

National Technical University of Athens
School of Electrical and Computer Engineering
Division of Computer Science

Improving the Expressibility and Ease of Use of State Machine Bug Finder

Diploma Thesis

of

kriton iordanidis

Supervisor: Konstantinos Sagonas
Associate Professor, NTUA

Athens, July 2025



National Technical University of Athens
School of Electrical and Computer Engineering
Division of Computer Science

Improving the Expressibility and Ease of Use of State Machine Bug Finder

Diploma Thesis

of

KRITON IORDANIDIS

Supervisor: Konstantinos Sagonas
Associate Professor, NTUA

Approved by the examination committee on 10th July 2025.

(Signature)	*(Signature)*	*(Signature)*
.....	.....	.....
Konstantinos Sagonas	Dimitrios Fotakis	Aris Pagourtzis
Associate Professor, NTUA	Professor, NTUA	Professor, NTUA

Athens, July 2025



National Technical University of Athens
School of Electrical and Computer Engineering
Division of Computer Science

Copyright © – Kriton Iordanidis, 2025
All rights reserved.

The copying, storage and distribution of this diploma thesis, exall or part of it, is prohibited for commercial purposes. Reprinting, storage and distribution for non - profit, educational or of a research nature is allowed, provided that the source is indicated and that this message is retained.

The content of this thesis does not necessarily reflect the views of the Department, the Supervisor, or the committee that approved it.

DISCLAIMER ON ACADEMIC ETHICS AND INTELLECTUAL PROPERTY RIGHTS

Being fully aware of the implications of copyright laws, I expressly state that this diploma thesis, as well as the electronic files and source codes developed or modified in the course of this thesis, are solely the product of my personal work and do not infringe any rights of intellectual property, personality and personal data of third parties, do not contain work / contributions of third parties for which the permission of the authors / beneficiaries is required and are not a product of partial or complete plagiarism, while the sources used are limited to the bibliographic references only and meet the rules of scientific citing. The points where I have used ideas, text, files and / or sources of other authors are clearly mentioned in the text with the appropriate citation and the relevant complete reference is included in the bibliographic references section. I fully, individually and personally undertake all legal and administrative consequences that may arise in the event that it is proven, in the course of time, that this thesis or part of it does not belong to me because it is a product of plagiarism.

(Signature)

.....
Kriton Iordanidis
Graduate of School of Electrical and Computer Engineering,
National Technical University of Athens
10th July 2025

to my parents

Περίληψη

Οι υλοποιήσεις των πρωτοκόλλων δικτύου που διατηρούν εσωτερικές καταστάσεις πρέπει να παρακολουθούν την παρουσία, τη σειρά και τον τύπο των μηνυμάτων που ανταλλάσσονται. Τυχόν σφάλματα, τα λεγόμενα σφάλματα μηχανής κατάστασης, μπορούν να θέσουν σε κίνδυνο την ασφάλεια. Το SMBugFinder είναι ένα εργαλείο λογισμικού που παρέχει ένα αυτοματοποιημένο πλαίσιο για την ανίχνευση τέτοιων σφαλμάτων σε υλοποιήσεις πρωτοκόλλων δικτύου. Λαμβάνει ως είσοδο ένα μοντέλο μηχανής κατάστασης της υλοποίησης του πρωτοκόλλου υπό εξέταση και έναν κατάλογο προτύπων σφαλμάτων για το πρωτόκολλο, όπου κάθε πρότυπο σφάλματος καθορίζεται κατάλληλα ως πεπερασμένο αυτόματο. Στη συνέχεια, παράγει ακολουθίες που εκθέτουν τα καταχωρημένα σφάλματα στην υλοποίηση που εξετάζεται.

Η τρέχουσα αναπαράσταση των προτύπων σφαλμάτων ως πεπερασμένων αυτομάτων πάνω σε ένα απλό αλφάβητο προσφέρει περιορισμένη εκφραστικότητα σε ορισμένες περιπτώσεις, και η μορφή εισόδου που χρησιμοποιείται επί του παρόντος από το SMBugFinder για τη δημιουργία του καταλόγου προτύπων σφαλμάτων δεν είναι πολύ φιλική προς τον χρήστη.

Η διπλωματική αυτή εργασία αντιμετωπίζει αυτές τις δύο ελλείψεις επεκτείνοντας τη λειτουργικότητα του εργαλείου σε δύο διαστάσεις. Πρώτον, εισάγουμε μια γραφική διεπαφή χρήστη (GUI) ώστε να διευκολύνουμε το σχεδιασμό και τον καθορισμό των προτύπων σφαλμάτων με διαδραστικό τρόπο. Η διεπαφή προσφέρει μια διαισθητική λειτουργία μεταφοράς και απόθεσης, καθιστά τον σχεδιασμό πιο φυσικό και υποστηρίζει την εξαγωγή σε διάφορες μορφές. Δεύτερον, εισάγουμε μια νέα γλώσσα ειδικού πεδίου (DSL) που συνοδεύει τα μοτίβα σφαλμάτων, επεκτείνοντας την ευελιξία και την εκφραστικότητά τους, επιτρέποντας τη χρήση δομών υψηλού επιπέδου, όπως σύνολα, συναρτήσεις και κατηγορήματα, καθώς και απαριθμημένους τύπους που αντιπροσωπεύουν παραμέτρους στα μηνύματα.

Λέξεις Κλειδιά

Επαλήθευση Λογισμικού, Συστήματα με Κατάσταση, Δικτυακά Πρωτόκολλα, Μεταγλωττιστές, Γραφικές Διεπαφές

Abstract

Implementations of stateful network protocols must keep track of the presence, order and type of exchanged messages. Any errors, so-called state machine bugs, can compromise security. SMBugFinder is a software tool that provides an automated framework for detecting such bugs in network protocol implementations. It takes as input a state machine model of the protocol implementation under test and a catalogue of bug patterns for the protocol, where each bug pattern is conveniently specified as a finite automaton. It then produces sequences that expose the catalogued bugs in the tested implementation.

The current representation of bug patterns as finite automata over a simple alphabet offers limited expressivity in certain cases, and the input format which is currently used by SMBugFinder to create the bug pattern catalogue is not very user friendly.

This diploma thesis addresses these two shortcomings by extending the tool's functionality in two dimensions. First, we introduce a graphical user interface (GUI) to ease the design and specification of bug patterns, and make it interactive. The GUI offers an intuitive drag n drop functionality, makes design more natural, and supports exporting to a variety of different formats. Second, we introduce a new domain specific language (DSL) that accompanies bug patterns, extending their versatility and expressiveness by allowing the use of high-level constructs such as sets, functions and predicates, as well as enumerated types that represent parameters in the messages.

Keywords

Software Testing, Stateful Systems, Network Protocols, Compilers, GUI

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Professor Konstantinos Sagonas, for his invaluable guidance throughout the thesis process. His deep knowledge of advanced Computer Science concepts and his inspiring lectures throughout my studies have been instrumental in shaping both this work and my academic path.

I am also deeply thankful to Professors Dimitrios Fotakis and Aris Pagourtzis for sharing with me their passion for Computer Science and for their role in fostering a strong academic foundation.

Special thanks are due to Paul Fiterau for his continuous support, insightful advice and thoughtful guidance throughout the thesis process.

Most importantly, I want to thank my family — my parents, Fanis and Katerina, for their unconditional love, support, and belief in me, and my brother and my sister, Romanos and Niovi, for their patience and encouragement.

Finally, I am deeply grateful to my friends, who provided motivation, balance, and a constant reminder to take breaks during this demanding period.

Athens, July 2025

Kriton Iordanidis

Table of Contents

Περίληψη 3

Abstract 5

Acknowledgements 7

1 Εκτενής Ελληνική Περίληψη 13
- 1.1 Υπόβαθρο . . . 13
  - 1.1.1 Συστήματα με κατάσταση και σφάλματα μηχανής κατάστασης . . . 13
  - 1.1.2 State Machine Bug Finder . . . 14
  - 1.1.3 Μοτίβα σφαλμάτων και μοντέλα SUT . . . 14
  - 1.1.4 Επεξήγηση της τεχνικής με βάση το DTLS . . . 15
  - 1.1.5 Απόκτηση του μοντέλου μηχανής κατάστασης του SUT . . . 15
  - 1.1.6 Κωδικοποίηση ευπαθειών και σφαλμάτων . . . 16
  - 1.1.7 Ανίχνευση σφαλμάτων στο μοντέλο του SUT . . . 17
  - 1.1.8 Επικύρωση των σφαλμάτων του μοντέλου στο SUT . . . 18
  - 1.1.9 *SMBugFinder* σε εφαρμογή στο DTLS . . . 18
  - 1.1.10 Το μοντέλο SUT . . . 19
  - 1.1.11 Κωδικοποίηση σφαλμάτων ως αυτόματα . . . 19
  - 1.1.12 Συγκέντρωση καταλόγου προτύπων σφαλμάτων . . . 20
- 1.2 Γλώσσα παραμετρικών προτύπων σφαλμάτων . . . 20
  - 1.2.1 Κίνητρο . . . 20
  - 1.2.2 Στόχοι σχεδιασμού . . . 21
  - 1.2.3 Προδιαγραφές γλώσσας . . . 21
  - 1.2.4 Ένα παραμετρικό πρότυπο σφάλματος με το αρχείο προδιαγραφών του . . . 22
  - 1.2.5 Η χρήση συναρτήσεων σε παραμετρικά μοτίβα σφαλμάτων . . . 22
  - 1.2.6 Υλοποίηση και ενσωμάτωση με το *SMBugFinder* . . . 23
  - 1.2.7 Σύνοψη . . . 23
- 1.3 Γραφικό περιβάλλον χρήστη για μοτίβα σφαλμάτων . . . 24
  - 1.3.1 Κίνητρο . . . 24
  - 1.3.2 Επισκόπηση του εργαλείου . . . 24
  - 1.3.3 Βασικά χαρακτηριστικά . . . 24
  - 1.3.4 Ενσωμάτωση ροής εργασίας με το SMBugFinder . . . 25
  - 1.3.5 Παράδειγμα χρήσης . . . 25
  - 1.3.6 Σημειώσεις υλοποίησης . . . 25

1.3.7 Σύνοψη 25

2 Introduction **27**
2.1 Motivation and Context 27
2.2 Goals and Contributions 27
2.3 Overview of the Thesis 28
2.4 Key Results and Implications 28

3 Background **29**
3.1 Stateful Systems and State Machine Bugs 29
3.2 State Machine Bug Finder 30
3.3 Bug Patterns and SUT Models 31
3.4 The Technique Illustrated on DTLS 32
3.4.1 Obtaining the SUT's State Machine Model 32
3.4.2 Encoding Vulnerabilities and Bugs 33
3.4.3 Detecting Bugs in the Model of the SUT 35
3.4.4 Validating the Model's Bugs in the SUT 36
3.5 *SMBugFinder*'s Usage Illustrated on DTLS 36
3.5.1 The SUT Model 36
3.5.2 Encoding Bugs as Automata 37
3.5.3 Assembling a Catalogue of Bug Patterns 39

4 Parametric Bug Pattern Language **41**
4.1 Motivation 41
4.2 Design Goals 41
4.3 Language Specification 41
4.3.1 Overview 41
4.3.2 Formal Specification 42
4.3.3 A Parametric Bug Pattern with its Specification File 42
4.3.4 The Use of Functions In Parametric Bug Patterns 43
4.4 Implementation and Integration with *SMBugFinder* 44
4.5 Summary 45

5 Graphical User Interface for Bug Patterns **47**
5.1 Motivation 47
5.2 Overview of the Tool 47
5.3 Core Features 47
5.4 Workflow Integration with SMBugFinder 48
5.5 Example Use Case 48
5.6 Implementation Notes 49
5.7 Summary 49

6 Future Work **51**

List of Figures

3.1 State machine of a vending machine. 29
3.2 Erroneous state machine of a vending machine. 30
3.3 State machine bug: the vending machine does not dispense the selected product. 30
3.4 Reduced model of a JSSE 12.0.2 DTLS server [4]. 32
3.5 DFA capturing the vulnerability of completing a handshake without a Certificate message from the client [4]. 33
3.6 DFA capturing the vulnerability of authenticating a client without it sending a CertificateVerify message [4]. 34
3.7 DFA capturing a server completing handshake, but consuming a ClientKeyExchange before a client Certificate [4]. 35
3.8 DFA capturing a server completing handshake, but consuming a CertificateVerify before ClientKeyExchange [4]. 35
3.9 Reduced model of MbedTLS 37
3.10 Non conforming cookie 38

4.1 Grammar of the parametric bug pattern language 42
4.2 Parametric non conforming cookie 43
4.3 Wrong certificate type 44

5.1 The State Machine Bug Designer 48
5.2 Non conforming cookie bug pattern designed in the GUI 49

Κεφάλαιο 1

Εκτενής Ελληνική Περίληψη

1.1 Υπόβαθρο

1.1.1 Συστήματα με κατάσταση και σφάλματα μηχανής κατάστασης

Πολλά συστήματα του πραγματικού κόσμου εξαρτώνται σε μεγάλο βαθμό από τη διατήρηση και τη διαχείριση μιας εσωτερικής κατάστασης κατά τη λειτουργία τους. Τέτοια συστήματα αναφέρονται ως συστήματα με κατάσταση. Παραδείγματα περιλαμβάνουν αυτόματους πωλητές, αυτόματους ταμειακούς μηχανισμούς (ATM), ανελκυστήρες, υλοποιήσεις πρωτοκόλλων δικτύου και πολλά άλλα.

Η συμπεριφορά αυτών των συστημάτων μοντελοποιείται συνήθως χρησιμοποιώντας μηχανές κατάστασης, οι οποίες περιγράφουν το σύνολο των πιθανών εσωτερικών καταστάσεων και τις μεταβάσεις μεταξύ τους. Η σωστή λειτουργία ενός συστήματος με κατάσταση εξαρτάται από την ορθή υλοποίηση της υποκείμενης μηχανής κατάστασης. Σφάλματα στο σχεδιασμό ή την υλοποίηση της μηχανής κατάστασης μπορούν να οδηγήσουν σε απροσδόκητη ή ανεπιθύμητη συμπεριφορά του συστήματος. Αναφερόμαστε σε τέτοια σφάλματα ως σφάλματα μηχανής κατάστασης. Στόχος μας είναι να εντοπίσουμε σφάλματα μηχανής κατάστασης σε μια δεδομένη υλοποίηση μηχανής κατάστασης. Ως παράδειγμα, παρέχουμε μια πιθανή μηχανή κατάστασης ενός αυτόματου πωλητή στο Σχ. 3.1.

Εσωτερικά, ένας αυτόματος πωλητής μπορεί να βρίσκεται σε μία εσωτερική κατάσταση κάθε φορά, και η τρέχουσα κατάστασή του αλλάζει με βάση τις αλληλεπιδράσεις του χρήστη ή άλλα γεγονότα. Για παράδειγμα, όταν η μηχανή δεν χρησιμοποιείται, βρίσκεται στην *idle* κατάσταση. Όταν εισάγονται κέρματα, μεταβαίνει στην *accepting_coins* κατάσταση. Εάν εισαχθούν επιπλέον κέρματα, παραμένει σε αυτήν την κατάσταση. Ωστόσο, εάν επιλεγεί ένα προϊόν, μεταβαίνει στην κατάσταση *selecting_product*. Οι υπόλοιπες καταστάσεις και μεταβάσεις στη μηχανή καταστάσεων μπορούν να ερμηνευθούν με παρόμοιο τρόπο.

