A symbolic-based passive testing approach to detect vulnerabilities in networking systems

Pramila Mouttappa

▶ **To cite this version:**

Pramila Mouttappa. A symbolic-based passive testing approach to detect vulnerabilities in networking systems. Other [cs.OH]. Institut National des Télécommunications, 2013. English. NNT : 2013TELE0023 . tel-01017860

HAL Id: tel-01017860
https://theses.hal.science/tel-01017860

Submitted on 3 Jul 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

DOCTORAT EN CO-ACCREDITATION
TELECOM SUDPARIS ET L'UNIVERSITE EVRY VAL D'ESSONNE

Spécialité : Informatique

Ecole doctorale : Sciences et Ingénierie

Présentée par
Pramila MOUTTAPPA

Pour obtenir le grade de
DOCTEUR DE TELECOM SUDPARIS

A Symbolic-based Passive Testing approach to detect Vulnerabilities in Networking Systems

Thèse dirigée par Ana CAVALLI

Soutenue le 16 Décembre 2013 devant le jury composé de :

Rapporteurs :	Fatiha ZAÏDI	-	Université Paris Sud XI
	Franz WOTAWA	-	Graz University of Technology
Directrice de thèse :	Ana CAVALLI	-	Mines Télécom-Télécom SudParis
Examinateurs :	Stéphane MAAG	-	Mines Télécom-Télécom SudParis
	Sébastien TIXEUIL	-	Université Pierre et Marie Curie
	Michel BOURDELLES	-	Thales Communications

Thèse N° 2013TELE0023

Abstract

Due to the increasing complexity of reactive systems, testing has become an important part in the process of the development of such systems. Conformance testing with formal methods refers to checking functional correctness, by means of testing, of a black-box system under test with respect to a formal system specification, i.e., a specification given in a language with a formal semantics. In this aspect, *passive* testing techniques are used when the implementation under test cannot be disturbed or the system interface is not provided. Passive testing techniques are based on the observation and verification of properties on the behavior of a system without interfering with its normal operation, it also helps to observe abnormal behavior in the implementation under test on the basis of observing any deviation from the predefined behavior.

The main objective of this thesis, is to present a new approach to perform passive testing based on the analysis of the control and data part of the system under test. During the last decades, many theories and tools have been developed to perform conformance testing. However, in these theories, the specifications or properties of reactive systems are often modeled by different variants of Labeled Transition Systems (LTS). However, these methodologies do not explicitly take into account the system's data, since the underlying model of LTS are not able to do that. Hence, it is mandatory to enumerate the values of the data before modeling the system. This often results in the state-space explosion problem. To overcome this limitation, we have studied a model called Input-Output Symbolic Transition Systems (IOSTS) which explicitly includes all the data of a reactive system.

Many passive testing techniques consider only the control part of the system and neglect data, or are confronted with an overwhelming amount of data values to process. In our approach, we consider control and data parts by integrating the concepts of symbolic execution and we improve trace analysis by introducing trace slicing techniques. Properties are described using Input Output Symbolic Transition Systems (IOSTSs) and we illustrate in our approach how they can be tested on real execution traces optimizing the trace analysis. These properties can be designed to test the functional conformance of a protocol as well as security properties.

In addition to the theoretical approach, we have developed a software tool that implements the algorithms presented in this paper. Finally, as a proof of concept of our approach and tool we have applied the techniques to two real-life case studies: the SIP and Bluetooth protocol.

Acknowledgements

It gives me great pleasure in expressing my gratitude to all those people who have supported me and had their contributions in making this thesis possible. First and foremost, I would like to thank, Professor Ana Cavalli, for her valuable time, guidance and funding to make my Ph.D. experience productive. I owe my deepest gratitude to my supervisor, Professor Stéphane Maag, for the patient guidance, encouragement and advice he has provided throughout my time as his student. I am really fortunate to have a supervisor who cared so much about my work, and who responded to my questions and query so promptly.

My sincere thanks must also go to the members of my thesis advisory and exam committee: Fatiha Zaïdi, Franz Wotawa, Michel Bourdelles and Sébastien Tixeuil. They generously gave their time to offer me valuable comments toward improving my work. It is no easy task, reviewing a thesis, and I am grateful for their thoughtful and detailed comments

I would like to extend my thanks to Prof. Nina Yevtushenko, Prof. Anis Laouiti and Dr. Alessandra Bagnato for their valuable comments, moral support and friendly advices.

My sincere thanks to Mme. Brigitte Laurent, for all the support she had given me in completing the non-technical part of my work.

My special thanks to my LOR friends: Felipe, Anderson, Khalifa, Xiaoping, Natalia, Jorge, Olga, Jimmy, Joao and Mohammed. They have been a source of friendship as well as contributed immensely to my personal and professional time.

I owe a lot to my parents, Savariraj and Mary, who encouraged and helped me at every stage of my personal and academic life, and longed to see this achievement come true. I also want to thank my in-laws for their unconditional support and confidence they had given me all these days. I am grateful to my sisters Sophia and Sheela and specially my brother Joe, who boosted me morally and provided me great information resources. My sincere thanks to my brother-in-law and sister-in-law for their valuable encouragement and support. Finally, I would like to dedicate my thesis to my lovable husband, David and my son Mervyn, who have always stood by me through the good and bad times. Thank you for your love, support, and unwavering belief in me.

Thanks a lot.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 General Context

The advent of high-performance networks has led to the development of a new set of technologies and new communication protocols and services for the systems. Reliable communication can be ensured only if the protocol implementations used within each system conform to their specifications. Although the application of these protocols and services in real-time may be satisfiable, still there can be flaws in the implementation which could be exploited by an attacker to compromise the network. Testing is then an activity in which the testers try to conform or guarantee that the protocol or service processes without fault or at least meets the requirements, but also to check security properties.

Software testing [Beizer 1990] distinguishes structural and functional testing. As shown in Figure 1.1 structural testing or white-box testing is based on the analysis of the internal structure of an implementation, while functional testing, also called black-box testing consists of checking whether an implementation of the software or hardware satisfies its specification without making any reference to its internal structures.

In this work, we are interested in so-called conformance testing where the aim is to check conformance of the implementation under test (IUT) to a set of desired properties and behaves in accordance with some predefined requirements. In general, conformance testing is understood as functional black-box testing, i.e., an IUT is given as a black-box and its functional behavior is defined in terms of inputs to and corresponding outputs

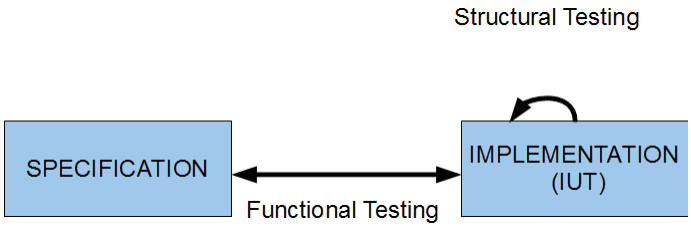


FIGURE 1.1: Functional and Structural testing

from the IUT. In this work we only take into account the black box conformance testing, where conformance testing refers to the activity of showing that the IUT executions follow all the properties defined in the corresponding specification. In order to say it informaly, if it deviates from the expected behavior we would like to see if it matches with any attack scenario.

Testing is generally performed based on the formal models. Conformance testing with formal methods refers to checking functional correctness, by means of testing, of a black-box system under test with respect to a formal system specification, i.e., a specification given in a language with a formal semantics. These formal specifications enables the analysis of systems and the reasoning about them with mathematical precision which aids testing. Commonly, two main classes of formal testing techniques are applied to check the conformance of protocols and software: active and passive testing (monitoring) techniques.

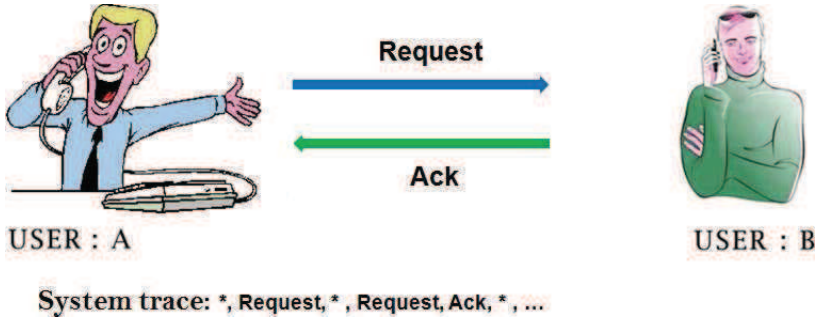
Most of the formal testing [Bentakouk et al. 2011, Gaston et al. 2006, Martijn et al. 2007, Rusu et al. 2005, Weiglhofer et al. 2010] approaches are said to be active which means that test cases are extracted from the specification and experimented on the implementation to conclude whether a test relation is satisfied. Active methods require to deploy a pervasive test environment (test architecture) to execute test cases and to observe implementation reactions. They may also interrupt the system normal functioning arbitrarily, for example by resetting it after each test case execution. However, when a system is deployed in an integrated environment, it becomes quite difficult to access it. Moreover, active methods may disturb the natural operation of the implementation under test. So, these ones may not be suitable in regards to the tested system. Passive testing represents another interesting alternative, which offers several advantages, e.g., to not disturb the system while testing.

Since the early 1980s there has been ongoing interest in passive testing, technically defined as a testing activity in which a tester does not influence (stimulate) an IUT in any way, it does not apply any test stimuli. Rather, the usual approach of passive testing consists on recording the trace (i.e., sequence of exchange of messages) produced by the implementation under test and mapped to the property to be tested or specification if it exists. Passive testing helps to observe abnormal behavior in the implementation under test on the basis of observing any deviation from the predefined behavior. This deviation can also sometimes match with certain attack patterns. Moreover, it is usually considered that the implementation is taken without knowledge of its internal state that is to say that we do not consider the event trace record to start from the initial state or a predefined state.

It is worth to point out that our passive testing approach is somehow related to runtime verification [Falcone et al. 2013] since they share similar objectives and procedures. Runtime verification techniques are used to dynamically verify the behavior observed in the target system with respect to some requirements. These requirements are often formally specified and verification techniques are used both to check that the observed behavior matches the specified properties and to explicitly recognize undesirable behaviors in the target system.

1.2 Motivation and Objectives

Reactive systems [Harel and Pnueli 1985] interact permanently with its environment by continuously exchanging information. Many systems in the real world can be considered as reactive: communication protocols, embedded systems, smart cards, etc. It is essential to realize how crucial reactive systems can be, and how important it is to perform testing or other validation techniques on them. For example, in network protocols, communication is established by exchanging messages between the entities, where each entity can independently act as an emitter or receiver. As the systems evolve, messages become richer with data values. These messages are defined as control and data portions based on the function of the protocol. Many works on passive testing [Andrés et al. 2008; 2012, Cavalli and Tabourier 1999, Lee et al. 1997] are focused only on checking the control portion of the protocol without taking into account the data part. However, it may result in producing false positive verdicts as illustrated below with an example.



(a) Control portion

(b) Control and data portion

FIGURE 1.2: Property - Request followed by Acknowledgment response

Let us consider a property, where a user A sends a *request* say, *Request(from:aron@ti.com, to:ben@info.com)* and expects for an *acknowledgment* response from user B, *Ack (from:aron@ti.com, to:ben@info.com)*. In Figure 1.2(a), the control portions are alone monitored. We observe from the system trace that the property of having a request followed by an acknowledgment is satisfied, hence the verdict for the trace results *pass*. But if we include the data portions of the messages then the verdict for the trace in Figure 1.2(b) results *fail* or *inconclusive*. It fails because there is no acknowledgment response from user B as required by the property where we could only see the acknowledgment response from a different user, *Ack(from:carl@pouf.com, to:ben@info.com)* and the verdict is *inconclusive* if the length of the trace is not sufficient to prove the invariant. Hence the data relationship between messages must be given importance to avoid such false positive verdicts.

In order to overcome the above problem, the data part of the protocols must also be

taken into account. This led to the development of specifications as extended finite state machines (EFSMs). In an EFSM, the transition can be expressed by an "if condition". If the trigger conditions are satisfied then the transition from one state to another is performed and the specified data operations are executed. However, applying the EFSM in passive testing requires the enumeration of data values which is a huge, time and space consuming activity. To overcome the problem of data enumeration for testing large reactive systems, we have adopted a symbolic approach, by modeling the properties of the system using an IOSTS formalism. In the IOSTS formalism, we represent the data in the form of *symbols* rather than concrete data values, which helps to avoid data enumeration.

1.3 Contributions

In the previous section we briefly described the current testing methods and its problems. In order to easily understand the contributions, we provide a short outline of our approach. In this work, the passive testing of a property on a real execution trace integrates two important techniques: symbolic execution of an Input-Output Symbolic Transition Systems and a parametric trace slicing approach. *Input-Output Symbolic Transition Systems* (IOSTS) are commonly used for formally modeling communicating systems interacting with their environment. In IOSTS, the parameters and variable values are represented by symbolic values (called fresh variables) instead of concrete ones. Enumeration of data values is therefore not required. This allows to reduce the huge amount of data values commonly applied in many passive testing approaches. In [Mouttappa et al. 2012b] we proposed this approach to monitor the conformance property alone and then as an improvement in [Mouttappa et al. 2012a] we monitored a property and an attack scenario. A more extended version of our approach is provided in the Journal article [Mouttappa et al. 2013b] to specify the protocol properties as well as several kinds of attack patterns. This helps to detect conformance as well as security anomalies.

Besides, a *Parametric trace slicing* technique [Chen and Rosu 2009] is used for trace analysis. Trace analysis plays a very important part in passive testing. A parametric trace is defined as a trace containing events with parameters that have been bound to a concrete data value (i.e., valuation) and parametric trace slicing is defined as a technique

to slice (or cut) the real protocol execution trace into various slices based on the valuation. Each slice corresponds to a particular valuation. These trace slices merged together constitutes the execution trace. We then apply the symbolic execution of our properties on the trace slices to provide a test verdict *Pass/Fail/Attack-Pass/Inconclusive*. The proposed symbolic passive testing approach was implemented in a tool called TestSym-P and applied to two different protocols: Session Initiation Protocol (SIP) and Bluetooth Protocol.

1.4 Thesis plan

This manuscript is organized as follows:

1. Chapter 2 presents the state of the art of conformance testing techniques. We begin with the general concepts of conformance testing and then present the different testing families, i.e. active and passive testing, as well as the different formalisms that have been identified to perform conformance testing. In addition, we also provide a general overview and some related works on the symbolic execution and the parametric trace slicing technique which have been used in our passive testing methodology.

2. Chapter 3 contains our main contribution. In our work we integrate two main techniques: *symbolic execution* and *parametric trace slicing*. First, we define the Input-Output Symbolic Transition Systems (IOSTSs) formalism that has been adopted to define the protocol behaviors/attacks. Then we detail how this IOSTS can be symbolically executed to obtain the tree-like structure, in which the branches correspond to the property/attack scenario we are interested in. Then, we define the parametric trace slicing technique that was used for the trace analysis. We describe the algorithm for the evaluation of the traces against the property/attack. Finally, we present the complexity of the described methodology and also compare with other passive testing techniques. The different steps in our methodology are explained taking a SIP property as an example.

3. Chapter 4 presents the application of the symbolic passive testing methodology to two illustrative use cases: Session Initiation Protocol (SIP) and Bluetooth protocol. The experiments and results obtained are detailed in this section.

4. The thesis ends with the Chapter 5, where we summarize our contributions in the field of passive testing and also present some perspectives and possible future directions to extend our work.

Chapter 2

State of the Art

Contents

2.1 Formal Testing

Testing is a way to check the correctness of a system implementation by means of experimenting with it. In general, tests are applied to the implementation under test and based on observations made during the execution of the tests, a verdict is given. The correctness of this verdict is based on the system specification. The main intent of system test is to find defects and correct them before go-live. Thus testing technique improves the confidence of the system quality.

Formal specification based testing is dependent on three key concepts: *implementation under test (IUT), specification and conformance*. The increasing usability of formal methods in the software/hardware development processes, as well as new developments in the theory of testing, have influenced the evolution of testing technology. This led to a joint project between the International Organization for Standardization (ISO) and

the International Telecommunication Union (ITU) on "Formal Methods in Conformance Testing" (FMCT) (ISO/IEC, 1996). The main objective of this project was to establish a theory and framework which may be used to assess conformance of an implementation to the behavior specified in a formal description. A more detailed definition of conformance is provided in [Tretmans 1994], and we recall few points here.

1. *Implementation Under Test (IUT)* is the basic element in testing. A implementation can be any reactive system i.e., an embedded system, a communication protocol, or any control system.

2. *Specification* describes the correct or expected behavior of the IUT. In formal testing the specification is expressed in some formal language, i.e., a language with a formal syntax and semantics. The set of all valid expressions (representing the property/behavior) represented in this language, be denoted by $SPEC$. If $s \in SPEC$, then in testing our goal is to check whether the behavior of the IUT conforms to s.

3. *Conformance* is a way to check whether the IUT conforms to the specification. This can be expressed as,

   $conform - to \subseteq IMPS \times SPEC$ where,

   -$IMPS$ is the set of all the implementations and

   -$SPEC$ set of all specifications (representing valid expressions).

   Conformance can be defined as a relation between the observable behavior of the IUT and the behavior of the corresponding model, which serves as system specification. An IUT conforms to its specification if both, the IUT and the specification, show the same behavior. Conformance testing belongs to the category of functional testing approaches. An IUT is solely tested according to its specification. The internal structure of the IUT (the code) is not known. Hence, in conformance testing, the IUT is a black-box. Only the observable behavior of the IUT, the interactions of the IUT with its environment, is testable.

Actually, there are three distinct ways of applying the conformance testing : through a *black-box*, a *white-box* or a *gray-box* approach.

1. *Black-box* testing, also called *functional* testing, is unaware of the internal structure of the IUT, as well as, it only observes the exchanged inputs and outputs between

the tester and the IUT without considering the internal actions. The verdict issued is based on the analysis of the observed events.

2. Whereas, in *White-box* or *structural* testing, the internal structure of the IUT is known (i.e., knowledge of the implementation code), and in addition to the observed events the internal actions are also taken into account.

3. *Gray-box* testing can be considered as an intermediate approach between the black-box and white-box testing. It is a technique to test the application with limited knowledge of the internal workings of an application. Unlike black box testing, where the tester only tests the application's user interface, in gray box testing, the tester has access to design documents and the database.

Our approach mainly concentrates on *Protocol conformance testing*. Protocol conformance testing is the process of testing the extent to which implementation of protocol entities adhere to the requirements stated in the relevant standard or specification. Similar to the above stated general conformance testing, this is a type of black-box testing or functional testing, where the evaluation is done based on the observable behavior (inputs and output messages) of the IUT. Further, conformance testing can be broadly classified into two main testing families: the active testing and the passive testing. Each of these two families encompasses various approaches, and each approach contains different techniques.

2.1.1 Active testing

In this family of conformance testing, the tester tries to show the conformance relation by executing a set of test scenarios on an implementation under test (IUT) and verifies whether its behavior matches with the specified requirements. In this type of test, the tester interacts directly with the IUT via its interfaces (external). According to the testing community, it is referred as black-box testing to qualify a test with no information about the internal structure of the implementation.

The Figure 2.1 depicts the general active testing architecture.

In general the following steps are followed for an active testing approach:

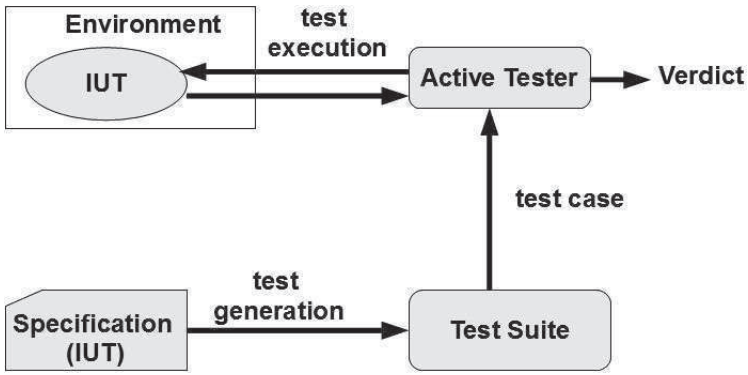


FIGURE 2.1: Active testing approach

1. generate (automatically) and apply (send) stimuli, or test input data, in order to control an IUT (mainly the system behavior);
2. observe phenomena as they appear under the influence of applied stimuli;
3. analyze the relation between observed phenomena and some references (such as a pre-computed, intended behavior);
4. decide on a suitable conformance verdict, which expresses the analysis result.

2.1.2 Passive testing

Passive testing consists in analyzing the traces (i.e., the input and output events) recorded from the IUT and trying to find a fault by comparing these traces with either the complete specification or with some specific requirements (or properties) during normal runtime. As the name implies, it does not disturb the natural run-time of an IUT. It is sometimes also referred to as monitoring. The record of the event observation (input/output) is called an event trace. This event trace will be verified against the specification (or requirements) in order to determine the conformance relation between the implementation and the specification. Then based on the result of that relation a final verdict is produced. We can distinguish two different approaches: *Online* and *Offline*. In the former, the passive tester tries to detect a fault during the execution of the system [Halle and Villemaire 2009, Simmonds 2011], where as, in the latter, the evaluation of the system is done by collecting the recorded traces [Alcalde et al. 2004, Andrés et al. 2008, Che et al. 2012, Lee et al. 1997]. Figure 2.2 shows the general passive testing architecture.

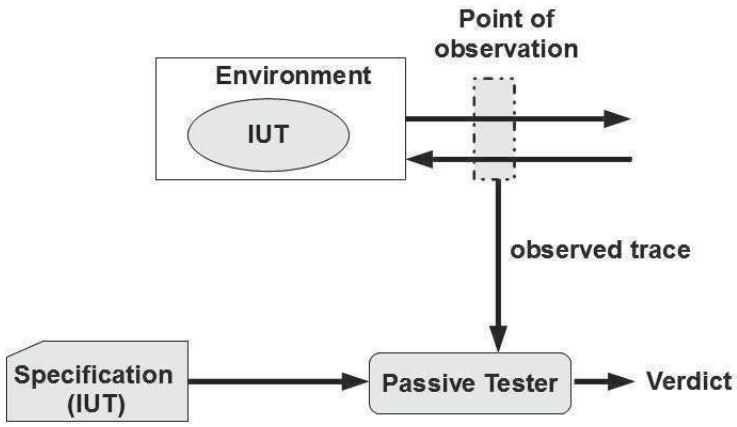


FIGURE 2.2: General passive testing approach

The objective of conformance testing is to check that an implementation performs all what is described in its specification or requirement (that is supposed to be correct). In order to minimize wrong interpretations of the specification it is preferable to describe them with the help of a formal language or formal model. According to the literature, we find that most of the passive testing approaches are based on such formal models or specifications.

Most of the formal testing [Bentakouk et al. 2011, Gaston et al. 2006, Martijn et al. 2007, Rusu et al. 2005, Weiglhofer et al. 2010] approaches consist in the generation of test cases that are applied to the implementation in order to check its correctness with respect to a specification. But, this is not always possible for large systems that are running continuously and cannot be shutdown or interrupted for a long period of time and also when direct interfaces are not provided. Indeed, interfering with such systems can result in misbehavior of the system. In these situations, there is a particular interest in using other types of validation techniques such as passive testing.

There is another branch of work which got importance in last years called *Model based testing* which can be considered as formal, specification based, **active**, black-box, functionality testing [Tretmans 2008]. The testing is said to be active here because the tester controls and observes the IUT in an active way by giving stimuli and triggers to the IUT, and observing its responses, as opposed to passive testing or monitoring. Few notable models used for model-based testing like Labeled Transition Systems (LTS) and its extensions and other important formalisms adopted for conformance testing are discussed in this chapter.

2.1.2.1 Labeled Transition Systems (LTS)

Most of the classical state-oriented testing approaches are based on simple machine models such as Labeled Transition Systems (LTS), in which data is represented by concrete values [Frantzen et al. 2005]. A labeled transition system is a structure consisting of states with transitions, labeled with actions, between them. The states model the system states; the labeled transitions model the actions that a system can perform.

Definition 2.1. **(LTS)** A Labeled Transition System is a tuple $\langle S, s_0, \Sigma, \rightarrow \rangle$ where,
- S is a (possibly infinite) set of states
- $s_0 \in S$ is the initial state
- Σ is a (possibly infinite) set of action labels
- $\rightarrow$ is the transition relation

Nevertheless, the labeled transition system defines the possible sequences of interactions that a system may have with its environment, but still these interactions are considered to be abstract. That is, there is no fundamental difference between the controllable (input) and observable (output) actions. However, this difference has a fundamental role in testing. Many works had been done in this direction, for example, Input-Output Automata (IOA) was proposed in [Lynch and Tuttle 1989], Input-Output State Machines (IOSM) [Phalippou 1994], Input-Output (Labeled) Transition Systems (IO(L)TS) [Jeron 2004, Tretmans 1994].

2.1.2.2 Input-Output Labeled Transition Systems (IOLTS)

An IOLTS consists of states and transitions labeled with actions between them. The states represent the states of the modeling system and the actions on the transitions model the actions which can be performed by the system. The Input-Output Labeled Transition Systems (IOLTS) is a variant of LTS, where the alphabet of observable actions is separated into two disjoint alphabets of input and output action. As in [Tretmans 1996], we also use the distinction between *inputs* and *ouputs*. Outputs are identified by the symbol !, which denotes the values sent from the system to the environment. Inputs, are identified by the symbol ?, corresponds to the values sent from the environment to the system.

Definition 2.2. **(IOLTS)** An Input-Output Labeled Transition System is a tuple $\langle S, s_0, \Sigma, \rightarrow \rangle$ where,
- S is a countable, non-empty set of states
- $s_0 \in S$ is the initial state
- $\Sigma = \Sigma^? \cup \Sigma^!$ is a countable alphabet of actions which consists of two disjoint alphabets of input $\Sigma^?$ and output $\Sigma^!$ actions
- $\rightarrow \subset S \times \Sigma \times S$ is the transition relation

The reactive systems often manipulate complex data structure. Moreover, they can exchange their data using input and output actions. However, the underlying model of (IO)LTS does not allow to explicitly describe the data of these systems. Therefore, in order to model a specification of such reactive system with (IO)LTS, it is necessary to enumerate values of each datum used by this system. This leads to an explosion of the state space. To overcome these problems, we introduce the *Symbolic Transition Systems* (STS).

2.1.2.3 Symbolic Transition Systems (STS)

The disadvantage of an LTS based transition system is the limited possibility of modeling data values and variables. For modeling data values and variables with an LTS all data is encoded in actions representing one concrete value. This mapping then leads to a state space explosion up to an LTS with infinitely many states. Another disadvantage of this method is that all additional information about the data, such as constraints, is lost. In order to model data values and variables without the state space explosion, and to maintain the additional data information, Symbolic Transition Systems (STS) have been introduced in [Henzinger et al. 1999]. Instead of mapping actions to concrete values the data in STS is treated symbolically.

Definition 2.3. **(STS)** A Symbolic Transition System is a tuple $\langle \mathcal{V}, \theta, \Sigma, T \rangle$ where,
- $\mathcal{V} = \langle v_1, v_2, \cdots, v_n \rangle$ is a tuple of variables
- θ is a predicate on V defining the initial condition on the variables
- Σ is a finite alphabet of actions and
- T is a finite set of symbolic transitions. Each symbolic transition is a tuple $t = \langle \sigma_t, G_t, A_t \rangle$ consisting of :
- $\sigma_t \in \Sigma$ is the action of t

- G_t is the predicate on $\mathcal{V}$ which guards the t
- A_t is the assignment function of t

In our approach, we define an *Input-Output Symbolic Transition System (IOSTS)* which is similar to the STS to model a specification behavior (property/attack). The main difference between IOSTS and IOLTS is the fact that IOSTS are a representation on a symbolic level, whereas IOLTS deal with concrete data values. This symbolic representation is enabled by enriching IOSTS with variables and parameters and has the advantage to avoid the state space explosion problem [Rusu et al. 2000]. But in our approach we adapt the IOSTS formalism to model a property/attack and perform the conformance by passive testing. A more detailed explanation of the syntax and semantics of IOSTS is provided in Chapter 3 of this thesis.

2.1.2.4 Finite State Machines (FSM)

Finite State Machine representation (FSM) which is also called automata has been widely used in system specification of various areas, like network protocols, high level software design, real-time reactive systems, etc. Usually, execution traces of the implementation are compared with the specification to detect faults in the implementation [Benharref et al. 2007, Cavalli and Tabourier 1999, Lee et al. 1997].

Definition 2.4. **(FSM)** An FSM is a 6-tuple $\langle I, O, S, s_0, T, \lambda \rangle$ where,

- I is a finite non-empty set of input symbols
- O is a finite set of output symbols
- S is a finite non-empty set of states
- s_0 is the initial state.
- $T : S \times I \rightarrow S$ is a transition function that brings the system into the state $s_j$when reading the event e in the state s_i $((s_i, e) \in S \times I)$
- $\lambda : S \times I \rightarrow O$ is a transduction function that produces the output o when reading the event e in the state s_i $((s_i, e) \in S \times I)$

So, when the machine is in the state s and receives the input symbol a then it moves to the state defined by $T(s, a)$ producing the output described by $\lambda(s, a)$. Although, the FSM has been used to model simple systems they quickly become complex or unpractical to use for complex systems. In addition, it has the provision to model only the control portion of a protocol.

