככל הניתן מהרצות סימבוליות שאינן נחוצות, וכן על מנת לייעל הרצות סימבוליות הכרחיות. האופטימיזציה הראשונה מתבססת על תכונות של פרוטוקולי תקשורת על מנת להימנע משאילתות שייכות שניתן לדעת מראש שתשובתן היא שלילית. האופטימיזציה השנייה מסתייעת בהרחבה שלנו לשאילתת השייכות על מנת לענות במהירות על שאילתות שקילות במקרים מסוימים. האופטימיזציה השלישית חוסכת הרצות סימבוליות חוזרות במקרה של רצפים המופיעים בתור רישות של רצפים אחרים.

מימשנו את השיטה שלנו בעזרת ספריות קיימות להרצה סימבולית וללמידת אוטומט. לאחר מכן, פנינו להרצת ניסויים של השיטה מול תרחישים אמיתיים. הניסוי העיקרי עליו עבדנו היה ניתוח של פרוטוקול ה-C&C של הנוזקה gh0st. במהלך הניסויים הראנו כי השיטה שלנו מצליחה ללמוד את מכונת המצבים של הפרוטוקול בצורה נכונה. יחד עם זאת, הוכח כי השיטה תלויה מאוד במנוע ההרצה הסימבולית עליו היא מתבססת. על כן, אנו מסכמים כי השיטה שלנו מסוגלת לנתח את פרוטוקול התקשורת של תכנית מסוימת, ככל שמנוע ההרצה הסימבולית מסוגל לבצע הרצה של התכנית.

לאחר מכן, אנו מציעים קירוב לשיטה התיאורטית שהוצעה, כיוון שאין זה מציאותי לממש אורקל מורה העונה על הדרישות במדויק כאשר אין ברשותנו מידע על הפרוטוקול.

תחילה אנו מציעים ייצוג יעיל לסוגי הודעות של הפרוטוקול. במקום לאחסן את כל ההודעות הקיימות עבור כל סוג של הודעה, אנו מציעים לייצג קבוצת הודעות בעזרת פסוק לוגי. פסוק לוגי זה מוגדר על הבתים המרכיבים את ההודעה. ערכו של הפסוק אמת עבור הודעות בעלות מבנה זהה למבנה ההודעות בקבוצה, ושקר עבור כל הודעה אחרת. אנו מציעים שיטה נאיבית לחלץ את הפסוק המתאים לקבוצת הודעות. שיטה זו בונה פסוק המבוסס על הבתים הזהים בתוכנם עבור כל ההודעות בקבוצה. שיטה זה מתאימה לפרוטוקולי תקשורת, כיוון שלרוב סוג ההודעה מסומן על ידי ערך מוסכם בהיסט מסוים של ההודעה ולכן יהיה משותף לכל ההודעות מאותו הסוג.

כיוון שאנו מציעים קירוב לאורקל מורה, אין הכרח שסוגי ההודעות שמתגלים על ידי שאילתות מהווים חלוקה של ההודעות בפרוטוקול לקבוצות זרות כנדרש. לכן אנו מציגים אלגוריתם לפיתרון של התנגשויות בין סוגי הודעות. כלומר, התנגשויות בין פסוקים לוגים שמתארים את סוגי ההודעות. סוגי ההודעות שמתגלות בעזרת שאילתות, נוספות לאלפבית לאחר פיתרון ההתנגשויות.

על מנת לקרב את התשובה לשאילתת שקילות, אנו משתמשים בשיטה דומה לשיטה שהוצעה בעבודה המקורית של L^*. בעבודה המקורית, הוצע לקרב את השאילתא בעזרת אוסף בדיקה. אוסף בדיקה מכיל רצפים מעל האלפבית, כאשר כל רצף נבדק בשאילתת שייכות. בשיטה שלנו, אני מקרבים שאילתת שקילות בעזרת אוסף בדיקה כאשר במקרה של אי-הסכמה בין האוטומט המוצע לבין שאילתת השייכות, נקבע כי האוטומט אינו שקול. כמו כן, אם עבור רצף בדיקה מסוים, שאילתת השייכות מזהה סוגי הודעות היכולים להופיע לאחר הרצף שנבדק, וסוגי הודעות אלה חסרים באלפבית הנוכחי, הקירוב של שאילתת השקילות יזהה מצב זה.