Το κρίσιμο σημείο εδώ είναι ότι η συμπεριφορά του αυτόματου πωλητή καθορίζεται εξ ολοκλήρου από τη μηχανή καταστάσεων. Τι συμβαίνει όμως εάν η μηχανή καταστάσεων περιέχει σφάλματα σχεδιασμού ή υλοποίησης;

Για να το εξηγήσουμε αυτό, ας εξετάσουμε τη μηχανή καταστάσεων με σφάλματα που φαίνεται στην Εικ. 3.2. Σε αυτή τη μηχανή καταστάσεων, μόλις επιβεβαιωθεί το επιλεγμένο προϊόν, το σύστημα μεταβαίνει στην κατάσταση *returning_change* χωρίς να διανείμει το προϊόν. Αυτή η συμπεριφορά είναι σαφώς λανθασμένη και υποδηλώνει ένα σφάλμα στο σχεδιασμό της μηχανής

καταστάσεων. Η διαδρομή που οδηγεί σε αυτή τη λανθασμένη συμπεριφορά επισημαίνεται με κόκκινο χρώμα.

Προσδιορίζουμε και απομονώνουμε τέτοιες εσφαλμένες συμπεριφορές, αναπαριστώντας τις ως μηχανές κατάστασης, τις οποίες ονομάζουμε σφάλματα μηχανής κατάστασης. Για παράδειγμα, η εσφαλμένη συμπεριφορά που συζητήθηκε προηγουμένως απεικονίζεται στο Σχ. 3.3.

1.1.2 State Machine Bug Finder

Πολλές εφαρμογές βασίζονται σε μεγάλο βαθμό στην ορθή υλοποίηση των πρωτοκόλλων δικτύου. Οι υλοποιήσεις πρωτοκόλλων δικτύου με κατάσταση, όπως SSH, TCP, TLS, QUIC κ.λπ., πρέπει να διατηρούν μια μηχανή κατάστασης με την οποία παρακολουθούν την παρουσία, τη σειρά και τον τύπο των μηνυμάτων που ανταλλάσσονται. Τα σφάλματα σε αυτή τη μηχανή κατάστασης, τα λεγόμενα σφάλματα μηχανής κατάστασης , μπορούν να κάνουν τις εφαρμογές ευάλωτες σε επιθέσεις, π.χ. τη δημιουργία σύνδεσης με έναν διακομιστή χωρίς την παροχή όλων των διαπιστευτηρίων.

Το *SMBugFinder* [7] είναι ένα εργαλείο ελέγχου ανεξάρτητο από πρωτόκολλα και πλατφόρμες , του οποίου στόχος είναι να καταστήσει την ανίχνευση και την έκθεση τέτοιων σφαλμάτων μηχανής κατάστασης μια πλήρως αυτοματοποιημένη διαδικασία. Λαμβάνει ως είσοδο ένα (ενδεχομένως εκμαθημένο) μοντέλο του συστήματος υπό δοκιμή (SUT) και έναν κατάλογο αυτομάτων κωδικοποιημένων προτύπων για σφάλματα μηχανής κατάστασης που ελέγχονται στο SUT. Χρησιμοποιεί τα πρότυπα για να ανιχνεύσει και, με τη χρήση ενός συστήματος δοκιμών, να επικυρώσει ακολουθίες που εκθέτουν αυτά τα σφάλματα στο μοντέλο. Οι ακολουθίες που επικυρώνονται με επιτυχία μετατρέπονται αυτόματα σε εκτελέσιμες δοκιμές.

1.1.3 Μοτίβα σφαλμάτων και μοντέλα SUT

Τα μοτίβα σφαλμάτων κωδικοποιούν μια συμπεριφορά που θεωρείται σφάλμα και που θέλουμε να ελέγξουμε αν αυτή η εσφαλμένη συμπεριφορά υπάρχει σε ένα SUT.

Το *SMBugFinder* επιτρέπει στους χρήστες να καθορίσουν μοτίβα σφαλμάτων ως DFAs σε μορφή DOT, όπου:

- **Καταστάσεις** αντιπροσωπεύουν την εσωτερική κατάσταση του SUT.
- **Μεταβάσεις** προκαλούνται από μηνύματα που λαμβάνονται ή εκπέμπονται από το SUT.
- **Καταστάσεις σφάλματος** υποδηλώνουν την ύπαρξη ενός μοτίβου σφάλματος στο SUT.

Για παράδειγμα, στο σφάλμα της μηχανής καταστάσεων που παρουσιάστηκε νωρίτερα στο Σχήμα 3.3, υπάρχουν τέσσερις καταστάσεις που αντιπροσωπεύουν τις εσωτερικές καταστάσεις του SUT, τρεις μεταβάσεις μεταξύ τους που προκαλούνται από ενέργειες που εκτελούνται στον αυτόματο πωλητή και μια κατάσταση σφάλματος που εμφανίζεται σε έναν κόκκινο διπλό κύκλο που υποδηλώνει την ύπαρξη σφάλματος στη μηχανή καταστάσεων του αυτόματου πωλητή.

Τα μοντέλα SUT κωδικοποιούν τη μηχανή κατάστασης ενός πρωτοκόλλου δικτύου που θέλουμε να ελέγξουμε σε σχέση με έναν καταλογό τύπων σφαλμάτων που έχει συλλεχθεί. Τα μοντέλα SUT καθορίζονται ως μηχανές Mealy σε μορφή DOT , την οποία το *SMBugFinder* μεταφράζει εσωτερικά σε DFAs.

Μια προσέγγιση που έχει αποδειχθεί αποτελεσματική για την εύρεση σφαλμάτων μηχανών κατάστασης είναι το *state fuzzing* [1] [2] [6]. Αυτόματα συνάγει περιγραφές μηχανών κατάστασης των υλοποιήσεων πρωτοκόλλων χρησιμοποιώντας *model learning* [9] [13]. Η μάθηση μοντέλων είναι μια αυτοματοποιημένη τεχνική δοκιμών μαύρου κουτιού που, με τη συστηματική αποστολή ερωτημάτων σε ένα SUT, συμπεραίνει ένα μοντέλο μηχανής κατάστασης (δηλ. μια μηχανή Mealy), καταγράφοντας τη συμπεριφορά εισόδου/εξόδου του SUT. Η εκμάθηση μοντέλων απαιτεί τα αλφάβητα εισόδου και εξόδου του SUT. Επιπλέον, συχνά απαιτείται κάποια αφαίρεση για να αλληλεπιδράσει με το SUT, δηλαδή ένας τρόπος αντιστοίχισης συγκεκριμένων μηνυμάτων πρωτοκόλλου με αφηρημένα σύμβολα αλφαβήτου και αντίστροφα. Στην πράξη, η αφαίρεση υλοποιείται σε ένα στοιχείο γνωστό ως «test harness» ή «mapper». Είναι σημαντικό να σημειωθεί ότι τα μοντέλα που δημιουργούνται από τη μάθηση μοντέλων είναι στις περισσότερες περιπτώσεις προσεγγιστικά και ενδέχεται να μην αντικατοπτρίζουν πάντα με ακρίβεια την πραγματική συμπεριφορά του SUT. Επομένως, τυχόν ελαττώματα που εντοπίζονται στο μοντέλο πρέπει να επικυρωθούν στο SUT με κατάλληλες δοκιμές. Αυτή η τεχνική έχει εφαρμοστεί σε διάφορες υλοποιήσεις πρωτοκόλλων δικτύου χρησιμοποιώντας εργαλεία όπως το DTLS-Fuzzer [5] και το EDHOC-Fuzzer [12].

1.1.4 Επεξήγηση της τεχνικής με βάση το DTLS

Θα παρουσιάσουμε ένα εκτενές παράδειγμα μοντέλου μηχανής κατάστασης που έχει εξαχθεί από μια υλοποίηση DTLS, μαζί με τέσσερα σφάλματα μηχανής κατάστασης. Το μεγαλύτερο μέρος του κειμένου και των εικόνων αυτής της ενότητας προέρχεται από το άρθρο NDSS'23 [4].

1.1.5 Απόκτηση του μοντέλου μηχανής κατάστασης του SUT

Η πρώτη είσοδος που χρειάζεται το *SMBugFinder* είναι το μοντέλο SUT. Για διάφορες υλοποιήσεις διακομιστή και πελάτη DTLS, αυτό μπορεί να γίνει με το εργαλείο DTLS-Fuzzer [5]. Το μοντέλο που θα περιγράψουμε εδώ αντιστοιχεί στην υλοποίηση διακομιστή JSSE 12.0.2. Ωστόσο, επειδή το DTLS-Fuzzer συνήγαγε ένα μοντέλο με 124 καταστάσεις, το οποίο είναι δυσανάγνωστο, παρουσιάζουμε στο Σχ. 3.4 μια μειωμένη έκδοση του, που αποτελείται μόνο από δώδεκα καταστάσεις.

Ας εξηγήσουμε πώς διαβάζεται αυτή η μηχανή Mealy. Οι ακμές της επισημαίνονται με στοιχεία του αλφαβήτου στις δύο πλευρές μιας κάθετου («/»), με εισόδους στα αριστερά και εξόδους στα δεξιά. Στην αρχική του κατάσταση (0), ο διακομιστής είτε δέχεται δεδομένα εφαρμογής (*App*) και δεν εξάγει καμία απόκριση, είτε δέχεται ένα μήνυμα *ClientHello (CH)* και απαντά με ένα *HelloVerifyRequest*. Στην κατάσταση 1, εκτός από το *App*, ο διακομιστής δέχεται επίσης ένα *CH* και στη συνέχεια απαντά με μια ακολουθία πέντε μηνυμάτων (*SH*, $Cert_s$, *SKE*, *CertReq* και *SHD*, με αυτή τη σειρά). Οι υπόλοιπες καταστάσεις είναι παρόμοιες, αλλά χρησιμοποιούμε επίσης μια συντομογραφία (στις αυτο-άκρες των καταστάσεων 6, 7 και 11) για να υποδηλώσουμε μια ένωση εισόδων και αντίστοιχων εξόδων. Μπορούμε επίσης να παρατηρήσουμε τη διαδρομή που ολοκληρώνει σωστά τη χειραψία, με μπλε χρώμα, ξεκινώντας από την αρχική κατάσταση και τελειώνοντας στην κατάσταση 7.

1.1.6 Κωδικοποίηση ευπαθειών και σφαλμάτων

Αφού αποκτήσουμε ένα μοντέλο SUT, η δεύτερη είσοδος που χρειάζεται το *SMBugFinder* είναι ένας κατάλογος προτύπων που θα ελεγχθούν σε σχέση με το μοντέλο SUT. Η ιδέα είναι να αναζητήσουμε στο δεδομένο μοντέλο SUT, M, διαδρομές που παραβιάζουν τις απαιτήσεις ασφάλειας και ορθότητας του πρωτοκόλλου ή μοιάζουν με σφάλματα.

Υπάρχουν τρία ζητήματα που πρέπει να αντιμετωπιστούν για να υλοποιηθεί αυτή η απλή ιδέα και να καταστεί αποτελεσματική στην πράξη: (1) Πώς καταγράφουμε πώς μοιάζει μια παραβίαση απαίτησης ή ένα σφάλμα στο M; (2) Πώς αναζητούμε αποτελεσματικά διαδρομές με σφάλματα στο M; (3) Πώς αποδεικνύουμε ότι το πραγματικό SUT περιέχει σφάλματα που ενεργοποιούνται κατά την εκτέλεση του κώδικα που αντιστοιχεί σε μια διαδρομή με σφάλματα στο M; Ας απαντήσουμε πρώτα στην πρώτη από αυτές τις ερωτήσεις.

Κάθε πρωτόκολλο δικτύου καθορίζει απαιτήσεις ασφάλειας και ορθότητας στις προδιαγραφές του (π.χ. στο RFC). Φυσικά, μπορούμε να εξετάσουμε αυτές τις προδιαγραφές και να εξαγάγουμε (ένα πλήρες σύνολο) αυτών των απαιτήσεων. Στη συνέχεια, μπορούμε να συντάξουμε έναν κατάλογο προτύπων σφαλμάτων, καθένα από τα οποία κωδικοποιείται ως ένα αυτόματο που δέχεται ακολουθίες μηνυμάτων που εκθέτουν το σφάλμα της μηχανής κατάστασης. Ας απεικονίσουμε αυτήν την κωδικοποίηση με ένα παράδειγμα. Ας υποθέσουμε ότι δοκιμάζουμε έναν διακομιστή DTLS ο οποίος έχει ρυθμιστεί να απαιτεί ένα έγκυρο πιστοποιητικό από τον πελάτη. Είναι προφανώς σφάλμα — στην πραγματικότητα, ευπάθεια — ο διακομιστής να ζητά πιστοποιητικό (με ένα μήνυμα *CertReq*) και στη συνέχεια να εισέρχεται στη φάση που ολοκληρώνει τη διαδικασία χειραψίας χωρίς να λάβει ένα μήνυμα πιστοποιητικού από τον πελάτη ($Cert_c$). Μπορούμε να καταγράψουμε αυτήν την ευπάθεια με ένα αυτόματο που δέχεται ακολουθίες μηνυμάτων που την εκθέτουν. Ένα τέτοιο αυτόματο φαίνεται στην Εικ. 3.5.

Ας εξηγήσουμε αυτό το Ντετερμινιστικό Πεπερασμένο Αυτόματο(DFA) καθώς και τη σημειογραφία που χρησιμοποιούμε. Οι πιθανές διαδρομές από την αρχική έως την τελική κατάσταση «bug» εκφράζουν την ευπάθεια που εξηγήσαμε παραπάνω: υπάρχει ένα μήνυμα *CertReq* ακολουθούμενο από ένα μήνυμα CCS_s από τον διακομιστή, χωρίς κανένα μήνυμα $Cert_c$ από τον πελάτη ενδιάμεσα. Χρησιμοποιούμε το σύμβολο $\mathscr{U}$ για αυτό που είναι γνωστό ως το σύνολο όλων των «άλλων» συμβόλων του αλφαβήτου Σ ενός DFA (δηλαδή, όλα τα σύμβολα εκτός από αυτά που εμφανίζονται ως επισημάνσεις στις άλλες εξερχόμενες μεταβάσεις από μια κατάσταση). Έτσι, για την αρχική κατάσταση, το $\mathscr{U}$ δηλώνει $\Sigma - \{CertReq\}$, και για τη μεσαία κατάσταση το $\mathscr{U}$ δηλώνει $\Sigma - \{SH, CCS_s\}$. Με την συμπερίληψη μιας μετάβασης SH πίσω στην αρχική κατάσταση, επιτρέπουμε στο DFA μας να αποφύγει ψευδείς κινδύνους (λόγω επαναδιαπραγμάτευσης). Στο DTLS, ένας πελάτης μπορεί να επανεκκινήσει τη διαδικασία χειραψίας σε οποιοδήποτε σημείο στέλνοντας ένα μήνυμα $ClientHello$ στον διακομιστή. Ορισμένοι διακομιστές DTLS θα απαντήσουν με ένα HVR σε αυτό το σημείο και η χειραψία θα επανεκκινηθεί. Ωστόσο, υπάρχουν επίσης υλοποιήσεις DTLS που παραλείπουν την επανάληψη του βήματος ανταλλαγής cookie και επανεκκινούν τη διαδικασία χειραψίας. Επομένως, είναι ασφαλέστερο και πιο ομοιόμορφο να χρησιμοποιείται το *δεύτερο* μήνυμα που στέλνει ένας διακομιστής (SH) για να υποδηλώσει ότι η διαδικασία χειραψίας επανεκκινείται και μπορεί να ολοκληρωθεί σωστά με ένα άλλο $CertReq$ από τον διακομιστή.