2.1.2.5 Extended Finite State Machines (EFSM)

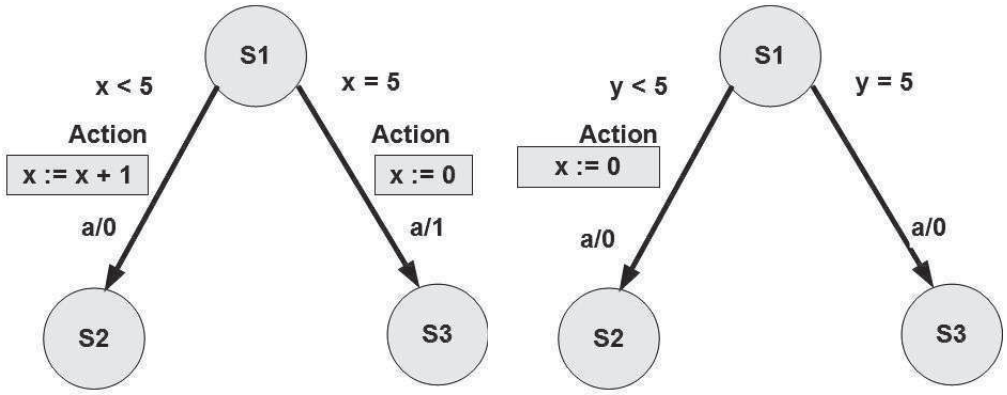
Real life protocols are too complex to be modeled with a simple Finite State Machine. In order to specify a real protocol, we then use the Extended Finite State Machine (EFSM) [Lee and Yannakakis 1996] formalism (i.e there will be internal variables and parameters).

Definition 2.5. **(EFSM)** An EFSM is a 6-tuple $\langle I, O, S, s_0, T, \vec{x} \rangle$ where,
- I is a finite non-empty set of input symbols
- O is a finite set of output symbols
- S is a finite non-empty set of states
- $s_0 \in S$ is the initial state.
- T is a finite set of transitions
- $\vec{x}$ is a variable vector

Each transition $t \in T$ is a 6-tuple $t = \langle s_t, f_t, i_t, o_t, P_t, A_t \rangle$ where,
- s_t is a starting state
- f_t is an ending state
- i_t is an input symbol
- o_t is an output symbol
- $P_t(\vec{x})$ is a predicate on the variable values
- $A_t(\vec{x})$ is an action on the variable values

An extended finite state machine differs from the traditional finite state machine in its definitions of the transitions. In a conventional FSM, the transition is associated with a set of input Boolean conditions and a set of output Boolean operations. In an EFSM model, the transition can be expressed by an *if statement* consisting of a trigger condition and a set of data operations. When the trigger condition is satisfied, the transition is fired, bringing the machine from the current state to the next state and performing the specified data operations. In general, the trigger conditions and the data operations may depend on the primary inputs as well as the data variables. EFSM-based testing considers the observable (input/output events) behavior of the model as the *control portion* and the variable and parameter values as *data portion*. An EFSM can be converted to an equivalent FSM for testing, but it might result in *state explosion problem* [Hong et al. 2002].

Passive testing by value determination Passive testing using EFSM, mainly concentrates on checking the correctness of event sequences (appearing in the collected trace), but it must also consider the variables and the parameter values. In [Cavalli and Tabourier 1999], the authors have proposed the first passive testing method based on deducing the variable and parameter values from an event trace with EFSM specifications. Figure 2.3(a) shows an example of this deduction process, where there are two possible transitions from state $s1$ upon receipt of input a, depending on the current value of variable x.



(a) Value determination (b) Loss of data problem

FIGURE 2.3: Example of value determination approach and the associated problem

Let us assume that the current state $s1$ is known, but that the current value of variable x is unknown. If the next I/O couple in the trace is $a/1$, we can deduce that, after the transition, the machine will be in state $s3$, and also that variable $x = 0$. Now the predicates can be evaluated having known the value of a variable, and will possibly be modified depending on actions.

According to this approach, a transition is said to be fireable if:
- the input/output of the trace matches the label of the transition; and
- either the predicate is true, or it cannot be evaluated (due to unknown variable values).

There can also be a problem in this type of approach when some variables have unknown values, since for some transitions, variables whose values had already been determined may become unknown again. This is defined as *loss of value* problem illustrated in Figure 2.3(b). Here, we assume that the value of variable x has already been determined say, $x = 3$ and that the value of y is still unknown. In all cases, both the

transitions $s1 \rightarrow s2$ and $s1 \rightarrow s3$ are fireable because the observable behavior is the same for both branches $a/0$, and in this case, since the two transitions give different values to x, x becomes *undefined*.

The proposed algorithm constitutes two important phases: *the homing phase* and *fault detection phase*. In the homing phase the following rules are considered:
- For a given I/O couple, if several transitions are possible resulting in different values for a variable, then the variable becomes *undefined*.
- If the predicate holds some undefined or unknown variable value then the predicate is ignored and only the I/O observations are used as a deciding factor for the execution of the transition.

In the fault detection phase, the conformance of the remaining trace with respect to the specification is performed with the intention of finding some faults in the I/O behavior.

Passive testing by interval determination In the previous approach, the algorithm suffers from an information loss problem. To overcome this issue, a more efficient passive testing approach for fault detection was proposed in [Lee et al. 2002]. Three main concepts are used for value determination.

1. **Intervals** are used to denote the integer variable values (i.e., the possible integer variable values). Using intervals, a variable v whose value is between two integers a and b has an interval $R(v) = [a, b]$, which shows $a \leq v \leq b$. Intervals are used to determine the satisfaction of predicates later in the evaluation. If the value of v is known to a, the interval is given by $R(v) = [a, a]$. The variable v is said **decided**.

2. The **Assertions**, $assert(\vec{x})$, records the possible constraints on variables. These constraints can be obtained from either predicates or actions. According to the transition rule, a transition is fired, if the predicate of this transition is assumed to be true and it is added to $assert(\vec{x})$. Assignments with undecided variables on the right side should also be included into $assert(\vec{x})$ as well as predicates.

3. **Candidate Configuration Sets (CCS)** are used to represent possible statuses of the machine. A CCS is a three tuple $\langle s, R(\vec{x}), assert(\vec{x})\rangle$ where,
   - s is the current specification state

- $R(\vec{x})$ is the set of intervals

- $assert(\vec{x})$ is an assertion on $\vec{x}$

Each candidate is evaluated according to the following rule:
- a transition with event e is possible from state s.
- if the predicate can be evaluated, the evaluation of the predicate holds, else it needs to be consistent with $R(\vec{x})$ and $assert(\vec{x})$.

If the above criteria are satisfied then a new CCS is created. The actions are executed and the interval is refined to calculate the $R(\vec{x})$ and $assert(\vec{x})$ for each new candidate. The algorithm finishes when there is only one candidate left, after which the fault detection phase starts.

Few more related works on EFSM-based passive fault detection :

In [Ural and Xu 2007] the authors have proposed an approach that provides information about possible starting state and possible trace compatibility with the observed I/O behavior at the end of passive fault detection. In addition, the proposed approach utilizes an Hybrid method to evaluate constraints in predicates associated with transitions in an EFSM which combines the use of both Interval Refinement and Simplex methods for performance improvement during passive fault detection.

Based on the works of [Lee et al. 2002], the authors of [Alcalde et al. 2004] developed a similar approach of passive testing but following the trace in the backward direction. In this approach the partial trace is processed backward to narrow down the possible specifications. The algorithm performs two steps. It first follows a given trace backward, from the current configuration to a set of starting ones, according to the specification. The goal is to find the possible starting configurations of the trace, which leads to the current configuration. Then, it analyzes the past of this set of starting configurations, also in a backward manner, seeking for configurations in which the variables are determined. When such configurations are reached, a decision is taken on the validity of the studied paths (traces are completed). Such an approach is usually applied as a complement to forward checking to detect more errors.

In [Benharref et al. 2007], the backward and forward methods are combined for *online* passive testing of web services. Here, the algorithm attempts to find a set of

candidates in the past of the trace, that matches the observed event. Using those information, passive fault-detection is carried out, using the forward approach.

The EFSM model has additional advantage compared to the classical FSM for specifications. However, applying the EFSM in passive testing requires the enumeration of data values which is a huge, time and space consuming activity for complex systems.

Invariant-based passive testing

An invariant-based passive testing is another interesting approach introduced initially by the authors of [Cavalli et al. 2003]. In this approach a set of properties are extracted from the EFSM specification, and then the trace resulting from the implementation is analyzed to determine whether it validates this set of properties. These extracted set of properties are called *invariants* because they have to hold true at every moment.

An overview of the different invariant-based passive testing approaches is provided as follows.

Input/Output invariants An I/O invariant consists of two parts, a *preamble* and *test*. The *preamble* is a sequence of events that needs to be found on the trace before the *test* can be evaluated, three types of such invariants are defined in [Cavalli et al. 2003].

1. **Output invariants** allow to express properties such as "immediately after the sequence *preamble* we must always have the output *test*". Some examples of output invariants are shown below:
   - $\underbrace{i_1}_{\text{preamble}} / \underbrace{o_1}_{\text{test}}$ denotes the property "i_1 is always followed by o_1".
   - $\underbrace{i_1/o_1, i_2}_{\text{preamble}} / \underbrace{o_2}_{\text{test}}$ denotes the property "after the sequence (i_1/o_1) and the input i_2 we always have o_2".

2. **Input invariants** allow to describe properties such as "immediately before the sequence *preamble* we must always have the input *test*". The following are few examples of input invariants:
   - $\underbrace{i_1}_{\text{test}} / \underbrace{o_1}_{\text{preamble}}$ denotes the property "o_1 is always preceded by i_1".

- $\underbrace{i_1}_{\text{test}} / \underbrace{o_1, i_2/o_2}_{\text{preamble}}$ denotes the property "the sequence $(o_1, i_2/o_2)$ must always be preceded by i_1".

3. **Succession invariants** are mostly used to describe complex properties such as loop problems. For example, the sequence
- $\underbrace{i_1/o_1, i_1/o_1, i_1}_{\text{preamble}} / \underbrace{o_2}_{\text{test}}$ denotes that the sequence i_1/o_1 is repeated twice before the output o_2 is returned. This kind of sequence is usually observed in a protocol while trying to establish a connection before it returns a failure.

   In this approach information was extracted from the specification and then used to process the trace. However, one of the drawbacks of this work is the limitation on the grammar used to express invariants.

Simple and Obligation invariants A new formalism to express invariants was presented in [Arnedo et al. 2003]. In this approach, wild-card characters to represent sequences of inputs or a single input/output in invariants was introduced. These invariants are called *simple invariants*. For example, an invariant such as $i_1/o_1, \cdots, i_{n-1}/o_{n-1}, i_n/O$ must be interpreted as "each time the implementation performs the sequence $i_1/o_1, \cdots, i_{n-1}/o_{n-1}, i_n$ the next observed output belongs to the set O. These are called *simple invariants*. In [Arnedo et al. 2003] the authors introduce a new notion of invariant to express properties as " if y happens then we must have that x had happened before". These invariants are called obligation invariants. Algorithms are also provided to decide the correctness of the proposed invariants with respect to a given specification.

However, most of the described invariant based testing approaches are derived from the FSM formalism, where only control parts are considered. Then, the authors of [Cavalli et al. 2003] proposed a method to extract the constraint information separately from the involved transitions in addition to extraction of control sequences. According to their approach the correct sequence must be found and the constraints must hold true, otherwise a fault is detected. In another work [Arnedo et al. 2003] proposes a minor modifications to the obligation invariant, to deal with constant data parameters. This approach paved the way for another work by the authors of [Ladani et al. 2005] to extend the simple and obligation invariant to match the EFSM formalism. In [Bayse et al. 2005] the authors assume that the current states of the observed trace are known. In our case we do not require such assumptions. Moreover, our points of observation are

set in a black-box framework that does not allow any homing phase. Since no formal specifications of the implementation is provided, the extracted traces are not related to any known states. Another recent interesting work on invariants was proposed by the authors in [Lalanne and Maag 2012]. In this approach, the authors discuss the importance of testing for data relations and constraints between exchanged messages and they also show how they can be tested directly on traces using logic programming.

2.2 Symbolic execution

Symbolic execution is a technique that consists in executing a program by using arbitrary symbols instead of real concrete data. Thus, computational operations involving guard conditions, assignments, etc., receive symbols as inputs and produce symbolic formulas as outputs. Thus this technique has the advantage of reducing the state space explosion and also fitting well with the IOSTSs. The main idea is to generate a tree-like structure which represents all the behaviors accepted by the IOSTS in a symbolic way. It is based on Input Output Symbolic Transition Systems (IOSTS), which extend Input Output Labeled Transition Systems (IOLTS) by the use of variables and parameters.

System specifications as well as test purposes/behaviors, which are used in conformance testing to specify what aspects of the system have to be tested, are defined as IOSTS. However, almost all the related works on symbolic execution are based on active testing, i.e., it is possible to generate test cases without enumerating the specification's state space. The resulting test cases are symbolic and can be made executable by instantiation of their variables. Our approach is different from the others in a way that we would like to model the behavior/property in the form of IOSTS and symbolically execute to perform the passive testing.

To have a basic idea of symbolic execution, let us consider the Figure 2.4. Figure 2.4 shows a sample program code and its associated symbolic execution, where for each variable in the program, a symbol is introduced to denote its initial value. Note that it has a tree-like structure, for every possible decision a new branch is added to the tree. This branch represents the decision that was taken or condition that has to be met in order to reach the states of the branch. Every state has an associated guard or path condition, which is the set of conditions that have to be met in order to reach

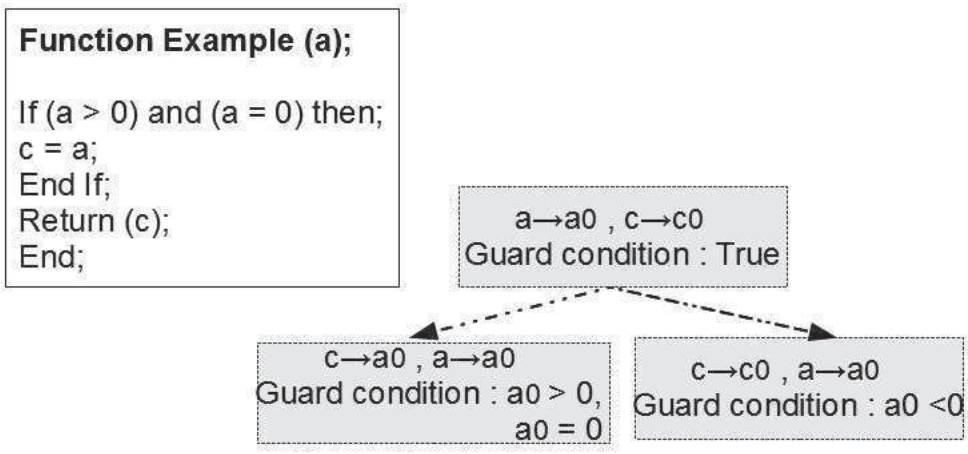


FIGURE 2.4: Symbolic execution of a sample program

a particular state. In this example, to reach the left branch, $a0$ must be greater than or equal to 0 and for the right branch, if $a0$ is less than 0. In addition to the guard conditions, each state in the tree can also store the symbols representing the value of the variables (symbolic values).

The authors of [Gaston et al. 2006] proposed an approach to test whether a system conforms to its specification given in terms of an Input/Output Symbolic Transition System. Here the test purposes are directly expressed as symbolic execution paths of the specification which are finite symbolic subtrees of its symbolic execution. In another work [Rusu et al. 2000] the authors describe an approach for generating symbolic test cases, in the form of input-output automata with variables and parameters.

In [Aiguier et al. 2005] the authors define a logic to express the properties of reactive system represented by IOSTS using $\mathcal{F}$ a new temporal logic formula which is an extension of CTL* [1]. Their main aim is to automatically generate test cases from test purposes given by properties in $\mathcal{F}$. In our approach, we do not have any model of the system, rather we express the properties using an IOSTS model and then try to verify directly on the trace by means of passive testing.

The authors of [Gall et al. 2007] propose a conformance testing based approach to check a refinement relation between reactive system specifications. They model the systems using IOSTS, a first order automata based formalism. Like the traditional conformance testing, some properties or behaviors (observable traces) are selected from the abstract specification and are submitted to the concrete specification to get a verdict

[1] CTL* - unifies Linear Temporal Logic (LTL) and Computational Tree Logic (CTL)

about the refinement relation. However, contradicting to conformance testing techniques, the execution of selected behaviors is not a black box procedure but a white box procedure based on static analysis called symbolic execution technique with constraints solver. In [Nguyen et al. 2012b] the authors propose a conformance checking framework based on symbolic models and an extension of the symbolic bisimulation equivalence. In this approach, the global specification and implementation description are provided as input to the framework and are transformed into STGs. Later the STGs (comprising the STG from specification and implementation) are checked for conformance. This leads to the generation of a large boolean formula which is then verified using SMT solver to reach a conformance verdict. Although this approach is interesting as it avoids state space explosion issues but still, complex constraints cannot be resolved and also it depends on a complete formal specification. The same authors as an improvement in [Nguyen et al. 2012a] discuss an interesting online verification of service choreographies considering complex data constraints. However, they assume that the IUT conforms to the model (which are based on Symbolic Transition Graph with Assignments (STGA)) and also they prove the scalability of their approach for a maximum of 20,000 packets. But, in our approach we consider only an informal specification to define the properties and do not depend on any model. In addition, we have proved the scalability of our approach to very large traces ($> 10^6$).

From the literature we see that most of the works stated above and also to mention few others [Bentakouk et al. 2011, Weiglhofer et al. 2010] are based on active testing using symbolic execution approach. However, in our work we represent the system behavior or property in the form of IOSTS and symbolically execute to obtain a tree-like structure. The branches or the behaviors of the symbolic tree are monitored against the real system trace using passive testing approach.

2.3 Parametric trace analysis

Trace analysis is a fundamental part in many approaches, such as runtime verification, testing, monitoring, and specification mining. For example, EAGLE [Barringer et al. 2003] was introduced as a general purpose rule-based temporal logic for specifying runtime monitors. As an improvement of their work the authors proposed another tool called RuleR [Barringer et al. 2010] a conditional rule-based system, which is more efficiently

implemented for run-time checking, and into which one can compile various temporal logics used for runtime verification. The Program Trace Query Language (PTQL), a language for writing expressive, declarative queries about program behavior was proposed in [Goldsmith and et al. 2005]. EAGLE [Barringer et al. 2003], RuleR [Barringer et al. 2010], and PTQL [Goldsmith and et al. 2005] are very general trace specification and monitoring systems, whose specification formalisms allow complex properties with parameter bindings anywhere in the specification. EAGLE and RuleR are based on fixed-point logics and rewrite rules, while PTQL is based on SQL relational queries. These systems try to define general specification formalisms supporting data binding.

In the literature although we come across several techniques that have been proposed to analyze parametric traces, they have limitations: some in the specification formalism, others in the type of traces they support. For example, JavaMOP [Chen and Rosu 2007] is an efficient parametric runtime monitoring framework, nevertheless, it can only handle a limited type of traces, where the first event for a particular property instance binds all the property parameters. This limitation prevents JavaMOP from supporting many useful parametric properties. Parametric trace slicing [Chen and Rosu 2009] provides solution to parametric trace analysis that is unrestricted by the type of parametric property or trace that can be analyzed. Trace slicing is actually a transformation technique that reduces the size of execution traces for the purpose of testing and debugging. Based on the appropriate use of antecedents, trace slicing tracks back reverse dependences and causality along execution traces and then cuts irrelevant information that does not influence the data observed from the trace. Other approaches that have been proposed to specify and monitor parametric properties are Tracematches [Avgustinov et al. 2007], J-LO [Bodden 2005]and LSC [Maoz and Harel 2006]. These ones support a limited number of parameters, and each has its own approach to handle parameterization specific to its particular specification formalism. On the contrary, the parametric trace slicing technique is generic in the specification formalism, and supports unlimited number of parameters. Our work is based on [Chen and Rosu 2009], performing parametric trace analysis for passive testing.

2.4 Runtime Verification (or Runtime Monitoring)

Usually, testing is performed with active approaches: test cases are generated from the specification and applied on its implementation to check whether the implementation meets desirable behaviors which defines the confidence level of the test between the specification and implementations. Active testing however, may give rise to some inconvenience of disturbing the implementation. Runtime verification which is also sometimes referred to as *runtime monitoring* is also an alternative to the active testing such as the passive testing approaches.

The primary goal of runtime verification is to check whether an implementation I, from which traces can be observed, meets a set of properties during the system execution. With runtime verification, detailed observations of the target system execution behavior are checked at runtime against properties that specify the intended system behavior. These properties are often derived from the target requirement specification, or indeed the properties to be monitored may form the entire specification. In either case, the properties are expressed formally, for example, such as regular expressions, temporal logics or state machines. Both approaches (passive testing and runtime verification) share some important research directions, such as methodologies for checking test relations and properties, or trace extraction techniques. However, while passive testing has the specific purpose of delivering a verdict about the conformance of a black-box implementation (IUT), runtime verification deals with the more general aspects of property evaluation and monitor generation, without necessarily attempting to provide a verdict about the system. The authors of [Leucker and Schallhart 2008] provide a good survey and introduction of methodologies in runtime verification.

A number of approaches to the runtime monitoring of systems in general have been suggested over the years, however, we have decided to present few interesting works that can be compared with ours over here. The authors of [Ghezzi and Guinea 2007] suggests a framework in which correlations between data in multiple messages are expressed and can be checked at runtime. To the best of our knowledge, the correlations imply a single request-response and do not involve messages arbitrarily far apart in time. However, in our approach the data relationship between messages is not limited to a single request-response message rather the complete trace is analyzed. In another work [Wehbi et al. 2012] the authors proposes an interesting tool called MMT (Montimage Monitoring

Tool) to perform passive monitoring for network protocols. In this technique, they do not inject traffic in the network, nor modify the traffic that is being transmitted in the network which is similar to our passive testing technique. However, enumeration of data values might be required to test large reactive systems.

In [Halle and Villemaire 2008], the authors suggest a new logic called, $LTL - FO+$ to model the properties with data parameterization. Here, data is a more significant part of the definition of formulas and LTL temporal operators are used to indicate temporal relations between messages in the trace. Messages are expressed as set of pairs $(label, value)$ which makes the syntax of the logic very flexible. Nevertheless, when more parameters are added to the syntax it loses its clarity. In our approach we consider symbolic data values and hence we do not enumerate and remains flexible for any set of data variables. The concept of parameterized propositions is introduced by the authors of [Stolz 2008]. In this approach the data values in formulas are fixed (i.e., enumerated). Although, it is an interesting approach, it becomes difficult to analyze large reactive systems which may possess huge set of data values.

Runtime verification has been steadily gaining popularity, but vagueness still exists regarding its applicability in real-time systems [Colombo et al. 2009]. The introduction of a monitor overseeing a system, normally slows down the system, which may prove to affect the system performance or real-time systems. However, the introduction of monitors also modifies the behavior of the system, changes which may lead to the creation of new bugs, or the eradication of others. But still, runtime verification and passive testing has its own pros and cons in the protocol testing domain.

Chapter 3

Methodology for Symbolic Passive Testing

Contents

3.1 Introduction

Passive testing as the name implies, it is intended to detect faults by passively observing the input/output actions of the implementation, without interrupting its normal behavior. Usually, passive testing methods extract traces by means of a trace analyzer or some sniffer-based tools, running in the same environment as the implementation. Then, the resulting traces can be used to check that the implementation behavior does not contradict the specification one, or to check the satisfaction of specific properties defined by means of some formal models. Hence the basic objective of passive testing is to provide a verdict about the conformance of an implementation under test, through the behavior observed in the system trace.

In this chapter we present what we define as the *symbolic passive testing* technique to perform conformance testing of communication protocols. In our approach we basically integrate two important techniques: *Symbolic execution* and *Parametric trace slicing*. Figure 3.1 shows the architecture of our symbolic passive testing approach.

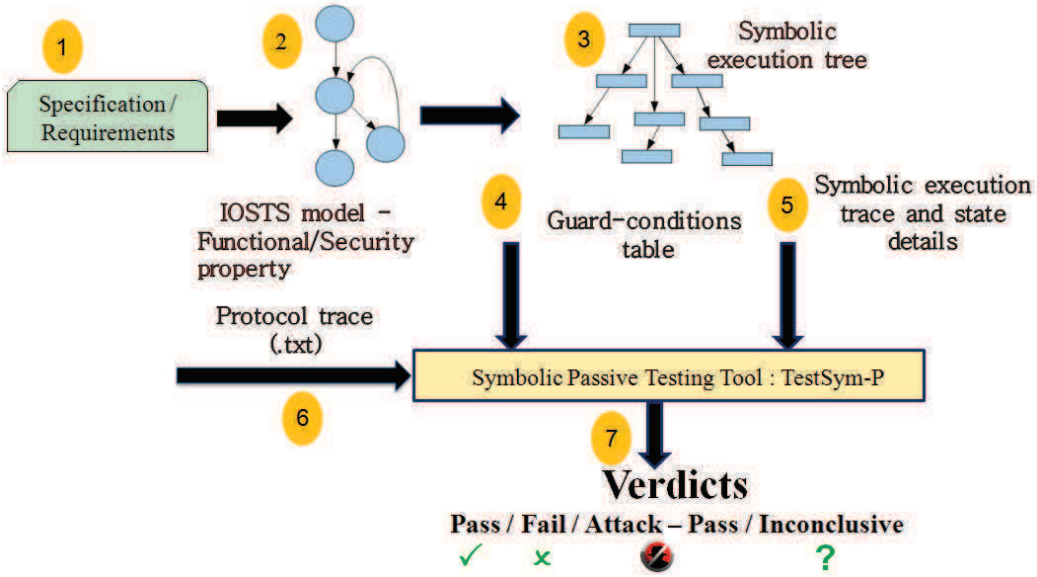


FIGURE 3.1: Architecture of our Symbolic Passive Testing approach.

Several stages are presented as part of our approach in this architecture.

1. We begin by defining the property/attack (from specification sheet if it exists or from requirements sheet) using *sequence diagrams*. The sequence diagram is a pictorial representation of the dataflow between the entities in the IUT.

2. The behavior or attack scenario is then represented using the *Input-Output Symbolic Transition Systems*.

3. Then the IOSTS is symbolically executed. The basic idea of the *symbolic execution* technique is to execute programs or specifications using symbols instead of concrete data as input values, and to derive a symbolic execution tree to represent all possible behaviors of the system or IUT in a symbolic way.

4. The *guard-conditions* are nothing but pre-conditions that must be satisfied by the states involved in the transition. The details of the guard-conditions are derived from the symbolic states in the symbolic execution tree.

5. The *symbolic trace* corresponds to the sequence of symbolic events or messages. The symbolic trace are the branches of the symbolic execution tree which represents actually the system behavior and the eventual attack scenarios, that need to be monitored.

6. *Raw traces* that are collected using trace analyzers are provided as the main source of input to the prototype tool, TestSym-P, to perform passive testing.

A detailed description of the implemented prototype tool, TestSym-P, is presented in Appendices A and B. Combining the techniques of parametric trace slicing and symbolic evaluation a final verdict *Pass/Fail/Inconclusive/Attack-Pass* is obtained.

Based on the Chapter 2, we herein focus on the IOSTS formalism to model the system behavior for the following reasons:

1. IOSTSs introduce the concept of attribute variables. That is, instead of enumerating all possible real data when modeling systems, an IOSTS uses these attribute variables (also called as *fresh variables or symbols*). This abstraction of real data helps to avoid *state explosion problem* in large reactive systems like communication protocol, embedded systems, etc.

2. Reduction of non-determinism. IOSTSs introduce the concept of *guards*, which are conditions of the transitions. Thus, when there are two transitions leaving from the same state, one can easily find out which one of the two transitions is executed.

3. IOSTSs are a general abstraction of the IOLTSs. Semantics of an IOSTS is also given by means of an IOLTS.

Since an IOSTS introduces the notion of attribute variables to represent concrete data they seem to fit well with the symbolic execution technique. The symbolic execution trees resulting from the symbolic execution of an IOSTS represents all the possible behaviors in the system, it suffices to find concrete data for the different symbols in the corresponding branch satisfying the guard conditions accordingly, so one can reach a specific state. In addition to verifying the system behaviors we also monitor certain vulnerabilities eventually described as attack scenarios, which may also be a deviation from the expected behavior.

Parametric trace slicing technique is used for trace analysis. Trace analysis plays a very important part in passive testing. A parametric trace is defined as a trace containing events with parameters that have been bound to a concrete data value (i.e., valuation) and parametric trace slicing is defined as a technique to slice (or cut) the real protocol execution trace into various slices based on the valuation. Each slice corresponds to a particular valuation. These trace slices merged together constitutes the execution trace. Finally, we apply the evaluation logic on the trace slices to provide a test verdict *Pass/Fail/Attack-Pass/Inconclusive*.

In this chapter, we begin with few basic definitions and then define the syntax and semantics of the IOSTS formalism. Then, we define how an IOSTS can be symbolically executed in Section 4.2.4. The parametric trace slicing technique is described in Section 3.3 and then the evaluation logic in Section 3.4. We conclude the chapter discussing on complexity analysis and comparison with other related works in Section 3.5. All of the concepts introduced will be detailed in the following sections. Part of the work was published in [Mouttappa et al. 2012b] and a more complete version has been published in the *Computer Networks* Journal [Mouttappa et al. 2013b].

