ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Analysis of EDHOC Implementations Using Protocol State Fuzzing

---

Συγγραφέας:
Αθανάσιος Χρίστος Τυπάλδος

Επιβλέπων:
Κωνσταντίνος Σαγώνας
Αναπληρωτής Καθηγητής

Αθήνα, 6 Απριλίου 2023

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Analysis of EDHOC Implementations Using Protocol State Fuzzing

---

Συγγραφέας:
Αθανάσιος Χρίστος Τυπάλδος

Επιβλέπων:
Κωνσταντίνος Σαγώνας
Αναπληρωτής Καθηγητής

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 6η Απριλίου 2023.

(Υπογραφή)	*(Υπογραφή)*	*(Υπογραφή)*
Κωνσταντίνος Σαγώνας Αναπληρωτής Καθηγητής	Δημήτριος Φωτάκης Καθηγητής	Αριστείδης Παγουρτζής Καθηγητής

Declaration of Authorship

Copyright © Αθανάσιος Χρίστος Τυπάλδος, 2023.
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.
Με επιφύλαξη παντός δικαιώματος. All rights reserved.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μή κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.

Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθέί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

Υπογραφή: ______________________________

Ημερομηνία: ______________________________

Περίληψη

Την τελευταία δεκαετία εμφανίζονται όλο και περισσότερες συσκευές που ανήκουν στο οικοσύστημα του Διαδικτύου των Πραγμάτων (IoT). Οι συσκευές αυτές έχουν τη δυνατότητα να συνδέονται στο διαδίκτυο και να ανταλλάσσουν δεδομένα κυρίως με τη χρήση ενσωματωμένων αισθητήρων. Το περιβάλλον στο οποίο λειτουργούν είναι συνήθως περιορισμένο σε πόρους και εύρος ζώνης, γεγονός που περιορίζει και την ασφάλεια που παρέχουν. Για την αντιμετώπιση αυτού του προβλήματος έχουν προταθεί και τυποποιηθεί πολλά νέα πρωτόκολλα διαδικτύου και ασφάλειας, τα οποία απευθύνονται σε τέτοιες συσκευές. Μεταξύ αυτών είναι και το πρωτόκολλο EDHOC, το οποίο είναι ένα ελαφρύ πρωτόκολλο ανταλλαγής κλειδιών ιδανικό για περιορισμένα περιβάλλοντα. Τα πρωτόκολλα δεν πρέπει να είναι μόνο καλά σχεδιασμένα στη θεωρία, αλλά και στην πράξη. Οι υλοποιήσεις των πρωτοκόλλων θα πρέπει να είναι ακριβείς, εύρωστες και να συμμορφώνονται με τις προδιαγραφές τους. Το κύριο αντικείμενο της παρούσας διπλωματικής είναι η ανάλυση των υλοποιήσεων του πρωτοκόλλου EDHOC. Ένας από τους πολλούς διαθέσιμους τρόπους για την ανάλυση τέτοιων υλοποιήσεων είναι να δημιουργηθεί πρώτα ένα (προσεγγιστικό) μοντέλο μηχανής καταστάσεων των υλοποιήσεων, το οποίο στη συνέχεια θα επιθεωρηθεί οπτικά ή θα χρησιμοποιηθεί σε αυτοματοποιημένο έλεγχο. Στην παρούσα διπλωματική παρουσιάζεται ο EDHOC-Fuzzer, το οποίο είναι ένα εργαλείο ικανό να μαθαίνει τέτοια μοντέλα μηχανών καταστάσεων χρησιμοποιώντας την τεχνική του protocol state fuzzing. Η τεχνική αυτή έχει ήδη εφαρμοστεί με επιτυχία σε πολλά άλλα πρωτόκολλα. Στην παρούσα διπλωματική διάφορες υλοποιήσεις EDHOC τέθηκαν υπό εκμάθηση και τα μοντέλα που μαθεύτηκαν αναλύθηκαν διεξοδικά. Η ανάλυσή τους παρέχει πληροφορίες για τον τρόπο με τον οποίο συμπεριφέρεται η υλοποίηση και μπορεί να αποκαλύψει ορισμένα καλά κρυμμένα λογικά σφάλματα, τα οποία μπορεί να οδηγήσουν ακόμη και σε ευπάθειες ασφαλείας. Αυτό αναδεικνύει όχι μόνο την αποτελεσματικότητα των τεχνικών που χρησιμοποιήθηκαν για την ανάλυση των υλοποιήσεων, αλλά και τον αντίκτυπο που μπορούν να έχουν τέτοια εργαλεία, όταν προστεθούν στην εργαλειοθήκη των ανθρώπων που υλοποιούν τέτοια πρωτόκολλα.