Σε σχέση με το γεγονός ότι όλα τα αυτόματα μας είναι *ντετερμινιστικά*, σημειώνουμε ότι για

όλα τα σύμβολα στο Σ για τα οποία δεν υπάρχει εξερχόμενη μετάβαση από μια κατάσταση, υπάρχουν υπονοούμενες μεταβάσεις για αυτά τα σύμβολα σε μια κατάσταση «sink», τις οποίες δεν εμφανίζουμε για να μην γεμίσουμε τις εικόνες. Για παράδειγμα, για το αυτόματο στο Σχ. 3.5 υπάρχουν υπονοούμενες μεταβάσεις από την τελική κατάσταση «bug» για όλα τα σύμβολα στο Σ, και υπάρχει επίσης μια μετάβαση για το $Cert_c$ από την κατάσταση «certreq» στην κατάσταση «sink», που σημαίνει ότι εάν ένα πιστοποιητικό αποσταλεί από τον πελάτη, τότε δεν φτάνουμε στην κατάσταση αποδοχής του DFA μας (δηλαδή, δεν υπάρχει σφάλμα).

Η δεύτερη ευπάθεια που εντοπίζουμε με ένα DFA (Εικ. 3.6) είναι παρόμοια. Σε έναν διακομιστή που έχει ρυθμιστεί να απαιτεί πιστοποιητικό από τον πελάτη, ο πελάτης πιστοποιείται (δηλαδή, ολοκληρώνεται η διαδικασία χειραψίας) με την αποστολή ενός πιστοποιητικού ($Cert_c$) από τον πελάτη, αλλά *χωρίς* την αποστολή ενός $CertificateVerify$ στον διακομιστή. Αυτό αποτελεί ένα σοβαρό κενό ασφαλείας. Μπορεί να υποδηλώνει ότι ένας εισβολέας μπορεί να πιστοποιηθεί χρησιμοποιώντας το πιστοποιητικό κάποιου άλλου χωρίς να αποδείξει την κατοχή του πιστοποιητικού με ένα $CertVer$.

Εκτός από τις ευπάθειες, σημειώστε ότι μπορούμε επίσης να χρησιμοποιήσουμε DFA για να καταγράψουμε άλλα είδη σφαλμάτων της μηχανής κατάστασης πρωτοκόλλου. Για παράδειγμα, το DFA της Εικ. 3.7 καταγράφει το σφάλμα ότι, σε μια έγκυρη χειραψία με έναν διακομιστή που χρησιμοποιεί ECDH, ο διακομιστής δεν μπορεί να καταναλώσει ένα $ClientKeyExchange$ πριν από το μήνυμα $Cert_c$ από τον πελάτη. Στην πραγματικότητα, μπορούμε να γενικεύσουμε την άκρη $CKE(ECDH)$ στο DFA του Σχ. 3.7 ώστε να λειτουργεί για όλους τους αλγόριθμους ανταλλαγής κλειδιών που υποστηρίζει ένας διακομιστής και οι οποίοι απαιτούν από τον πελάτη να παρέχει ένα πιστοποιητικό. Για παράδειγμα, μπορούμε να ορίσουμε ως ετικέτα αυτής της άκρης το σύνολο $\{CKE(DH), CKE(ECDH), CKE(RSA)\}$. Θα χρησιμοποιήσουμε τη συντομογραφία $CKE^{\mathcal{K}}$ για να αναφερθούμε σε ένα τέτοιο σύνολο.

Το τελευταίο μας παράδειγμα DFA (Εικ. 3.8) είναι παρόμοιο. Καταγράφει μια άλλη περίπτωση μη έγκυρης αλληλουχίας μηνυμάτων: ένας διακομιστής ολοκληρώνει τη χειραψία ενώ καταναλώνει ένα $CertVer$ πριν από το μήνυμα CKE από τον πελάτη.

Αφού εξηγήσαμε πώς να κωδικοποιήσουμε τις απαιτήσεις σε αυτόματα, ας εξετάσουμε πώς να αποκτήσουμε τέτοιες απαιτήσεις αρχικά. Υπάρχουν διάφορες δυνατότητες: Τα DFA μπορούν να κατασκευαστούν και να βελτιωθούν απευθείας από i) συγκεκριμένες απαιτήσεις σε μια προδιαγραφή πρωτοκόλλου (π.χ., «Το μήνυμα $Cert_c$ αποστέλλεται από τον πελάτη μόνο εάν ο διακομιστής ζητήσει πιστοποιητικό.» [[10], σ. 55]), ii) από οποιοδήποτε προηγουμένως αναφερθέν σφάλμα μηχανής κατάστασης για υλοποιήσεις πρωτοκόλλου, ή iii) χρησιμοποιώντας τις γνώσεις μας σχετικά με το πρωτόκολλο και εφαρμόζοντας την κοινή λογική. Σημειώνουμε ότι το σύνολο των DFAs που καταγράφουν απαιτήσεις, ευπάθειες και κοινά σφάλματα μιας υλοποίησης πρωτοκόλλου μπορεί να καθοριστεί εκτός σύνδεσης και σταδιακά.

1.1.7 Ανίχνευση σφαλμάτων στο μοντέλο του SUT

Βρισκόμαστε στο σημείο όπου έχουμε ένα μοντέλο M ενός SUT με τη μορφή μηχανής Mealy, και έχουμε επίσης ένα σύνολο προτύπων σφαλμάτων που εκφράζονται ως DFAs. Προκειμένου να ανιχνεύσουμε τα σφάλματα στο μοντέλο του SUT, πρέπει να αντιμετωπίσουμε δύο ζητήματα. Πρώτον, τα μοντέλα SUT περιγράφονται μέσω μηχανών Mealy, αλλά τα μοτίβα σφαλμάτων περι-

γράφονται μέσω DFAs. Αυτοί οι δύο φορμαλισμοί μοιάζουν παρόμοιοι, αλλά τεχνικά είναι πολύ διαφορετικοί. Οι μηχανές Mealy καθορίζουν τις σχέσεις εισόδου- εξόδου των συμβόλων στις μεταβάσεις τους, ενώ τα DFAs ορίζουν μια γλώσσα: το σύνολο των λέξεων που δέχεται το DFA. Δεύτερον, η αναζήτηση όλων των διαδρομών στο μοντέλο δεν θα λειτουργήσει. Τα περισσότερα μοντέλα, εκτός από το ότι έχουν πολλές διαδρομές λόγω των πολλών καταστάσεων, μπορεί να έχουν ακόμη και άπειρο αριθμό διαδρομών. Για παράδειγμα, υπάρχουν εσωτερικές ακμές σε όλες τις καταστάσεις του Σχ. 3.4.

Με λίγα λόγια, αυτό που κάνει το *SMBugFinder* σε αυτό το βήμα είναι να μετατρέψει το μοντέλο μηχανής Mealy του SUT σε DFA και να το διασταυρώσει με κάθε ένα από τα DFA που περιγράφουν τα μοτίβα σφαλμάτων. Εφαρμόζοντας αυτή τη μεθοδολογία στο τρέχον παράδειγμά μας, τον διακομιστή JSSE 12.0.2, το *SMBugFinder* θα εντοπίσει ότι: i) η ευπάθεια που περιγράφεται από το DFA του Σχ. 3.5 (ολοκλήρωση χειραψίας χωρίς μήνυμα $Cert$) υπάρχει στη διαδρομή 0→1→2→8→11→7; ii) η ευπάθεια που περιγράφεται από το DFA του Σχ. 3.6 (αυθεντικοποίηση πελάτη χωρίς $CertVer$) εμφανίζεται στη διαδρομή 0→1→2→8→10→6→7; iii) το σφάλμα που καταγράφηκε από το DFA του Σχ. 3.7 (CKE πριν από $Cert_c$) εμφανίζεται στη διαδρομή 0→1→2→8→10→5→6→7 και iv) το σφάλμα που αποτυπώθηκε από το DFA του Σχ. 3.8 ($CertVer$ πριν από CKE) εμφανίζεται στη διαδρομή 0→1→2→3→9→5→6→7. Επίσης, το *SMBugFinder* θα ανακαλύψει ότι το σφάλμα $CertVer$ πριν από το CKE υπάρχει επίσης στη διαδρομή 0 έως 7 η οποία περνάει από τις ακμές 2→8 και 8→10 (χοντρές κόκκινες γραμμές).

1.1.8 Επικύρωση των σφαλμάτων του μοντέλου στο SUT

Δυστυχώς, το γεγονός ότι έχει εντοπιστεί ένα σφάλμα στο μοντέλο ενός SUT δεν αποδεικνύει από μόνο του ότι το σφάλμα υπάρχει πράγματι στο μοντέλο. Υπάρχουν διάφοροι παράγοντες που πρέπει να λάβουμε υπόψη. Πρώτον, το μοντέλο του DTLS-Fuzzer μπορεί να μην έχει συγκλίνει (βλ. [5]) και το μοντέλο που έχει συναχθεί μπορεί να είναι κατά προσέγγιση. Αυτό σημαίνει ότι το προηγούμενο βήμα μπορεί να αναφέρει σφάλματα που δεν υπάρχουν στο SUT. Δεύτερον, ακόμη και αν το μοντέλο είναι ακριβές, η αναφορά ενός σφάλματος χωρίς την παροχή ενός δοκιμαστικού περιπτώματος που το εμφανίζει, καθιστά την αναπαραγωγή και την επιδιόρθωση του σφάλματος πολύ δύσκολη.

Έτσι, για να επικυρώσουμε την παρουσία σφαλμάτων στο πραγματικό σύστημα, κατασκευάζουμε ακολουθίες πακέτων πρωτοκόλλου και τα εκτελούμε σε αυτό. Με αυτόν τον τρόπο μπορούμε να τα επιστρέψουμε ως μαρτυρίες σφαλμάτων (bug witnesses) στις αναφορές προς τους προγραμματιστές. Το *SMBugFinder* θα επικυρώσει αυτόματα όλα τα σφάλματα στο SUT.

1.1.9 *SMBugFinder* σε εφαρμογή στο DTLS

Ως επίδειξη της χρήσης του *SMBugFinder* στην τρέχουσα μορφή του, θα παρουσιάσουμε ένα συγκεκριμένο παράδειγμα σε μια εφαρμογή διακομιστή DTLS, συγκεκριμένα στο μοντέλο MbedTLS v2.26.0. Θα αναφερόμαστε σε αυτή την εφαρμογή SUT ως MbedTLS.

Όπως έχει ήδη σημειωθεί, το *SMBugFinder* απαιτεί πρώτα την παραγωγή των εισόδων για την αυτοματοποιημένη δοκιμή: το μοντέλο SUT και τα μοτίβα σφαλμάτων για το πρωτόκολλο, τα οποία συγκεντρώνουμε σε έναν κατάλογο. Στη συνέχεια, εκτελούμε το *SMBugFinder* σε αυτές τις εισόδους. Το *SMBugFinder* θα παράγει αποτελέσματα τα οποία μπορούμε να αναλύσουμε.

1.1.10 Το μοντέλο SUT

Μπορούμε να χρησιμοποιήσουμε ένα μοντέλο που περιγράφει τον διακομιστή MbedTLS που δημιουργήθηκε από το DTLS-Fuzzer[5], όπως αναφέρθηκε προηγουμένως. Αυτό το εργαλείο παράγει το μοντέλο μηχανής Mealy σε μορφή DOT, έτοιμο να χρησιμοποιηθεί ως έχει από το *SMBugFinder*. Να σημειωθεί ότι αυτή η μορφή DOT είναι κοινή σε όλα τα εργαλεία εκμάθησης κατάστασης πρωτοκόλλου που γνωρίζουμε, πράγμα που σημαίνει ότι το *SMBugFinder* μπορεί να λειτουργήσει με τα μοντέλα που παράγουν όλα αυτά τα εργαλεία. Το σχήμα 3.9 παρέχει μια οπτικοποίηση ενός μικρού μέρους του μοντέλου, το οποίο δημιουργείται αυτόματα χρησιμοποιώντας το εργαλείο dot του Graphviz.

Εξετάζοντας αυτό το μοντέλο, μπορεί κανείς να δει τη διαδρομή $s0 \rightarrow s1 \rightarrow s2$ του βήματος ανταλλαγής cookie. Αυτή η διαδρομή σχετίζεται με ένα σφάλμα μηχανής κατάστασης, σύμφωνα με το DTLS RFC [11]. Στη συνέχεια, θα δούμε πώς μπορούμε να παρέχουμε μοτίβα σφαλμάτων που επιτρέπουν στο *SMBugFinder* να ανιχνεύει αυτόματα αυτό το ελάττωμα.

1.1.11 Κωδικοποίηση σφαλμάτων ως αυτόματα

Η διαδρομή από την κατάσταση s0 στην κατάσταση s2, αποκαλύπτει ένα σφάλμα μηχανής κατάστασης: ο διακομιστής ολοκληρώνει το βήμα ανταλλαγής cookie μετά τη λήψη ενός δεύτερου `CLIENT_HELLO` με το `RSA` ciphersuite παρά το γεγονός ότι το πρώτο `CLIENT_HELLO` χρησιμοποιεί το `PSK` ciphersuite. Αυτή η συμπεριφορά παραβιάζει την προδιαγραφή RFC του DTLS [11]. Το μοτίβο σφάλματος που χρησιμοποιούμε για την ανίχνευση αυτού του σφάλματος φαίνεται στο Σχήμα 3.10(a), μαζί με τον πηγαίο κώδικα που το υλοποιεί (Σχήμα 3.10(b)). Όπως το μοντέλο SUT, το μοτίβο σφάλματος είναι επίσης σε μορφή DOT και μπορεί να προβληθεί με το εργαλείο dot.

Το παραπάνω αυτόματο καταγράφει (αποδέχεται) την ακόλουθη ακολουθία μηνυμάτων: `I_PSK_CLIENT_HELLO` $\rightarrow$ `O_HELLO_VERIFY_REQUEST` $\rightarrow$ `I_RSA_CLIENT_HELLO` $\rightarrow$ `O_SERVER_HELLO` η οποία παραβιάζει την προδιαγραφή RFC του DTLS, όπως αναφέρθηκε προηγουμένως. Το μοτίβο κωδικοποιεί ένα ντετερμινιστικό πεπερασμένο αυτόματο (DFA) που καθορίζεται για όλα τα σύμβολα εισόδου και εξόδου του μοντέλου SUT, τα οποία εμφανίζονται στο μοτίβο με πρόθεμα I_ και O_, αντίστοιχα. Είναι σημαντικό να σημειωθεί ότι, επί του παρόντος, οι ακμές του γραφήματος μπορούν να περιέχουν πιο σύνθετες δομές από απλά μηνύματα. Αυτές περιλαμβάνουν:

- Το σύμβολο U, το οποίο αντιπροσωπεύει το σύνολο των συμβόλων SUT που δεν περιλαμβάνονται σε καμία εξερχόμενη ακμή από μια κατάσταση.