3.2 Symbolic transition systems

In this section we formally define the *Input-Output Symbolic Transition Systems (IOSTS)*. IOSTS is an extended version of IOLTS defined in Subsection 2.2 of Chapter 2. It explicitly includes data of the reactive systems, and symbolically manipulates with them.

An IOSTS is a kind of automata model which is extended with sets of variables and with guards and assignments on transitions, giving the possibilities to model the system states and constraints on actions. The fact of using symbolic variables helps to describe infinite state transition systems in a finite manner i.e., IOSTS represents the finite transitions of large or infinite state-based systems and in this formalism data are exchanged in input/output messages. At the beginning of this section, we define few definitions, then, we define the IOSTS formalism and discuss its syntax and semantics. We present an example of IOSTS and provide adequate explanation to model protocol behaviors and certain vulnerabilities.

3.2.1 Basic definitions

Definition 3.1. **(Typed Data)** Let $\mathfrak{T} = t_1, ..., t_{|\mathfrak{T}|}$ be a finite set of data types which consists of basic types, e.g., Boolean, natural, integer, enumerated types, and complex types obtained by combining two or more basic types, e.g., arrays, records, structures and queues. Let $D = d_1, ..., d_{|D|}$ be a finite *set of typed data*, where the type of each datum from D belongs to $\mathfrak{T}$.

Definition 3.2. **(Well-Typed Expressions)** An expression is a combination of values and/or variables using operators that can be evaluated to a value. A well-typed expression is an expression, say *exp*, over the typed data D if the value returned by *exp* is of type $t \in \mathfrak{T}$.

Definition 3.3. **(Syntax of IOSTS)** An IOSTS $\mathfrak{M}$ is a tuple $\langle D, I, L, l^0, \Sigma, T \rangle$ where:
- $D = V \cup P$ is a finite **set of typed data** which consists of set V of variables and set P of parameters.
- I is the **initial condition**, a boolean expression on V.
- L is a non-empty, finite **set of locations**.
- $l^0 \in$ L is the **initial location**.
- $\Sigma = \Sigma^? \cup \Sigma^!$ is a non-empty, finite **alphabet of actions** which consists of two mutually disjoint alphabets of **input actions** $\Sigma^?$ and **output actions** $\Sigma^!$, *i.e.,* $(\Sigma^? \cap \Sigma^! = \emptyset)$. For each action $a \in \Sigma$, its signature $sig(a) = \langle p_1, ..., p_k \rangle \in P^k$ $(k \in \mathbb{N})$ is a tuple of parameters.
- T is a finite **set of symbolic transitions**. Each symbolic transition is a tuple $t = \langle l, a, G, A, l' \rangle$ consisting of :

- a location $l \in L$ called the **origin** of the symbolic transition.

- an action $a \in \Sigma$ called the **action** of the transition.

- a predicate G on D, called the **guard** is a boolean expression containing the truth values *true, false*.

- a **set of assignment** A, each assignment is of the form $(x := A_x)_{x \in V \cup P}$, such that, for each $x \in V \cup P$, the right-hand side A_x of the assignment is an expression on $V \cup P$. These assignments are well-typed, that is, the expressions A_x returns a data type which is the same as that of x.

- a location $l' \in L$ called the **destination** of the symbolic transition.

In order to distinguish an input from an output action, we may respectively attach the '?' and '!' symbols to the actions respectively.

Example 3.1. *For better understanding of the IOSTS formalism, we consider an example based on the SIP protocol with real data and control parts. The IOSTS* $\mathfrak{M}$ *shown in Figure 3.2(b), comprises of:*

- the set of typed data, $D = \underbrace{vfrom, vto, vcid, vcontact1, vrealm}_{V} \cup \underbrace{from0, to0, cid0, contact0, realm0}_{P}$. *We consider here, the sets of symbolic variables* V *and parameters* P *are mutually disjoint.*

- the initial condition, $vfrom = from - i, vto = to - i, vcid = cid - i$.

- the set of locations, $L = l0, l1, l2, l1.1$

- the initial location, $l^0 = l0$

- the alphabet of actions, $\Sigma = \underbrace{200Ok, 401Unauth}_{input\ actions} \cup \underbrace{Register, Registerw/cred}_{output\ actions}$. *The set of input* $\Sigma^?$ *and output* $\Sigma^!$ *actions are mutually disjoint.*

- the set of transitions, T. *For instance* $t \in T$ *is given by,*

$$t : \langle \underbrace{l0}_{source}, \underbrace{Register}_{a}, \underbrace{true}_{G}, \underbrace{((vfrom := from\_i) \wedge (vcid := cid\_i) \wedge (vto := to\_i))}_{A}, \underbrace{l1}_{destination} \rangle$$

For instance in the above transition, for any input or output actions ($!Register$*), if the guard conditions* G *associated with any transition, say, from location* $l0$ *to* $l1$ *is true, then the set of assignments* A *happens. Note that the sequence involved in each transition is: first the guard-conditions are verified; if the conditions are satisfied then a new value for the variables are assigned and finally the system reaches the destination state.*

3.2.2 Semantics of an IOSTS

In general, valuation means assigning concrete values to the elements of V or P. In our approach, the concrete input values and initialization values of variables are replaced by symbolic ones, called *fresh variables*. We represent the set of fresh variables by F, where $F \cap V = \emptyset$.

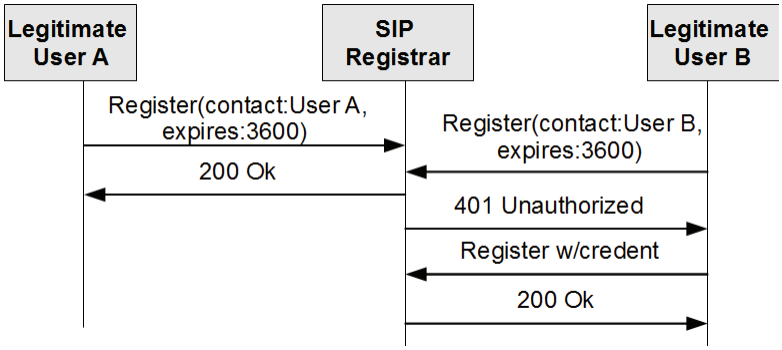
Input-Output Labeled Transitions Systems (IOLTS) are rooted from Labeled Transition systems (LTS) with distinguished inputs and outputs. The semantics of an IOSTS $\langle D, I, L, l_0, \Sigma, T \rangle$ is an IOLTS $\langle S, S_0, \Lambda, \rightarrow \rangle$ defined as:
- the **set of states** is $S = L \times \mathcal{V}$, where $\mathcal{V}$ **is the set of valuation for the variables** V. Formally a *state* is a pair $\langle l, v \rangle$ where $l \in L$ is a location and $v \in (\mathcal{V}_x)_{x \in V}$ corresponds to the valuation for the variables $x \in V$.
- the **set of initial states** is $S_0 \subseteq S$, an *initial* state is a state $\langle l0, v0 \rangle$ such that $l0 \in L$ is the initial location and $v0$ is the valuation of the variables that satisfies the initial condition I, i.e., $S0 = \{\langle l0, v0 \rangle | v0 \in \mathcal{V} \wedge I(v0) = true\}$.
- the **set of valued actions** $\Lambda = \{\langle a, \vartheta \rangle | a \in \Sigma, \vartheta \in \Pi_{sig(a)}\}$, where a is an input or output action and ϑ is a valuation for the parameter(s) carried by the action a. The set of **valued parameters** P **is given by** Π. $\Pi_{sig(a)}$ corresponds to the all possible valuations for the parameters seen in the action a.
- $\rightarrow$ is the **transition relation**, which is a 3-tuple $\langle s, \alpha, s' \rangle$ where, $s, s' \in S$ are the source and destination states respectively and α is a valued action.

3.2.2.1 Examples of a SIP property defined as an IOSTS

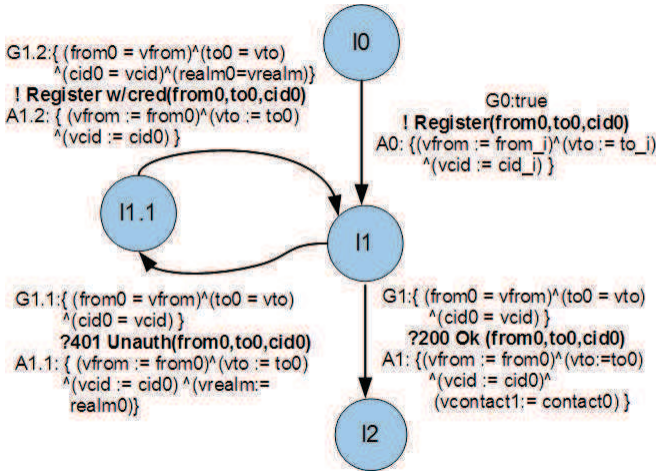
The Session Initiation Protocol (SIP) is an application-layer control protocol for creating, modifying and terminating sessions with one or more participants [Rosenberg et al. 2002]. These sessions include Internet telephone calls, multimedia distribution, and multimedia conferences. When users request to use a SIP service, they need to be registered and authenticated in order to get the service from the proxy server. For a better understanding of the IOSTS formalism, we represent a sample SIP property called *Registration*. For explanation we have considered only few parameters in the IOSTS like *(From, To,* and *Call-ID* fields) corresponding to the SIP protocol. However, a more

detailed description of the SIP protocol and its properties with the attack scenarios will be provided in Chapter 4 of this thesis.

3.2.2.2 Registration Property in SIP



(a) Sequence Diagram



(b) IOSTS model

FIGURE 3.2: SIP: Registration Property in SIP

The SIP registration mechanism allows a user agent to create a binding in a location service for a particular domain that associates an address-of-record with one or more contact addresses. Hence, before a session can be established, a legitimate user sends a registration request that contains information about the IP address, phone number, name, an expiration value, etc. Figure 3.2(a) shows the sequence diagram of how two

legitimate users A and B register with the domain registrar who is responsible for maintaining a database of records of all subscribers and Figure 3.2(b) shows the equivalent manually designed IOSTS model.

In the IOSTS model, as there is no initial condition to begin the transition, we assume the initial guard-condition ($G0$) to be *true*. As per the Definition 3.3, only if the guard-conditions associated with each state (G) are satisfied or *true* for any actions (input or output), there can be a transition from the source state to the destination state and new assignments (A) performed. If the registrar does not require authentication, it accepts the registration shown by the transition from state *l0-l1* and sends a confirmation message *200 Ok*, given by the transition from state *l1-l2* (Registration done by user A in Figure 3.2(a)). Otherwise, the registrar will ask the user to identify using a challenge/response method. The user will have to answer the challenge using a password that was initially shared with the registrar shown by the transition from state *l1-l1.1*. The registrar verifies the response and if the response is as expected, the registrar will accept the packet given by the transition from state *l1.1-l1-l2* (Registration done by user B in Figure 3.2(a)). After every successful registration the session can be established.

In the next section, we represent the symbolic execution of the IOSTS model representing the SIP registration property as a symbolic tree. The branches in the tree correspond to the behavior of the property presented in the Section 3.2.2.1.

3.2.3 Symbolic Execution

Symbolic execution was first developed for program testing [King 1976]. The interesting idea behind symbolic execution mainly consists in replacing concrete input values and initialization values of variables by symbolic ones in order to compute constraints induced on these variables by different possible executions. Recently, symbolic execution has been applied on models for verification or conformance testing purposes [Bannour et al. 2012, Gall et al. 2007, Gaston et al. 2006]. The symbolic execution of an IOSTS results in a symbolic execution tree with interesting properties which are worth mentioning: (i) each symbolic state in the tree-like structure can hold any number of child states, there by each branch in the tree represents a particular property or a vulnerability scenario (ii) every branch has a unique decision making criteria, which means, the decision that have to be taken in order to reach a particular state of one branch of the tree are different

from the other branches. These decisions or so called *guard-conditions* are stored in each symbolic state.

Symbolic execution tree in our work represents the behaviors or attacks in a symbolic way, thus to passively test each branch, it suffices to find concrete data for the different symbols in the corresponding branch satisfying the guard-conditions. The symbolic execution technique has been widely used to test different types of systems, mostly contributing to the active testing techniques. However, in our work we utilize this technique to perform passive testing of system behaviors.

Thus, the symbolic execution (SE for short) of IOSTS serves two main objectives: (i) to use symbolic values (fresh variables) for action messages and initialization values for IOSTS variables instead of concrete data values (ii) to obtain a tree-like structure which represents all the behaviors accepted by the IOSTS in a symbolic way. In Figure 4.9, the symbolic execution of an IOSTS is represented as a tree with different branches (or paths) in which the vertices's are symbolic extended states and edges are labeled by symbolic communication actions, which are computed from the transition communication action and from the symbolic values associated to attribute variables in symbolic state. The behavior of an IOSTS, also referred to as *semantics*, can be represented by the symbolic traces obtained from it. Traces are nothing but succession of communication actions that are specified by an IOSTS. In this section we provide the definitions related to symbolic execution of the IOSTS.

3.2.3.1 Basic definitions

Definition 3.4. **(Symbolic extended states)** Let $\mathfrak{M} = \langle D, I, L, l_0, \Sigma, T \rangle$ be an IOSTS and $\mathfrak{M}' = \langle S, S_0, \Lambda, \rightarrow \rangle$ be the corresponding IOLTS. A symbolic extended state for an IOSTS is given by the tuple $\eta = (s, A, G)$ where $s \in S$, A a set of assignments and G a set of guard conditions.

The above definition clearly states that the symbolic extended states are used to store information concerning the symbolic execution, the current state of the symbolic execution, the assignment of symbolic values to the attribute variables, and the constraints on those symbolic values after the execution. The constraints which are stored

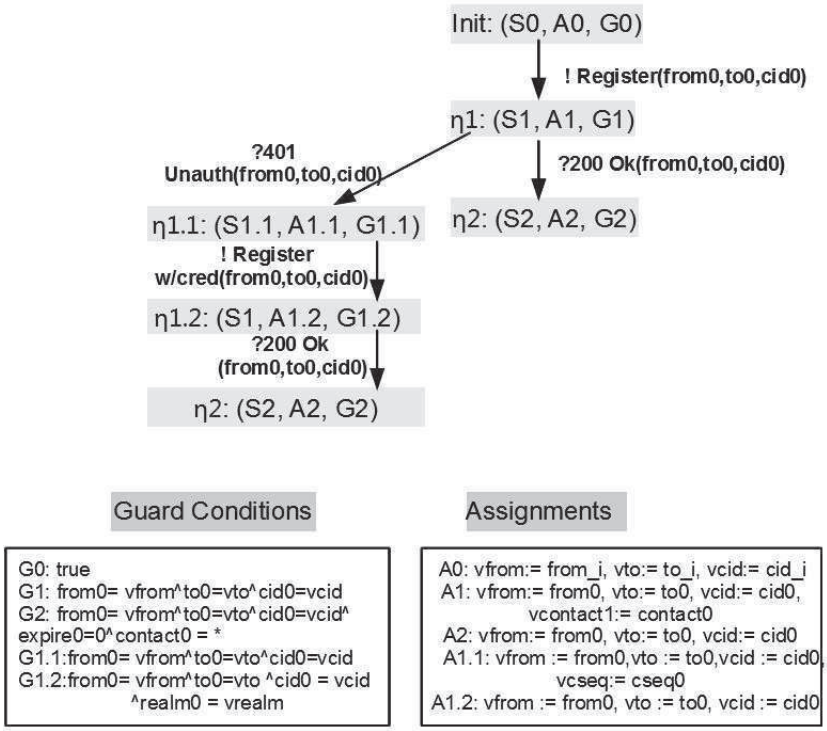


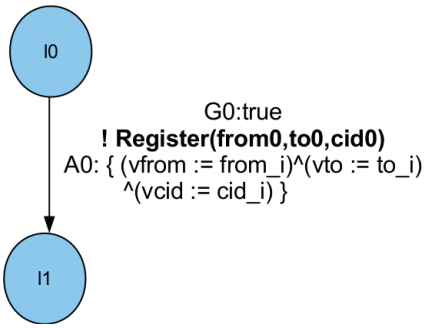
FIGURE 3.3: Symbolic Execution of IOSTS.

in each symbolic state are called *guard-conditions*, which are nothing but constraints that *must* be satisfied for an execution to follow a particular path.

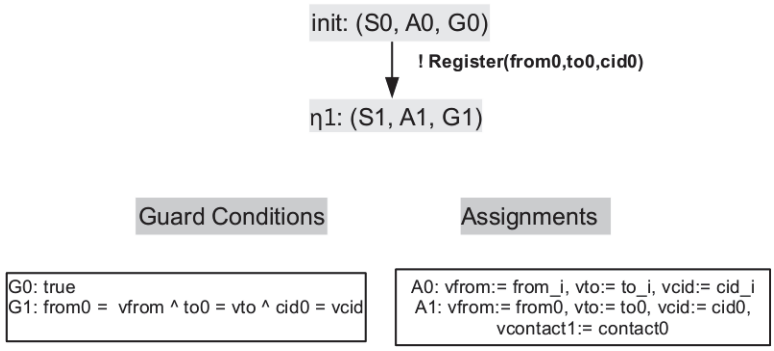
Definition 3.5. **(Satisfiable symbolic state)** Let $\mathfrak{M}$ be an IOSTS, K the set of all symbolic extended states. A symbolic extended state η is said to be a **satisfiable** symbolic state η_{sat} if the guard conditions G associated with the symbolic state are evaluated to true. K_{sat} is the set of all satisfiable symbolic states.

Definition 3.6. **(Symbolic execution of a transition)** Let $\mathfrak{M} = \langle D, I, L, l_0, \Sigma, T \rangle$ be an IOSTS. For any $t \in T$ and $\eta \in S$, a symbolic execution of t from η is defined as a 3-tuple $(\eta, \alpha, \eta^{'}) \in S \times \Lambda_F \times S$. The source of the transition (η) is the symbolic state from which the transition is executed, α is the symbolic interpretation of the communication actions introduced in the transition, the target of the transition ($\eta^{'}$) is the symbolic state reached by the execution. Λ_F corresponds to the set of valued communication actions obtained by assigning fresh variables F as values for the parameters.

Example 3.2. *Let us consider the Figure 3.4(a) which depicts a transition of an IOSTS (the IOSTS represented in Figure 3.2(b) is taken here for illustration). In Figure 3.4(b), there are two symbolic extended states (as per the Definition 3.6) init and* $\eta 1$.



(a) A sample transition of an IOSTS



(b) Symbolic execution of the transition

FIGURE 3.4: Example illustrating the symbolic execution of one transition

The symbolic state *init* holds the information before executing the transition: the associated state of the IOSTS is $l0$, the assignment of values to the attribute variables ($A0$), and the guard conditions ($G0$). Since it is the initial state, there are no guard conditions and hence we assume $G0 = true$. The symbolic transition is labeled with the communication action of t, substituting the variables with some fresh variables. Here, the communication action $!Register(from0, to0, cid0)$ contains new values $from0, to0, cid0$ to be stored in the symbolic variables $vfrom, vto, vcid$. The symbolic target state is $\eta 1$, which has a new set of values for their variables and a set of guard-conditions $G1$ that needs to be satisfied.

Subsequently, the symbolic execution of an IOSTS can be depicted as a tree whose edges are symbolic extended states and vertexes are labeled by symbolic communication actions. The root is the symbolic extended state made of the IOSTS initial state, the guard condition *true* and an arbitrary initialization of the variables to some symbolic

values. The symbolic communication action is computed from the transition communication action and from the symbolic values associated to the attribute variables in η. A target symbolic state is then computed, it stores the target state of the transition, a new path condition derived from the path condition of the symbolic extended state and from the transition guard, and finally the new symbolic values associated to attribute variables. Now, we formally define the symbolic execution of an IOSTS.

Definition 3.7. **(Symbolic execution of an IOSTS)** Let $\mathfrak{M} = \langle D, I, L, l_0, \Sigma, T \rangle$ be an IOSTS. The root of the symbolic execution tree, is given by $init = (S0, A0, true)$. The root ($init$) is a symbolic state made of the IOSTS initial state ($S0 \in S$), the arbitrary initialization $A0$ of the variables to some fresh variables F and set of path conditions $G0$ to be *true* (there is no constraint to begin the execution). $R_{sat} \subseteq K_{sat} \times \Lambda_F \times K_{sat}$ is the symbolic execution of the transitions of $\mathfrak{M}$, where Λ_F is the set of valued communication actions obtained by assigning fresh variables F as values for the parameters and K_{sat} is the set of satisfiable symbolic state. Thus, the symbolic execution $SE(\mathfrak{M})$ of an IOSTS can be defined by the tuple $(init, R_{sat})$.

Paths of a symbolic execution are the sequences of symbolic transitions that start at the initial state.

Definition 3.8. **(Paths of SE)** Let $SE(\mathfrak{M}) = (init, R_{sat})$ be a symbolic execution. R_{sat} is the set of symbolic transitions where the symbolic states are satisfiable. The set of paths of $SE(\mathfrak{M})$, denoted by $Path(SE(\mathfrak{M}))$, contains all the finite sequences $t_1 ... t_n$ of transitions of R_{sat}, such that $source(t_1) = init$ and for every $i, 1 \leq i \leq n, target(t_i) = source(t_{i+1})$.

- $source(t_1) = init$
- for every $i, 1 \leq i \leq n, target(t_i) = source(t_{i+1})$

Example 3.3. *Let us consider the symbolic execution tree in Figure 3.3 representing the registration property in SIP*[1] *as explained in Section 3.2.2.1. For readability reasons, the assignments and guard conditions are not directly shown inside the symbolic states but in separate frames. In this symbolic execution tree we have two different paths.*[2].

[1]The symbolic parameters in the message corresponds to the $From, To$ and $Call - ID$ field values in SIP. However, there is no limitation in choosing the number of parameters.

[2]Paths of a symbolic execution are the sequences of symbolic transitions that start at the initial state, as per the Definition 3.8

(1) Registration without authentication - $(\mathsf{init}, !\mathsf{Register}(\mathsf{from0}, \mathsf{to0}, \mathsf{cid0}), \eta 1), (\eta 1, ?\mathsf{200Ok}(\mathsf{from0}, \mathsf{to0}, \mathsf{cid0}), \eta 2)$

(2) Registration with authentication - $(\mathsf{init}, !\mathsf{Register}(\mathsf{from0}, \mathsf{to0}, \mathsf{cid0}), \eta 1), (\eta 1, ?\mathsf{401\ Unauth}(\mathsf{from0}, \mathsf{to0}, \mathsf{cid0}), \eta 1.1), (\eta 1.1, !\mathsf{Registerw/cred}(\mathsf{from0}, \mathsf{to0}, \mathsf{cid0}), \eta 1.2), (\eta 1.2, ?\mathsf{200Ok}(\mathsf{from0}, \mathsf{to0}, \mathsf{cid0}), \eta 2)$

We define the behaviors of a symbolic execution by means of its symbolic traces.

Definition 3.9. **(Traces of a symbolic execution)** Let $t_1 \dots t_n$ be a finite sequence of symbolic executions of the transitions of $\mathfrak{M}$ such that for i, $1 \leq i \leq n$, we have that $t_i = (\eta_i, \alpha_i, \eta_{i+1})$ and $t_1 = (init, \alpha_i, t_2)$.

Let $\langle (init, \alpha_1, t_2), (t_2, \alpha_2, t_3), ..., (t_n, \alpha_n, t_{n+1}) \rangle \in SE(\mathfrak{M})$ be a symbolic trace. We will say that the sequence $\langle \alpha_1 \alpha_2 ... \alpha_n \rangle$ is a symbolic trace of $\mathfrak{M}$. We will denote by $Trace(SE(\mathfrak{M}))$ the set of all symbolic traces of $\mathfrak{M}$. Let $\langle \alpha_1 \alpha_2, ..., \alpha_n \rangle$ be a symbolic trace, we will say that $\langle a_1, a_2, ..., a_n \rangle$ is the control portion of this trace if, $\alpha_i = (a_i, \vartheta_i)$ for $1 \leq i \leq n$. We will denote the set of traces with only the control portion of the action by $CP[Trace(SE(\mathfrak{M}))]$.

Example 3.4. *Based on the definition above, we obtain the symbolic traces for the paths defined in Example 3.3 (for Figure 3.3) ,*
$Trace(SE(\mathfrak{M})) = \{!\mathsf{Register}(\mathsf{from0}, \mathsf{to0}, \mathsf{cid0})?\mathsf{200Ok}(\mathsf{from0}, \mathsf{to0}, \mathsf{cid0}), !\mathsf{Register}(\mathsf{from0}, \mathsf{to0}, \mathsf{cid0})?\mathsf{401Unauth}(\mathsf{from0}, \mathsf{to0}, \mathsf{cid0})!\mathsf{Registerw/cred}(\mathsf{from0}, \mathsf{to0}, \mathsf{cid0})?\mathsf{200Ok}(\mathsf{from0}, \mathsf{to0}, \mathsf{cid0})\}$.

The set of traces with only the control portion of the action is given by,
$CP[Trace(SE(M))] = \{!\mathsf{Register}?\mathsf{200Ok}, !\mathsf{Register}?\mathsf{401Unauth}!\mathsf{Register}?\mathsf{200Ok}\}$.

In this section we have obtained the traces of the symbolic execution of an IOSTS representing a protocol property, in the next section we explain the parametric trace slicing approach to passively test these symbolic traces on real network traces.

3.3 Parametric Trace Slicing

Parametric traces, i.e., traces containing events or messages with parameter bindings, observed during the system execution. Parameter bindings are actually abstract parameters (i.e., variables) bound to concrete data values during runtime. Most properties of

the parametric traces are also found to be parametric. Due to the dynamic nature and unlimited number of parameter bindings of the parametric properties, formal verification and testing against implementation under test (IUT) becomes very difficult. However, in this work we have shown, how we can apply the parametric trace slicing technique to test system behaviors.

3.3.1 Basic definitions

Trace slicing is a widely used technique for analyzing a real network trace [Chen and Rosu 2009]. The parametric trace slicing technique splits the real protocol execution trace into different slices based on the data portions of each event (i.e., packet) in the trace. The events corresponding to a particular valuation are grouped in the order they appear in the trace in a particular slice, and all the other events that are unrelated to the given valuation are dropped. In this section, we first define some basic definitions (parametric/non-parametric events, trace slicing, etc.) and then present an offline parametric trace slicing algorithm that plays an important part in the passive testing technique.

Note: In general, *valuation* means assigning some values to the parameters or variables (see Section 3.2.2). These values are *symbolic* in the case of symbolic execution and *concrete* in the case of parametric trace slicing. In order to show the relationship with the symbolic approach and parametric trace slicing, and to ease the reading, we have decided to use similar notations like Λ, ϑ here also.

Definition 3.10. **(Non-Parametric event and traces)** A non-parametric event (or base event), say $a \in \Sigma$ is defined as an event with no data portions or in simple words, it can be said as an event without parametric bindings (i.e., events that do not carry concrete data instantiating abstract parameters).

A non-parametric trace is an element in Σ^*, that is a sequence of actions (as defined by Definition 3.3) without data portions.

Example 3.5. *Consider a non-parametric trace* $\rho_{non} : \langle a1, \rangle\langle a2, \rangle\langle a3\rangle$. *Each event in this trace, say* $a1, a2, a3$ *can be referred to as a non-parametric event.*

Definition 3.11. **(Parametric event and traces)** A parametric event is defined as an element in the set of valued actions $\langle a, \vartheta\rangle \in \Lambda$.

A parametric trace is defined as a word in the set of valued actions Λ^*.

Example 3.6. *Consider three valued actions* $\langle a1, \vartheta' : (x \mapsto 1, y \mapsto 2)\rangle$, $\langle a2, \vartheta'' : (x \mapsto 1, y \mapsto 2, z \mapsto 3)\rangle$ *and* $\langle a3, \vartheta''' : (x \mapsto 1)\rangle$. ρ *defined as* $\rho : \langle a1, \vartheta'\rangle\langle a2, \vartheta''\rangle\langle a3, \vartheta'''\rangle$ *is a parametric trace. Each valued action in this trace can be referred to as parametric event.*

Definition 3.12. **(Less and More Informative valued actions.)** Let $Q \subseteq P$ be a set of parameters in an IOSTS. We denote by Π^Q the set of valued parameters Q. Let $\vartheta', \vartheta \in \Pi^Q$ be two valuations of the parameters Q, we say that ϑ' is **less informative** than ϑ (or ϑ is **more informative** than ϑ'), denoted by $\vartheta' \sqsubseteq \vartheta$, if for any $q \in Q$, if $\vartheta'(q)$ is defined it implies $\vartheta(q)$ is also defined and $\vartheta(q) = \vartheta'(q)$.

Example 3.7. *Let us represent two valuations* $\vartheta' : \langle x \mapsto 1, y \mapsto 2\rangle$ *and* $\vartheta : \langle x \mapsto 1, y \mapsto 2, z \mapsto 3\rangle$. *Here,* ϑ' *is less informative than* ϑ *since all the parameters* (x, y) *defined in* ϑ' *are also defined in* ϑ *and* ϑ *carries additional information (parameter* z*).*

Definition 3.13. **(Trace slicing)** Let $\rho = (\langle a_1, \vartheta_1\rangle, \langle a_2, \vartheta_2\rangle, ..., \langle a_n, \vartheta_n\rangle) \in \Lambda^*$ be a parametric trace, and $\vartheta \in \Pi^Q$ be a valuation. We define recursively (for $i = 1$ to n) the function $\rho \restriction_\vartheta$ as:

$$\rho \restriction_\vartheta = \begin{cases} <> & if \ \ \rho = <> \\ (\langle a_i, \vartheta_i\rangle), (\langle a_2, \vartheta_2\rangle, ..., \langle a_n, \vartheta_n\rangle) \restriction_\vartheta & if \ \ \vartheta_i \sqsubseteq \vartheta \\ (\langle a_2, \vartheta_2\rangle, ..., \langle a_n, \vartheta_n\rangle) \restriction_\vartheta & if \ \ \vartheta_i \not\sqsubseteq \vartheta \end{cases}$$

It means that for a finite parametric trace ρ, the trace slice for ϑ can be obtained under two different cases. If ϑ_i is less informative than ϑ, then the action a_i is added at the end of the trace slice else we leave the trace slice undisturbed as defined above.