Λέξεις Κλειδιά: ασφάλεια πρωτοκόλλων, ασφάλεια λογισμικού, εκμάθηση μοντέλων, ενεργητική εκμάθηση αυτομάτων, protocol state fuzzing, πρωτόκολλο EDHOC

Abstract

The last decade an increasing number of devices have come to light that belong to the ecosystem of Internet of Things (IoT). These devices have the ability to connect to the internet and exchange data mainly using embedded sensors. The environment that those devices operate in are usually limited in resources and bandwidth, which is a fact that also restricts the security that they provide. In order to address this problem a lot of new network and security protocols have been proposed and standardized, targeting such devices operating in constrained environments. Among them is the EDHOC protocol, which is a lightweight key exchange protocol ideal for constrained environments. Not only the protocols should be well designed in theory, but also in practice. The implementations of the protocols ought to be precise, robust and should comply with their specifications. The main subject of this thesis is the analysis of the implementations of the EDHOC protocol. One of the many available ways to analyze such implementations is to first generate a close approximation of the implementations' underlying state machine model and then inspect the resulting model or use it for model-based testing. This thesis presents EDHOC-Fuzzer, which is a tool capable of learning such state machine models using the technique of protocol state fuzzing. This technique has been already successfully applied to many other protocols. In this thesis several EDHOC implementations were put under learning and their learned models have been analyzed thoroughly. Their analysis provides insights in the way that the implementation behaves and can uncover some well-hidden logical flaws that can lead to minor bugs or even security vulnerabilities. This showcases not only the effectiveness of the techniques used to analyze these implementations, but also the impact that such tools can have, when they are added in the toolbox of people implementing such protocols.

Keywords: protocol security, software security, model learning, active automata learning, protocol state fuzzing, EDHOC protocol

Acknowledgements

First I would like to thank my advisor, Professor Kostis Sagonas, for all the support and guidance he offered me and continues to do so. His help was invalueable and involved both technical and external topics. I am very glad that we met and cooperated. I would also like to thank the other examining commitee members for their kind words. I would like to make a special mention for Professor Nikolaos Papaspyrou, who was an inspiration to me from the beginning of my undergraduate studies and I always enjoyed his classes. All of the aforementioned Professors changed my perspective and sparked my interest in computer science.

Next, I would like to thank all of my friends that supported me throughout these years and were present when I needed them. A lot of unforgettable memories have been formed that will follow me along.

Last but not least, I would like to thank my family for their constant effort and support. Without them I would not have been the same person nor would I have achieved what I have now. I am very grateful for my parents, my brother, and my sister, whom I know I can always count on.

Contents

Περίληψη **iv**

Abstract **vi**

Acknowledgements **viii**

1 Εκτεταμένη Ελληνική Περίληψη **1**
1.1 Εισαγωγή 1
1.2 Θεωρητικό Υπόβαθρο 3
1.3 Υλοποίηση 10
1.4 Πειράματα 21
1.5 Συμπέρασμα 35

2 Introduction **37**
2.1 Contribution 38
2.2 Outline 38

3 Background **39**
3.1 CBOR Data Format 39
3.2 COSE Standard 39
3.3 CoAP Protocol 39
3.4 OSCORE Protocol 40
3.5 EDHOC Protocol 41
3.5.1 Roles and Messages 41
3.5.2 Authentication Method 42
3.5.3 Connection Identifiers 42
3.5.4 Authentication Parameters 43
3.5.5 Cipher Suites 44
3.5.6 Ephemeral Public Keys 45
3.5.7 Application Profile 45
3.5.8 OSCORE Context Derivation 45
3.6 Deterministic Finite Automaton 45
3.7 Mealy Machine 46
3.8 Active Automata Learning 46
3.9 Protocol State Fuzzing 47
3.10 Related Work 48

4 Implementation **49**
4.1 EDHOC-Fuzzer 50

4.1.1 Abstract Symbol SUL . . . 50
4.1.2 Mapper . . . 51
4.1.3 Alphabets . . . 52
4.1.4 Remarks . . . 54
4.1.5 Outcome . . . 54
4.2 ProtocolState-Fuzzer . . . 55
4.2.1 Data Flow . . . 55
4.2.2 State Fuzzer . . . 56
4.2.3 Learner . . . 56
4.2.4 Membership SUL Oracle . . . 57
4.2.5 Equivalence Oracle . . . 58
4.2.6 Equivalence SUL Oracle . . . 58
4.2.7 SUL Wrappers . . . 59
4.2.8 Outcome . . . 60