על מנת לקרב שאילתת שייכות, אנו משתמשים בהרצה סימבולית של התכנית הנתונה. את הריצה הסימבולית אנו מבצעים באופן מבוקר, כך שאנו מחלקים אותה לשני שלבים. אנו מכנים את השלב הראשון שלב המעקב כיוון שבמהלכו אנו מחפשים ריצות של התכנית המתאימות לרצף סוגי ההודעות שהתקבל בשאילתת השייכות. השלב השני נקרא שלב הגישוש כיוון שמהלכו אנו מחפשים סוגי הודעות היכולים להופיע לאחר הרצף שנבדק.

במהלך שלב המעקב, אנו מוסיפים לריצה הסימבולית אילוצים המבוססים על הפסוקים הלוגים שמשוייכים לרצף סוגי ההודעות עליו נשאלנו בשאילתא. אילוצים אלה גורמים לריצה הסימבולית לאתר ריצות המתאימות לרצף בלבד. אם בסוף שלב המעקב, נמצאו ריצות המתאימות לרצף, הרי שהרצף מייצג רצף סוגי הודעות אפשרי בתכנית, ועל כן שאילתת השייכות תשיב בחיוב. לעומת זאת, אם לא נמצאו ריצות מתאימות שאילתת השייכות תשיב בשלילה.

שלב הגישוש יורץ רק במקרה בו נקבע שהרצף עליו נשאלנו שייך לפרוטוקול. בשלב זה אנו מאתרים ריצות בהן התכנית שולחת או מקבלת הודעות נוספות. כאשר התכנית שולחת הודעה, אנו נייצר מועמד לסוג הודעה לאחר ניתוח של מצב התכנית בו נשלחה ההודעה. מנגד, עבור הודעות שמתקבלות על ידי התכנית, נדרש פיתרון אחר. הודעות אלה מהוות קלט לתכנית ולכן אין לנו שום מידע לגביהן. בעבודה שלנו אנו מציגים פיתרון חדשני לבעיה זו. הפיתרון שלנו מתבסס על ההנחה שהתכנית מפענחת את ההודעות שמתקבלות וכך אנו מגלים את המבנה המאפיין הודעות אלה. בשני המקרים, אנו מייצרים מועמדים לסוג הודעה, אשר מוכללים באלפבית לאחר פיתרון התנגשויות.

תקציר

תכניות מחשב רבות בימינו עושות שימוש ברשת האינטרנט כדי להעביר מידע בין מחשבים ברחבי העולם. תכניות אלה עושות שימוש בפרוטוקולי תקשורת: אוסף של כללים המגדירים כיצד צד בתקשורת אמור או יכול להגיב בעת ביצוע תקשורת עם מחשבים אחרים. במקרה הנפוץ, פרוטוקול ממומש בשפת תכנות ומתורגם לשפת מכונה על ידי מפתח התכנית. הסקת פרוטוקול מתייחסת לתהליך ההפוך, בו נדרש להסיק את מאפייני הפרוטוקול מתוך תכנית מחשב המשתמשת בו. הסקת פרוטוקול הינה שימושית בתחום של אבטחת מידע. בתחום זה חוקרים מעוניינים פעמים רבות לבחון את פרוטוקול השליטה והבקרה (C&C) של נוזקה שנמצאה במחשב. בנוסף, אפשרי כי חוקרים ירצו לבחון את הפרוטוקול של תוכנה לגיטימית במטרה לאתר פרצות אבטחה ולתקן אותן.

בעבודה זו אנו מציגים שיטה אוטומטית להסקת פרוטוקול תקשורת מתוך תכנית המשתמשת בו. השיטה מחלצת את מכונת המצבים של הפרוטוקול ואת הפורמט של סוגי ההודעות בפרוטוקול. בהגדרת עולם הבעיה, אנו מניחים כי ניתן לסווג את כלל ההודעות הנשלחות והמתקבלות (בנפרד) לקבוצות זרות הנקראות סוגי הודעות. סוג הודעה יכול לייצג הודעות הנשלחות או הודעות המתקבלות. אנו מניחים כי שפת הפרוטוקול מורכבת מרצפים של סוגי הודעות, המייצגים רצפי הודעות אפשריות בפרוטוקול.