Example 3.8. *Consider a parametric trace* $\rho : \langle a1, \vartheta' : (x \mapsto 1, y \mapsto 2)\rangle\langle a2, \vartheta'' : (x \mapsto 1, y \mapsto 2, z \mapsto 3)\rangle\langle a3, \vartheta''' : (x \mapsto 1)\rangle$ *and* $\vartheta : \langle (x \mapsto 1, y \mapsto 2)\rangle$.
The ϑ*-trace slice is given by* $\rho \restriction_\vartheta = a1a3$ *where* $a1a3 \in \Sigma^*$.
Consider the first event in the trace $\langle a1, \vartheta' : (x \mapsto 1, y \mapsto 2)\rangle$, *since* $\vartheta' = \vartheta$, *the action* $a1$ *is initially added to* $\rho \restriction_\vartheta$. *Then coming to the next event* $\langle a2, \vartheta'' : (x \mapsto 1, y \mapsto 2, z \mapsto 3)\rangle$, *since* ϑ'' *is more informative than* ϑ, *the trace slice is left undisturbed. Coming to the last event in the trace slice,* $\langle a3, \vartheta''' : (x \mapsto 1)\rangle$, ϑ''' *is less informative than* ϑ, *hence the action* $a3$ *is added at the end of the existing trace slice for* ϑ.

3.3.2 Parametric Trace Slicing Algorithm

In this section the algorithm to build the trace slices of a parametric trace (i.e., a real execution trace) is presented. This algorithm takes the trace ρ in event order (i.e., from the first event to the last one) as input and provides ts, the set of all possible trace slices of ρ as output. $ts \in \Sigma^*$ is obtained by the table $\mathfrak{L} : \Pi \rightarrow \Sigma^*$ analyzing the valuation of parameters observed in ρ as defined in Definition 3.13.

Algorithm 1: Parametric Trace Slicing Algorithm

```
1  Algorithm : Parametric Trace slicing;
   Input: parametric trace ρ ∈ Λ*
   Output: A table 𝔏 : Π → Σ*
2  Initialization : 𝔏 ← ϵ; Θ ← {ϵ} ;
3  begin
4      foreach ordered parametric event ⟨a, ϑ⟩ in ρ do
5          foreach parametric instance ϑ' in Θ do
6              if (ϑ' ⊑ ϑ) then
7                  𝔏(ϑ) ← ρ ↾_ϑ a
8              else
9                  𝔏(ϑ) ← ρ ↾_ϑ
10             end
11         end
12         Θ ← Θ ∪ {ϑ}
13     end
14 end
```

The Algorithm 1 is defined as follows. The outer for-loop (lines 4-13) takes each event $\langle a, \vartheta \rangle$ incrementally in the trace ρ and the existence of the valuation of the current event is checked in Θ using the inner for-loop (lines 5-11). If the valuation ϑ' is less informative than ϑ observed in the outer for-loop as per line 6, then the action a is added at the end of the trace slice as per line 7, if not, the action is not added to the trace slice. The procedure described for the inner for-loop is continued until all the parametric instances in Θ are evaluated against $\langle a, \vartheta \rangle$ in the outer for-loop. Later, Θ is updated with the new instance ϑ as per line 12. Now, the next event in the outer for-loop is taken and the whole procedure is repeated until line 12. This procedure is continued until we reach the end of the trace ρ in the outer for-loop. At the end, we obtain a table $\mathfrak{L}$ with all possible trace slices ts and Θ with all possible valuations contained in the trace.

Example 3.9. *Consider a sample SIP trace* $\rho = !Register\langle from1, to1, cid1\rangle$ $?401Unauth\langle from1, to1, cid1\rangle !Register\langle from2, to1, cid2\rangle !Registerw/cred$

$\langle from1, to1, cid1\rangle ?200Ok\langle from1, to1, cid1\rangle ?200Ok\langle from2, to1, cid2\rangle$.

For simplicity, we only consider the values of (From, To, Call-ID) *parameters in SIP, represented by* $(from, to, cid)$ *respectively. For clarity we represent the valuations* $\langle from \mapsto from1\rangle$ *by* $\langle from1\rangle$ *and* $\langle from \mapsto from1, to \mapsto to2\rangle$ *by* $\langle from1, to2\rangle$*. Applying our trace slicing algorithm on* ρ *we obtain the Table 3.1.*

As per the Algorithm 1, initially the trace slice table and Θ *are empty. Let us consider the first event in the trace,* $!Register\langle from1, to1, cid1\rangle$*, the action* $!Register$ *is added in the trace slice table for the corresponding valuation* $\langle from1, to1, cid1\rangle$ *and also updated in* Θ*. Then coming to the next event,* $?401Unauth\langle from1, to1, cid1\rangle$*, since the valuation is already available in the trace slice, the action* $?401Unauth$ *is appended with the already existing event in the trace slice table. Next for the event,* $!Register\langle from2, to1, cid2\rangle$ *since the valuation* $\langle from2, to1, cid2\rangle$ *does not exist in the slice table, a new entry is created and the action* $!Register$ *is added in the trace slice table for the corresponding valuation* $\langle from2, to1, cid2\rangle$ *and* Θ *updated. For the next event,* $!Registerw/cred\langle from1, to1, cid1\rangle$ *since the valuation already exists in the trace slice table, the action* $!Registerw/cred$ *is appended with the already existing events. Similar approach is carried out for all the other events in the trace.*

For instance, $\mathfrak{L}(\langle from1, to1, cid1\rangle) = !Register?401Unauth!Registerw/cred?200Ok$. *Here,* $\mathfrak{L}$ *corresponds to the trace slice table that we obtain after applying the trace slicing algorithm and* $\mathfrak{L}(\langle from1, to1, cid1\rangle)$ *gives the trace slice value for the particular valuation,* $\vartheta = \langle from1, to1, cid1\rangle$

TABLE 3.1: Slice table $\mathfrak{L}$ for a sample SIP trace ρ.

Valuation (ϑ)	ts - the ϑ-trace slice
$\langle from1, to1, cid1\rangle$	!Register ?401Unauth !Register w/cred ?200Ok
$\langle from2, to1, cid2\rangle$	!Register ?200Ok

In this section, we have defined an algorithm to slice a real network trace. In the next section, we discuss how the obtained parametric trace slices are used for evaluating a SIP property defined as an IOSTS.

3.4 Testing the IOSTS property on real execution traces

As stated earlier, our approach is the integration of symbolic execution and parametric trace slicing technique. In this section, we clearly detail our evaluation logic and the different verdicts obtained to prove the conformance. Here, we take the advantage of expressing the parametric properties or the behavior using IOSTS formalism. This gives a wide space to passively test our symbolic property for all possible parameter instances (concrete values) observed in the trace.

3.4.1 Evaluation of a Property on Trace slices

Our objective is to check an expected behavior formally specified as an IOSTS property $\mathcal{P}$ against an execution trace slice in ts. We also target here the test of deviant behaviors for testing. It means that if our conformance property is not satisfied on real traces, we check the presence of an eventually defined vulnerability/security attack in the monitored trace.

In our evaluation approach, we evaluate the control and the data portions of the messages observed in the symbolic traces. $CP[Trace(SE(\mathfrak{M}))]$ and $\mathfrak{L}(\vartheta)$ are used to check the control portion. For the data portion, we check the guard conditions associated with each state in the symbolic execution of the IOSTS. The evaluation is done for each slice against the symbolic traces and the verdicts *Pass, Fail, Inconclusive, Attack-Pass, Attack-Fail* are emitted based on the Table 3.2.

Definition 3.14. **(AttackSeq)** AttackSeq is a variable defined to differentiate the conformance property and the security property during the evaluation.

$$AttackSeq = \begin{cases} 0 & for \quad conformance \ \ property \\ 1 & for \quad security \ \ property \end{cases}$$

Pass and *Fail* are provided for the test of a conformance property while *Attack-Pass* and *Attack-Fail* are dedicated to the test of a security property. *Inconclusive* is emitted if we cannot firmly decide (e.g., in case of a too short execution trace).

A formal description of the different cases involved in the evaluation of the property on the trace slice is provided below.

TABLE 3.2: Evaluation table for each trace slice.

AttackSeq	Control Portion	Data Portion	Verdict
0	✓	✓	Pass
0	✓	×	Fail
1	✓	✓	Attack-Pass
1	✓	×	Attack-Fail
0 or 1	×	-	Inconclusive

- ***Pass***:
  - The control portions are identical for two sequences in $\mathfrak{L}(\vartheta)$ and $CP[Trace(SE(\mathfrak{M}))]$.
  - The data portions are satisfied (all states in $SE(\mathfrak{M})$ are satisfiable, Definition 3.5).

$$\forall ts_i \in ts, (\exists \mathfrak{X} \in CP[Trace(SE(\mathfrak{M}))], \mathfrak{X} = \mathfrak{L}(\vartheta_i) \wedge \forall \eta \in SE(\mathfrak{M}), \eta \in K_{sat}, AttackSeq = 0)$$

- ***Fail***:
  - The control portions are identical.
  - The data portions are not satisfied.

$$\forall ts_i \in ts, (\exists \mathfrak{X} \in CP[Trace(SE(\mathfrak{M}))], \mathfrak{X} = \mathfrak{L}(\vartheta_i)) \wedge \exists \eta \in SE(\mathfrak{M}), \eta \notin K_{sat}, AttackSeq = 0)$$

- ***Attack-Pass***:
  - The control portions are identical for the two sequences: it exists one slice $ts_i \in ts$ identical to one element of $CP[Trace(SE(\mathfrak{M}))]$ holding the attack sequence.
  - The data portions are satisfied.

$$\forall ts_i \in ts, (\exists \mathfrak{X} \in CP[Trace(SE(\mathfrak{M}))], \mathfrak{X} = \mathfrak{L}(\vartheta_i) \wedge \forall \eta \in SE(\mathfrak{M}), \eta \in K_{sat}, AttackSeq = 1)$$

- ***Attack-Fail***:
  - The control portions are identical for the two sequences: it exists one slice $ts_i \in ts$ identical to one element of $CP[Trace(SE(\mathfrak{M}))]$ holding the attack sequence.
  - The data portions are not satisfied.

$$\forall ts_i \in ts, (\exists \mathfrak{X} \in CP[Trace(SE(\mathfrak{M}))], \mathfrak{X} = \mathfrak{L}(\vartheta_i) \wedge \exists \eta \in SE(\mathfrak{M}), \eta \notin K_{sat}, AttackSeq = 1)$$

- ***Inconclusive***, if the control portions are not identical. Indeed, the execution trace has eventually been extracted from the IUT starting from a time t that does not correspond to the initial state (reset). Thus, the obtained finite trace may not be practically sufficient to prove the property or attack on the trace slice, which results in an *inconclusive* verdict. Since the control portion is not satisfied we do not check the data portions.

  $\forall ts_i \in ts, \nexists \mathfrak{X} \in CP[Trace(SE(\mathfrak{M}))], \mathfrak{X} = \mathfrak{L}(\vartheta_i), AttackSeq = 0 \text{ or } 1$

The Algorithm 2 is defined as follows. In order to evaluate the conformance or security properties on the trace slices, two steps are performed: control portion must be checked and then we verify the data portion. In line 6, the control portion is checked by comparing $\mathfrak{L}(\vartheta_i)$ and $\mathfrak{X}_k$. If the sequences are identical and if the verdict is not *Pass*, then we process the lines 7-19, else returns *Inconclusive* and skips the data-portion checking. In the if-loop, we check for the type of sequence. If ($AttackSeq = 0$), then it is a conformance property sequence and the data portion is verified using the function *data-check-fn*. In *data-check-fn*, the set of assignments A corresponding to each state is performed and the guard conditions are evaluated. If the function returns a *Pass*, then the output verdict is *Pass* else *Fail*. If ($AttackSeq \neq 0$) it is an attack sequence and the data portions are evaluated using the same function. If the function returns a *Pass*, then the output verdict is *Attack-Pass* else *Attack-Fail*.

Algorithm 2: Evaluation on the Trace Slices

Algorithm : Evaluation logic for each slice ;

Input: symbolic traces of $SE(M)$, State details of $SE(M)$, $AttackSeq$, Trace slices ts_i.

Output: $verdict(ts_i) = Pass/Fail/Inconclusive/Attack-Pass/Attack-Fail$

Initialization : $i \leftarrow 0$ to n, $verdict(ts_i) \leftarrow$ ' ';

begin

  foreach $(ts_i \in ts)$ **do**

    foreach $(\mathfrak{X}_k \in CP[Trace(SE(\mathfrak{M}))])$ **do**

      if $(\mathfrak{L}(\vartheta_i) = \mathfrak{X}_k) \wedge (verdict(ts_i) \neq Pass)$ **then** // Control check

        if *(AttackSeq = 0)* **then**

          if *data-check-fn*$(ts_i, G) = Pass$ **then** // Data check

            $verdict(ts_i) \leftarrow Pass$;

          else

            $verdict(ts_i) \leftarrow Fail$;

          end

        else

          if *data-check-fn*$(ts_i, G) = Pass$ **then** // Data check

            $verdict(ts_i) \leftarrow Attack - Pass$;

          else

            $verdict(ts_i) \leftarrow Attack - Fail$;

          end

        end

      else

        $verdict(ts_i) \leftarrow Inconclusive$;

      end

    end

  end

end

In the next section, we explain how these results can be used to evaluate a system implementation $\mathcal{I}$ against the property $\mathcal{P}$.

3.4.2 Evaluation of Property/Attack on the Implementation Traces

Our main objective is to test if a trace of the black-box implementation $\mathcal{I}$ satisfies our IOSTS behavior $\mathcal{P}$ by passive testing. For each trace slice obtained, we check if the control portion of the protocol is matched with our behavior, if matched we perform the data portion check, and generate a verdict accordingly. In general, all the trace slices put together constitutes a trace, thus based on the verdicts obtained for each trace slice we can conclude if the property is satisfied by the implementation or if there is any violation leading to an attack.

Finally, based on the verdicts obtained on the slices, we may define the final verdict of the property testing on the entire real trace by:

- Pass, if $(\forall Verdict(ts_i) = Pass)$
- Attack-Pass, if $(\exists Verdict(ts_i) = Attack - Pass)$
- Fail, if $[(\exists Verdict(ts_i) = Fail) \wedge (\exists Verdict(ts_i) \neq Attack - Pass)]$
- Inconclusive, if $[(\exists Verdict(ts_i) = Attack - Fail) \wedge otherwise]$.

The implementation $\mathcal{I}$ satisfies the property $\mathcal{P}$ results in a *Pass* if all the trace slices in ts satisfy the property, i.e., if our evaluation algorithm provides a verdict *Pass* for every trace slice in ts. But, during the evaluation if any trace slice results in an *Attack-Pass*, then we conclude that the implementation does not satisfy the property, which implies an attack has happened. But if we do not observe an *Attack-Pass* and observe a *Fail* verdict for any of the trace slice, then the final verdict is *Fail*. An *Attack-Fail*, can be due to the short trace length (or insufficient trace length) then trace slicing will result in incomplete slices. So, during the evaluation of the trace slices (control portion checking) it results in *Inconclusive* and also in all the other cases it results in *Inconclusive*.

3.5 Time Complexity Analysis

The complexity of an algorithm can be defined as a function $f(n)$ which measures the time and space used by an algorithm in terms of input size n. The main focus of complexity analysis is to study on how execution time increases with the data set to

be processed. In this section, we present the time complexity analysis of the proposed algorithms and also the comparison report with other passive testing tools.

3.5.1 Complexity of our approach

The overall time complexity of our symbolic approach depends upon the algorithms proposed for slicing and evaluation logic in Sections 3.3.2 and 3.4.1 respectively.

3.5.1.1 Complexity of Slicing logic

Our trace slicing logic depends upon the number of messages n in the trace, and the number of available parametric instances m in the monitored trace. The proposed slicing algorithm computes trace slices for all the parametric instances observed in the trace rather than computing all possible combinations of parametric instances. Although, the complexity remains the same in both cases, yet in our approach the number of slices would be significantly reduced thereby results in improved evaluation time complexity. In the best case, the time complexity is linear with respect to the number of messages in the trace as the number of parametric instances observed in the trace is less than the length of the trace, $m < n$. But in the worst case, since m is not constant, the number of parametric instances can be equal to the length of the trace, $m = n$, hence we can obtain a quadratic time complexity. However, practically the number of instances observed in a trace remains comparatively smaller than the trace length.

We therefore have, $\mathcal{O}_{slicing} = \mathcal{O}(n \times m)$. In worst case, where $m = n$, $\mathcal{O}_{slicing} = \mathcal{O}(n^2)$.

3.5.1.2 Complexity of Evaluation logic

The proposed evaluation logic depends upon the number of symbolic sequences (property or vulnerability/attack) that has to be monitored say, t_{seq} (that is, $Trace(SE(M))$ as defined in Definition 3.9) and the total number of slices obtained from the slicing logic t_s. As per the parametric trace slicing logic defined in Section 3.3, the number of slices is dependent directly on the number of different parametric instances observed in the trace, hence we have $t_s = m$. However, t_{seq} is not significant (can be a constant)

compared to the number of slices. The time complexity for the evaluation logic is given by $\mathcal{O}_{eval} = \mathcal{O}(t_s \times t_{seq})$. Since $t_s = m$ and t_{seq} is merely a constant, $\mathcal{O}_{eval} = \mathcal{O}(m)$.

In the worst case, where $m = n, \mathcal{O}_{eval} = \mathcal{O}(n)$

However, theoretically speaking, the number of trace slices obtained is always less than the trace length, (i.e., $t_s < n$) and the length of the trace sequence is also comparatively smaller. Thus, the time complexity is linear.

The overall evaluation time complexity (worst-case) depends upon the proposed two algorithms and is given by, $\mathcal{O}_{overall} = \mathcal{O}_{slicing} + \mathcal{O}_{eval}$
$\mathcal{O}_{overall} = \mathcal{O}(n^2 + n)$.

3.5.2 Comparison with other Passive Testing tools

Passive testing mainly deals with monitoring functional and security related properties. Several research works has been carried out to formalize these properties (e.g., FSM, EFSM, LTL, etc.). However, each of them has their own limitations in terms of expressibility [Hewlett-Packard 2004], specification-dependency [Lee et al. 2002, Ural and Xu 2007], etc. Many passive testing solutions assume that the current state of the observed trace are known. But, in our approach we do not assume to have any specification of the implementation under test, hence the extracted traces are not related to any known states. Table 3.3 shows the analyzed time complexities for different passive testing tools.

TABLE 3.3: Time complexity - Different Passive Testing tools.

Tool	Time Complexity	Data constraints
TIPS	n^2	✓
PASTE	$kn^2 + n(p-k)$	×
TestInv-P	n^2	✓
DataMon	n^k	✓
TestSym-P	$n^2 + n$	✓

Note: p-No. of operators, k-No. of quantifiers, n-No. of messages in the trace.

TIPS (Testing Invariants for Protocols and Services) is an interesting passive testing tool which considers data and time analysis. As our approach is symbolic, we are not restricted with specific data values. The time complexity is quadratic, but the authors have proposed as future work to reduce the time complexity to linear in [Morales et al. 2010]. The tool PASTE (Passive Testing) [Andrés et al. 2012] has efficient passive testing algorithms implemented in their tool, but they do not consider the causality between

the data portions in a trace. TestInv-P efficiently monitors functional and security requirements on the captured trace considering the data constraints. Nevertheless, the time complexity still remains quadratic and enumeration of data values is required in this tool [Bagnato et al. 2010].

DataMon is another interesting tool based on Horn logic [Che et al. 2012], but the time complexity depends upon the number of quantifiers used in the formula. In addition, the prototype tool failed to provide verdict for large trace files ($> 10^6$) mainly because of the way of expressing the property and also the need to check the enumerated values. In our tool, TestSym-P, we give importance in considering the causality between the data portions in the trace. The worst-case time complexity remains quadratic like other approaches, practically the complexity remains less than quadratic, based on the description given in Subsection 3.5.1. Although, the time complexity when compared to other passive testing approaches was promising but still an inordinate amount of system memory was required to generate the verdict. During the evaluation, additional system memory was required to store the resultant slice table and the evaluation output. Nevertheless, we believe that we can improve the system memory usage by applying online testing. In order to study the scalability of our tool we have carried experiments on very large traces ($> 10^6$) and the results were promising. The experimental results are discussed in Chapter 4.

3.6 Conclusion

In this chapter we described our approach for symbolic passive testing, by introducing the IOSTS structures, which are the basis of the work presented in our thesis. In order to provide a clear understanding of the approach we have shown an example of a SIP (Session Initiation Protocol) property. IOSTS are used to model properties or behaviors of the implementation under test (IUT), which are then symbolically executed to obtain a tree-like structure that represents all the valid behaviors that any implementation of the specification should satisfy. IOSTS are executed by applying the symbolic execution technique, whose main idea is to replace symbols instead of concrete data. The symbolic traces obtained from the symbolic execution tree serves as the basis when testing the conformance of the properties with respect to their specifications.

Then, we introduced the concepts of parametric trace slicing for trace analysis, where in, based on the different parametric instances observed in the trace, the trace was cut into different slices. These slices are validated against the symbolic traces and verdicts for each trace slice is obtained like *pass, fail, attack-pass, attack-fail, inconclusive*. After evaluating each trace slice, the logic for the final evaluation was discussed. For the final evaluation logic we obtain *pass, fail, attack-pass, inconclusive* verdicts to prove the conformance of the properties on the system trace. In the later part of this chapter, we described the time complexity involved in our approach and also provided a comparison report with the other existing passive testing approaches.

With the introduction of the symbolic passive testing technique, we are now ready to show in the Chapter 4 how we have applied the work to two different real case studies: Session Initiation Protocol (SIP) and Bluetooth protocol.

Chapter 4

Application to Real-Time Case Studies

Contents

4.1 Introduction

For the validation of efficacy of the proposed symbolic passive testing technique, we applied it to two different real-time protocols, the SIP and Bluetooth protocol in a real

network environment. We divide this chapter into two sections comprising a detailed study of the two case studies. The experimentation and results that have been obtained are depicted and detailed in this chapter.

4.2 Case study 1: The Session Initiation Protocol (SIP)

This section gives an overview of the 3GPP IP Multimedia Subsystem (IMS), which is based on the IETF Session Initiation Protocol (SIP) and other protocols. A basic overview of the IMS Architecture is first introduced and then, technical details of the SIP protocol are presented. The basic IMS procedures like registration and session establishment are introduced with the various kinds of threats/vulnerabilities on the SIP based systems such as flooding attacks (DoS), message flow attacks, etc. Our symbolic passive testing approach discussed in Chapter 3 is applied to SIP and the experiments and results are discussed.

4.2.1 Basic overview of the IMS

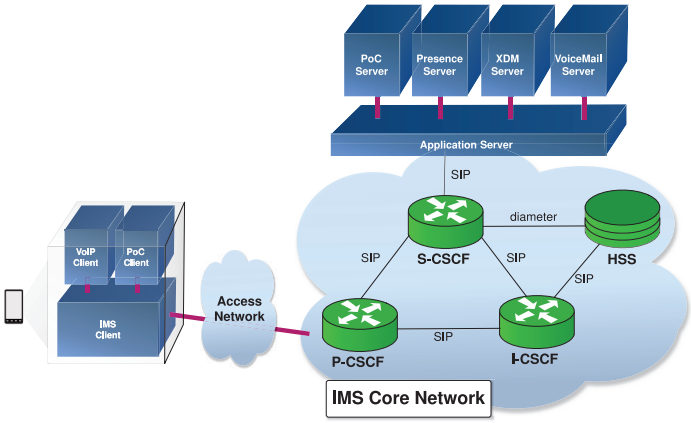


FIGURE 4.1: An IMS Architecture.

The IP Multimedia Subsystem (IMS) [IMS 2012] is a standardized framework for delivering IP multimedia services to users in mobility. The IMS aims at facilitating the access to voice or multimedia services in an access-independent way in order to develop the fixed-mobile convergence. Figure 4.1 shows the core of the IMS network consisting of the Call Session Control Functions (CSCF), which redirect requests depending on the type of service, the Home Subscriber Server (HSS), a database for the provisioning

of users, and the Application Server (AS), where the different services run and interoperate. Most communication with the core network and between the services is done using the SIP. The traces contain all communication between the client, the IMS core, and the AS. Given the point of observation, messages exchange for the Presence service, as well as the PoC (Push-to-talk Over Cellular)[Alliance 2006] service can be observed in the collected traces. In addition to SIP packets other protocols (TCP, RTCP, TalkBurst) also appears. However, during our experiments we have filtered these protocol packets by the support of the trace analyzer tool in order to specifically test SIP.

4.2.2 Overview of SIP

The Session Initiation Protocol is a protocol designed to provide session management functionalities such as establish, terminate and modify multimedia sessions [Rosenberg et al. 2002]. SIP is a very simple text based protocol similar to that of HTTP and it follows the request/response model. This has made SIP a very popular protocol in the VoIP system implementations. H.323 [Vineet Kumar and Sengodan 2001] protocol provides very similar functions but the SIP has better features such as simplicity, extensibility and scalability.

SIP is being widely used in building VoIP networks. Unlike the traditional telephone networks, VoIP networks do not have a closed communication which makes communication medium vulnerable to the intruders. The attacks on the SIP systems may cause severe consequences such as making system unavailable for the services, hijacking of information or user credentials and more.

4.2.2.1 SIP components

Although SIP works in conjunction with other technologies and protocols, there are two fundamental components that are used by the Session Initiation Protocol as shown in Figure 4.2:

- *User agents*, which are endpoints of a call (i.e., each of the participants in a call)
- *SIP servers*, which are computers on the network that service requests from clients, and send back responses

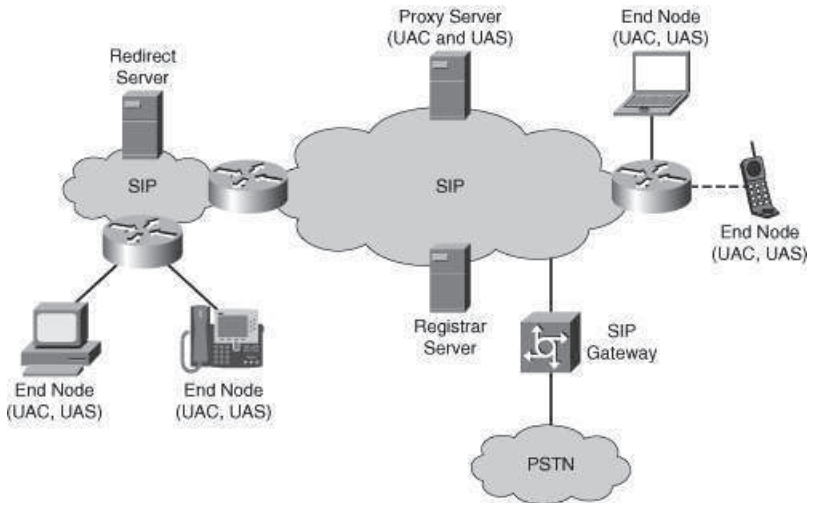


FIGURE 4.2: SIP Components.

User Agents (UA) User agents are both the computer that is being used to make a call, and the target computer that is being called. These make the two endpoints of the communication session. There are two components to a User agent: a *client* and a *server*. When a User agent makes a request (such as initiating a session), it is the *User Agent Client (UAC)*, and the User agent responding to the request is the *User Agent Server (UAS)*. Because the User agent will send a message, and then respond to another, it will switch back and forth between these roles throughout a session.

One UA will invite the other into a session, and SIP can then be used to manage and tear down the session when it is complete. During this time, the UAC will use SIP to send requests to the UAS, which will acknowledge the request and respond to it.

SIP Server The SIP server is used to resolve usernames to IP addresses, so that requests sent from one User agent to another can be directed properly. A User agent registers with the SIP server, providing it with their username and current IP address, thereby establishing their current location on the network. This also verifies that they are online, so that other User agents can see whether they are available and invite them into a session. Since the User agent probably would not know the IP address of another User agent, a request is made to the SIP server to invite another user into a session. The SIP server then identifies whether the person is currently online, and if so, compares the username to their IP address to determine their location. If the user is not part of that domain, and thereby uses a different SIP server, it will also pass on requests to other servers.

In performing these various tasks of serving client requests, the SIP server will act in any of several different roles:

- *Registrar Server*: Registrar servers are used to register the location of a User agent who has logged onto the network. It obtains the IP address of the user and associates it with their username on the system. This creates a directory of all those who are currently logged onto the network, and where they are located. When someone wishes to establish a session with one of these users, the Registrar server's information is referred to, thereby identifying the IP addresses of those involved in the session.

- *Proxy Server*: Proxy servers are computers that are used to forward requests on behalf of other computers. If a SIP server receives a request from a client, it can forward the request onto another SIP server on the network. While functioning as a proxy server, the SIP server can provide such functions as network access control, security, authentication, and authorization.