5 Experiments **61**
5.1 EDHOC-RS . . . 62
5.1.1 Default Client . . . 62
5.1.2 Default Server . . . 63
5.2 RISE . . . 64
5.2.1 Default Client . . . 64
5.2.2 Patched Client . . . 65
5.2.3 Default Server . . . 67
5.2.4 Patched Server . . . 70
5.3 SIFIS-HOME . . . 71
5.3.1 Default Client Phases 1, 2 . . . 71
5.3.2 Default Client Phases 3, 4 . . . 73
5.3.3 Default Server Phases 1, 2, 3, 4 . . . 74
5.4 uOSCORE-uEDHOC . . . 75
5.4.1 Default Client EDHOC . . . 75
5.4.2 Default Client EDHOC and OSCORE . . . 76
5.4.3 Default Server EDHOC . . . 77
5.4.4 Default Server EDHOC and OSCORE . . . 78
5.5 Automated Bug Detection . . . 79
5.5.1 Bug Patterns . . . 80

6 Conclusion **85**
6.1 Future Work . . . 85

Bibliography **87**

List of Figures

1.1 Αφαιρετικά Επίπεδα του CoAP 3
1.2 Αφαιρετικά Επίπεδα του CoAP με OSCORE 4
1.3 Ροή Μηνυμάτων EDHOC 5
1.4 Learner και SUL στη Θεωρία 7
1.5 Learner, Mapper και SUL στη Θεωρία 8
1.6 Αρχιτεκτονική του EDHOC-Fuzzer 10
1.7 Υλοποίηση του Abstract Symbol SUL 11
1.8 Υλοποίηση του EDHOC Mapper 13
1.9 Αρχιτεκτονική του ProtocolState-Fuzzer 16
1.10 Membership SUL Oracle 18
1.11 Equivalence SUL Oracle 19
1.12 SUL Wrappers 20
1.13 Μοντέλο του EDHOC-RS Default Client 22
1.14 Μοντέλο του EDHOC-RS Default Server 23
1.15 Μοντέλο του RISE Default Client 24
1.16 Μοντέλο του RISE Default Server 27
1.17 Μοντέλο των SIFIS-HOME Default Client Phases 1 και 2 29
1.18 Μοντέλο των SIFIS-HOME Default Client Phases 3 και 4 30
1.19 Μοντέλο του uOSCORE-uEDHOC Default EDHOC and OSCORE Client 31
1.20 Μοντέλο του uOSCORE-uEDHOC Default EDHOC and OSCORE Server 32
1.21 Bug Pattern: O Client Initiator Διατηρεί τη Σύνοδο Ζωντανή μετά την Αποστολή Μηνύματος EDHOC Error 34

3.1 Abstract Layers of CoAP 40
3.2 CoAP Message Format 40
3.3 Abstract Layers of CoAP with OSCORE 41
3.4 EDHOC Message Flow 42
3.5 Authentication Keys for Method Types 42
3.6 EDHOC exchange of CoAP Client Initiator and CoAP Server Responder 43
3.7 EDHOC exchange of CoAP Client Responder and CoAP Server Initiator 44
3.8 Learner and SUL Components in Theory 47
3.9 Learner, Mapper and SUL in Theory 48

4.1 EDHOC-Fuzzer Architecture 49
4.2 Abstract Symbol SUL Implementation 50

4.3 EDHOC Mapper Implementation 52
4.4 ProtocolState-Fuzzer Architecture 55
4.5 Membership SUL Oracle 57
4.6 Equivalence SUL Oracle 59
4.7 SUL Wrappers 60

5.1 EDHOC-RS Default Client Learned Model 62
5.2 EDHOC-RS Default Server Learned Model 63
5.3 RISE Default Client Learned Model 64
5.4 RISE Patched Client Learned Model 66
5.5 RISE Default Server Learned Model 69
5.6 RISE Patched Server Learned Model 70
5.7 SIFIS-HOME Default Client Phases 1 and 2 Learned Model 72
5.8 SIFIS-HOME Default Client Phases 3 and 4 Learned Model 73
5.9 SIFIS-HOME Default Server Phases 1, 2, 3 and 4 Learned Model . . 74
5.10 uOSCORE-uEDHOC Default EDHOC Client Learned Model 75
5.11 uOSCORE-uEDHOC Default EDHOC and OSCORE Client Learned Model 76
5.12 uOSCORE-uEDHOC Default EDHOC Server Learned Model 77
5.13 uOSCORE-uEDHOC Default EDHOC and OSCORE Server Learned Model 78
5.14 Bug Pattern: Client Initiator Keeps Session Alive After Sending EDHOC Error Message 80
5.15 Bug Pattern: Server Exchanges OSCORE Messages and Forced to Terminate 81
5.16 Bug Pattern: Server's OSCORE Resource Responds to Plaintext Messages 82
5.17 Bug Pattern: Initiator Without the Required Message_4 Sends OSCORE Message 83
5.18 Bug Pattern: Receiving EDHOC Error Message Harms Derived OSCORE Context 84