- Εκφράσεις συνόλου, όπως η αφαίρεση ενός συνόλου από ένα άλλο.

Επιπλέον, τα σύμβολα για τα οποία δεν έχει καθοριστεί μετάβαση οδηγούν σε μια υπονοούμενη κατάσταση απόρριψης.

Η συμπαγής μορφή της κωδικοποίησης είναι κρίσιμη για τη χρηστικότητα του *SMBugFinder* και θα συζητήσουμε πώς επεκτείνεται περαιτέρω στο κεφάλαιο 4.

1.1.12 Συγκέντρωση καταλόγου προτύπων σφαλμάτων

Τα μοτίβα σφαλμάτων παρέχονται στο *SMBugFinder* μέσω ενός αρχείου καταλόγου σφαλμάτων με μορφή XML. Αυτό το αρχείο περιέχει για κάθε σφάλμα που εξετάζεται, μια διαδρομή προς το μοντέλο DOT του μοτίβου μαζί με μεταδεδομένα σχετικά με το σφάλμα, όπως μια περιγραφή και (προαιρετικά) τη σοβαρότητα, τα οποία θα εμφανίζονται στον χρήστη εάν εντοπιστεί το σφάλμα. Παρακάτω, παρατίθεται ένα απόσπασμα του καταλόγου που χρησιμοποιήθηκε για την επίδειξή μας.

```
<bugPatterns>
 ...
    <bugPattern>
        <name>Non-conforming Cookie</name>
        <bugLanguage>non-conforming_cookie.dot</bugLanguage>
      <description>The first two ClientHello messages use different
        cipher suites, indicating that cipher suites are not included
            in the cookie computation.
        </description>
        <severity>LOW</severity>
    </bugPattern>

    <bugPattern>
        <name>ClientKeyExchange before Certificate</name>
      <bugLanguage>clientkeyexchange_before_certificate.dot</bugLanguage>
      <description>A server completes the handshake with the client,
        but consumes a ClientKeyExchange before a client Certificate.
        </description>
        <severity>HIGH</severity>
    </bugPattern>
 ...
<bugPatterns>
```

1.2 Γλώσσα παραμετρικών προτύπων σφαλμάτων

1.2.1 Κίνητρο

Όπως αναφέρθηκε, το σύστημα *SMBugFinder* δέχεται πρότυπα σφαλμάτων που περιγράφονται ως αυτόματα πεπερασμένων καταστάσεων χρησιμοποιώντας τη γλώσσα DOT. Αν και αποτελεσματική για απλά, συγκεκριμένα σφάλματα, η DOT από μόνη της στερείται μηχανισμών αφαίρεσης, καθιστώντας δύσκολο ή αδύνατο τον ορισμό επαναχρησιμοποιήσιμων, επεκτάσιμων ή γενικών προτύπων σφαλμάτων.

Στην πράξη, πολλά σφάλματα ακολουθούν γενικές μορφές. Συγκεκριμένα:

- Μπορεί να θέλουμε να υποδηλώσουμε σχέσεις (ιδιότητες) μεταξύ των παραμέτρων δύο ή περισσότερων μηνυμάτων.

- Μπορεί να θέλουμε να χρησιμοποιήσουμε συναρτήσεις στις παραμέτρους των μηνυμάτων.

Για να εκφράσουν τέτοια πρότυπα αποκλειστικά σε DOT, οι χρήστες πρέπει να ορίσουν ρητά όλες τις ακολουθίες μηνυμάτων που προκύπτουν από όλες τις πιθανές τιμές των παραμέτρων, με αποτέλεσμα να δημιουργούνται σύνθετα πρότυπα σφαλμάτων με πολλές παρόμοιες διαδρομές που είναι δύσκολο να διαβαστούν. Για να ξεπεραστούν αυτοί οι περιορισμοί, εισάγουμε μια *παραμετρική γλώσσα για πρότυπα σφαλμάτων*, που επιτρέπει στους χρήστες να ορίζουν επαναχρησιμοποιήσιμες, αφηρημένες και εκφραστικές προδιαγραφές.

1.2.2 Στόχοι σχεδιασμού

Η παραμετρική γλώσσα προτύπων σφαλμάτων έχει σχεδιαστεί με τους ακόλουθους στόχους:

- **Εκφραστικότητα**: Να επιτρέπει τον γενικό ορισμό προτύπων, με παραμέτρους που λαμβάνουν τιμές από απαριθμημένους τύπους.
- **Ενσωμάτωση**: Να είναι συμβατή με τη ροή εργασίας ανάλυσης και επικύρωσης του *SMBugFinder*.
- **Αναγνωσιμότητα**: Να παρέχει μια καθαρή και δηλωτική σύνταξη.

1.2.3 Προδιαγραφές γλώσσας

Η γλώσσα απαιτεί από τον χρήστη να ορίσει σε ένα αρχείο τα απαιτούμενα πεδία (παραμέτρους) ως τύπους δεδομένων, καθορίζοντας τις τιμές τους και, στη συνέχεια, να δηλώσει σε ποιο πεδίο αντιστοιχεί κάθε μήνυμα. Προαιρετικά, μπορούν να δηλωθούν συναρτήσεις στο τέλος του αρχείου. Αναφερόμαστε σε ένα τέτοιο αρχείο ως *αρχείο προδιαγραφών*.

Η γραμματική της γλώσσας καθορίζεται σε μορφή EBNF στο Σχήμα 4.1. Να σημειωθεί ότι *<λέξη>* δηλώνει ένα τερματικό σύμβολο της γραμματικής, δηλαδή μια συμβολοσειρά.

Ας εξηγήσουμε ποιες εκφράσεις παράγει. Ένα πρόγραμμα αυτής της γραμματικής αποτελείται από ορισμούς πεδίων, αντιστοιχίσεις μηνυμάτων σε πεδία και συναρτήσεις. Να σημειωθεί ότι οποιοδήποτε από αυτά μπορεί να παραλειφθεί.

Εισάγουμε τους ορισμούς πεδίων με τη λέξη-κλειδί `` `fields =' `` και τους δηλώνουμε αμέσως παρακάτω. Ένα πεδίο αντιστοιχίζεται στις τιμές που επιτρέπεται να λάβει. Για παράδειγμα, μια έγκυρη δήλωση πεδίου είναι `field` → `value1, value2`. Ουσιαστικά, τα πεδία είναι απαριθμημένοι τύποι δεδομένων.

Στη συνέχεια, εισάγουμε αντιστοιχίσεις μηνυμάτων σε πεδία με τη λέξη-κλειδί `` `parametric_messages =' `` και τις δηλώνουμε ακριβώς από κάτω. Κάθε αντιστοιχία δηλώνεται με ένα `message` σε συνδυασμό με είτε `(field)` είτε `{field}`. Για παράδειγμα, οι έγκυρες εκφράσεις περιλαμβάνουν `message (field)` και `message {field}`.

Τέλος, μια συνάρτηση ορίζεται χρησιμοποιώντας τη λέξη-κλειδί `` `func“ `` ακολουθούμενη από το όνομά της και ένα σύμβολο `` `=” ``. Προαιρετικά, ο τύπος της μπορεί να δηλωθεί αμέσως μετά το όνομά της και πριν από το σύμβολο `` `=“ `` με ένα `` `:” `` ακολουθούμενο από τον τύπο της εισόδου, ένα σύμβολο `` `→' `` και τον τύπο της εξόδου. Να σημειωθεί ότι οι τύποι είναι ουσιαστικά πεδία που πρέπει να οριστούν στην ενότητα ορισμών πεδίων. Ακριβώς παρακάτω, ορίζουμε ρητά την έξοδο που πρέπει να παράγει για κάθε μία από τις εισόδους της. Έτσι, κάθε γραμμή περιέχει μια τιμή εισόδου ακολουθούμενη από ένα σύμβολο `` `→' `` και την τιμή εξόδου.

1.2.4 Ένα παραμετρικό πρότυπο σφάλματος με το αρχείο προδιαγραφών του

Με τη δυνατότητα ορισμού παραμέτρων και συναρτήσεων παράλληλα με ένα μοτίβο σφάλματος, το μοτίβο σφάλματος που περιγράψαμε νωρίτερα στο Σχήμα 3.10(a) μπορεί να γενικευτεί. Θα πρέπει να προσθέσουμε μια παράμετρο (κωδικοποιημένη σε DOT) σε κάθε μήνυμα `I_CLIENT_HELLO` και επίσης να δηλώσουμε τις τιμές που μπορεί να πάρει (χρησιμοποιώντας τη νέα παραμετρική γλώσσα) σε ένα ξεχωριστό αρχείο. Αυτά τα αρχεία φαίνονται στο Σχήμα 4.2.

Αυτό το παραμετρικό μοτίβο είναι σημαντικά πιο γενικό από το αρχικό. Καταγράφει όλες τις πιθανές διαδρομές στις οποίες η σουίτα κρυπτογράφησης που διαφημίστηκε αρχικά από το `CLIENT_HELLO` είναι διαφορετική από αυτή που διαφημίστηκε στο δεύτερο μήνυμα `CLIENT_HELLO`. Ας εξηγήσουμε τη σύνταξη που χρησιμοποιούμε. Το `` `Xlist:=cipher_suites' `` που συνοδεύει τη μετάβαση `I_CLIENT_HELLO`, ορίζει μια παράμετρο στο μήνυμα που αποστέλλεται. Αυτό σημαίνει ότι μπορούμε να καταγράψουμε μια τιμή που συνοδεύει κάθε `CLIENT_HELLO` που αποστέλλεται στον διακομιστή SUT. Συγκεκριμένα σε αυτό το παράδειγμα, η τιμή αντιπροσωπεύει το *σύνολο κρυπτογράφησης* που ο πελάτης ανακοινώνει στον διακομιστή. Μπορούμε αργότερα να αναφερθούμε σε αυτήν την τιμή χρησιμοποιώντας το αναγνωριστικό της, δηλαδή `Xlist`. Το `` `cipher_suites"' `` χρησιμοποιείται για να ορίσει τις τιμές που μπορεί να πάρει το `Xlist`. Παρομοίως, το `[Ylist:=cipher_suites]` που ορίζεται παρακάτω, ορίζει μια δεύτερη παράμετρο που και πάλι παίρνει τιμές από την ίδια λίστα `cipher_suites`. Το τμήμα `` `where Ylist != Xlist"' `` ορίζει ένα κατηγόρημα που εκφράζει μια σχέση μεταξύ `Ylist` και `Xlist`, συγκεκριμένα ότι οι παράμετροι `Xlist` και `Ylist` δεν μπορούν να έχουν την ίδια τιμή.

Το αρχείο προδιαγραφών δηλώνει ότι το *cipher_suites* μπορεί να έχει τις τιμές *RSA*, *PSK*, *ECDH* ή *DH* και επίσης ότι αυτές οι τιμές εμφανίζονται ως παράμετροι στο μήνυμα CLIENT_HELLO. Όταν το μοτίβο παραμετρικού σφάλματος χρησιμοποιείται σε συνδυασμό με αυτό το αρχείο, μπορεί να επεκταθεί, εσωτερικά από το *SMBugFinder*, σε ένα απλό μοτίβο σφάλματος.

Με αυτόν τον τρόπο, ουσιαστικά, έχουμε εκφράσει την ιδέα ότι «ένας διακομιστής DTLS που ολοκληρώνει το βήμα ανταλλαγής cookie μετά τη λήψη ενός δεύτερου CLIENT_HELLO με διαφορετικά υποστηριζόμενα σύνολα κρυπτογράφησης από αυτά του πρώτου CLIENT_HELLO» είναι ένα σφάλμα.

1.2.5 Η χρήση συναρτήσεων σε παραμετρικά μοτίβα σφαλμάτων

Για να δείξουμε τη χρήση των συναρτήσεων στα μοτίβα σφαλμάτων, θα παρουσιάσουμε ένα ακόμα παραμετρικό μοτίβο σφάλματος που βρέθηκε σε υλοποιήσεις πελάτη DTLS, το οποίο κωδικοποιεί τη συμπεριφορά ότι το «πιστοποιητικό που παρέχεται από τον πελάτη ΠΡΕΠΕΙ να περιέχει ένα κλειδί που είναι συμβατό με τα πιστοποιητικά_types» [3, p. 53] σε ένα μήνυμα `CertReq` από τον διακομιστή. Το μοτίβο σφάλματος που χρησιμοποιήσαμε για να το απεικονίσουμε εμφανίζεται σε DOT στο Σχήμα 4.3, συνοδευόμενο από ένα αρχείο προδιαγραφών, όπως απαιτείται.

Ας το εξηγήσουμε. Παρόμοια με αυτό που έχουμε ήδη αναφέρει στο προηγούμενο πρότυπο, ορίζεται μια παράμετρος στο `I_CERTIFICATE_REQUEST` την οποία μπορούμε να αναφερθούμε αργότερα ως `CTset`. Οι τιμές που μπορεί να πάρει ορίζονται στο παραμετρικό αρχείο μέσω του πεδίου `` `cert_type' ``. Στη μετάβαση `certreq` → `bug`, ορίζουμε μια άλλη παράμετρο που ονομάζεται `kt` και χρησιμοποιούμε το πρόθεμα `` `where cert_type(kt) !in CTset"' `` για

να υποδηλώσουμε ότι αυτή η διαδρομή ακολουθείται εάν η τιμή της συνάρτησης `cert_type` όταν εφαρμόζεται στο `kt` δεν βρίσκεται στο `CTset` που ορίστηκε νωρίτερα.

Όπως είναι αναμενόμενο, το πεδίο *key_type* δηλώνεται ότι λαμβάνει τιμές από *RSA* και *ECDSA* και το πεδίο *cert_type* δηλώνεται ότι λαμβάνει τιμές από *RSA_SIGN* και *ECDSA_SIGN*. Το μήνυμα CERTIFICATE_REQUEST ορίζεται ώστε να υποστηρίζει ένα σύνολο από αυτές τις τιμές. Να σημειωθεί η χρήση των «{» και «}». Χρησιμοποιούνται για να δηλώσουν ότι η παράμετρος αυτού του μηνύματος αντιπροσωπεύει ένα σύνολο, ενώ τα «(» και «)» χρησιμοποιούνται για να υποδηλώσουν ότι η παράμετρος λαμβάνει μία μόνο τιμή. Το κυριότερο είναι ότι ορίζουμε τη συνάρτηση *cert_type*, η οποία καθορίζει ρητά την αντιστοίχιση μεταξύ των τιμών εισόδου και εξόδου χρησιμοποιώντας μια βολική δηλωτική σύνταξη.