- *Redirect Server*: The Redirect servers are used by SIP to redirect clients to the User agent they are attempting to contact. If a User agent makes a request, the Redirect server can respond with the IP address of the User agent being contacted. This is different from a Proxy server, which forwards the request on your behalf, as the Redirect server essentially tells you to contact them yourself.

4.2.2.2 Message Syntax

Since SIP is a text-based protocol like HTTP, it is used to send informations, as a series of requests and responses between clients and servers, and User Agent clients and User Agent servers. When requests are made, there are a number of possible signaling commands that might be used:

- *REGISTER*: Used when a User agent first goes online and registers their SIP and IP addresses with a Registrar server.

- *INVITE*: Used to invite another User agent to communicate, and then establish a SIP session between them.

- *ACK*: Used to accept a session and confirm reliable message exchanges.
- *OPTIONS*: Used to obtain information on the capabilities of another User agent, so that a session can be established between them. When this information is provided a session is not automatically created as a result.
- *SUBSCRIBE*: Used to request updated presence information on another User agent's status. This is used to acquire updated information on whether a User agent is online, busy, offline, and so on.
- *NOTIFY*: Used to send updated information on a User agent's current status. This sends presence information on whether a User agent is online, busy, offline, and so on.
- *CANCEL*: Used to cancel a pending request without terminating the session.
- *BYE*: Used to terminate the session. Either the User agent who initiated the session, or the one being called can use the BYE command at any time to terminate the session.

When a request is made to a SIP server or another User agent, one of a number of possible responses may be sent back. These responses are grouped into six different categories, with a three-digit numerical response code that begins with a number relating to one of these categories. The various categories and their response code prefixes are as follows:

- *Informational (1xx)*: The request has been received and is being processed.
- *Success (2xx)*: The request was acknowledged and accepted.
- *Redirection (3xx)*: The request cannot be completed and additional steps are required (such as redirecting the User agent to another IP address).
- *Client error (4xx)*: The request contained errors, so the server cannot process the request.
- *Server error (5xx)*: The request was received, but the server cannot process it.
- *Global failure (6xx)*: The request was received and the server is unable to process it.

SIP headers follow similar grammar rules to HTTP headers. Requests and responses share a common message format which consists of a start-line, one or more header fields, an empty line indicating the end of the header fields, and an optional message-body. The start-line in SIP messages can be either a request or a status line as shown in Figure 4.3. Request messages use the request line to set the type of request. Response messages indicate whether the processing of a request is successful or not in the status line.

```
Session Initiation Protocol
    Request-Line: REGISTER sip:CA.cym.com SIP/2.0
        Method: REGISTER
        Request-URI: sip:CA.cym.com
            Request-URI Host Part: CA.cym.com
        [Resent Packet: False]
    Message Header
        Via: SIP/2.0/UDP 192.168.1.6:5061;branch=z9hG4bK-2491-1-0
            Transport: UDP
            Sent-by Address: 192.168.1.6
            Sent-by port: 5061
            Branch: z9hG4bK-2491-1-0
        From: ua1 <sip:ua1@nnl.cym:5061>;tag=2491SIPpTag071
            SIP Display info: ua1
            SIP from address: sip:ua1@nnl.cym:5061
                SIP from address User Part: ua1
                SIP from address Host Part: nnl.cym
                SIP from address Host Port: 5061
            SIP tag: 2491SIPpTag071
        To: ua1 <sip:ua1@nnl.cym:5061>
            SIP Display info: ua1
            SIP to address: sip:ua1@nnl.cym:5061
                SIP to address User Part: ua1
                SIP to address Host Part: nnl.cym
                SIP to address Host Port: 5061
        Call-ID: 1-2491@192.168.1.6
        CSeq: 1 REGISTER
            Sequence Number: 1
            Method: REGISTER
        Contact: sip:ua1@192.168.1.6:5061
            Contact-URI: sip:ua1@192.168.1.6:5061
                Contactt-URI User Part: ua1
                Contact-URI Host Part: 192.168.1.6
                Contact-URI Host Port: 5061
        Content-Length: 0
        Expires: 300
```

FIGURE 4.3: Example of a SIP message.

SIP message headers consist of fields with name value pairs. Where some fields are optional such as content type and length, some fields are mandatory for every SIP message. Table 4.1 lists the mandatory header fields for SIP messages.

4.2.2.3 SIP transactions and Dialogs

Although SIP messages are sent independently over the network, they are usually arranged into transactions by User agents and certain types of proxy servers. Hence, SIP is said to be a transactional protocol. A *transaction* is a sequence of SIP messages exchanged between SIP network elements. A transaction consists of one request and all responses to that request. That includes zero or more provisional responses and one or

Table 4.1: SIP messages mandatory header fields

Field name	Description
To	The request destination's SIP address
From	Indicates the originator of the request
CSeq	The command sequence that ensures messages are dealt with, in the order they were generated.
Call-ID	A randomly generated string that uniquely identify SIP sessions. SIP proxy servers use Call-ID to identify messages belonging to a SIP session.
Via	Contains information about SIP devices a message has passed through as it moves between caller and callee. The Via field is also used to route responses in the reverse direction.
Contact	Contains the actual location of the callee, which might be different from the address of the originator in the From header

more final responses (remember that an INVITE might be answered by more than one final response when a proxy server forks the request).

If a transaction was initiated by an INVITE request then the same transaction also includes ACK, but only if the final response was not a 2xx response. If the final response was a 2xx response then the ACK is not considered part of the transaction.

SIP dialogs represent peer-to-peer relationships between two SIP endpoints. It provides the context for sequencing and routing of messages between SIP User agents. Dialogs are identified by the following: *Call-ID* header, and the *From Tag* and *To Tag* parameters. The value of these three fields are the same for messages that belong to the same dialog. The header field CSeq is used to sequence messages within a dialog. The value is increased monotonically from request to request, thereby identifying the transactions within a dialog. In effect, a dialog is a sequence of transactions. This is illustrated in Figure 4.4.

4.2.3 IOSTS modeling of SIP behaviors/attacks

In Chapter 3, we discussed our approach considering the SIP registration property as an example. In this section, we describe the IOSTS modeling of the de-registration attack and DoS scenarios associated with SIP registration and also session establishment property with their associated attack scenarios.

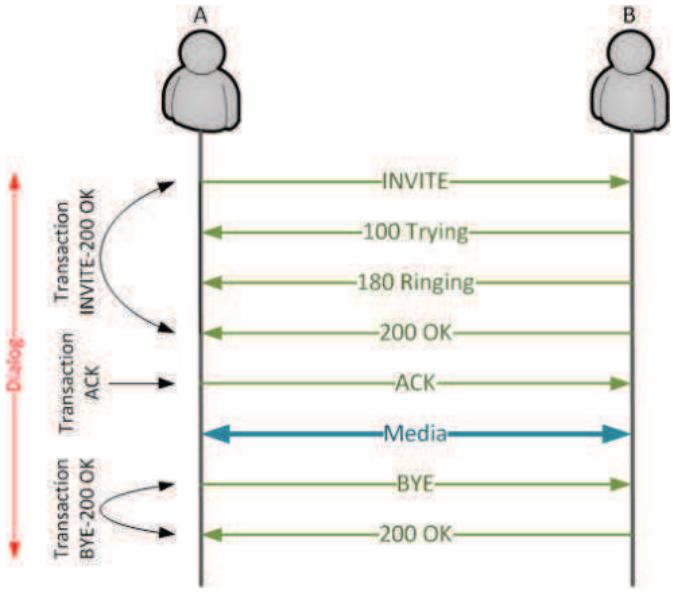
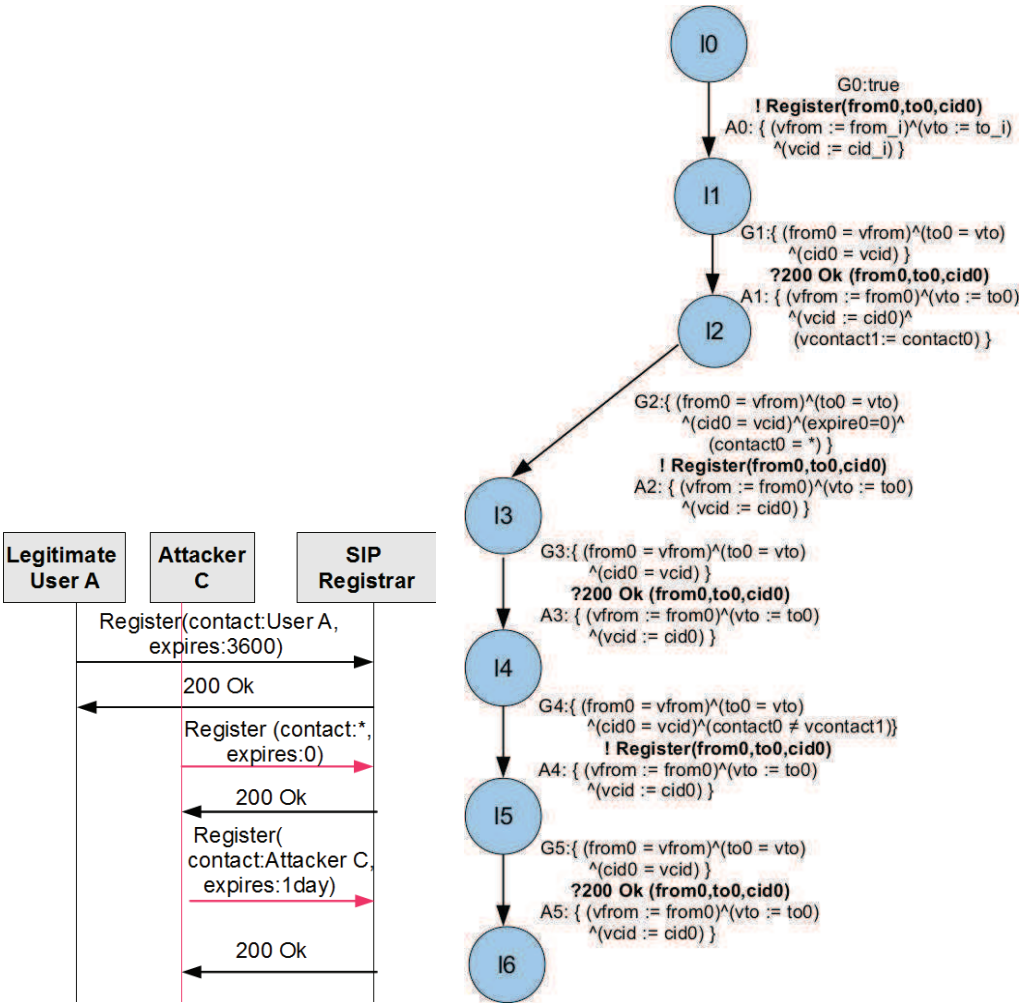


FIGURE 4.4: Dialog and transactions during the establishment of a SIP session.

4.2.3.1 Registration Hijack attack in SIP

In general, the user's registration will be valid for the time period specified in the *expires* field in the *contact* parameter or in *expires* parameter in SIP packets [Collier 2005]. The user has to send periodical refreshing registration requests to the registrar to keep its contact valid; if not the user contacts will be removed after the expiration time. In registration hijacking attack, the attacker is assumed to impersonate a legitimate User agent, de-registers all the existing contacts and register their own device as the appropriate contact address, thereby directing all requests for the legitimate user to the attacker's device. Figure 4.5(a) shows the sequence diagram of how a legitimate user A performs registration with a registrar and how an attacker performs a registration hijack attack. The corresponding IOSTS model is shown in Figure 4.5(b).

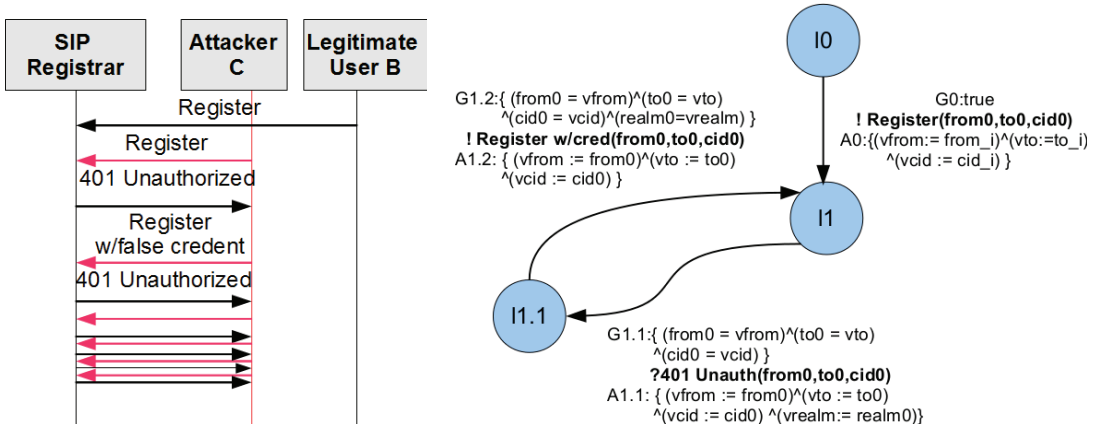
The state transitions for the legitimate registration of user A is shown by *l0-l1-l2* (as explained in Subsection 3.2.2.2). The attacker performs the de-registration of a User agent by sending a registration request with all the message headers and parameters retaining the same as the legitimate user A, except for the parameters in the *contact* field (contact: *, expires: 0) as shown by the state transition from *l2-l3* (Registration done by attacker C in Figure 4.5(a)). Since the registrar takes the *From* address as the asserted identity of the originator of the request, all the contacts of the affected User agent are removed given by the state transition from *l3-l4*. Now, the attacker



(a) Sequence Diagram (b) IOSTS model

FIGURE 4.5: SIP: Registration Hijack Attack

can register himself by modifying the *contact* address (contact: Attacker C's address, expires: 1 day) shown by the state transition from *l4-l5*. The registrar accepts the new registration request and sends a confirmation message *200 Ok* shown by the transition from *l5-l6*. After this attack, if the legitimate user A tries to establish a session, the user's identity might have been removed and would need to register with the registrar again.

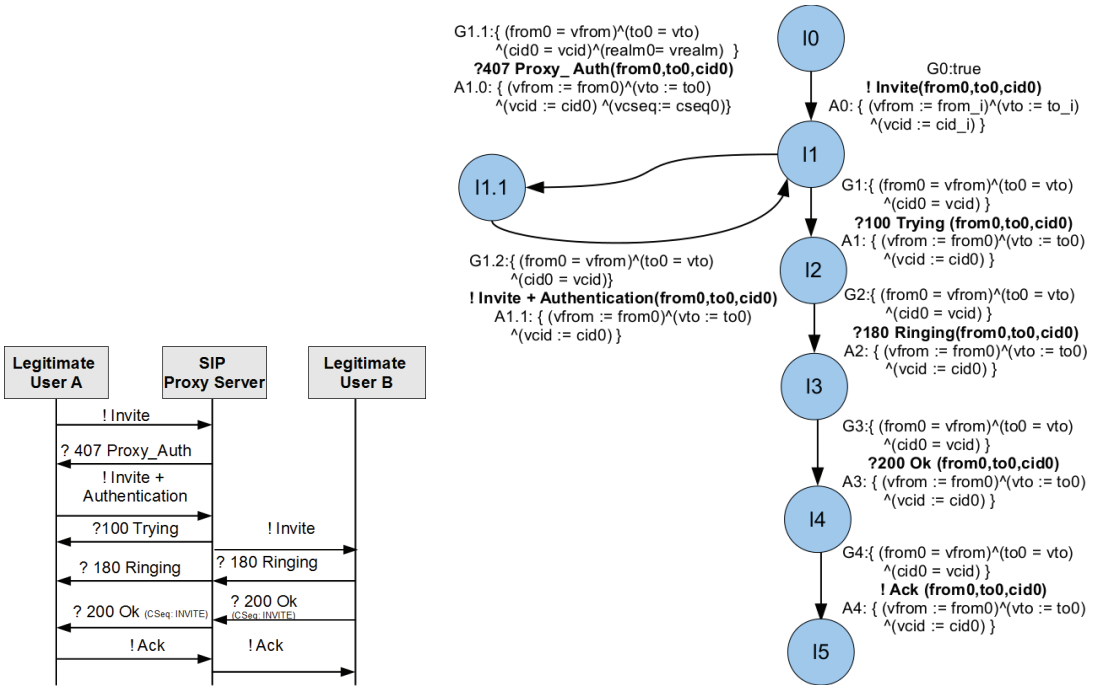


(a) Sequence Diagram (b) IOSTS model

FIGURE 4.6: Denial of Service Attack

4.2.3.2 Denial of Service (DoS) attack in SIP

Usually, during the registration phase, the user might be challenged by the registrar. Figure 4.6(a) shows how an attacker C tries to create a denial of service attack. An equivalent IOSTS model for DoS attack is represented as an infinite loop as shown in the Figure 4.6(b). The attacker C during the challenge/response method tries to respond to the challenge using fake passwords shown by the transition from the state *l1.1-l1*. If the registrar finds the user credentials send by the faulty attacker, the message *401 Authorization required* (the challenge/response method) would be send until the user credentials are proved to be correct as shown by the state transition from *l1-l1.1*. This might result in a DoS attack causing a heavy traffic on the registrar which might hinder the registrar to provide services to the legitimate users. The unfolding of this loop during the symbolic execution (explained in Section 4.2.4) may produce paths of an infinite length. To ensure that the computation terminates, we need to define a bound to limit the unfolding of those loops. Hence, we decided to manually terminate the unfolding of the loop to a certain definite length (In our case, we decided to allow $n \in \mathbb{N}$ consecutive occurrences, the bound was set to, $n = 10$).



(a) Sequence Diagram (b) IOSTS model

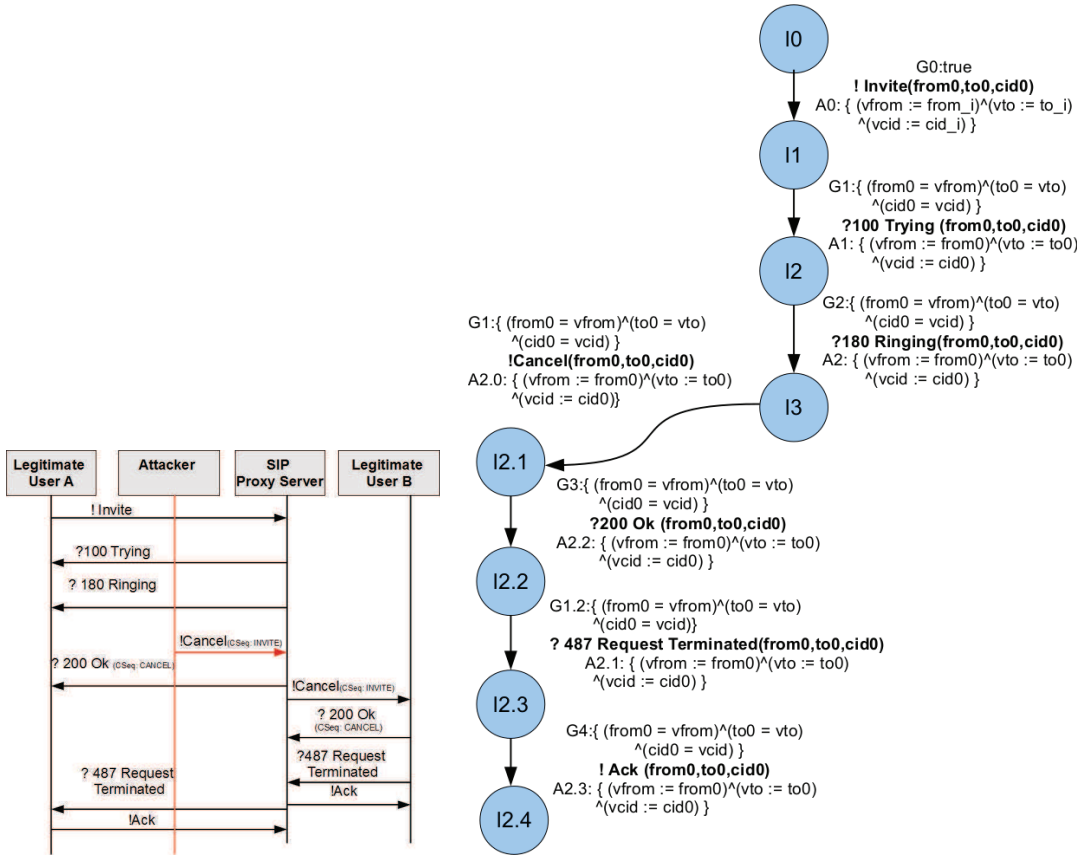
FIGURE 4.7: SIP: Session Establishment Property in SIP.

4.2.3.3 Session Establishment Property in SIP

Figure 4.7(a) shows the sequence diagram for SIP session establishment between the user's using a single proxy. The formal IOSTS model is shown in Figure 4.7(b). If the user A is previously registered to the proxy server, then an *Invite* request is send via a proxy server to the user B shown by the state transition from *l0-l1*. The *Invite* request asks the server to establish a dialog. A *100 Trying* response is send back to the proxy server to indicate that it has received the *Invite* and is processing the request which is shown by the state transition from *l1-l2*. *180 Ringing* response is send back to user A to indicate the SIP phone of user B is ringing shown by *l2-l3*. Then the user B can respond with a *200 Ok* response if it accepts the call from the user A, *l3-l4*. A *200 Ok* response to an *Invite* establishes a session, and it also creates a dialog between the user A that issued the *Invite* and the user B that generated the *200 Ok* response. Now, the user A needs to send an *Ack* for every final response it receives from the user B shown by the state transition from *l4-l5*. In another scenario, if the server requires the authorization credentials from the user A before establishing the call, the server

sends a *407 Proxy Authorization* response containing the challenge information shown by the state transition from *l1-l1.1*. A new *Invite* is then send containing the correct credentials, *l1.1-l1* , if the information are verified correct by the proxy, the call proceeds as described earlier and the session is established successfully.

4.2.3.4 Session Teardown attack in SIP



(a) Sequence Diagram (b) IOSTS model

FIGURE 4.8: Session Teardown Attack.

An attacker can cause session teardown attack by sending *Cancel/Bye* message as shown in Figure 4.8(a). The corresponding IOSTS model is shown in Figure 4.8(b). A *Cancel* message is used to cancel the ongoing transaction (before the session is established, i.e. before receiving the *200 Ok* response from the other user) as shown by the transition from state *l3-l2.1*. In this attack the attacker impersonates as a legitimate user *A* and send the *Cancel* request to cancel the *Invite* request generated by a user *A*[1].

[1]http://tools.ietf.org/html/rfc3665

Thus, the attacker send a *Cancel* and gives up on the call before user B answers (send a *200 OK* response). Any *Cancel* message is acknowledged with a *200 OK* on a hop by hop basis, rather than end to end as shown by the state transition from *l2.1-l2.2*. The proxy server that receives a *Cancel* requests for an *Invite*, but has not yet send a final response, would respond to the *Invite* with a *487 Request Terminated* error response as shown by the state transition from *l2.2-l2.3* and finally the call termination is acknowledged by the user A by sending an *Ack* shown by the state transition from *l2.3-l2.4*.

In the next section, we represent the symbolic execution of the IOSTS model representing the SIP properties and attacks as a symbolic tree. The branches in the tree correspond to the behavior of the property and attack scenarios presented in this section.

4.2.4 Symbolic Execution

Example 4.1. *Let us consider the symbolic execution tree in Figure 4.9(a) representing the registration property and attacks in SIP*[2] *as explained in Section 3.2.2.1. For readability reasons, the assignments and guard conditions are not directly illustrated inside the symbolic states but in separate frames. In this symbolic execution tree we have four different paths.*[3]

(1) Registration without authentication - (init, !Register(from0, to0, cid0), η1), (η1, ?200Ok (from0, to0, cid0), η2)

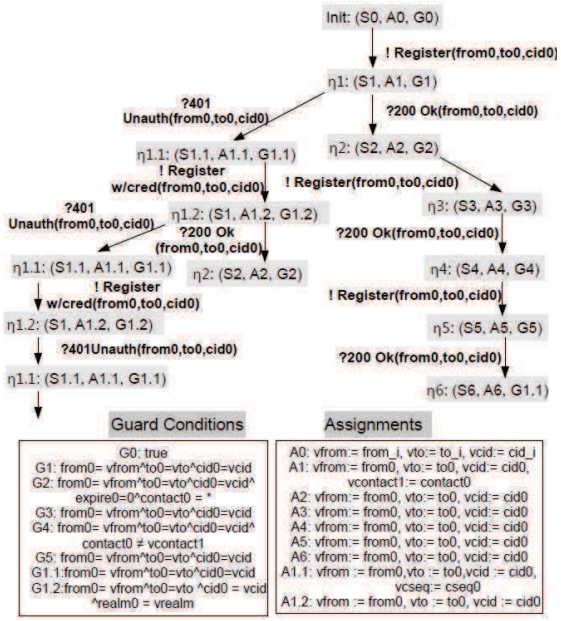
(2) Registration with authentication - (init, !Register(from0, to0, cid0), η1), (η1, ?401Unauth (from0, to0, cid0), η1.1), (η1.1, !Registerw/cred(from0, to0, cid0), η1.2), (η1.2, ?200Ok(from0, to0, cid0), η2)

(3) Registration Hijack attack - (init, !Register(from0, to0, cid0), η1), (η1, ?401Unauth(from0 , to0, cid0), η2), (η2, !Register(from0, to0, cid0), η3), (η3, ?200Ok(from0, to0, cid0), η4), (η4, !Register(from0, to0, cid0), η5), (η5, ?200Ok(from0, to0, cid0), η6)

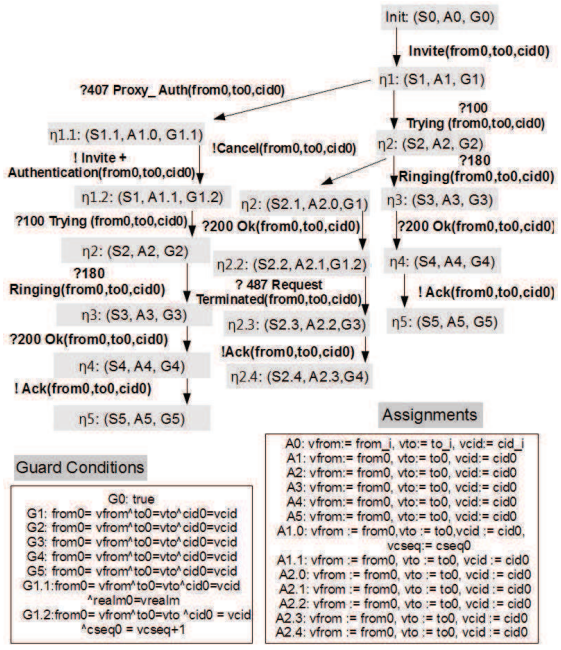
(4) DoS attack - (init, !Register(from0, to0, cid0), η1), (η1, ?401Unauth(from0, to0, cid0), η1.1), (η1.1, !Registerw/cred(from0, to0, cid0), η1.2), (η1.2, ?401Unauth(from0, to0, cid0) , η1.1), (η1.1, !Register(from0, to0, cid0), η1.2), (η1.2, ?401Unauth(from0, to0, cid0), η1.1)

[2]The symbolic parameters in the message corresponds to the $From, To$ and $Call-ID$ field values in SIP (dialog parameters). However, there is no limitation in choosing the number of parameters.

[3]Paths of a symbolic execution are the sequences of symbolic transitions that start at the initial state (Section 3.2.2, Page 25 of [Bannour 2012]).



(a) SIP - Registration



(b) SIP - Session Establishment

FIGURE 4.9: Symbolic Execution of IOSTS with Security Attack scenarios.

Based on the definition 3.9, we obtain the symbolic traces for the paths defined above(for the Figure 4.9(a)) ,

$Trace(SE(\mathfrak{M})) = \{!\mathsf{Register}(\mathsf{from0}, \mathsf{to0}, \mathsf{cid0})?\mathsf{200Ok}(\mathsf{from0}, \mathsf{to0}, \mathsf{cid0}), !\mathsf{Register}(\mathsf{from0}, \mathsf{to0},$

cid0)?401Unauth(from0, to0, cid0)!Registerw/cred(from0, to0, cid0)?200Ok(from0, to0, cid0) , !Register(from0, to0, cid0)?200Ok(from0, to0, cid0)!Register(from0, to0, cid0)?200Ok (from0, to0, cid0)!Register(from0, to0, cid0)?200Ok(from0, to0, cid0), !Register(from0, to0, cid0)?401Unauth(from0, to0, cid0)!Registerw/cred(from0, to0, cid0)?401Unauth(from0, to0, cid0)!Registerw/cred(from0, to0, cid0)?401Unauth(from0, to0, cid0)}

.

The set of traces with only the control portion of the messages is given by, $CP[Trace(SE(M))] = \{$!Register?200Ok, !Register?401Unauth!Register?200Ok, !Register ?200Ok!Register?200Ok!Register?200Ok, !Register?401Unauth!Register?401Unauth !Register?401Unauth$\}$.