השיטה משתמשת בלמידת אוטומט כדי לבנות את מכונת המצבים של הפרוטוקול ובהרצה סימבולית כדי לגלות את סוגי ההודעות בפרוטוקול וכדי לבחון מה הם הרצפים האפשריים של הודעות בפרוטוקול. אנו מניחים שברשותנו קוד בשפת מכונה של התכנית אותה אנו נדרשים לנתח. כמו כן, אנו מניחים שהפרוטוקול שהתכנית מממשת ניתן לתיאור בעזרת אוטומט ושאורך הודעה בפרוטוקול הוא מוגבל. מנגד, בעבודה זו אין אנו מניחים גישה למימוש נוסף של הפרוטוקול, הקלטות של תעבורת רשת או תכנית אחרת המשתמשת בפרוטוקול וזמינה לצורך תשאול.

ראשית, אנו מציגים שיטה תיאורטית ללמידת אוטומט אשר מותאמת ללמידת מכונת המצבים של פרוטוקול תקשורת לא ידוע. השיטה מבוססת על אלגוריתם L^* ללמידת אוטומט שהוצג בעבודה קודמת. אנחנו מבצעים התאמה שמאפשרת להתמודד עם ההנחה שהאלפבית של מכונת המצבים, קרי סוגי ההודעות בפרוטוקול, אינו ידוע בתחילת תהליך הלמידה. בדומה לאלגוריתם L^*, השיטה שאנחנו מציגים מניחה את קיומו של אורקל מורה המסוגל לענות על שני סוגים של שאילתות. השאילתא הראשונה נקראת שאילתת שייכות, ומטרתה לבדוק האם רצף סוגי הודעות מתאר רצף המתאים לפרוטוקול, וכן לספק מידע על סוגי ההודעות היכולים להופיע לאחר הרצף הנבדק. השאילתא השנייה נקראת שאילתת שקילות, ומטרתה לבדוק האם אוטומט מסוים מתאר נכונה את הפרוטוקול וכן האם חסרים סוגי הודעות באלפבית שנצבר עד כה. סוגי הודעות שנחשפים בעזרת שאילתות נוספים לאלפבית הידוע עד כה.

המחקר בוצע בהנחייתם של פרופסור ארנה גרומברג ודוקטור גבי נקיבלי בפקולטה למדעי המחשב.

חלק מן התוצאות בחיבור זה פורסמו כמאמרים מאת המחבר ושותפיו למחקר בכנסים ובכתבי-עת במהלך תקופת מחקר המאסטר של המחבר, אשר גרסאותיהם העדכניות ביותר הינן:

Ron Marcovich and Gabi Nakibly. Automatic protocol reverse engineering. In *Black Hat USA*, 2022.

תודות

אני רוצה להודות למנחים שלי, ארנה וגבי, על הזמן, התמיכה וההכוונה המקצועית והפורה שנתנו לי.

אני רוצה להודות בנוסף למשפחתי היקרה, שהיו לי גב במשך כל לימודיי בטכניון ובפרט גם במהלך תואר המסטר.

כמו כן, אני מבקש להודות לחבריי היקרים, שהיו שם לצידי תמיד, לתמוך, לייעץ ולכוון.

ללירון אהובתי, על התמיכה, ההכלה וההבנה.

אני מודה לטכניון על התמיכה הכספית הנדיבה בהשתלמותי.

הסקת פרוטוקול תקשורת מקובץ הרצה בעזרת הרצה סימבולית ולמידת אוטומט

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר
מגיסטר למדעים במדעי המחשב

רון מרקוביץ

הוגש לסנט הטכניון — מכון טכנולוגי לישראל
תמוז התשפ"ב חיפה יולי 2022

הסקת פרוטוקול תקשורת מקובץ הרצה
בעזרת הרצה סימבולית ולמידת אוטומט

רון מרקוביץ