1.2.6 Υλοποίηση και ενσωμάτωση με το *SMBugFinder*

Η γλώσσα έχει υλοποιηθεί σε OCaml ως ένας μεταγλωττιστής που μεταφράζει το αρχείο προδιαγραφών σε δομές δεδομένων Java, ώστε οι προδιαγραφές να μπορούν να χρησιμοποιηθούν από το *SMBugFinder*. Η θέση του αρχείου προδιαγραφών μπορεί να καθοριστεί εύκολα στον κατάλογο XML των προτύπων σφαλμάτων, έτσι ώστε οι παράμετροί τους να μπορούν να ερμηνευθούν κατά την εκτέλεση του *SMBugFinder*. Στην πράξη, το αρχείο αναλύεται και συνδυάζεται με κάθε πρότυπο σφάλματος που μεταφράζεται σε εσωτερικές δομές αυτομάτων συμβατές με τον πυρήνα του *SMBugFinder*.

Αυτή η αρχιτεκτονική εξασφαλίζει ότι το SMBugFinder μπορεί να επεξεργαστεί παραμετρικά πρότυπα όπως και τα μη παραμετρικά, χρησιμοποιώντας την υπάρχουσα υποδομή. Πρόσθετες πληροφορίες:

- Η γραμματική της γλώσσας υλοποιείται στον γλωσσικό αναλυτή Menhir της OCaml.
- Ο μεταγλωττιστής της γλώσσας παράγεται ως shared object που φορτώνεται σε Java μέσω κώδικα stubs χρησιμοποιώντας το JNI.
- Έχει ληφθεί μέριμνα ώστε να εντοπίζονται όσο το δυνατόν περισσότερα σφάλματα και να εμφανίζονται στον χρήστη, μέσω των σταδίων λεκτικής, συντακτικής και σημασιολογικής ανάλυσης της μεταγλώττισης.

1.2.7 Σύνοψη

Η παραμετρική γλώσσα προτύπων σφαλμάτων αυξάνει σημαντικά τη δύναμη και τη χρηστικότητα του *SMBugFinder*. Με την επέκταση των επιλογών δήλωσης των προτύπων σφαλμάτων, οι χρήστες μπορούν να προσδιορίσουν πιο εύκολα τα πραγματικά σφάλματα. Ωστόσο, τα πρότυπα σφαλμάτων είναι γραμμένα σε DOT, γεγονός που παρακινεί τη δημιουργία ενός καλύτερου εργαλείου, το οποίο συζητείται στο επόμενο κεφάλαιο.

1.3 Γραφικό περιβάλλον χρήστη για μοτίβα σφαλμάτων

1.3.1 Κίνητρο

Ο σχεδιασμός μοτίβων σφαλμάτων με τη χρήση της γλώσσας DOT είναι όχι μόνο χρονοβόρος, αλλά και επιρρεπής σε σφάλματα. Ο χρήστης πρέπει να εξοικειωθεί με τη σύνταξη της γλώσσας DOT και είναι εύκολο να χάσει την εικόνα των καταστάσεων και των μεταβάσεων. Επί του παρόντος, η ροή εργασίας απαιτεί τη σταδιακή σύνταξη προτύπων σφαλμάτων και την οπτικοποίησή τους με τη χρήση της σουίτας λογισμικού `dot`, προκειμένου να διαπιστωθεί εάν το πρότυπο έχει εκφραστεί σωστά. Αυτό είναι αντίθετο στην κοινή λογική και, καθώς αυξάνεται ο αριθμός των προτύπων ή η πολυπλοκότητα των προτύπων, αυξάνεται και ο κίνδυνος ασυνέπειας και επιπλέον κόστους συντήρησης.

Για να το αντιμετωπίσουμε αυτό, αναπτύξαμε ένα εργαλείο γραφικής διεπαφής χρήστη (GUI) που επιτρέπει στους χρήστες να κατασκευάζουν και να επεξεργάζονται οπτικά μοτίβα σφαλμάτων και στη συνέχεια να τα εξάγουν ως μηχανές καταστάσεων σε μορφή DOT. Αυτό καθιστά το *SMBugFinder* πιο φιλικό προς τον χρήστη και απλοποιεί τη διαδικασία δημιουργίας μοτίβων σφαλμάτων.

1.3.2 Επισκόπηση του εργαλείου

Το GUI, με την ονομασία **State Machine Bug Designer**, επιτρέπει στους χρήστες να σχεδιάζουν αυτόματα μηχανήματα απευθείας σε ένα πρόγραμμα περιήγησης. Υποστηρίζει τη διαδραστική κατασκευή και τροποποίηση ντετερμινιστικών πεπερασμένων αυτόματων (DFA), τα οποία μπορούν στη συνέχεια να εξαχθούν σε μορφή DOT, καθώς και σε άλλες υποστηριζόμενες μορφές, όπως JSON, PNG, SVG και LaTeX.

Το κύριο παράθυρο του εργαλείου απεικονίζεται στο Σχήμα 5.1(a).

1.3.3 Βασικά χαρακτηριστικά

Το GUI παρέχει τις ακόλουθες βασικές λειτουργίες:

- **Προσθήκη καταστάσεων**: Κάντε διπλό κλικ στον καμβά για να προσθέσετε έναν νέο κόμβο κατάστασης.
- **Προσθήκη μεταβάσεων**: Χρησιμοποιήστε το `Shift` και σύρετε για να δημιουργήσετε μεταβάσεις μεταξύ καταστάσεων.
- **Ορισμός καταστάσεων αποδοχής (σφάλματος)**: Κάντε διπλό κλικ σε μια υπάρχουσα κατάσταση για να ενεργοποιήσετε μια κατάσταση σφάλματος.
- **Μετακίνηση στοιχείων**: Σύρετε τις καταστάσεις για να αναδιοργανώσετε τη διάταξη.
- **Διαγραφή στοιχείων**: Επιλέξτε μια κατάσταση ή μια άκρη και πατήστε `Delete`.
- **Μορφές εξαγωγής**: Εξαγάγετε το μοτίβο σφάλματος ως PNG, SVG, LaTeX, JSON ή DOT.
- **Εισαγωγή μοτίβων**: Ανεβάστε ένα αρχείο JSON για να φορτώσετε ένα μοτίβο σφάλματος που έχετε αποθηκεύσει προηγουμένως.

Αυτές οι δυνατότητες καθιστούν το εργαλείο κατάλληλο τόσο για το σχεδιασμό νέων μοτίβων όσο και για την επεξεργασία υπαρχόντων.

1.3.4 Ενσωμάτωση ροής εργασίας με το SMBugFinder

Το εργαλείο έχει σχεδιαστεί για να ενσωματώνεται απρόσκοπτα με την υλοποίηση του *SMBugFinder*:

1. Ο χρήστης κατασκευάζει ένα μοτίβο σφάλματος οπτικά χρησιμοποιώντας το GUI.
2. Το μοτίβο εξάγεται ως αρχείο DOT.
3. Το μοτίβο περιλαμβάνεται σε ένα αρχείο καταλόγου.
4. Το SMBugFinder φορτώνει το αρχείο DOT μόλις εκτελεστεί, σαν να είχε γραφτεί από τον χρήστη.

Αυτός ο μηχανισμός εξαγωγής σε DOT διατηρεί τη συμβατότητα με την αρχική λειτουργικότητα του *SMBugFinder*, προσφέροντας παράλληλα μια σημαντικά βελτιωμένη εμπειρία χρήστη για τη δημιουργία προτύπων.

1.3.5 Παράδειγμα χρήσης

Θα παρέχουμε ένα παράδειγμα για σκοπούς επίδειξης. Το σχήμα 5.2 δείχνει το μοτίβο σφάλματος Non Conforming Cookie που συζητήσαμε νωρίτερα (Ενότητα 3.10(a)). Έχει σχεδιαστεί στον browser χρησιμοποιώντας τις λειτουργίες που αναφέραμε στην ενότητα 5.3. Είναι αμέσως προφανές ότι η κατασκευή του είναι πολύ πιο εύκολη και η οπτικοποίησή του είναι πλέον μέρος της διαδικασίας σχεδιασμού. Κάνοντας κλικ στην επιλογή *DOT* στην ενότητα *Export as:* του GUI, μας ζητείται να αποθηκεύσουμε τοπικά το πρότυπο σε μορφή DOT.

1.3.6 Σημειώσεις υλοποίησης

Το GUI υλοποιείται ως εφαρμογή βασισμένη σε πρόγραμμα περιήγησης χρησιμοποιώντας HTML5 και JavaScript. Έχει σχεδιαστεί για να επιτρέπει την εύκολη μετακίνηση των καταστάσεων, καθώς και των ακμών, καθιστώντας τη διαδικασία σχεδιασμού διαισθητική για τον χρήστη, ανεξάρτητα από το μοτίβο σφάλματος. Εσωτερικά, η μηχανή καταστάσεων αναπαρίσταται ως δομή γραφήματος, η οποία σειριοποιείται από και προς DOT ή JSON χρησιμοποιώντας τυπικές βιβλιοθήκες. Αυτή η αρχιτεκτονική επιτρέπει στο GUI να είναι φορητό και διαλειτουργικό.

1.3.7 Σύνοψη

Ο γραφικός σχεδιαστής προτύπων απλοποιεί δραστικά τη δημιουργία προτύπων σφαλμάτων βασισμένων σε μηχανές κατάστασης. Αποσυνθέτει τη σύνταξη DOT και επιτρέπει στους χρήστες να σκέφτονται οπτικά τις συμπεριφορές και τις παραβιάσεις των πρωτοκόλλων. Γεφυρώνοντας το χάσμα μεταξύ της σύλληψης και της υλοποίησης των προτύπων, το GUI συμβάλλει στο να καταστεί το SMBugFinder ένα πιο αποτελεσματικό και προσβάσιμο εργαλείο για την ανίχνευση σφαλμάτων πρωτοκόλλων.

Chapter 2

Introduction

2.1 Motivation and Context

As software systems grow in complexity, so does the need for reliable techniques to identify and prevent bugs. Over the past years, various techniques have emerged which help detect bugs during or after the development of software. Among such tools, bug-finding systems based on state machines offer an intuitive method for specifying and discovering erroneous patterns of behavior in software and more specifically in network protocol implementations.

State Machine Bug Finder (SMBugFinder) [7] is a cross-platform, automated bug detection framework implementing an automata-based technique thoroughly described in a paper published at NDSS'23 [4]. By modeling incorrect sequences of messages, *SMBugFinder* can detect violations of expected behavior found in models of network protocol implementations.

The current implementation of *SMBugFinder* lacks flexibility in terms of expressing more abstract or reusable patterns, limiting its ability to detect certain unwanted behaviors. Furthermore, bug patterns need to be described in a format which makes their expression difficult for the user, namely DOT [8].

2.2 Goals and Contributions

This thesis extends the capabilities of *SMBugFinder* in two major directions:

1. **Parametric Bug Pattern Language:** We introduce a domain-specific language (DSL) for defining *parametric* bug patterns. These patterns generalize the idea of state-machine-based specifications by utilizing more expressive constructs such as sets, functions and predicates, as well as enumerated types that represent the allowed values of parameters in the messages. This enables users to write reusable and modular bug patterns applicable across multiple scenarios.

2. **Graphical User Interface (GUI):** We develop an interactive graphical interface that assists users in designing, visualizing, and validating their bug patterns. This GUI lowers the barrier to entry for users who are not familiar with writing bug patterns in DOT format, making *SMBugFinder* more accessible and intuitive.

Together, these contributions aim to enhance the usability, expressiveness, and versatility of *SMBugFinder* as a bug detection tool.

2.3 Overview of the Thesis

The rest of the thesis is structured as follows:

- **Chapter 2: Background.** We provide an overview of State Machine Bug Finder, its design principles and current implementation.
- **Chapter 3: Parametric Bug Pattern Language.** We describe the design and implementation of the parametric language, including its syntax, semantics, and integration with *SMBugFinder*. Through examples we demonstrate how this language can express real-world bug patterns that were previously unrepresentable.
- **Chapter 4: Graphical User Interface.** We introduce the GUI developed to support visual design and interaction with bug patterns. We discuss its features, architecture, and benefits for end-users.
- **Chapter 5: Future Work.** We discuss the limitations of the thesis, and outline future directions.

2.4 Key Results and Implications

Preliminary evaluations show that the proposed language permits the specification of several non-trivial bug patterns while maintaining compatibility with *SMBugFinder*'s internal architecture. The graphical interface significantly reduces the time and effort required to design patterns, especially for new users.

Together, these enhancements lay the groundwork for making *SMBugFinder* a more practical and powerful tool for state machine bug detection.

Chapter 3

Background

3.1 Stateful Systems and State Machine Bugs

Many real-world systems depend critically on maintaining and managing an internal state during their operation. Such systems are referred to as stateful systems. Examples include vending machines, automated teller machines (ATMs), elevators, network protocol implementations, and numerous others.

The behavior of these systems is typically modeled using state machines, which describe the set of possible internal states and the transitions between them. The correct functioning of a stateful system depends on the proper implementation of its underlying state machine. Errors in the design or implementation of its state machine can lead to unexpected or undesirable system behavior. We refer to such errors as state machine bugs. Our goal is to detect state machine bugs in a given state machine implementation.

As an example, we provide a possible state machine of a vending machine in Fig. 3.1.

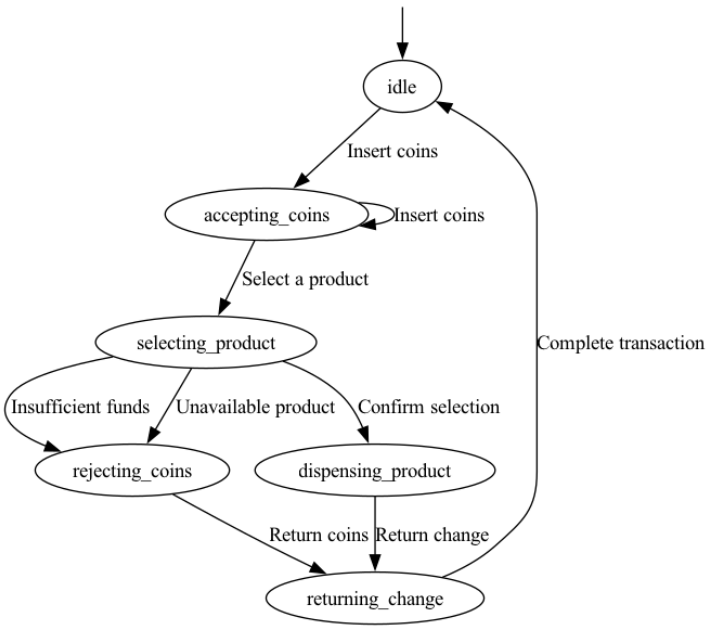


Figure 3.1. *State machine of a vending machine.*

Internally, a vending machine can be at one internal state at a time, and its current state changes based on user interactions or other events. For example, when the machine is

not being used, it is waiting at the *idle* state. When coins are inserted, it transitions to the *accepting_coins* state. If additional coins are inserted, it remains at this state; however, if a product is selected, it transitions to the *selecting_product* state. The remaining states and transitions in the state machine can be interpreted in a similar manner.

The crucial point here is that the behavior of the vending machine is entirely determined by its state machine. But what happens if the state machine contains design or implementation errors?

To illustrate this, consider the erroneous state machine shown in Fig. 3.2. In this state machine, once the selected product is confirmed, the system transitions to the *returning_change* state without dispensing the product. This behavior is clearly incorrect and indicates a flaw in the state machine design. The path leading to this erroneous behavior is highlighted in red.