A similar description can be given for the Figure 4.9(b) denoting the session establishment property and attacks. The symbolic execution tree constitutes three different paths:
(1)Session establishment without authentication - (init, !Invite(from0, to0, cid0), η1), (η1, ?100Trying(from0, to0, cid0), η2), (η2, ?180Ringing(from0, to0, cid0), η3), (η3, ?200Ok (from0, to0, cid0), η4), (η4, !Ack(from0, to0, cid0), η5)
(2)Session establishment with authentication - (init, !Invite(from0, to0, cid0), η1), (η1, ?407 Proxy-Auth(from0, to0, cid0), η1.1), (η1.1, !Invite + Auth(from0, to0, cid0), η1.2), (η1.2, ?100Trying(from0, to0, cid0), η2), (η2, ?180Ringing(from0, to0, cid0), η3), (η3, ?200Ok(from0, to0, cid0), η4), (η4, !Ack(from0, to0, cid0), η5)
(3)Session Teardown attack - (init, !Invite(from0, to0, cid0), η1), (η1, ?100Trying(from0 , to0, cid0), η2), (η2, !Cancel(from0, to0, cid0), η2.1), (η2.1, ?200Ok(from0, to0, cid0), η2.2), (η2.2, ?487Request-terminated(from0, to0, cid0), η2.3), (η2.3, !Ack(from0, to0, cid0), η2.4)

The symbolic traces for the above defined paths are,
$Trace(SE(\mathfrak{M})) = \{$!Invite(from0, to0, cid0)?100Trying(from0, to0, cid0)?180Ringing (from0, to0, cid0)?200Ok(from0, to0, cid0)!Ack(from0, to0, cid0), !Invite(from0, to0, cid0) ?407Proxy-Auth(from0, to0, cid0)!Invite + Auth(from0, to0, cid0)?100Trying(from0, to0, cid0)?180Ringing(from0, to0, cid0)?200Ok(from0, to0, cid0)!Ack(from0, to0, cid0), !Invite(from0, to0, cid0)?100Trying(from0, to0, cid0)!Cancel(from0, to0, cid0)?200Ok (from0, to0, cid0)?487Request-terminated(from0, to0, cid0)!Ack(from0, to0, cid0)$\}$.

Likewise, the set of traces with only the control portion of the messages is given by, $CP[Trace(SE(M))] = \{$!Invite?100Trying?180Ringing?200Ok!Ack, !Invite?407Proxy-Auth

!Invite + Auth?100Trying?180Ringing?200Ok!Ack, !Invite?100Trying!Cancel?200Ok?487 Request-terminated!Ack}.

These symbolic traces and the state details of the SE tree (i.e., guard-conditions and assignments) are provided as input to the tool (Tool description is provided in Appendix A of this thesis). These symbolic traces are validated against the trace slice (as discussed in Chapter 3 of this thesis) to provide the final verdict for the identified SIP properties and attacks/vulnerabilities discussed in this section.

4.2.5 Experimental results

For the experiments, SIP traces were obtained from two different sources: (A) Trace 1 in Table 4.2 was obtained from a production IMS implementation, provided by the Alcatel-Lucent company and extracted from the interfaces of the IMS core as shown in Figure 4.1. (B) Traces 2-7 in Table 4.2 were obtained from SIPp [Hewlett-Packard 2004]. SIPp, provided by the Hewlett-Packard Company, is an Open Source SIP implementation of a test system conforming to the IMS as well as a testing tool and traffic generator for the SIP protocol. The SIP traces were captured using the protocol trace analyzer Wireshark [Wireshark 2006] as text format (.txt). The implementation and the sample files used for the experiments can be found at [TestSym-P 2013]. For the prototype tool description, TestSym-P, please refer to Appendix A and B.

In order to evaluate the efficiency of our approach, we performed our experiments in two ways: with unmodified traces and by manually introducing some errors like modifying the $From, To$ parameter fields in few packets and also by introducing few attack messages (as defined in Section 3.2.2.1) in the real traces (for traces 1,2,3,4). The resultant output verdicts[4] are shown in Table 4.2 and 4.3 for the session registration and establishment property in SIP respectively[5].

For example, consider the following sample messages taken from SIP trace, here we have modified the $From$ field in message 2 from $sip : zack@ims04.alu.net$ to $sip :$ **XXX**$@ims04.alu.net$.

[4]Final Verdicts: P-Pass, F-Fail, I-Inconclusive, AP-Attack pass.

[5]Note: The Attack Fail verdict is only obtained during the trace slice evaluations. For the final verdict, the Attack Fail is considered to be an Inconclusive (I) verdict (refer Subsection 3.4.2)

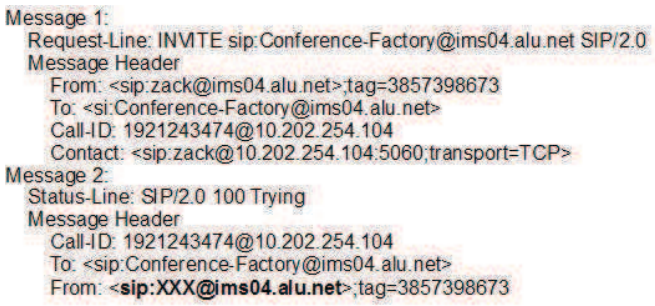


FIGURE 4.10: Sample SIP trace

TABLE 4.2: Results of Testing the Session Registration Property on sample SIP traces (Without Filters).

Trace	No.Packets	No.Slices	Trace Output without errors				Trace Outputs with errors and attacks				
			P	F	I	Verdict	P	F	I	AP	Verdict
1	5000	2719	680	-	2039	I	679	1	2039	-	F
2	25000	13895	3468	-	10427	I	3467	-	10427	1	AP
3	50000	27084	6764	-	20320	I	6762	1	20320	1	AP
4	100000	53286	13304	-	39981	I	13303	-	39981	1	AP
5	500000	272717	68179	-	204538	I	68179	-	204538	-	I
6	1000000	522102	130525	-	391577	I	130525	-	391577	-	I
7	5000000	1580288	395070	-	1185218	I	395070	-	1185218	-	I

TABLE 4.3: Results of Testing the Session Establishment Property on sample SIP traces (Without Filters).

Trace	No.Packets	No.Slices	Trace Output without errors				Trace Outputs with errors and attacks				
			P	F	I	Verdict	P	F	I	AP	Verdict
1	5000	2719	122	-	2597	I	121	-	2597	1	AP
2	25000	13895	328	-	13567	I	327	1	13566	1	AP
3	50000	27084	1282	-	25802	I	1280	2	25802	-	F
4	100000	53286	3307	-	49979	I	3303	4	49979	-	F
5	500000	272717	13635	-	259081	I	13636	-	259081	-	I
6	1000000	522102	26106	-	495996	I	26106	-	495996	-	I
7	5000000	1580288	79014	-	1501274	I	79014	-	1501274	-	I

Introducing errors in the message caused the guard conditions associated with a symbolic state to *Fail*. The results were successful and the errors and attacks introduced were correctly detected by the tool. Table 4.2 and 4.3 provide the verdicts obtained before and after introducing the errors and attack (before using filters). We observe that there are many inconclusive verdicts in our results. The main reason is that the trace is not filtered to verify a particular SIP property, hence the trace slicing will result in many unwanted slices (other SIP properties like subscription, notify, etc).

So experiments were conducted on unmodified sample SIP traces and the results obtained before and after adding filters are shown in Table 4.4 and 4.5 respectively. As a result, we were able to obtain improvement in the evaluation time[6]. Nevertheless adding filters improved the number of inconclusive verdicts. It could be further improved

[6]Evaluation Time is represented by - days:hrs:min:sec

if time constraints are included, as we need not wait until the end of the trace to provide the verdict.

TABLE 4.4: Results of Testing the Session Registration Property on sample SIP traces (With Filters).

Trace	No.Packets	Before filters			After filters		
		Time	I	No.Slices	Time	I	No.Slices
1	5000	00:00:01:36	2039	2719	00:00:01:17	680	1360
2	25000	00:00:19:49	10427	13895	00:00:15:56	3480	6948
3	50000	00:01:12:02	20320	27084	00:00:48:40	6778	13542
4	100000	00:05:37:11	39981	53286	00:02:29:38	13341	26646
5	500000	01:02:07:26	204538	272717	00:23:35:20	68179	136358
6	1000000	03:20:31:19	391577	522102	03:01:14:40	130526	261051
7	5000000	19:06:25:14	1185216	1580288	17:23:24:15	395075	790145

TABLE 4.5: Results of Testing the Session Establishment Property on sample SIP traces (With Filters).

Trace	No.Packets	Before filters			After filters		
		Time	I	No.Slices	Time	I	No.Slices
1	5000	00:00:01:30	2597	2719	00:00:01:18	1237	1359
2	25000	00:00:28:46	13567	13895	00:00:19:29	6619	6947
3	50000	00:01:12:04	25802	27084	00:00:49:49	12260	13542
4	100000	00:04:01:15	49979	53286	00:02:51:29	23333	26640
5	500000	00:23:15:20	259081	272717	00:18:24:32	122713	136349
6	1000000	03:04:37:08	305996	522102	02:18:55:13	234932	261038
7	5000000	18:21:13:32	1501274	1580288	17:13:08:54	711126	790140

In [Che et al. 2012] the prototype tool failed to provide verdict for large trace files ($> 10^6$), because of the way of expressing the property, analyzing them on the traces and the need to check the enumerated values. But we were able to successfully apply our approach to such traces to study the scalability and the results were promising. Moreover, the evaluation time taken for passively testing the SIP properties was also lesser than their approach. From the Figure 4.11, we observe that the filters have reduced the number of inconclusive verdicts, improved the evaluation time and also reduced the number of unwanted slices.

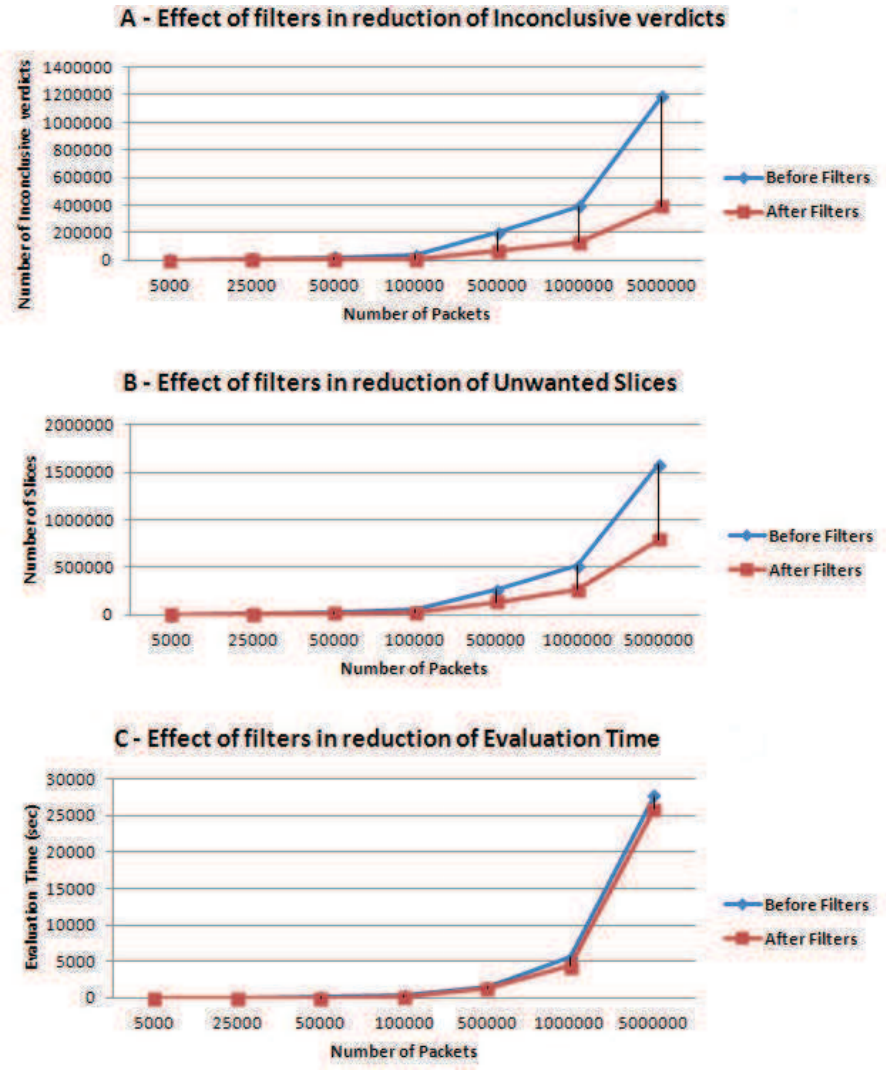


FIGURE 4.11: Effect of filters on sample SIP traces (Table 4.4) .

4.3 Case study 2: Bluetooth Protocol

As Information and Communication Technology (ICT) systems become more and more part of our daily lives, current and future vehicles are more and more integrated into ICT networks. The consumer's smart phones, multimedia devices etc. are linked to the vehicles and allow the drivers or passengers to use the internet, to access their private phone books or to run their individual applications through the vehicle's integrated infotainment systems. Today's most common technology to link consumer devices to in-vehicle electronics is *Bluetooth* [A.Miller and Bisdikian 2000], which latest version is 4.0. Such a wireless connection provides the most comfortable way for the driver to access a variety of services. However, this connection also implies the risk of possible misuse which leads to enhanced security issues and risks for the automotive electronics and with that for the passengers.

Recently the complexity of ICT systems and automotive electronics increases drastically and requires modern testing and validation methods, which allow handling of complex systems fast and efficient way. This is what we address in the automotive case study, where in we had access to the real infotainment platform through the ITEA2 DIAMONDS project[7]. This case study is provided by Dornier Consulting, an international consulting and project management company that operates in the fields of: traffic, transport, the environment and water.[8]. The System Under Test (SUT) or IUT is an automotive connectivity module, which provides the driver an ability to connect a mobile phone to the infotainment system. The module itself is connected via the controller area network (CAN) bus to the vehicle. The phone can be linked via the Bluetooth technology. In this case study, the Bluetooth specification 2.0 [Group 1999] was used. An overview of the SUT is shown in Figure 4.12. The connection to a mobile phone is possible over the Bluetooth network, whereas additional USB devices can be attached via a USB interface. In this section we illustrate our symbolic passive testing approach by its application to a set of real execution traces extracted from a real automotive Bluetooth framework with functional and security properties. Part of this work was published in [Mouttappa et al. 2013a].

[7]Research supported by the European project DIAMONDS (EUREKA-ITEA2 project (2010-2013)(http://www.itea2-diamonds.org/index.html)

[8]http://www.dornier-consulting.com/

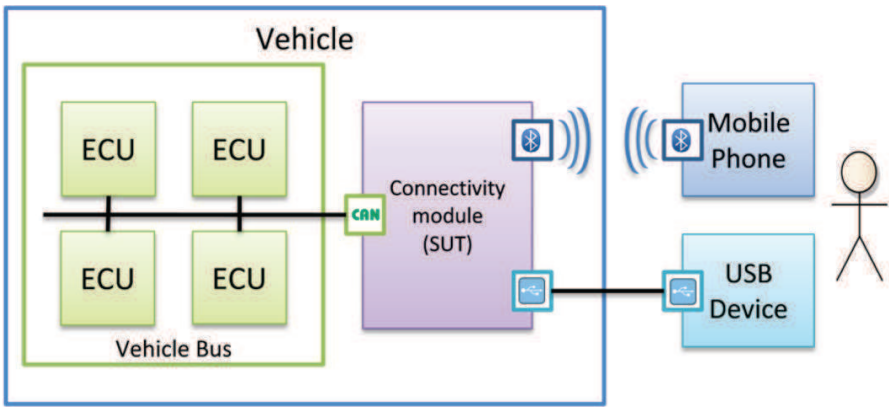


FIGURE 4.12: Overview of the SUT

4.3.1 Overview of Bluetooth protocol

The Bluetooth protocol operates at 2.4 GHz frequency in the free ISM-band (Industrial, Scientific, and Medical) by using frequency hopping. Bluetooth is a technology for short range wireless data and real-time two-way voice transfer providing data rates up to 3 Mb/s. Moreover, Bluetooth networks are formed by radio links, which means that there are additional security aspects whose impact is not yet well understood. Almost any device can be connected to another device by using Bluetooth.

Bluetooth devices that communicate with each other form a piconet. The device that initiates a connection is the piconet master. One piconet can have a maximum of seven active slave devices and one master device. All communication within a piconet goes through the piconet master. The clock of the piconet master and frequency hopping information are used to synchronize the piconet slaves with the master. Two or more piconets together form a scatternet, which can be used to eliminate Bluetooth range restrictions. A scatternet environment requires that different piconets must have a common device, called a scatternet member, to relay data between the piconets. Many kinds of Bluetooth devices, such as mobile phones, headsets, PCs, laptops, printers, mice and keyboards, are widely used all over the world.

4.3.1.1 Bluetooth Stack

The core protocols form a five-layer stack as shown in Figure 4.13 consisting of the following elements:

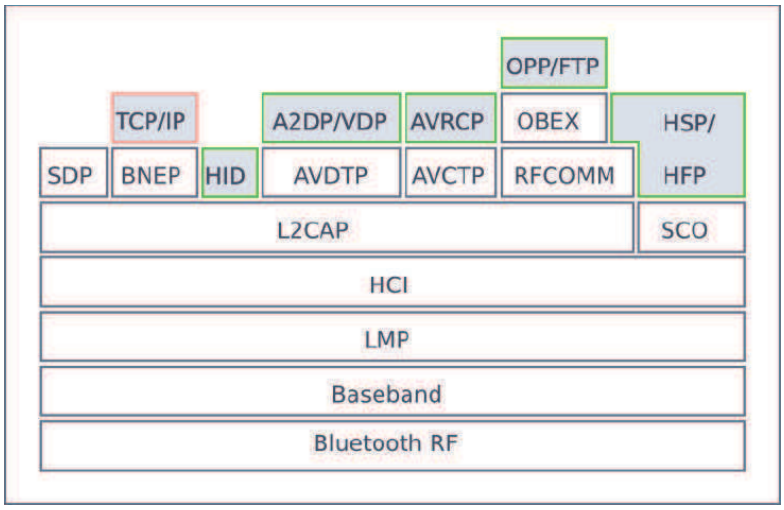


FIGURE 4.13: Bluetooth Stack

- Radio: Specifies the details of the air interface, including frequency, the use of frequency hopping, modulation scheme, and transmit power.
- Baseband: Concerned with connection establishment within a piconet, addressing, packet format, timing, and power control.
- Link Manager Protocol (LMP): Responsible for linking setup between Bluetooth devices and ongoing link management (this includes security aspects such as authentication and encryption, plus the control and negotiation of baseband packet sized).
- Logical Link Control and Adaptation Protocol (L2CAP): Adapts upper-layer protocols to the baseband layer. L2CAP provides both connectionless and connection-oriented services.
- Service Discovery Protocol (SDP): Device information, services, and the characteristics of the services can be queried to enable the establishment of a connection between two or more Bluetooth devices.

The stack is primary divided into a Controller part and a Host part. The Controller comprises of the Bluetooth Radio, Baseband and the Link Manager Protocol. It is performed in hardware for obvious reasons. Host deals with high level data, and is usually built in software. Between the Host and the Controller, there is an *Host/Controller Interface* (HCI), whose messages are mainly considered for our analysis in this thesis.

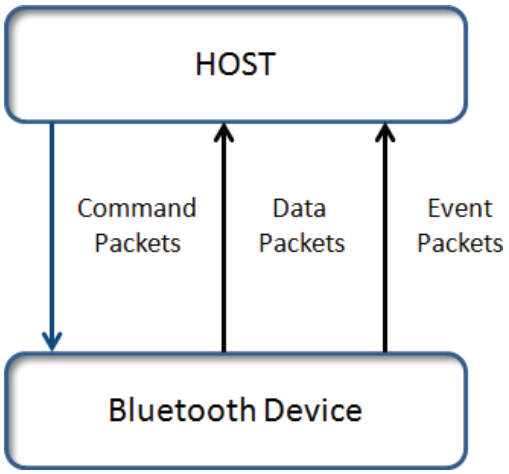


FIGURE 4.14: HCI packets flow

Host Controller Interface (HCI) The Host Controller Interface (HCI) forms the interface between the software protocol stack and the Link Manager underneath it, which is implemented in the firmware of a Bluetooth device. Notice that this is a packet-oriented communication between HCI and the Link Manager rather than a device driver. The difference is that HCI does not access the register and the memory locations of a Bluetooth device directly. Instead, it sends command and data packets to the device and receives data packets and event-message packets from this device, see Figure 4.14. This means that the Host Controller Interface offers a uniform interface for accessing the hardware.

The Bluetooth standard for the host controller interface defines the following:
- Command packets used by the host to control the module
- Event packets used by the module to inform the host of changes in the lower layers
- Data packets to pass voice and data between host and module
- Transport layers which can carry HCI packets

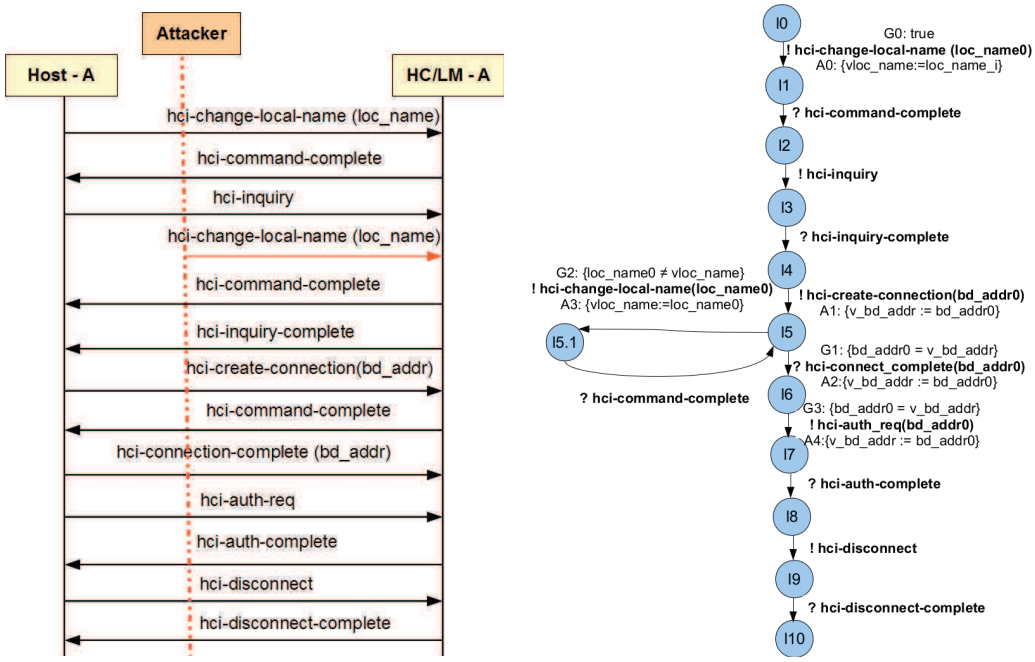
- HCI Commands: The host to control the Bluetooth module and to monitor its status uses HCI commands. Commands are transferred using HCI command packets. If a command can complete immediately, an HCI-Command-Complete is returned to indicate that the command has been dealt with. If a command cannot complete immediately, an HCI-Command-Status event is returned immediately, and another event is returned later when the command has completed.

FIGURE 4.15: A piconet hacked by an attacker

- HCI Data Packets: HCI Data Packets are used to pass both data (ACL) and voice (SCO) information over the HCI. Different packets are used for ACL and SCO data.
- HCI Event Packets: The format of the HCI event packets is similar to the HCI Command Packets. They carry an event code identifying the event.
- The HCI Transport Layer: A transport layer is required to get HCI packets from the host to the Bluetooth module.

4.3.2 IOSTS modelling of a Bluetooth behavior/attack

In Bluetooth, a piconet is a basic networking unit consisting of one master and a maximum of 7 active slave devices. The smallest piconet consists of two Bluetooth devices, one master and one slave as shown in Figure 4.15. The device which initiates the process of forming a piconet is considered as the master and the device which is ready to get connected with the master is considered as the slave. Like many other communication technologies, Bluetooth is composed of a hierarchy of components referred to as a stack [Group 1999]. The automotive case study partners provided us few traces mainly concerning the HCI layer messages to depict the device discovery and connectivity behavior between the Bluetooth devices. We tried to apply our symbolic passive testing approach to verify the aforementioned Bluetooth protocol behaviors.



(a) Sequence Diagram (b) IOSTS model

FIGURE 4.16: Bluetooth Call establishment and Bluestabbing attack.

4.3.2.1 Bluetooth call establishment property

Figure 4.16(a) shows the message sequences captured from the HCI layer of the master (car's Bluetooth device) while trying to achieve the Bluetooth connectivity with the slave device (mobile phone). Each Bluetooth device has a device local name, a user-friendly name to identify the different Bluetooth devices. This device name can be initially configured by each host by sending an *hci-change-local-name* message to the host controller (HC/LM-A). A Bluetooth device in *discoverable* mode can communicate or be visible to other Bluetooth devices. If it does not prefer to be visible, it could be in *non-discoverable* mode. Devices which are in discoverable mode are only eligible to participate in the piconet. So when one Bluetooth device wants to connect to another one, it must go through certain steps to learn and authenticate with the remote device. The master first finds the other devices which are in "discoverable" mode, and then performs an *inquiry* on each device by sending an *hci-inquiry* message. Thus the inquiry process gives the master a list of hardware addresses called *bd_addr* which are available to be connected and the important device feature information. The *bd_addr* is a unique address of a Bluetooth device, similar to MAC address of a network device. This address is needed for

further communications with a device. Having received the slave addresses, the master can establish an actual connection with one or more of the devices it found via the *paging* process. During the paging process the master sends an *hci-create-connection* message to establish a connection with a particular slave (based on the parameter *bd_addr*). The connection is successfully established upon receiving an *hci-connection-complete* message. Authentication can be explicitly executed at any time after a connection has been established by an *hci-auth-req* message. And the established connection can be anytime detached by the master device by an *hci-disconnect* message [Group 1999].

4.3.2.2 Bluetooth attack - Bluestabbing

Usually, the list of discovered Bluetooth devices displays only the name of the located device, and it does not show the actual Bluetooth address. If the slave devices are familiar with the located device name they are in discoverable mode else invisible. This local name can be changed anytime by any one, hence prone to *Bluestabbing attack* [Browning and Kessler 2009] as shown in Figure 4.16(a). In the Bluestabbing attack, the attacker impersonates as a legitimate user and modifies the Bluetooth device name of a legitimate user by resending an *hci-change-local-name* message with a badly formatted device name by causing the slave device to confuse during the device discovery phase (Inquiry). But this attack could be more severe, if the Bluetooth attacker modifies his own local device name as a legitimate user's device name, and tries to establish a connection with the other Bluetooth device by capturing the passwords and sensitive informations from the device.

For a better understanding of the IOSTS formalism, we represent the Bluetooth behavior along with the attack scenario in Figure 4.16(b). We observe that there is a transition from state $l5$ to state $l5.1$, a deviation from the regular scenario due to the *hci-change-local-name* message inserted by the attacker, during the device inquiry phase, resulting in the Bluestabbing attack. For explanation we have considered only few parameters bd_addr, loc_name corresponding to the Bluetooth protocol, but in practice there is no limitation in considering the number of parameters. In Figure 4.16(b), $vloc\_name$ and v_bd_addr belongs to the set of system variables V and loc_name0 and bd_addr0 constitutes the set of formal parameters P. In an IOSTS, all the values of the parameters and variables are represented symbolically. Intuitively, variables are data

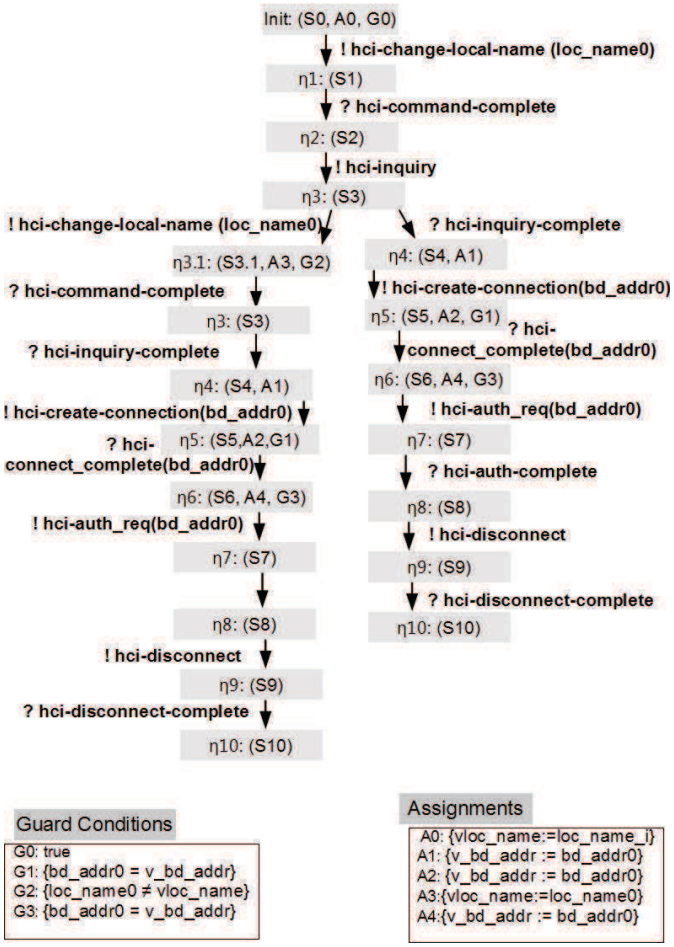


FIGURE 4.17: Symbolic execution of IOSTS.

to be compared with and parameters are data to communicate with the environment. For instance, the IOSTS tuple for the symbolic transition $t \in T$ from l_0 to l_1 can be expressed as below.

$$t : \langle l_0, (hci\text{-}change\text{-}local\text{-}name), true, ((vloc\_name := loc\_name\_i)), l_1 \rangle$$

Here, for any output action say, *hci-change-local-name*, if the guard-condition associated with the state is *true*, then there is a transition from state l_0 to l_1 and new assignments for the variables $(vloc\_name := loc\_name\_i)$ are performed.

4.3.3 Symbolic Execution

Figure 4.17 shows the symbolic execution of an IOSTS, where in the branches depicts the call establishment and security vulnerabilities in Bluetooth protocol. The symbolic execution tree consists of two paths:

(1)Call establishment - (init, !hci-change-local-name(loc_name0), η1), (η1, ?hci-command -complete, η2), (η2, !hci-inquiry, η3), (η3, ?hci-inquiry-complete, η4), (η4, !hci-create-connection(bd_addr0), η5), (η5, !hci-connect-complete(bd_addr0), η6), (η6, !hci-auth-req (bd_addr0), η7), (η7, !hci-auth-complete, η8), (η8, !hci-disconnect, η9), (η9, !hci-disconnect-complete, η10)

(2)Bluestabbing attack - (init, !hci-change-local-name(loc_name0), η1), (η1, ?hci-command -complete, η2), (η2, !hci-inquiry, η3), (η3, !hci-change-local-name(loc_name0), η3.1), (η3.1, !hci-command-complete, η3), (η3, ?hci-inquiry-complete, η4), (η4, !hci-create-connection(bd_addr0), η5), (η5, !hci-connect-complete(bd_addr0), η6), (η6, !hci-auth-req(bd_addr0), η7), (η7, !hci-auth-complete, η8), (η8, !hci-disconnect, η9), (η9, !hci-disconnect-complete, η10)

The symbolic traces for the above defined paths are,
$Trace(SE(\mathfrak{M}))$ = {!hci-change-local-name(loc_name0)?hci-command-complete!hci-inquiry ?hci-inquiry-complete!hci-create-connection(bd_addr0)!hci-connect-complete(bd_addr0)!hci-auth-req(bd_addr0)!hci-auth-complete!hci-disconnect!hci-disconnect-complete, !hci-change-local-name(loc_name0)?hci-command-complete!hci-inquiry!hci-change-local-name(loc_name0) !hci-command-complete?hci-inquiry-complete!hci-create-connection(bd_addr0)!hci-connect-complete(bd_addr0)!hci-auth-req(bd_addr0)!hci-auth-complete!hci-disconnect!hci-disconnect-complete}.

The set of traces with only the control portion of the messages is given by,
$CP[Trace(SE(M))]$ = {!hci-change-local − name?hci-command-complete!hci-inquiry?hci-inquiry-complete!hci-create-connection!hci-connect-complete!hci-auth-req!hci-auth-complete !hci-disconnect!hci-disconnect-complete, !hci-change-local-name?hci-command-complete !hci-inquiry!hci-change-local-name!hci-command-complete?hci-inquiry-complete!hci-create-connection!hci-connect-complete!hci-auth-req!hci-auth-complete!hci-disconnect!hci-disconnect-complete}.

4.3.4 Experimental Results

For the experiments, 10 traces for a Bluetooth session establishment were provided by the Dornier Consulting company. Each of the obtained Bluetooth trace had different device local name. Figure 4.18 shows a snapshot of the sample Bluetooth trace. In order to evaluate the efficiency of our approach, we performed our experiments in two ways:

TABLE 4.6: Prototype Tool results on sample Bluetooth traces ρ.

Trace	No. Messages	No.Slices	Trace Output without errors				Trace Outputs with errors and attacks				
			P	F	I	Final O/P	P	F	I	AP	Final O/P
1	81	2	1	-	1	I	-	1	1	-	F
2	89	3	1	-	2	I	-	-	2	1	AP
3	81	2	1	-	1	I	-	1	1	-	F
4	81	2	1	-	1	I	-	-	1	1	AP
5	81	2	1	-	1	I	-	1	1	-	F
6	81	2	1	-	1	I	-	-	1	1	AP
7	81	2	1	-	1	I	-	-	1	1	AP
8	81	2	1	-	1	I	-	-	1	1	AP
9	81	2	1	-	1	I	-	1	1	-	F
10	81	2	1	-	1	I	-	-	1	1	AP

Verdicts: P-Pass, F-Fail, I-Inconclusive, AP-Attack pass

```
HCI-EVT: HCE_CONNECT_REQUEST +00.000s [Length 10]
         BD_ADDR:00:26:7E:4A:77:DC ConnType:ACL_Connection
         MajorService:Rendering|Object Transfer|Audio MajorDevice:Audio MinorDevice:08
HCI-EVT: HCE_CONNECT_COMPLETE +00.001s [Length 11]
         Status:HOST_TIMEOUT ConnHndl:11 BD_ADDR:00:26:7E:4A:77:DC
         LinkType:ACL_Connection EncryptMode:Disabled
HCI-CMD: HCC_RESET +00.001s [Length 0]
HCI-EVT: HCE_COMMAND_COMPLETE +00.004s [Length 4]
         NumHCICommandPackets:1 Command:HCC_RESET
         Status:NO_ERROR
HCI-CMD: HCC_READ_BUFFER_SIZE +00.004s [Length 0]
HCI-EVT: HCE_COMMAND_COMPLETE +00.006s [Length 11]
         NumHCICommandPackets:1 Command:HCC_READ_BUFFER_SIZE
         Status:NO_ERROR
         ACL Buffer Len =  1021, Num ACL Buffers = 8
         SCO Buffer Len =    64, Num SCO Buffers = 1
HCI-CMD: HCC_HOST_BUFFER_SIZE +00.006s [Length 7]
         ACL Buffer Len =  1021, Num ACL Buffers = 120
         SCO Buffer Len =    60, Num SCO Buffers = 20
HCI-EVT: HCE_COMMAND_COMPLETE +00.008s [Length 4]
         NumHCICommandPackets:1 Command:HCC_HOST_BUFFER_SIZE
         Status:NO_ERROR
```

FIGURE 4.18: Sample Bluetooth trace

with unmodified traces and by manually introducing errors and also by introducing few fake messages to create a vulnerability in the real trace. For example, in the traces 1,3,5,9 we tried to modify manually the parameter *bd_addr* in Bluetooth message like *hci-connect-complete* so that we can obtain *Fail* verdict. Introducing error in the message caused the guard conditions associated with the symbolic state $\eta 5$ to *Fail* as shown in Figure 4.17.

In order to detect the attack scenario explained in the Section 4.3.2.2, we introduced few fake messages to the traces 2,4,6,7,8,10 to create a Bluestabbing attack scenario. For example, as shown in the Figure 4.17 we manually introduced fake message like *hci-change-local-name* before the inquiry complete phase in the real-time trace obtained. The attacks introduced were also correctly detected by our tool.

The verdicts obtained before and after introducing the errors are provided in the Table 4.6. The evaluation time to passively test the sample traces were approximately less than 1 second for each trace. As the main objective of the automotive case study was to detect some abnormalities during the Bluetooth connection establishment phase, only HCI layer messages were monitored. That was the reason for the short traces

in our experiments, however, we would like to extend our approach to passively test more complex systems as future works. Our prototype and the sample files used for the experiments can be found at http://www-public.it-sudparis.eu/~mouttapp/TestSym.html.

4.4 Conclusion

In this chapter we presented the results of applying our symbolic passive testing approach to two different case studies: Session Initiation Protocol (SIP) and Bluetooth protocol. In our work, the sequence of observed events which are called *traces* are collected offline and are analyzed to check whether they meet the defined property or system behaviors. Our approach is the integration of two different techniques, symbolic execution to model the property/attack and parametric trace slicing to perform trace analysis. The properties to passively test are modeled using the IOSTS (Input-Output Symbolic Transition Systems) formalism and the system trace are partitioned into several slices based on the different parametric instances of the trace. The symbolic property is verified against the trace slices and a verdict *(Pass/Fail/Inconclusive/Attack-Pass)* on the conformance of the property is provided.

In order to prove the efficiency of the implemented approach, failures and attacks were manually added to the system traces and analyzed. Experimental results for the two case studies prove the capability of the implemented approach to detect failures and attacks in the traces. Although the number of parameters that were considered in the messages seems to be small, there is actually no limit in the number of parameters, which seems to be an added advantage of this approach. Some perspectives regarding the future works and research ideas are provided along with the general conclusion of our work in the following chapter and a more detailed study of the implemented prototype model, TestSym-P is given in the Appendix A and B.

The work done so far in the course of this thesis already contributes to the efficiency and scalability of passive testing technique. Further, it stimulates further research in the field of symbolic passive testing.

Chapter 5

General Conclusion

Contents

5.1 Summary

Motivation The main objective of the presented work was to address some of the limitations of the existing passive testing for conformance, and to present a new symbolic passive testing approach applicable to communication protocols (i.e. message-based or text-based protocols).

This thesis has been placed in the area of black-box conformance testing for reactive systems. In general, the model of IOLTS is more suitable for the testing of reactive systems than the FSM. Nevertheless, the theories and tools based on IOLTS are somewhat limited, i.e., they do not explicitly take into account the system data because the underlying model of IOLTS does not allow to do it. Thus, in order to formally model the properties of the reactive system modeled by an IOLTS, it is necessary to enumerate the values of each datum used by the system. This may result in the classical state-space explosion problem. Another important aspect that needs to be considered is the data relationship between the messages. Most of the current passive conformance testing approaches focuses only on monitoring the control portion of the messages neglecting the data portion. Considering the control portion alone sometimes leads to

several *false-positive verdicts* (as explained with an example in Chapter 1) which needs to be eliminated or reduced during passive testing. In addition to the aspects mentioned above, we are also interested in the practical implementation of the proposed work, as a tool *TestSym-P*.

Summary of the Thesis. In Chapter 1, we introduced the importance of reactive systems in daily life. Such systems are usually large and complex, and it can be error prone. Even a small error may lead to serious dis-functioning of the system. In this thesis we focused our work on *passive testing* techniques to monitor the system behaviors and attacks leading to the misbehavior of the system. In Chapter 2, we first presented briefly the state of the art of the most relevant works in the active and passive conformance testing families with formal specification. Some of these techniques have been based on the model of Finite State Machines (or FSM), and others on the model of (Input-Output) Labeled Transition Systems (or (IO)LTS).

In order to solve the issues mentioned above, we proposed a new symbolic passive testing approach in Chapter 3, which extends the Input-Output labeled transition systems (IOLTS) with variables, symbolic constants and communication parameters. These systems are called Input-Output Symbolic Transition Systems (or IOSTS). The symbolic passive testing technique integrates the concepts of *Symbolic execution* and *Parametric trace slicing* techniques. We have identified in the state of the art that the existing formalism for modeling large systems (property or behavior) lacks to define the data relationship between the messages and also have problem with data enumeration which results in state explosion. To avoid these problems, we adapted the symbolic execution technique, where the variables and parameters are represented as symbolic values rather than concrete data. In the traditional dynamic symbolic execution technique the data dependencies for symbolic variables are well captured. This symbolic representation eliminates the necessity for data enumeration.

In our work, we have defined the syntax and semantics of the IOSTS model and described how the symbolic formalism can be used to represent the system behavior and eventual vulnerabilities or attacks. This IOSTS formalism has the ability to model the system behavior by taking into account the control and data portion, thereby, defining the data relationship between the messages which helps to reduce *false-positive verdicts*. The symbolic execution of the IOSTS, basically results in a tree-like structure with

different branches constituting the behavior or attack scenarios that need to be passively tested.

For the trace analysis, we adapted the *parametric trace slicing* technique. The most important aspect in passive testing is the *homing-phase*, that is identification of the current configuration of the IUT. However, in our approach we do not consider the homing phase because the system trace is split into different slices based on certain parametric instance of interest. These slices are then monitored against the symbolic property. As mentioned in the Chapter 2, the identified trace analysis technique is formalism independent, hence could be applied to any existing formalism. An algorithm to perform the parametric trace slicing is defined in our work, which considers all the parametric instances observed in the system trace. An algorithm to evaluate the symbolic property against the obtained trace slices was implemented. The evaluation logic checks the data relationship between the messages by taking into account the control and data portions of the messages and provides a final verdict *Pass/Fail/Inconclusive/Attack-Pass*.

Finally, we presented in Chapter 4, the applicability of our symbolic passive testing procedure in the course of two case studies. The first case study dealt with the Session Initiation protocols (SIP) and interesting scenarios and attacks were defined and modeled using our symbolic approach. The second case study is of industrial relevance since it has been processed on a popular protocol, Bluetooth used in various devices, e.g., phones, laptop, etc. Our approach has been implemented into a prototype framework (TestSym-P, a brief discussion is provided in Appendix A and B of this thesis) and experimental traces have been provided to illustrate how properties and attack scenarios are defined with our symbolic methodology. We also discussed the results obtained to illustrate the effectiveness of the approach to detect correct and incorrect behaviors.

We believe that our symbolic passive testing approach deserves future attention, and that research to come can improve its applicability to industrial reactive systems. The perspectives that can be drawn from the case studies will be discussed in the following section.

5.2 Perspectives

In our work, we have provided a novel symbolic approach for testing conformance properties, which provides an interesting work when compared to testing of data portion in other passive testing approaches. Although interesting results have been obtained (see Chapter 4), there is still room for improvement. The future aspects and other possible improvements to our approach are discussed in the following.

Time constraints As it can be seen in the experimental results provided in Section 4.2.5, the number of inconclusive verdicts obtained was large. For this reason, we introduced filters in our work to monitor only specific protocol properties, the results obtained were so promising since it reduced the number of inconclusive verdicts. However, it would be an added advantage to include *time* constraints in the definition of properties, whenever provided by the protocol specification or requirements. Timing constraints can reduce further the number of inconclusive verdicts by limiting the evaluation algorithm to a set of well defined slices.

In order to include time constraints, we need to formally model our properties using *Timed Input-Output Symbolic Transition System (TIOSTS)*. This model is actually an extension of two existing models: Timed Automata and IOSTS. Basically, a TIOSTS is an automaton with a finite set of locations, variables used to represent the system data, and a finite set of clocks used to represent time evolution. An edge comprises a guard on variables and clocks, an action carrying parameters for the communication with its environment, an assignment of variables, and resets of clocks. Extending the current semantics and evaluation algorithm should be simple enough to handle the time constraints.

Fuzzing technique In order to prove the efficiency of the implemented prototype, the obtained traces are manually modified to include attack scenarios and to report failure verdicts. However, it would be interesting to apply fuzzing approaches to modify the data parameters in the trace automatically instead of manually as performed in the current work. The proposed trace slicing algorithm considers the order of the message in the trace, so introducing fuzzing concepts to introduce fake messages in the system trace should be carefully studied.

Modeling the system properties In the present work, the identified properties and attack scenarios are shown as sequence diagrams and then manually modeled using IOSTS formalism. However, in many cases it would be interesting to obtain the IOSTS model automatically. For this purpose, if we could represent the behaviors and attack scenarios using UML State Charts then we can use an already existing software for translating UML State Charts into IOSTS [Thurnher 2008]. By allowing the user to define system behaviors via UML State Charts, a better support for the modeling of large systems is provided.

The symbolic execution tree obtained by symbolically executing the IOSTS is also performed manually. Nevertheless, an algorithm can be implemented to obtain the symbolic execution tree from the IOSTS model. Thus, automatizing the formal modeling of system properties can avoid errors due to human interpretation and also saves lot of time.

Online testing Online testing, i.e., evaluation of properties as the implementation is being run, might also be a desired improvement, for instance for the detection of certain attacks and abnormal behavior of the implementation. Currently, we have applied our methodology to test a trace length $> 10^6$, to study the scalability. Although, the time complexity when compared to other passive testing approaches was promising but still an inordinate amount of system memory was required to generate the verdict. The reason is that, the large trace which was several GigaBytes was stored as SQL database in our prototype framework. During the evaluation, additional system memory was required to store the resultant slice table and the evaluation output. Nevertheless, online testing could hopefully improve the system memory usage, but the question of how the trace can be collected online and stored in the disk for processing requires some study.

Improvements on the evaluation logic In the IOSTS formalism, the transition between any two states is actually controlled by the guard-conditions. In the current logic, the guard-conditions are limited to '=' and $' \neq'$ (i.e., SQL logic was defined to perform pattern matching or string comparison) hence, a simple improvement can also be provided as future works to allow to define more complex relations between data fields. Addition of complex relations between the data fields in the guard-conditions, can definitely improve the testing of complex system properties and attack scenarios.

Although, the current evaluation logic is computationally solvable in principle, but still it can be improved by considering parallel processing of the evaluation properties.

Appendix A

Symbolic Framework for Passive Testing

TestSym-P is a tool that implements the symbolic passive testing methodology introduced in Chapter 3 in order to perform conformance testing and also to analyze some misbehaviors of the IUT. The proposed symbolic tool was implemented using SQL. SQL is used to process the huge amount of data contained in the captured traces. The architecture of the TestSym-P tool comprises of three key modules: Trace parsing, Trace slicing and Trace evaluation. Figure A.1 shows the input/output interaction between the key modules. A brief illustration of each module is described in the following sections. More details on the symbolic tool is available at the URL http://www-public.it-sudparis.eu/~mouttapp/TestSym.html

A.1 Trace Parsing

Trace parsing is mainly performed to filter the trace files keeping only the relevant information for the protocol(s) under test. The raw trace file collected from Wireshark or any network analyzer is exported in the form of raw text (.txt format) file and is given as input to the trace parsing module. The tool converts the text file format *.txt* to a tabular file format, $dbo.InputoExcel$ with the required parameters. This table $dbo.InputoExcel$ becomes the source database table for SQL. A snapshot of the database table obtained after parsing the SIP trace is shown in Figure A.2.

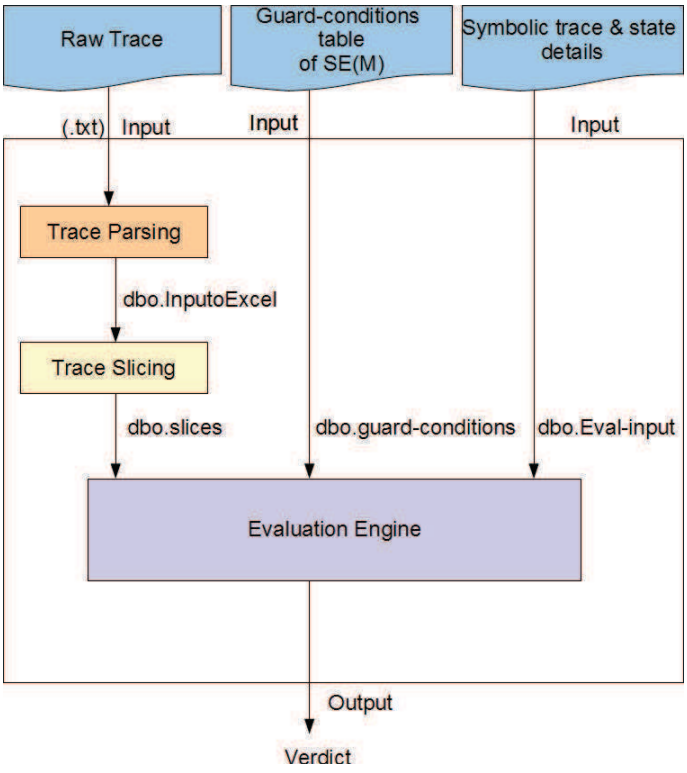


FIGURE A.1: TestSym-P, the prototype tool.

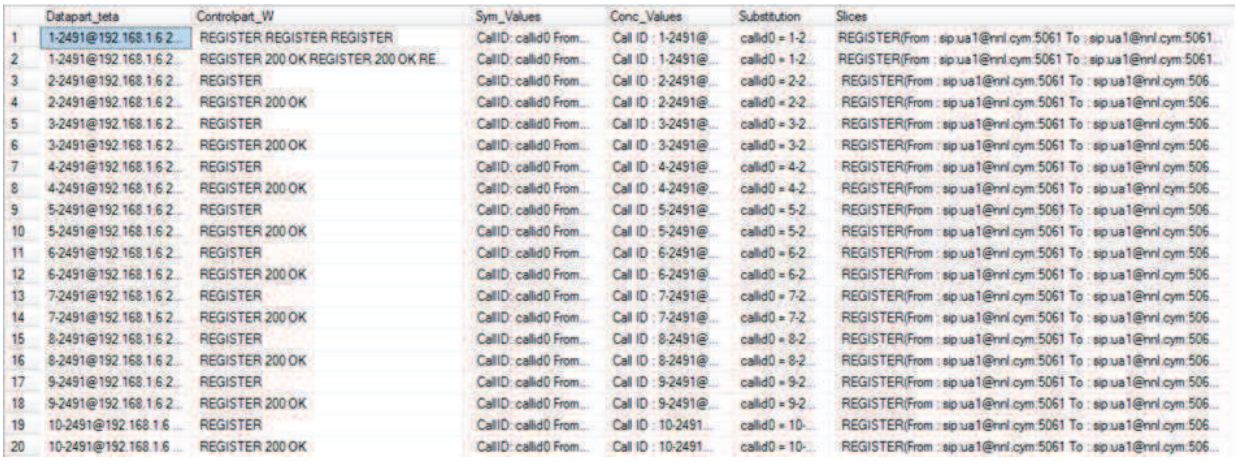
	ID	Method	From_	To_	Call_ID	FromTag	ToTag	Contact_expires	Contact_URI	CSeq_value	Expires	realm
1	1	REGISTER	sip:ua1@nnl.cym:5061	sip:ua1@nnl.cym:5061	1-2491@192.168.1.6	2491SIPpTag071			192.168.1.6	1	300	
2	2	200 OK	sip:ua1@nnl.cym:5061	sip:ua1@nnl.cym:5061	1-2491@192.168.1.6	2491SIPpTag071	2916SIPpTag081		192.168.1.5	1	300	
3	3	REGISTER	sip:ua1@nnl.cym:5061	sip:ua1@nnl.cym:5061	1-2491@192.168.1.6	2491SIPpTag071			*	1	0	
4	4	200 OK	sip:ua1@nnl.cym:5061	sip:ua1@nnl.cym:5061	1-2491@192.168.1.6	2491SIPpTag071	2916SIPpTag081		*	1	0	
5	5	REGISTER	sip:ua1@nnl.cym:5061	sip:ua1@nnl.cym:5061	1-2491@192.168.1.6	2491SIPpTag071			192.167.1.6	1	300	
6	6	200 OK	sip:ua1@nnl.cym:5061	sip:ua1@nnl.cym:5061	1-2491@192.168.1.6	2491SIPpTag071	2916SIPpTag081		192.167.1.6	1	300	
7	7	REGISTER	sip:ua1@nnl.cym:5061	sip:ua1@nnl.cym:5061	2-2491@192.168.1.6	2491SIPpTag072			192.168.1.6	1	300	
8	8	200 OK	sip:ua1@nnl.cym:5061	sip:ua1@nnl.cym:5061	2-2491@192.168.1.6	2491SIPpTag072	2916SIPpTag082		192.168.1.5	1	300	
9	9	REGISTER	sip:ua1@nnl.cym:5061	sip:ua1@nnl.cym:5061	3-2491@192.168.1.6	2491SIPpTag073			192.168.1.6	1	300	
10	10	200 OK	sip:ua1@nnl.cym:5061	sip:ua1@nnl.cym:5061	3-2491@192.168.1.6	2491SIPpTag073	2916SIPpTag083		192.168.1.5	1	300	
11	11	REGISTER	sip:ua1@nnl.cym:5061	sip:ua1@nnl.cym:5061	4-2491@192.168.1.6	2491SIPpTag074			192.168.1.6	1	300	
12	12	200 OK	sip:ua1@nnl.cym:5061	sip:ua1@nnl.cym:5061	4-2491@192.168.1.6	2491SIPpTag074	2916SIPpTag084		192.168.1.5	1	300	
13	13	REGISTER	sip:ua1@nnl.cym:5061	sip:ua1@nnl.cym:5061	5-2491@192.168.1.6	2491SIPpTag075			192.168.1.6	1	300	
14	14	200 OK	sip:ua1@nnl.cym:5061	sip:ua1@nnl.cym:5061	5-2491@192.168.1.6	2491SIPpTag075	2916SIPpTag085		192.168.1.5	1	300	
15	15	REGISTER	sip:ua1@nnl.cym:5061	sip:ua1@nnl.cym:5061	6-2491@192.168.1.6	2491SIPpTag076			192.168.1.6	1	300	
16	16	200 OK	sip:ua1@nnl.cym:5061	sip:ua1@nnl.cym:5061	6-2491@192.168.1.6	2491SIPpTag076	2916SIPpTag086		192.168.1.5	1	300	
17	17	REGISTER	sip:ua1@nnl.cym:5061	sip:ua1@nnl.cym:5061	7-2491@192.168.1.6	2491SIPpTag077			192.168.1.6	1	300	
18	18	200 OK	sip:ua1@nnl.cym:5061	sip:ua1@nnl.cym:5061	7-2491@192.168.1.6	2491SIPpTag077	2916SIPpTag087		192.168.1.5	1	300	
19	19	REGISTER	sip:ua1@nnl.cym:5061	sip:ua1@nnl.cym:5061	8-2491@192.168.1.6	2491SIPpTag078			192.168.1.6	1	300	
20	20	200 OK	sip:ua1@nnl.cym:5061	sip:ua1@nnl.cym:5061	8-2491@192.168.1.6	2491SIPpTag078	2916SIPpTag088		192.168.1.5	1	300	

FIGURE A.2: A snapshot of the trace parsing table.

A.2 Trace Slicing

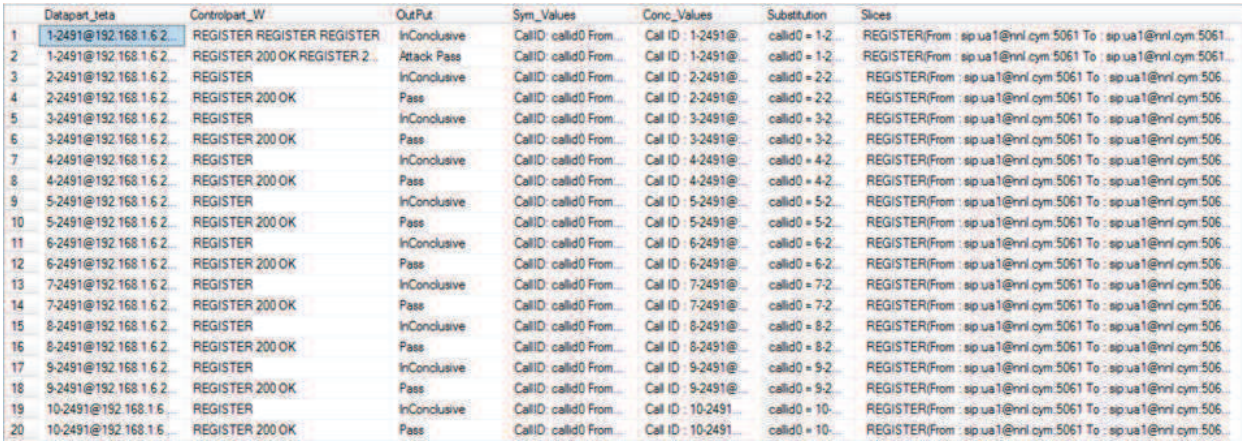
The traces are sliced based on certain parameters of interest, say for example, for SIP the dialog parameters: *Call-ID, From Tag* and *To tag* was chosen.*Call-ID* is a unique string that identifies a call, *From tag* is generated by the caller and uniquely identifies the dialog in the caller's user agent. *To tag* is generated by a callee and uniquely identifies

the dialog in the callee's user agent. In this module, the trace slicing algorithm presented in Section 3.3 is implemented. The formatted trace file *dbo.InputoExcel* is provided as input to the trace slicing module and the resultant slice table *dbo.Slices* is obtained as output. A snapshot of the resultant trace slicing table for SIP is shown in Figure A.3.