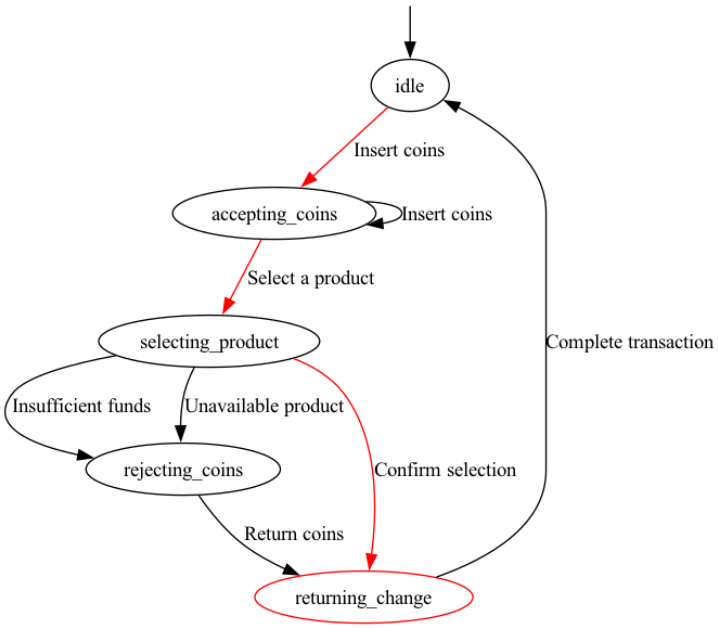


Figure 3.2. *Erroneous state machine of a vending machine.*

We identify and isolate such erroneous behaviors by representing them as state machines, which we refer to as state machine bugs. For instance, the erroneous behaviour discussed earlier is depicted in Fig. 3.3.

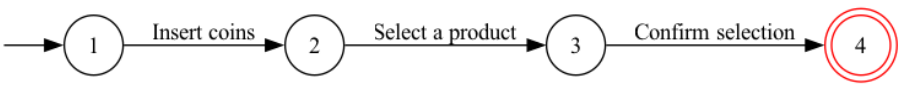


Figure 3.3. *State machine bug: the vending machine does not dispense the selected product.*

3.2 State Machine Bug Finder

Many applications heavily rely on network protocols being properly implemented. Implementations of stateful network protocols such as SSH, TCP, TLS, QUIC, etc. must

maintain a state machine by which they keep track of the presence, order and type of the messages exchanged. Flaws in this state machine, so-called state machine bugs, can render implementations vulnerable to attacks, e.g., establishing a connection with a server without providing all the credentials.

SMBugFinder [7] is a cross-platform protocol-agnostic testing tool whose aim is to make the detection and exposition of such state machine bugs a fully automated process. It takes as input a (possibly learned) model of the system under test (SUT) and a catalogue of automata-encoded patterns for state machine bugs that the SUT is checked for. It uses the patterns to detect and, with provision of a test harness, validate sequences exposing these bugs in the model. Successfully validated sequences are turned automatically into executable test cases.

3.3 Bug Patterns and SUT Models

Bug Patterns encode a behaviour which is considered a bug and that we want to check if this erroneous behavior exists on a SUT.

SMBugFinder allows users to specify bug patterns as DFAs in DOT format where:

- **States** represent the internal state of the SUT.
- **Transitions** are triggered by messages received or emitted by the SUT.
- **Error states** signify the existence of a bug pattern in the SUT.

For example, in the state machine bug shown earlier in Figure 3.3, there are four states that represent internal states of the SUT, three transitions between them which are triggered by actions performed on the vending machine, and an error state shown in a red double circle which signifies the existence of the bug in the state machine of the vending machine.

SUT models encode the state machine of a network protocol which we want to test against a collected catalogue of bug patterns. SUT models are specified as Mealy machines in DOT format which *SMBugFinder* translates internally into DFAs.

An approach that has proven effective for finding state machine bugs is *state fuzzing* [1] [2] [6]. It automatically infers state machine descriptions of protocol implementations using *model learning* [9] [13]. Model learning, is an automated black-box testing technique that, by systematically querying a SUT, infers a state machine model (i.e. a Mealy machine), capturing the SUT's input/output behavior. Model learning needs the SUT's input and output alphabets. Additionally, some abstraction is often required in order to interact with the SUT, i.e., a way to map concrete protocol messages to abstract alphabet symbols and vice versa. In practice, abstraction is implemented in a component known as a test harness or mapper. It is important to note that the models generated by model learning are in most cases approximate, and may not always accurately reflect the SUT's actual behavior. Hence, any flaws found in the model must be validated on the SUT by appropriate tests. This technique has been applied to various network protocol implementations using tools such as DTLS-Fuzzer [5] and EDHOC-Fuzzer [12].

3.4 The Technique Illustrated on DTLS

We will provide an extensive example of a state machine model inferred from a DTLS implementation, along with four state machine bugs. Most text and figures of this section are taken from the NDSS'23 paper [4].

3.4.1 Obtaining the SUT's State Machine Model

The first input that *SMBugFinder* needs is the SUT model. For various DTLS server and client implementations, this can be done with the DTLS-Fuzzer [5] tool. The model which we will describe here corresponds to the JSSE 12.0.2 DTLS server implementation. However, because DTLS-Fuzzer inferred a model with 124 states, which is unreadable, we present in Fig. 3.4 a reduced version of it, consisting of only twelve states.

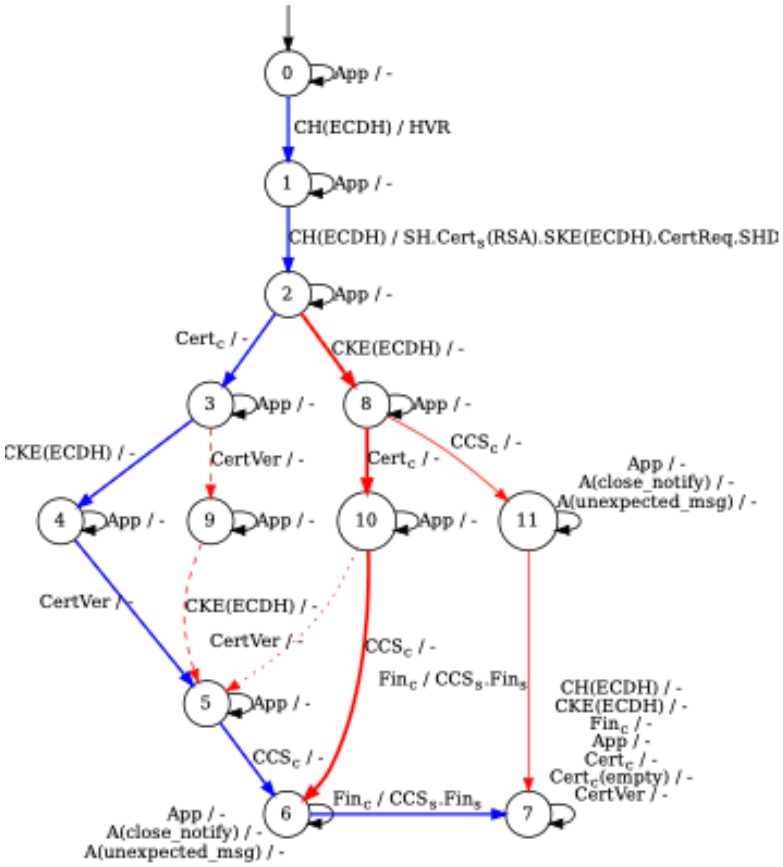


Figure 3.4. *Reduced model of a JSSE 12.0.2 DTLS server [4].*

Let us explain how to read this Mealy machine. Its edges are labeled with elements of the alphabet(s) on two sides of a slash ('/'), with inputs on the left and outputs on the right. At its initial state (0), the server either accepts application data (*App*) and does not output any response, or accepts a *ClientHello (CH)* message and responds with a *HelloVerifyRequest*. At state 1, besides *App*, the server also accepts a *CH* and then replies with a sequence of five messages (*SH*, *Cert*$_s$, *SKE*, *CertReq*, and *SHD*, in this order). The

remaining states are similar, but we also use a shorthand notation (on the self-edges of states 6, 7, and 11) to denote a union of inputs and corresponding outputs. We can also notice the path that correctly completes the handshake, colored in blue, starting from the initial state and ending in state 7.

3.4.2 Encoding Vulnerabilities and Bugs

After obtaining a SUT model, the second input that *SMBugFinder* needs, is a catalogue of patterns that will be checked against the SUT model. The idea is to search the given SUT model, M, for paths that violate the security and correctness requirements of the protocol or look suspiciously like bugs.

There are three issues that need to be addressed to realize this simple idea and make it effective in practice: (1) How do we capture what a requirement violation or bug looks like in M? (2) How do we efficiently search for buggy paths in M? (3) How do we demonstrate that the actual SUT contains bug(s) that are triggered when executing the code that corresponds to a buggy path in M? Let us first answer the first of these questions.

Every network protocol specifies security and correctness requirements in its specification (e.g., in its RFC). Naturally, we can go over this specification and extract (a complete set of) these requirements. Then, we can compose a catalogue of bug patterns, each of them encoded as an automaton that accept sequences of messages that expose the state machine bug. Let us illustrate this encoding by an example. Assume that we are testing a DTLS server which is configured to require a valid certificate from the client. It is obviously an error — in fact, a vulnerability — for the server to request a certificate (with a *CertReq* message) and then enter the phase that completes the handshake without receiving a certificate message from the client ($Cert_c$). We can capture this vulnerability with an automaton that accepts sequences of messages that expose it. Such an automaton is shown in Fig. 3.5.

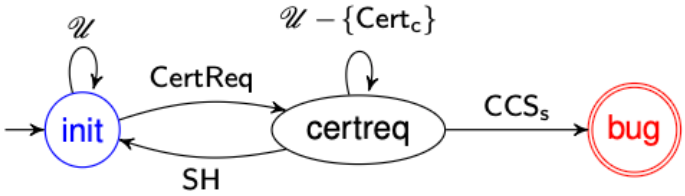


Figure 3.5. *DFA capturing the vulnerability of completing a handshake without a Certificate message from the client [4].*

Let us explain this Deterministic Finite Automaton (DFA) and also the notation that we use. The possible paths from the initial to the final "bug" state express the vulnerability we explained above: there is a *CertReq* followed by a CCS_s message from the server, without any $Cert_c$ message from the client in between. We use the symbol $\mathscr{U}$ for what is often known as the set of all "other" symbols of the alphabet Σ of a DFA (i.e., all symbols except those that are shown as labels in the other outgoing transitions from a state). So, for the initial state $\mathscr{U}$ denotes $\Sigma - \{CertReq\}$, and for the middle state $\mathscr{U}$

denotes $\Sigma - \{SH, CCS_s\}$. By including a SH transition back to the initial state, we allow our DFA to avoid false alarms (due to renegotiation). In DTLS, a client can restart the handshake process at any point by sending a $ClientHello$ message to the server. Some DTLS servers will respond with a HVR at this point and the handshake will be restarted. However, there also exist DTLS implementations that skip repeating the cookie exchange step and restart the handshake. It is therefore safer and more uniform to use the *second* message that a server sends (SH) to denote that the handshake is restarting, and may be completed correctly with another $CertReq$ from the server.

Related to the fact that all our automata are *deterministic*, we note that for all symbols in Σ for which there is no outgoing transition from a state, there are implicit transitions for these symbols to a "sink" state, which we do not show in order not to clutter the pictures. For example, for the automaton in Fig. 3.5 there are implicit transitions out of the final "bug" state for all symbols in Σ, and there is also a transition for $Cert_c$ out of the "certreq" state to the sink state, signifying that if a certificate is sent from the client, then we do not reach the accepting state of our DFA (i.e., there is no bug).

The second vulnerability we capture with a DFA (Fig. 3.6) is similar. In a server configured to require a certificate from the client, a client is authenticated (i.e., handshake is completed) with the client sending a certificate ($Cert_c$) but *without* sending a $CertificateVerify$ to the server. Note that this is a serious security hole. It may indicate that an attacker can authenticate using someone else's certificate without proving ownership of the certificate with a $CertVer$.

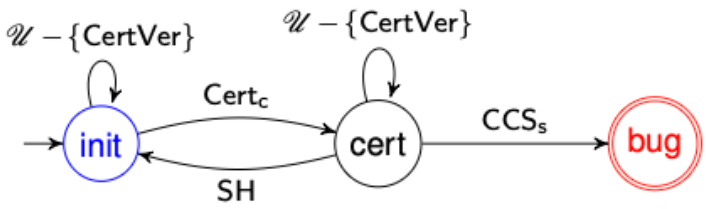


Figure 3.6. *DFA capturing the vulnerability of authenticating a client without it sending a CertificateVerify message [4].*

Besides vulnerabilities, note we can also employ DFAs to capture other kinds of protocol state machine errors. For example, the DFA of Fig. 3.7 captures the bug that, in a valid handshake with a server that uses ECDH, the server cannot consume a $ClientKeyExchange$ before the $Cert_c$ message from the client. In fact, we can generalize the $CKE(ECDH)$ edge in the DFA of Fig. 3.7 to work for all key exchange algorithms that a server supports and which require the client to provide a certificate. For example, we can specify as label of that edge the set $\{CKE(DH), CKE(ECDH), CKE(RSA)\}$. We will use the shorthand $CKE^{\mathscr{K}}$ to refer to such a set.

Our last DFA example (Fig. 3.8) is similar. It captures another case of invalid sequencing of messages: a server completing the handshake while consuming a $CertVer$ before the CKE message from the client.

Having explained how to encode requirements into automata, let us consider how to obtain such requirements in the first place. There are several possibilities: DFAs can be

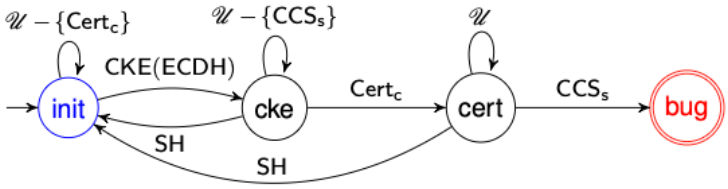


Figure 3.7. *DFA capturing a server completing handshake, but consuming a ClientKeyExchange before a client Certificate [4].*

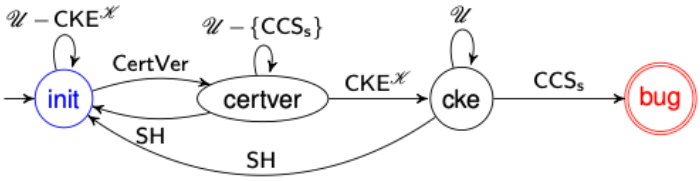


Figure 3.8. *DFA capturing a server completing handshake, but consuming a CertificateVerify before ClientKeyExchange [4].*

directly constructed and refined from i) specific requirements in a protocol's specification (e.g., "The $Cert_c$ message is sent by the client only if the server requests a certificate." [[10], p. 55]), ii) from any previously reported state machine bug for protocol implementations, or iii) by using our knowledge about the protocol and exercising common sense. We note that the set of DFAs capturing requirements, vulnerabilities, and common bugs of a protocol implementation can be specified offline and incrementally.