	Datapart_teta	Controlpart_W	Sym_Values	Conc_Values	Substitution	Slices
1	1-2491@192.168.1.6 2...	REGISTER REGISTER REGISTER	CallID: callid0 From...	Call ID : 1-2491@...	callid0 = 1-2...	REGISTER(From : sip:ua1@nnl.cym:5061 To : sip:ua1@nnl.cym:5061...
2	1-2491@192.168.1.6 2...	REGISTER 200 OK REGISTER 200 OK RE...	CallID: callid0 From...	Call ID : 1-2491@...	callid0 = 1-2...	REGISTER(From : sip:ua1@nnl.cym:5061 To : sip:ua1@nnl.cym:5061...
3	2-2491@192.168.1.6 2...	REGISTER	CallID: callid0 From...	Call ID : 2-2491@...	callid0 = 2-2...	REGISTER(From : sip:ua1@nnl.cym:5061 To : sip:ua1@nnl.cym:506...
4	2-2491@192.168.1.6 2...	REGISTER 200 OK	CallID: callid0 From...	Call ID : 2-2491@...	callid0 = 2-2...	REGISTER(From : sip:ua1@nnl.cym:5061 To : sip:ua1@nnl.cym:506...
5	3-2491@192.168.1.6 2...	REGISTER	CallID: callid0 From...	Call ID : 3-2491@...	callid0 = 3-2...	REGISTER(From : sip:ua1@nnl.cym:5061 To : sip:ua1@nnl.cym:506...
6	3-2491@192.168.1.6 2...	REGISTER 200 OK	CallID: callid0 From...	Call ID : 3-2491@...	callid0 = 3-2...	REGISTER(From : sip:ua1@nnl.cym:5061 To : sip:ua1@nnl.cym:506...
7	4-2491@192.168.1.6 2...	REGISTER	CallID: callid0 From...	Call ID : 4-2491@...	callid0 = 4-2...	REGISTER(From : sip:ua1@nnl.cym:5061 To : sip:ua1@nnl.cym:506...
8	4-2491@192.168.1.6 2...	REGISTER 200 OK	CallID: callid0 From...	Call ID : 4-2491@...	callid0 = 4-2...	REGISTER(From : sip:ua1@nnl.cym:5061 To : sip:ua1@nnl.cym:506...
9	5-2491@192.168.1.6 2...	REGISTER	CallID: callid0 From...	Call ID : 5-2491@...	callid0 = 5-2...	REGISTER(From : sip:ua1@nnl.cym:5061 To : sip:ua1@nnl.cym:506...
10	5-2491@192.168.1.6 2...	REGISTER 200 OK	CallID: callid0 From...	Call ID : 5-2491@...	callid0 = 5-2...	REGISTER(From : sip:ua1@nnl.cym:5061 To : sip:ua1@nnl.cym:506...
11	6-2491@192.168.1.6 2...	REGISTER	CallID: callid0 From...	Call ID : 6-2491@...	callid0 = 6-2...	REGISTER(From : sip:ua1@nnl.cym:5061 To : sip:ua1@nnl.cym:506...
12	6-2491@192.168.1.6 2...	REGISTER 200 OK	CallID: callid0 From...	Call ID : 6-2491@...	callid0 = 6-2...	REGISTER(From : sip:ua1@nnl.cym:5061 To : sip:ua1@nnl.cym:506...
13	7-2491@192.168.1.6 2...	REGISTER	CallID: callid0 From...	Call ID : 7-2491@...	callid0 = 7-2...	REGISTER(From : sip:ua1@nnl.cym:5061 To : sip:ua1@nnl.cym:506...
14	7-2491@192.168.1.6 2...	REGISTER 200 OK	CallID: callid0 From...	Call ID : 7-2491@...	callid0 = 7-2...	REGISTER(From : sip:ua1@nnl.cym:5061 To : sip:ua1@nnl.cym:506...
15	8-2491@192.168.1.6 2...	REGISTER	CallID: callid0 From...	Call ID : 8-2491@...	callid0 = 8-2...	REGISTER(From : sip:ua1@nnl.cym:5061 To : sip:ua1@nnl.cym:506...
16	8-2491@192.168.1.6 2...	REGISTER 200 OK	CallID: callid0 From...	Call ID : 8-2491@...	callid0 = 8-2...	REGISTER(From : sip:ua1@nnl.cym:5061 To : sip:ua1@nnl.cym:506...
17	9-2491@192.168.1.6 2...	REGISTER	CallID: callid0 From...	Call ID : 9-2491@...	callid0 = 9-2...	REGISTER(From : sip:ua1@nnl.cym:5061 To : sip:ua1@nnl.cym:506...
18	9-2491@192.168.1.6 2...	REGISTER 200 OK	CallID: callid0 From...	Call ID : 9-2491@...	callid0 = 9-2...	REGISTER(From : sip:ua1@nnl.cym:5061 To : sip:ua1@nnl.cym:506...
19	10-2491@192.168.1.6 ...	REGISTER	CallID: callid0 From...	Call ID : 10-2491...	callid0 = 10-...	REGISTER(From : sip:ua1@nnl.cym:5061 To : sip:ua1@nnl.cym:506...
20	10-2491@192.168.1.6 ...	REGISTER 200 OK	CallID: callid0 From...	Call ID : 10-2491...	callid0 = 10-...	REGISTER(From : sip:ua1@nnl.cym:5061 To : sip:ua1@nnl.cym:506...

FIGURE A.3: A snapshot of the trace slicing table.



	Datapart_teta	Controlpart_W	OutPut	Sym_Values	Conc_Values	Substitution	Slices
1	1-2491@192.168.1.6 2...	REGISTER REGISTER REGISTER	InConclusive	CallID: callid0 From...	Call ID : 1-2491@...	callid0 = 1-2...	REGISTER(From : sip:ua1@nnl.cym:5061 To : sip:ua1@nnl.cym:5061...
2	1-2491@192.168.1.6 2...	REGISTER 200 OK REGISTER 2...	Attack Pass	CallID: callid0 From...	Call ID : 1-2491@...	callid0 = 1-2...	REGISTER(From : sip:ua1@nnl.cym:5061 To : sip:ua1@nnl.cym:5061...
3	2-2491@192.168.1.6 2...	REGISTER	InConclusive	CallID: callid0 From...	Call ID : 2-2491@...	callid0 = 2-2...	REGISTER(From : sip:ua1@nnl.cym:5061 To : sip:ua1@nnl.cym:506...
4	2-2491@192.168.1.6 2...	REGISTER 200 OK	Pass	CallID: callid0 From...	Call ID : 2-2491@...	callid0 = 2-2...	REGISTER(From : sip:ua1@nnl.cym:5061 To : sip:ua1@nnl.cym:506...
5	3-2491@192.168.1.6 2...	REGISTER	InConclusive	CallID: callid0 From...	Call ID : 3-2491@...	callid0 = 3-2...	REGISTER(From : sip:ua1@nnl.cym:5061 To : sip:ua1@nnl.cym:506...
6	3-2491@192.168.1.6 2...	REGISTER 200 OK	Pass	CallID: callid0 From...	Call ID : 3-2491@...	callid0 = 3-2...	REGISTER(From : sip:ua1@nnl.cym:5061 To : sip:ua1@nnl.cym:506...
7	4-2491@192.168.1.6 2...	REGISTER	InConclusive	CallID: callid0 From...	Call ID : 4-2491@...	callid0 = 4-2...	REGISTER(From : sip:ua1@nnl.cym:5061 To : sip:ua1@nnl.cym:506...
8	4-2491@192.168.1.6 2...	REGISTER 200 OK	Pass	CallID: callid0 From...	Call ID : 4-2491@...	callid0 = 4-2...	REGISTER(From : sip:ua1@nnl.cym:5061 To : sip:ua1@nnl.cym:506...
9	5-2491@192.168.1.6 2...	REGISTER	InConclusive	CallID: callid0 From...	Call ID : 5-2491@...	callid0 = 5-2...	REGISTER(From : sip:ua1@nnl.cym:5061 To : sip:ua1@nnl.cym:506...
10	5-2491@192.168.1.6 2...	REGISTER 200 OK	Pass	CallID: callid0 From...	Call ID : 5-2491@...	callid0 = 5-2...	REGISTER(From : sip:ua1@nnl.cym:5061 To : sip:ua1@nnl.cym:506...
11	6-2491@192.168.1.6 2...	REGISTER	InConclusive	CallID: callid0 From...	Call ID : 6-2491@...	callid0 = 6-2...	REGISTER(From : sip:ua1@nnl.cym:5061 To : sip:ua1@nnl.cym:506...
12	6-2491@192.168.1.6 2...	REGISTER 200 OK	Pass	CallID: callid0 From...	Call ID : 6-2491@...	callid0 = 6-2...	REGISTER(From : sip:ua1@nnl.cym:5061 To : sip:ua1@nnl.cym:506...
13	7-2491@192.168.1.6 2...	REGISTER	InConclusive	CallID: callid0 From...	Call ID : 7-2491@...	callid0 = 7-2...	REGISTER(From : sip:ua1@nnl.cym:5061 To : sip:ua1@nnl.cym:506...
14	7-2491@192.168.1.6 2...	REGISTER 200 OK	Pass	CallID: callid0 From...	Call ID : 7-2491@...	callid0 = 7-2...	REGISTER(From : sip:ua1@nnl.cym:5061 To : sip:ua1@nnl.cym:506...
15	8-2491@192.168.1.6 2...	REGISTER	InConclusive	CallID: callid0 From...	Call ID : 8-2491@...	callid0 = 8-2...	REGISTER(From : sip:ua1@nnl.cym:5061 To : sip:ua1@nnl.cym:506...
16	8-2491@192.168.1.6 2...	REGISTER 200 OK	Pass	CallID: callid0 From...	Call ID : 8-2491@...	callid0 = 8-2...	REGISTER(From : sip:ua1@nnl.cym:5061 To : sip:ua1@nnl.cym:506...
17	9-2491@192.168.1.6 2...	REGISTER	InConclusive	CallID: callid0 From...	Call ID : 9-2491@...	callid0 = 9-2...	REGISTER(From : sip:ua1@nnl.cym:5061 To : sip:ua1@nnl.cym:506...
18	9-2491@192.168.1.6 2...	REGISTER 200 OK	Pass	CallID: callid0 From...	Call ID : 9-2491@...	callid0 = 9-2...	REGISTER(From : sip:ua1@nnl.cym:5061 To : sip:ua1@nnl.cym:506...
19	10-2491@192.168.1.6 ...	REGISTER	InConclusive	CallID: callid0 From...	Call ID : 10-2491...	callid0 = 10-...	REGISTER(From : sip:ua1@nnl.cym:5061 To : sip:ua1@nnl.cym:506...
20	10-2491@192.168.1.6 ...	REGISTER 200 OK	Pass	CallID: callid0 From...	Call ID : 10-2491...	callid0 = 10-...	REGISTER(From : sip:ua1@nnl.cym:5061 To : sip:ua1@nnl.cym:506...

FIGURE A.4: A snapshot of the trace evaluation table.

A.3 Final Evaluation

The evaluation algorithm defined in Section 3.4.1 is implemented in the evaluation engine module. Inputs to this module are: the trace slice table, *dbo.Slices*, a table comprising the symbolic traces, $Trace(SE(\mathfrak{M}))$, total number of states involved in the symbolic execution of the IOSTS and a table with the set of associated guard-conditions G for each state of the symbolic execution. First, the evaluation is carried on for each trace slices and the verdicts *Pass/Fail/Inconclusive/Attack-Pass/Attack-Fail* are obtained. A

snapshot of the verdicts obtained for each trace slice is shown in Figure A.4. The final evaluation logic is dependent on the verdicts obtained for each trace slice (as defined in Section 3.4.2) to prove the conformance of the system property on the trace. As a result, we obtain the final evaluation verdicts *Pass/Fail/Inconclusive/Attack-Pass*.

Appendix B

Inputs to the TestSym-P

The symbolic passive testing architecture, TestSym-P presented in Appendix A requires three different inputs (colored blue) as shown in the Figure A.1 in Appendix A:
- The communication traces represented in .txt format (captured using Wireshark for instance).
- The Guard conditions associated with each state of the symbolic executions.
- The symbolic traces collected from the symbolic execution of the IOSTS property, $Traces(SE(M))$ (i.e. the property that is to be verified on the IUT traces) and the number of states involved.

B.1 Raw Traces

The raw traces for the analysis are obtained by using a trace analyzer. The traces are exported in the form of raw text files (.txt) and these text files are provided as one of the input to the tool. For example, the SIP traces were collected using Wireshark and exported as text files for the analysis and for the Bluetooth protocol, Bluetooth sniffer was used to capture the traces from the Bluetooth stack implementation.

B.2 Guard-conditions

Guard-conditions are the necessary conditions that are required for the transition from one state to another in an IOSTS. These guard conditions are provided as SQL tables

to the tool. This table consists of two columns, the *ID* and the *GuardCondition*. The SQL procedures defined in the tool check the satisfiability of these guard-conditions for each defined state.

B.2.0.1 Guard-conditions table - SIP

Figure B.1 shows the snapshot of the guard-conditions table for the SIP properties explained in Chapter 4. For explanation, we have taken very few constraints mainly based on the SIP dialog parameters, i.e., *Call-ID, To-tag, From-tag*. However, the tool has no limitations on the number of parameters that can be defined to illustrate the constraints of the symbolic state.

ID	GuardCondition
1	True
2	from0 = s1.From , to0 = s1.To , cid0 = s1.Call-ID
3	from0 = s1.From , to0 = s1.To, cid0 = s1.Call-ID, expires0 = 0, contact0 = *
4	from0 = s1.From , to0 = s1.To, cid0 = s1.Call-ID
5	from0 = s1.From , to0 = s1.To, cid0 = s1.Call-ID, contact0 <> s1.Contact_URI
6	from0 = s1.From , to0 = s1.To, cid0 = s1.Call-ID
7	from0 = s1.From , to0 = s1.To , cid0 = s1.Call-ID, realm0 = s1.realm
8	from0 = s1.From , to0 = s1.To, cid0 = s1.Call-ID

(a) Registration property

ID	GuardCondition
1	True
2	from0 = s1.From , to0 = s1.To , cid0 = s1.Call-ID
3	from0 = s2.From , to0 = s2.To, cid0 = s1.Call-ID
4	from0 = s3.From , to0 = s3.To, cid0 = s1.Call-ID
5	from0 = s4.From , to0 = s4.To, cid0 = s1.Call-ID
6	from0 = s5.From , to0 = s5.To, cid0 = s1.Call-ID

(b) Session establishment property

FIGURE B.1: SQL table for Guard conditions - SIP

B.2.0.2 Guard-conditions table - Bluetooth

For the Bluetooth protocol, *bd-addr* was the most interesting parameter in the Bluetooth messages. The symbolic states with no guard-conditions are marked *NULL* in the table. Figure B.2 shows the guard-conditions table for analysing the Bluetooth trace.

ID	GuardCondition
1	True
2	NULL
3	NULL
4	NULL
5	NULL
6	bdaddr0 = s5.bdaddr
7	bdaddr0 = s5.bdaddr
8	bdaddr0 = s5.bdaddr
9	NULL
10	NULL
11	NULL
12	bdaddr0 = s5.bdaddr
13	NULL
14	NULL
15	NULL
16	bdaddr0 = s4.bdaddr
17	bdaddr0 = s4.bdaddr
18	bdaddr0 = s4.bdaddr
19	bdaddr0 = s4.bdaddr

FIGURE B.2: SQL table for Guard conditions - Bluetooth protocol

B.3 Symbolic state details

The symbolic state table provides the following details:
(i) the symbolic sequence, *Seqs*, which represents the property or attack scenario (with symbolic data)
(ii) the ID number from the guard-conditions table, *GuardCondRowNr*,associated with each sequence
(iii) number of states in the symbolic sequence, *NrOfStates* and
(iv) the type of sequence, *AttackSeq*, i.e., if $AttackSeq = 0$, it corresponds to a property else an attack sequence.

The SQL procedure uses this table to validate all the traces in the database.

B.3.0.3 Symbolic state table - SIP

Figure B.3 shows the snapshot of the symbolic state details for the identified properties and attack scenarios in SIP.

ID	Seqs	GuardCondRowNr	NrOfStates	AttackSeq
1	Register(from0,to0,cid0) 200 Ok(from0,to0,cid0)	1,2	3	0
2	Register(from0,to0,cid0) 401 Unauthorized(from0,to0,cid0) Register(from0,to0,cid0) 200 Ok(from0,to0,cid0)	1,2,7,8	5	0
3	Register(from0,to0,cid0) 200 Ok(from0,to0,cid0) Register(from0,to0,cid0) 200 Ok(from0,to0,cid0) Register(fro...	1,2,3,4,5,6	7	1
4	Register(from0,to0,cid0) 401 Unauthorized(from0,to0,cid0) Register(from0,to0,cid0) 401 Unauthorized(from0,t...	1,2,7,8,7,8	7	1

(a) Registration property

ID	Seqs	GuardCondRowNr	NrOfStates	AttackSeq
1	INVITE(from0,to0,cid0) 100 Trying(from0,to0,cid0)	1,2	2	0
2	INVITE(from0,to0,cid0) 100 Trying(from0,to0,cid0) 180 Ringing(from0,to0,cid0)	1,2,3,4	4	0
3	INVITE(from0,to0,cid0) 100 Trying(from0,to0,cid0) 180 Ringing(from0,to0,cid0) 200 OK(from0,to0,cid0)	1,2,3,4,5	5	1
4	INVITE(from0,to0,cid0) 100 Trying(from0,to0,cid0) 180 Ringing(from0,to0,cid0) 200 OK(from0,to0,ci...	1,2,3,4,5,6	6	1

(b) Session establishment property

FIGURE B.3: SQL table for Symbolic state details - SIP

B.3.0.4 Symbolic state table - Bluetooth protocol

Figure B.4 shows the snapshot of the symbolic state details for the identified property and attack scenario in Bluetooth protocol.

ID	Seqs	GuardCondRowNr	NrOfStates	AttackSeq
1	hcc_chng_local_name() hcc_inquiry() hce_inquiry_complete() hcc_create_connection(...	1,2,3,4,5,16,17,18,9,10,2,19,14,15	15	0
2	hcc_chng_local_name() hcc_inquiry() hcc_chng_local_name() hce_inquiry_complete() ...	1,2,3,4,4,5,6,7,8,9,10,11,12,13,14,15	17	1

FIGURE B.4: SQL table for Symbolic state details - Bluetooth protocol

Based on the final evaluation Algorithm illustrated in Chapter 3 (Section 3.4.2), the tool outputs the final verdict *Pass/Fail/Inconclusive/Attack-Pass*. A *Pass* if the property is satisfied, a *Fail*, if the property is not satisfied (i.e., if any of the guard-conditions fail to satisfy), an *Attack-pass*, if the attack sequence is satisfied and an *Inconclusice*, if the trace length is not sufficient to prove the verdict.

Bibliography

Aiguier, M., Gaston, C., Gall, P. L., Longuet, D., and Touil, A. (2005). A temporal logic for input output symbolic transition systems. In *Proceedings of the 12th Asia-Pacific Software Engineering Conference*, pages 43–50.

Alcalde, B., Cavalli, A. R., Chen, D., Khuu, D., and Lee, D. (2004). Network protocol system passive testing for fault management: A backward checking approach. In *Formal Techniques for Networked and Distributed Systems (FORTE)*, pages 150–166.

Alliance, O. M. (2006). Push to talk over cellular requirements, approved ver. 1.0.

A.Miller, B. and Bisdikian, C. (2000). *Bluetooth Revealed: The Insiders Guide to an Open Specification for Global Wireless Communications*. Prentice-Hall, NJ, USA.

Andrés, C., Merayo, M. G., and Núñez, M. (2008). Passive testing of timed systems. In *Automated Technology for Verification and Analysis (ATVA)*, pages 418–427.

Andrés, C., Merayo, M. G., and Núñez, M. (2012). Formal passive testing of timed systems: theory and tools. *Software: Testing, Verification and Reliability*, 22:365–405.

Arnedo, J., Cavalli, A., and Núñez, M. (2003). Fast testing of critical properties through passive testing. In *TestCom 2003*, pages 295–310.

Avgustinov, P., Tibble, J., and de Moor, O. (2007). Making trace monitoring feasible. In *In: OOPSLA07: ACM Conference on Object-Oriented Programming, Systems and Languages*, pages 589–608.

Bagnato, A., Raiteri, F., Mallouli, W., and Wehbi, B. (2010). Practical experience gained from passive testing of web based systems. In *Third International Conference on Software Testing, Verification, and Validation Workshops*, pages 394–402.

Bannour, B. (2012). *Symbolic analysis of scenario based timed models for component based systems: Compositionality results for testing.* PhD thesis, Institut CARNOT CEA LIST.

Bannour, B., Escobedo, J. P., Gaston, C., and Gall, P. L. (2012). Off-line test case generation for timed symbolic model-based conformance testing. In *ICTSS*, volume 7641 of *Lecture Notes in Computer Science*, pages 119–135. Springer.

Barringer, H., Goldberg, A., Havelund, K., and Sen, K. (2003). Rule-based runtime verification.

Barringer, H., Rydeheard, D., and Havelund, K. (2010). Rule systems for run-time monitoring: From eagle to ruler. *J. Log. Comput.,*, 20:675–706.

Bayse, E., Cavalli, A., Núñez, M., and Zaidi, F. (2005). A passive testing approach based on invariants: Application to the wap. *Computer Networks*, 48:247–266.

Beizer, B. (1990). *Software testing techniques (2nd ed.).* Van Nostrand Reinhold Co., New York, NY, USA.

Benharref, A., Dssouli, R., Serhani, M. A., En-Nouarry, A., and Glitho, R. (2007). New approach for efsm-based passive testing of web services. In *7th International Conference on Testing of Software and Communicating Systems*, pages 13–27.

Bentakouk, L., Poizat, P., and Zaidi, F. (2011). Checking the behavioral conformance of web services with symbolic testing and an smt solver. In *Proceedings of the 5th international conference on Tests and proofs*, pages 33–50.

Bodden, E. (2005). J-lo, a tool for runtime-checking temporal assertions. Master's thesis, RWTH Aachen University.

Browning, D. and Kessler, G. (2009). Bluetooth hacking: A case study. In *Proceedings of the Conference on Digital Forensics, Security and Law*, pages 20–22.

Cavalli, A., Prokopenko, S., and Gervy, C. (2003). New approaches for passive testing using an extended finite state machine specification. *Information and Software Technology*, 45:837–852.

Cavalli, A. and Tabourier, M. (1999). Passive testing and application to the gsm-map protocol. In *Information and Software Technology*, pages 813–821.

Che, X., Lalanne, F., and Maag, S. (2012). A logic-based passive testing approach for the validation of communicating protocols. In *Proceedings of the 7th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)*, pages 53–64.

Chen, F. and Rosu, G. (2007). Mop: An efficient and generic runtime verification framework. In *Proceedings of the OOPSLA'07*, pages 569–588.

Chen, F. and Rosu, G. (2009). Parametric trace slicing and monitoring. In *15th Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 246–261.

Collier, M. (2005). VoIP Vulnerabilities - Registration Hijacking. Technical report, SecureLogix Corportion.

Colombo, C., Pace, G. J., and Schneider, G. (2009). Safe runtime verification of real-time properties.

Falcone, Y., Havelung, K., and Reger, G. (2013). A Tutorial on Runtime Verification. In Manfred Broy, Doron Peled, G. K., editor, *Engineering Dependable Software Systems*, volume 34, pages 141–175. IOS Press.

Frantzen, L., Tretmans, J., and Willemse, T. A. C. (2005). Test generation based on symbolic specifications. In *FATES 2004, number 3395 in LNCS*, pages 1–15. Springer-Verlag.

Gall, P. L., Rapin, N., and Touil, A. (2007). Symbolic execution techniques for refinement testing. In *1st International conference on Tests and proofs (TAP'07)*, pages 131–148.

Gaston, C., Gall, P. L., Rapin, N., and Touil, A. (2006). Symbolic execution techniques for test purpose definition. In *18th IFIP Testing of Communicating Systems (TestCom)*, pages 1–18.

Ghezzi, C. and Guinea, S. (2007). Run-time monitoring in service-oriented architectures. In *Test and Analysis of Web Services 2007*, pages 237–264.

Goldsmith, S. and et al. (2005). Relational queries over program traces.

Group, B. S. I. (1999). Bluetooth specification version 2.0 + edr [vol 0]. Website. http://www.bluetooth.org/.

Halle, S. and Villemaire, R. (2008). Runtime monitoring of message-based workflows with data. In *EDOC*, pages 63–72. IEEE Computer Society.

Halle, S. and Villemaire, R. (2009). Runtime monitoring of web service choreographies using streaming xml. In *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 2118–2125.

Harel, D. and Pnueli, A. (1985). Logics and models of concurrent systems. chapter On the development of reactive systems, pages 477–498. Springer-Verlag New York, Inc., New York, NY, USA.

Henzinger, Mazumdar, T. A., and Rupak (1999). A classification of symbolic transition systems.

Hewlett-Packard (2004). SIPp. Website. http://sipp.sourceforge.net/.

Hong, H. S., Lee, I., Sokolsky, O., and Ural, H. (2002). A temporal logic based theory of test coverage and generation. In *TACAS 2002*, pages 327–341.

IMS (2012). Ims procedures and protocols: The lte user equipment perspective. White Paper.

Jeron, T. (2004). *Contribution la gnration automatique de tests pour les systmes ractifs*. PhD thesis, Universit de Rennes.

King, J. (1976). Symbolic execution and program testing. In *Com. ACM'76*, pages 385–394.

Ladani, B., Alcalde, B., and Cavalli, A. R. (2005). Passive testing - a constrained invariant checking approach. In *17th IFIP Testing of Communicating Systems (TestCom)*, pages 9–22.

Lalanne, F. and Maag, S. (2012). A formal data-centric approach for passive testing of communication protocols. *IEEE/ACM Transactions on Networking*, PP(99).

Lee, D., Chen, D., Hao, R., Miller, R. E., Wu, J., and Yin, X. (2002). A formal approach for passive testing of protocol data portions. In *10th IEEE International Conference on Network Protocols (ICNP 2002)*, pages 122–131.

Lee, D., Netravalli, A. N., Sabnani, K. K., Sugla, B., and John, A. (1997). Passive testing and applications to network management. In *Proceedings of International Conference Network Protocols*, pages 113–122.

Lee, D. and Yannakakis, M. (1996). Optimization problems from feature testing of communication protocols. In *Proceedings of the ICNP*, pages 66–75.

Leucker, M. and Schallhart, C. (2008). A brief account of runtime verification.

Lynch, N. A. and Tuttle, M. R. (1989). An introduction to input/output automata. *CWI Quarterly*, 2:219–246.

Maoz, S. and Harel, D. (2006). From multi-modal scenarios to code: compiling lscs into aspectj. In *in SIGSOFT FSE*.

Martijn, O., Vlad, R., Jan, T., G., D. V. R., and C., W. T. A. (2007). Integrating verification, testing, and learning for cryptographic protocols. In *Proceedings of the 6th international conference on Integrated formal methods*, IFM'07, pages 538–557.

Morales, G., Maag, S., Cavalli, A., Mallouli, W., and de Oca, E. (2010). Timed extended invariants for the passive testing of web services. In *Proceedings of the 8th IEEE ICWS*, pages 592–599.

Mouttappa, P., Maag, S., and Cavalli, A. (2012a). Improving protocol validation by an iosts based passive testing approach. In *System Testing and Validation Workshop (STV'12)*, pages 87–95.

Mouttappa, P., Maag, S., and Cavalli, A. (2012b). An iosts based passive testing approach for the validation of data-centric protocols. In *12th International Conference on Quality Software (QSIC'12)*, pages 49–58.

Mouttappa, P., Maag, S., and Cavalli, A. (2013a). Monitoring based on iosts for testing functional and security properties: application to an automotive case study. In *37th Annual IEEE International Computer Software and Applications Conference (COMPSAC 2013)*.

Mouttappa, P., Maag, S., and Cavalli, A. (2013b). Using passive testing based on symbolic execution and slicing techniques: Application to the validation of communication protocols. *Computer Networks*, 57:2992–3008.

Nguyen, H. N., Poizat, P., and Zaïdi, F. (2012a). Online verification of value-passing choreographies through property-oriented passive testing. In *HASE*, pages 106–113.

Nguyen, H. N., Poizat, P., and Zaidi, F. (2012b). A symbolic framework for the conformance checking of value-passing choreographies. In *Proceedings of the ICSOC*, pages 525–532.

Phalippou, M. (1994). *Relations d'Implantations et Hypothses de Test sur les Automates Entrees et Sorties*. PhD thesis, Bordeaux.

Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M., and Schooler, E. (2002). SIP: Session Initiation Protocol. RFC 3261. Technical report, Internet Engineering Task Force.

Rusu, V., du Bousquet, L., and Jéron, T. (2000). An approach to symbolic test generation. In *2nd International Conference on Integrated Formal Methods (IFM 2000)*, pages 338–357.

Rusu, V., Marchand, H., and Jéron, T. (2005). Automatic verification and conformance testing for validating safety properties of reactive systems. In *International Symposium of Formal Methods Europe*, pages 189–204.

Simmonds, J. (2011). *Dynamic Analysis of Web Services*. PhD thesis, University of Toronto.

Stolz, V. (2008). Temporal assertions with parameterized propositions. *Journal of Logic and Computation.*, 20(3):743–757.

TestSym-P (2013). Symbolic passive testing tool. Website. http://www-public.it-sudparis.eu/~mouttapp/TestSym.html.

Thurnher, C. (2008). Model transformation from uml state machines to input/output symbolic transition systems. Master's thesis, Institute of Information Systems - Vienna University of Technology.

Tretmans, J. (1994). A formal approach to conformance testing. In *The 6th International Workshop on Protocol Test Systems*, pages 257–276.

Tretmans, J. (1996). Test generation with inputs, outputs ans repetitive quiescence. In *Software - concepts and tools*, pages 103–120.

Tretmans, J. (2008). In Hierons, R. M., Bowen, J. P., and Harman, M., editors, *Formal methods and testing*, chapter Model based testing with labelled transition systems, pages 1–38. Springer-Verlag, Berlin, Heidelberg.

Ural, H. and Xu, Z. (2007). An EFSM-based passive fault detection approach. In *Testing of Software and Communicating Systems, 19th IFIP*, pages 335–350.

Vineet Kumar, M. K. and Sengodan, S. (2001). *IP Telephony with H.323*. Wiley, 605, Third aveneue, New York, NY10158-0012.

Wehbi, B., de Oca, E. M., and Bourdelles, M. (2012). Events-based security monitoring using mmt tool. *Software Testing, Verification, and Validation, 2008 International Conference on*, 0:860–863.

Weiglhofer, E. J. M., Aichernig, B. K., and Wotawa, F. (2010). When bdds fail: Conformance testing with symbolic execution and smt solving. In *ICST*, pages 479–488.

Wireshark (2006). Wireshark network analyser. Website. http://www.wireshark.org/.