3.4.3 Detecting Bugs in the Model of the SUT

We are at the point where we have a model M of a SUT in the form of a Mealy machine, and we also have a set of bug patterns expressed as DFAs. In order to detect the bugs in the model of the SUT, we have to address two issues. First, SUT models are described through Mealy machines, but bug patterns are described through DFAs. These two formalisms look similar but technically are very different. Mealy machines specify input- output relations of the symbols in their transitions, while DFAs define a language: the set of words that the DFA accepts. Second, searching for all paths in the model will not work. Most models, besides having many paths due to many states, may even have an infinite number of paths. For example, there are self edges in all states of Fig. 3.4.

In a nutshell, what *SMBugFinder* does in this step is convert the Mealy machine model of the SUT to a DFA and intersect it with each of the DFAs describing the bug patterns. By applying this methodology on our running example, the JSSE 12.0.2 server, *SMBugFinder* will detect that: i) the vulnerability described by the DFA of Fig. 3.5 (completing a handshake without a $Cert$ message) is present in the path 0→1→2→8→11→7; ii) the vulnerability described by the DFA of Fig. 3.6 (authenticating a client without a $CertVer$) appears in the path 0→1→2→8→10→6→7; iii) the bug captured by the DFA of Fig. 3.7 (CKE before $Cert_c$) appears in the path 0→1→2→8→10→5→6→7;

and iv) the bug captured by the DFA of Fig. 3.8 ($CertVer$ before CKE) appears in the path 0→1→2→3→9→5→6→7. Also, *SMBugFinder* will discover that the $CertVer$ before CKE bug also exists on the 0 to 7 path which passes via the edges 2→8 and 8→10 (thick red lines).

3.4.4 Validating the Model's Bugs in the SUT

Unfortunately, the fact that a bug has been detected in the model of a SUT does not prove by itself that the bug exists indeed in the model. There are several considerations we need to take into account. First, DTLS-Fuzzer's model may have not converged (see [5]), and the inferred model may be approximate. This means that the previous step may report bugs that do not exist in the SUT. Second, even if the model is precise, reporting a bug without providing a test case that exhibits it, makes reproducing and fixing the bug very hard.

Thus, in order to validate the presence of bugs in the actual system, we construct sequences of protocol packets and we run them against it. That way we can return them as bug witnesses in the reports to the developers. *SMBugFinder* will automatically validate all bugs on the SUT.

3.5 *SMBugFinder*'s Usage Illustrated on DTLS

As demonstration of usage of *SMBugFinder* in its current state, we will provide a concrete example on a DTLS server implementation, specifically the MbedTLS v2.26.0 model. We will refer to this SUT implementation as MbedTLS.

As noted already, *SMBugFinder* requires first producing the inputs for automated testing: the SUT model and the bug patterns for the protocol which we assemble into a catalogue. We then deploy and run *SMBugFinder* on these inputs. *SMBugFinder* will produce results which we can analyze.

3.5.1 The SUT Model

We can use a model describing the MbedTLS server generated by DTLS-Fuzzer[5] as mentioned earlier. This tool produces the Mealy machine model in DOT format, ready to be used as-is by *SMBugFinder*. Note that this DOT format is common across all protocol state learning tools we know of, meaning that *SMBugFinder* is able to work with the models that all these tools produce. Figure 3.9 provides a visualization of a small part of the model, which is automatically generated using Graphviz's dot utility.

Inspecting this model, one can see the path $s0 \rightarrow s1 \rightarrow s2$ of the cookie exchange step. This path relates to a state machine bug, according to the DTLS RFC [11]. We will next see how we can supply bug patterns that enable *SMBugFinder* to detect this flaw automatically.

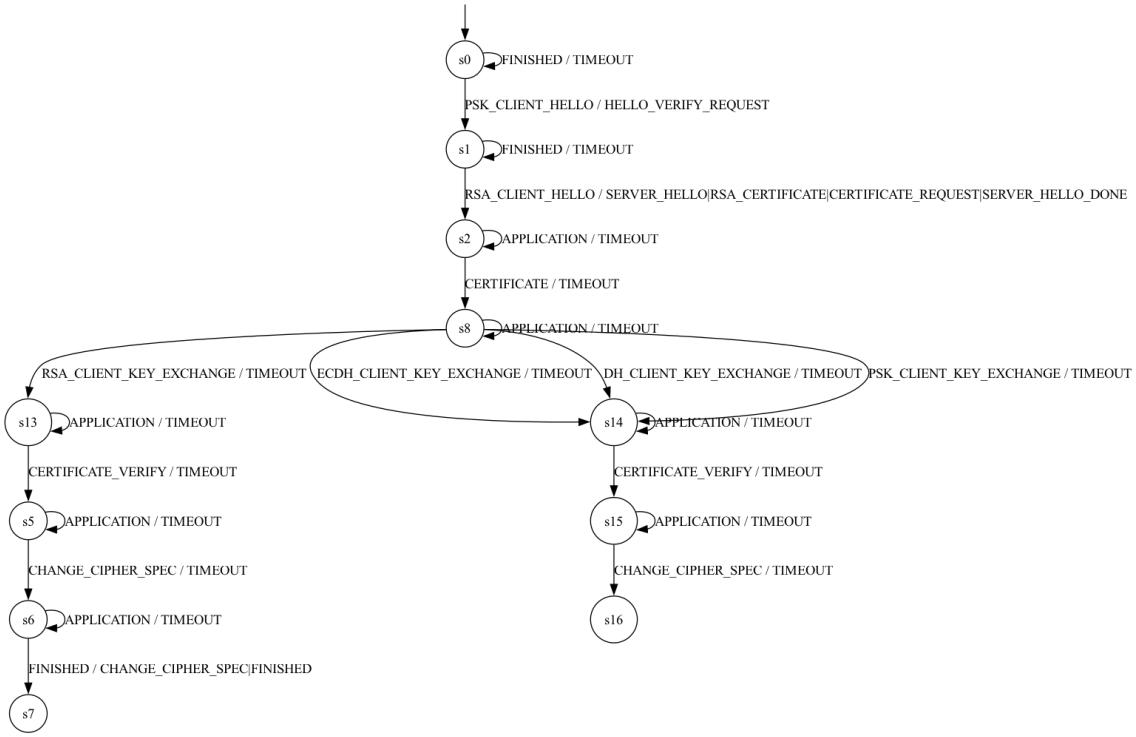


Figure 3.9. *Reduced model of MbedTLS*

3.5.2 Encoding Bugs as Automata

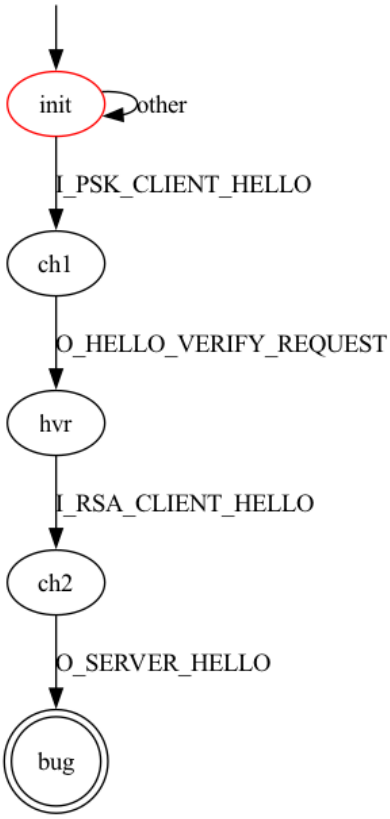
The path from state s0 to state s2, exposes a state machine bug; the server completes the cookie exchange step after receiving a second `CLIENT_HELLO` with the `RSA` ciphersuite despite the first `CLIENT_HELLO` using the `PSK` ciphersuite. This behaviour violates the DTLS's RFC specification [11]. The bug pattern we use to detect this bug is shown in Figure 3.10(a), alongside the source code implementing it (Figure 3.10(b)). Like our SUT model, the bug pattern is also in DOT format and can be viewed with the dot utility.

The above pattern automaton captures (accepts) the following sequence of messages; `I_PSK_CLIENT_HELLO` → `O_HELLO_VERIFY_REQUEST` → `I_RSA_CLIENT_HELLO` → `O_SERVER_HELLO` which violates DTLS's RFC specification as mentioned earlier. The pattern encodes a deterministic finite automaton (DFA) that is specified for all the SUT model's input and output symbols, which appear in the pattern prefixed by I_ and O_, respectively. It is important to note that currently, edges of the graph can contain more advanced constructs than just plain messages. These include:

- The symbol U which represents the set of SUT symbols which are not included in any outgoing edge from a state.
- Set expressions, such as the subtraction of a set from another.

Additionally, symbols for which no transition is specified lead to an implicit rejecting sink state.

The compactness of the encoding is critical to the usability of *SMBugFinder* and we will discuss how it is further extended in chapter 4.



(a) Non conforming cookie visualized

```
digraph G {
  init [color="red"]
  bug [shape="doublecircle"]

  init -> ch1 [label="I_PSK_CLIENT_HELLO"]
  init -> init [label="other"]
  ch1 -> hvr [label="O_HELLO_VERIFY_REQUEST"]
  hvr -> ch2 [label="I_RSA_CLIENT_HELLO"]
  ch2 -> bug [label="O_SERVER_HELLO"]

  __start0 [label="" shape="none"
            width="0" height="0"];
  __start0 -> init;
}
```

(b) Non conforming cookie in DOT format

Figure 3.10. *Non conforming cookie*

3.5.3 Assembling a Catalogue of Bug Patterns

Bug patterns are provided to *SMBugFinder* via a bug catalogue file formatted in XML. This file contains for each bug considered, a path to the pattern DOT model along with metadata on the bug, such as a description and (optionally) severity, which will be shown to the user if the bug is detected. Below, an excerpt of the catalogue used for our demonstration is given.

```
<bugPatterns>
 ...
    <bugPattern>
        <name>Non-conforming Cookie</name>
        <bugLanguage>non-conforming_cookie.dot</bugLanguage>
        <description>The first two ClientHello messages use different
            cipher suites, indicating that cipher suites are not included
            in the cookie computation.
        </description>
        <severity>LOW</severity>
    </bugPattern>

    <bugPattern>
        <name>ClientKeyExchange before Certificate</name>
        <bugLanguage>clientkeyexchange_before_certificate.dot</bugLanguage>
        <description>A server completes the handshake with the client,
            but consumes a ClientKeyExchange before a client Certificate.
        </description>
        <severity>HIGH</severity>
    </bugPattern>
 ...
<bugPatterns>
```

Chapter 4

Parametric Bug Pattern Language

4.1 Motivation

As mentioned, *SMBugFinder* system accepts bug patterns described as finite-state automata using the DOT language. While effective for simple, concrete bugs, DOT by itself lacks abstraction mechanisms, making it difficult or impossible to define reusable, extensible or general bug patterns.

In practice, many bugs follow generic forms. Notably:

- We might want to denote relations (properties) on the parameters of two or more messages.
- We might want to use functions on the parameters of messages.

To express such patterns solely in DOT, users must explicitly define all sequences of messages that arise from all possible values of parameters, resulting in complex bug patterns with many similar routes which are difficult to read. To overcome these limitations, we introduce a *parametric language for bug patterns*, allowing users to define reusable, abstract, and expressive specifications.

4.2 Design Goals

The parametric bug pattern language is designed with the following goals:

- **Expressiveness**: Allow patterns to be defined generically, with parameters that take values from enumerated types.
- **Integration**: Be compatible with *SMBugFinder*'s analysis and validation workflow.
- **Readability**: Provide a clean and declarative syntax.

4.3 Language Specification

4.3.1 Overview

The language requires the user to define in a file the needed fields (parameters) as data-types by specifying their values and subsequently declare which field each message

corresponds to. Optionally, functions can be declared at the end of the file. We refer to such a file as a *specification file*.

4.3.2 Formal Specification

The grammar of the language is specified in EBNF form in Figure 4.1. Note that <*word*> denotes a terminal symbol of the grammar, that is a string.

Let us explain what expressions it produces. A program of this grammar consists of field definitions, message to field mappings, and functions. Note that any of these can be omitted.

We introduce field definitions with the keyword '`fields =`' and we declare them right below. A field is mapped to the values that it is allowed to take. For example, a valid field declaration is `field` → `value1`, `value2`. Essentially, fields are enumerated data-types.

Then, we introduce message to field mappings with the keyword '`parametric_messages =`' and we declare them right below. Each mapping is declared with a `message` combined with either `(field)` or `{field}`. For example, valid expressions include `message` (`field`) and `message` {`field`}.

Finally, a function is defined using the keyword '`func`' followed by its name and a '`=`' symbol. Optionally, its type can be declared right after its name and before the '`=`' symbol with a '`:`' followed by the type of the input, a '→' symbol, and the type of the output. Note that types are essentially fields that must be defined in the field definitions section. Right below, we define explicitly the output that it should produce given each of its inputs. Thus, each line contains an input value followed by a '→' and the output value.

$$
\begin{aligned}
\langle program \rangle &::= \langle fields_def \rangle? \ \langle messages_def \rangle? \ \langle func \rangle * \\
\langle fields_def \rangle &::= \textbf{fields} = \langle field \rangle + \\
\langle field \rangle &::= \langle word \rangle \rightarrow \langle field_values \rangle \\
\langle field_values \rangle &::= \langle word \rangle \ (, \langle word \rangle) * \\
\langle messages_def \rangle &::= \textbf{parametric_messages} = \langle message \rangle + \\
\langle message \rangle &::= \langle word \rangle \ \langle message_val \rangle \\
\langle message_val \rangle &::= \textbf{(}\langle word \rangle\textbf{)} \mid \textbf{\{}\langle word \rangle\textbf{\}} \\
\langle func \rangle &::= \textbf{func} \ \langle word \rangle \ \langle func_ty \rangle? = \langle func_body \rangle \\
\langle func_ty \rangle &::= : \langle word \rangle \rightarrow \langle word \rangle \\
\langle func_body \rangle &::= (\langle word \rangle \rightarrow \langle word \rangle) *
\end{aligned}
$$

Figure 4.1. *Grammar of the parametric bug pattern language*

4.3.3 A Parametric Bug Pattern with its Specification File

With the ability to define parameters and functions alongside a bug pattern, the bug pattern we described earlier in Figure 3.10(a) can be generalised. We will have to add a parameter (encoded in DOT) in each `I_CLIENT_HELLO` message and also declare the

values it can take (using the new parametric language) in a seperate file. These files are shown in Figure 4.2.

```
digraph G {
    init -> ch1 [label="I_CLIENT_HELLO[Xlist:=cipher_suites]"]
    init -> init [label="other"]
    ch1 -> hvr [label="O_HELLO_VERIFY_REQUEST"]
    hvr -> ch2 [label="I_CLIENT_HELLO[Ylist:=cipher_suites]
        where Ylist != Xlist"]
    ch2 -> bug [label="O_SERVER_HELLO"]
}
```

(a) Parametric non conforming cookie in DOT format

```
fields =
    cipher_suites -> RSA, PSK, ECDH, DH

parametric_messages =
    CLIENT_HELLO {cipher_suites}
```

(b) Accompanying specification file

Figure 4.2. *Parametric non conforming cookie*

This parametric pattern in significantly more general than the original. It captures all possible paths in which the cipher suite first advertised by `CLIENT_HELLO` is different to the one that is advertised in the second `CLIENT_HELLO` message. Let us explain the syntax we use. '`Xlist:=cipher_suites`' which accompanies the `I_CLIENT_HELLO` transition, defines a parameter on the message being sent. This means that we are able to capture a value that comes with each `CLIENT_HELLO` sent to the server SUT. Particularly in this example, the value represents the *cipher suite* that the client advertises to the server. We can later reference this value using its identifier, namely `Xlist`. '`cipher_suites"`' is used to define the values that `Xlist` is allowed to take. Similarly, `[Ylist:=cipher_suites]` which is defined further down, defines a second parameter which again takes values from the same pool of `cipher_suites`. The '`where Ylist != Xlist"`' part defines a predicate that expresses a relation between `Ylist` and `Xlist`, particularly that the parameter `Xlist` and `Ylist` cannot hold the same value.

The specification file states that *cipher_suites* can have the values *RSA*, *PSK*, *ECDH* or *DH* and also that these values appear as a parameter in the CLIENT_HELLO message. When the parametric bug pattern is used in conjuction with this file, it can be expanded, internally by *SMBugFinder*, into a plain bug pattern.

That way, in essense, we have expressed the idea that "a DTLS server that completes the cookie exchange step after receiving a second CLIENT_HELLO with different supported ciphersuites to those in the first CLIENT_HELLO" is a bug.

4.3.4 The Use of Functions In Parametric Bug Patterns

To show the use of functions in bug patterns, we will present another parametric bug pattern found in DTLS client implementations that encodes the behaviour that the

"certificate provided by the client MUST contain a key that is compatible with certificate_types" [3, p. 53] in a `CertReq` message from the server. The bug pattern we have used to illustrate this is shown in DOT in Figure 4.3, accompanied by a specification file as needed.

```
init -> certreq [label="I_CERTIFICATE_REQUEST[CTset:=cert_type]"]
init -> init [label="other"]

certreq -> certreq [label="other"]
certreq -> bug [label="O_CERTIFICATE[kt:=key_type]
    where cert_type(kt) !in CTset"]
```

(a) Wrong certificate type in DOT format

```
fields =
        key_type -> RSA, ECDSA
        cert_type -> RSA_SIGN, ECDSA_SIGN

parametric_messages =
        CERTIFICATE (key_type)
        CERTIFICATE_REQUEST {cert_type}

func cert_type =
        RSA -> RSA_SIGN
        ECDSA -> ECDSA_SIGN
```

(b) Accompanying specification file

Figure 4.3. *Wrong certificate type*

Let us explain it. Similarly to what we have already mentioned in the previous pattern, a parameter is being defined in `I_CERTIFICATE_REQUEST` which we can reference later as `CTset`. The values which it can take are defined in the parametric file via the '`cert_type`' field. On the `certreq` → `bug` transition, we define another parameter called `kt` and we use the predicate '`where cert_type(kt) !in CTset`"' to denote that this path is taken if the value of the function `cert_type` when applied to `kt` is not in the `CTset` defined earlier.

Expectedly, the *key_type* field is declared to take values from *RSA* and *ECDSA* and the *cert_type* field is declared to take values from *RSA_SIGN* and *ECDSA_SIGN*. The CERTIFICATE_REQUEST message is defined to support a set of those values. Note the use of "{" and "}". They are being used to declare that the parameter of this message represents a set, whereas "(" and ")" are being used to denote that the parameter takes a single value. Most importantly, we define the *cert_type* function, which explicitly specifies the mapping between input and output values using a convenient declarative syntax.

4.4 Implementation and Integration with *SMBugFinder*

The language has been implemented in OCaml as a parser and compiler that translates the specification file into Java data structures so that they can be used by *SMBugFinder*.

The specification file's location can conveniently be specified in the XML catalogue of bug patterns so that their parameters can be interpreted during *SMBugFinder*'s execution. In practice, the file is parsed and combined with each bug pattern it is translated into internal automata structures compatible with *SMBugFinder*'s core.

This architecture ensures that SMBugFinder can process parametric patterns just like non-parametric ones, using the existing infrastructure.

Importantly:

- The language's grammar is implemented in OCaml's parser generator Menhir.
- The language's compiler is produced as shared object to be loaded onto Java through stubs code using the JNI.
- Care has been taken to catch as many errors as possible and display them to the user, through the lexical, syntax and semantics analysis stages of compiling.

4.5 Summary

The parametric bug pattern language significantly increases the power and usability of *SMBugFinder*. By extending bug patterns' declaration options, users can more easily specify real-world bugs. Yet, bug patterns are written in DOT which motivates the creation of a better tool which is discussed in the next chapter.

Chapter 5

Graphical User Interface for Bug Patterns

5.1 Motivation

Designing bug patterns manually using the DOT language is not only time consuming but also error prone. The user is required to familiarize themselves with the DOT syntax, and it is easy to lose track of the states and transitions. Currently, the workflow requires gradually writing bug patterns and visualizing them using `dot`'s software suite in order to find if the pattern has been correctly expressed. This is counterintuitive, and as the number of patterns or the complexity of the patterns increases, so does the risk of inconsistencies and maintenance overhead.

To alleviate this, we developed a graphical user interface (GUI) tool that allows users to visually construct and edit bug patterns and then export them as state machines in DOT format. This makes *SMBugFinder* more user-friendly and simplifies the bug pattern creation process.

5.2 Overview of the Tool

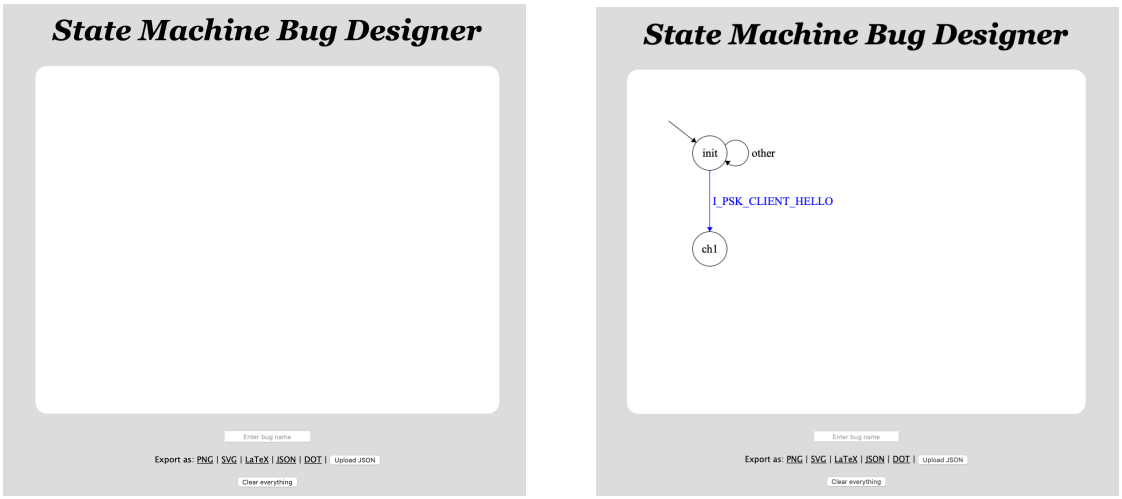
The GUI, named **State Machine Bug Designer**, allows users to design automata directly in a browser. It supports interactive construction and modification of deterministic finite automata (DFAs), which can then be exported to DOT format, as well as other supported formats including JSON, PNG, SVG and LaTeX.

The main pane of the tool is depicted in Figure 5.1(a).

5.3 Core Features

The GUI provides the following key functionalities:

- **Add states**: Double-click on the canvas to add a new state node.
- **Add transitions**: Use `Shift` and drag to create transitions between states.
- **Define accepting (bug) states**: Double-click on an existing state to toggle an error state.
- **Move elements**: Drag states around to reorganize the layout.



(a) GUI (b) Designing a bug in the GUI

Figure 5.1. *The State Machine Bug Designer*

- **Delete elements**: Select a state or edge and press `Delete`.
- **Export formats**: Export the bug pattern as PNG, SVG, LaTeX, JSON, or DOT.
- **Import patterns**: Upload a JSON file to load a previously saved bug pattern.

These capabilities make the tool suitable for both designing new patterns and editing existing ones.

5.4 Workflow Integration with SMBugFinder

The tool is designed to integrate seamlessly with *SMBugFinder*'s implementation:

1. The user constructs a bug pattern visually using the GUI.
2. The pattern is exported as a DOT file.
3. The pattern is included in a catalogue file.
4. SMBugFinder loads the DOT file once it is executed as if it was written by the user.

This export-to-DOT mechanism preserves compatibility with the original *SMBugFinder* functionality while offering a significantly improved user experience for pattern construction.

5.5 Example Use Case

We will provide an example for demonstration purposes. Figure 5.2 shows the Non Conforming Cookie bug pattern we discussed earlier (Section 3.10(a)). It has been designed in the browser using the functionalities we mentioned in section 5.3. Immediately, it is obvious that its construction is much easier and its visualization is now part of the

designing process. By clicking on the *DOT* option in the *Export as:* section of the GUI we are prompted to save locally the pattern in DOT format.

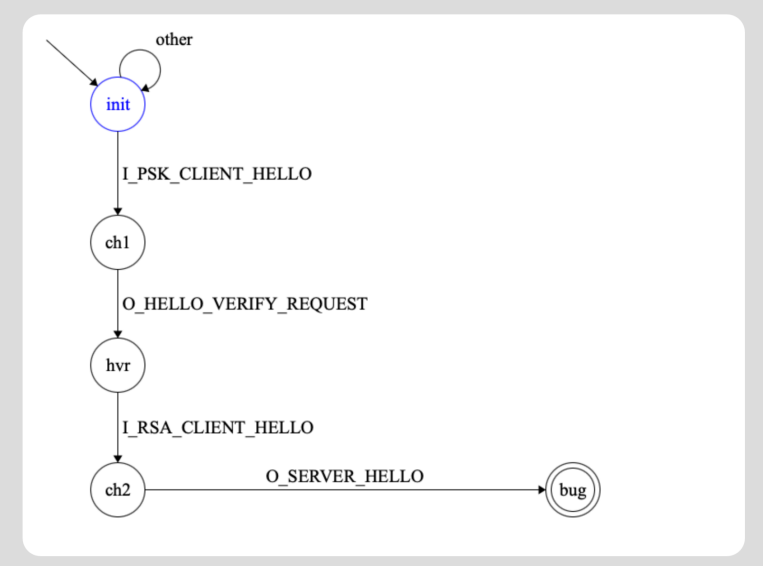


Figure 5.2. *Non conforming cookie bug pattern designed in the GUI*

5.6 Implementation Notes

The GUI is implemented as a browser-based application using HTML5 and JavaScript. It has been designed to conveniently allow moving states around, as well as edges making the designing process intuitive for the user, no matter the bug pattern.

Internally, the state machine is represented as a graph structure, which is serialized to and from DOT or JSON using standard libraries. This architecture allows the GUI to be portable and cross-platform.

5.7 Summary

The graphical pattern designer drastically simplifies the task of creating state-machine-based bug patterns. It abstracts away DOT syntax and allows users to think visually about protocol behaviors and violations. By bridging the gap between pattern conception and implementation, the GUI helps make SMBugFinder a more effective and accessible tool for protocol bug discovery.

Chapter 6

Future Work

While the additions and ameliorations that have been described so far significantly improve the expressibility and ease of use of State Machine Bug Finder, there are certain aspects that can be further improved.

Notably, the parametric bug language needs to be further refined and adapted as parametric bug patterns grow in complexity. The constructs needed to express them will have to be supported by the compiler in a future version. Namely, higher order functions could have a pivotal role in expressing even more advanced ideas in such patterns.

The GUI can also be refined further, by integrating a modern front-end design. More importantly, the ability to directly import and render DOT files can elliminate the need to save bug patterns in JSON.

Bibliography

[1] Benjamin Beurdouche et al. "A Messy State of the Union: Taming the Composite State Machines of TLS." In: *Commun. ACM* 60.2 (Feb. 2017), pp. 99–107. DOI: 10.1145/3023357. URL: https://doi.org/10.1145/3023357.

[2] Joeri de Ruiter and Erik Poll. "Protocol State Fuzzing of TLS Implementations." In: *24th USENIX Security Symposium*. Washington, D.C., USA: USENIX Association, Aug. 2015, pp. 193–206. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/de-ruiter.

[3] Tim Dierks and Eric Rescorla. *The Transport Layer Security TLS Protocol Version 1.2*. RFC 5246. RFC. Aug. 2008. URL: http://www.rfc-editor.org/rfc/rfc5246.txt.

[4] Paul Fiterau-Brostean et al. "Automata-Based Automated Detection of State Machine Bugs in Protocol Implementations." In: *30th Annual Network and Distributed System Security Symposium*. NDSS 2023. San Diego, CA, USA: The Internet Society, Feb. 2023. URL: https://www.ndss-symposium.org/ndss-paper/automata-based-automated-detection-of-state-machine-bugs-in-protocol-implementations/.

[5] Paul Fiterau-Brostean et al. "DTLS-Fuzzer: A DTLS Protocol State Fuzzer." In: *15th IEEE Conference on Software Testing, Verification and Validation*. ICST 2022. Valencia, Spain: IEEE, Apr. 2022, pp. 456–458. DOI: 10.1109/ICST53961.2022.00051. URL: https://doi.org/10.1109/ICST53961.2022.00051.

[6] Paul Fiterău-Broştean et al. "Analysis of DTLS Implementations Using Protocol State Fuzzing." In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2523–2540. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/fiterau-brostean.

[7] Paul Fiterău-Broştean et al. "SMBugFinder: An Automated Framework for Testing Protocol Implementations for State Machine Bugs." In: Artifact for the ISSTA '24 paper with the same title. Sept. 2024. DOI: 10.1145/3650212.3685310. URL: https://doi.org/10.1145/3650212.3685310.

[8] Graphviz. *DOT Documentation*. URL: https://graphviz.org/documentation/.

[9] Harald Raffelt et al. "Dynamic testing via automata learning." In: *Int. J. Softw. Tools Technol. Transf.* 11.4 (2009), pp. 307–324. DOI: 10.1007/s10009-009-0120-7. URL: https://doi.org/10.1007/s10009-009-0120-7.

[10] E. Rescorla et al. *Transport Layer Security (TLS) Renegotiation Indication Extension*. RFC 5746. RFC. Feb. 2010. URL: http://www.rfc-editor.org/rfc/rfc5746.txt.

[11] Eric Rescorla and Nagendra Modadugu. *Datagram Transport Layer Security Version 1.2*. RFC 6347. RFC. Jan. 2012. URL: http://www.rfc-editor.org/rfc/rfc6347.txt.

[12] Konstantinos Sagonas and Thanasis Typaldos. "EDHOC-Fuzzer: An EDHOC protocol state fuzzer." In: *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA '23. Seattle, WA, USA: ACM, July 2023, pp. 1495–1498. DOI: 10.1145/3597926.3604922. URL: https://doi.org/10.1145/3597926.3604922.

[13] Frits W. Vaandrager. "Model learning." In: *Commun. ACM* 60.2 (2017), pp. 86–95. DOI: 10.1145/2967606. URL: https://doi.org/10.1145/2967606.