List of Tables

1.1 Αλφάβητο Συμβόλων Εισόδου 13
1.2 Αλφάβητο Συμβόλων Εξόδου 14

4.1 The Input Alphabet Symbols 52
4.2 The Output Alphabet Symbols 53

Κεφάλαιο 1

Εκτεταμένη Ελληνική Περίληψη

1.1 Εισαγωγή

Την τελευταία δεκαετία παρατηρήθηκε τεράστια αύξηση του αριθμού των συσκευών χαμηλής ισχύος που συνδέονται στο διαδίκτυο. Οι συσκευές αυτές αναφέρονται ως Διαδίκτυο των Πραγμάτων (IoT) και έχουν τη δυνατότητα να συλλέγουν και να ανταλλάσσουν δεδομένα σε πραγματικό χρόνο χρησιμοποιώντας κυρίως ενσωματωμένους αισθητήρες. Ορισμένα βασικά παραδείγματα περιλαμβάνουν φώτα, θερμοστάτες, ψυγεία, έξυπνα ρολόγια, ακόμη και αισθητήρες σε αυτοκίνητα. Το περιβάλλον στο οποίο λειτουργούν αυτές οι συσκευές είναι συνήθως περιορισμένο όσον αφορά την πρόσβαση στο διαδίκτυο και τους διαθέσιμους πόρους, γεγονός που περιορίζει την ασφάλεια που προσφέρουν αυτές οι συσκευές. Για την αντιμετώπιση αυτής της πρόκλησης, έχουν προταθεί πολλά πρωτόκολλα διαδικτύου και ασφάλειας, προκειμένου να καταστεί δυνατή η αξιόπιστη και ασφαλής ανταλλαγή δεδομένων μεταξύ αυτών των συσκευών.

Οι περιορισμοί που θέτει το περιβάλλον καθιστούν τη χρήση του πρωτοκόλλου μεταφοράς UDP κατάλληλη για την επικοινωνία των συσκευών IoT. Πάνω σε αυτό έχει χτιστεί το CoAP [RFC7252], ένα διαδικτυακό πρωτόκολλο μεταφοράς εξειδικευμένο για περιορισμένα περιβάλλοντα που καθιστά δυνατή την επικοινωνία. Η ασφάλεια της επικοινωνίας παρέχεται από το OSCORE [RFC8613], ένα πρωτόκολλο που χρησιμοποιείται για την κρυπτογράφηση και συνεπώς την προστασία του απορρήτου των ανταλλασσόμενων δεδομένων. Ωστόσο, το πρωτόκολλο OSCORE απαιτεί την καθιέρωση συμμετρικών μυστικών παραμέτρων από τους κόμβους που επικοινωνούν, γεγονός που υποδηλώνει την ανάγκη για ένα ασφαλές και αξιόπιστο πρωτόκολλο ανταλλαγής κλειδιών. Αυτό το κενό καλύπτει το EDHOC [SMP23], ένα πρόσφατα προτεινόμενο πρωτόκολλο που βρίσκεται επί του παρόντος σε κατάσταση προσχεδίου, το οποίο είναι ένα ελαφρύ πρωτόκολλο ανταλλαγής κλειδιών ιδανικό για περιορισμένα περιβάλλοντα.

Από τη μία πλευρά στη θεωρία, τα πρωτόκολλα προτείνονται σταδιακά, προκειμένου να αντιμετωπιστούν τα προβλήματα που προκύπτουν. Από την άλλη πλευρά, στην πράξη, οι υλοποιήσεις των πρωτοκόλλων είναι αυτές που συναντώνται καθημερινά. Για το λόγο αυτό είναι ζωτικής σημασίας οι υλοποιήσεις αυτές να είναι ακριβείς, εύρωστες και σύμφωνες με τις προδιαγραφές των πρωτοκόλλων. Αυτό το πρόβλημα προσπαθεί να αντιμετωπίσει η παρούσα διπλωματική,

αναπτύσσοντας εργαλεία και τεχνικές για τον αποτελεσματικό έλεγχο των υλοποιήσεων των πρωτοκόλλων EDHOC.

Η κύρια τεχνική που χρησιμοποιείται ονομάζεται protocol state fuzzing, η οποία ουσιαστικά χρησιμοποιεί ενεργητική εκμάθηση αυτομάτων, μια τεχνική που επιδιώκει να μάθει με καλή προσέγγιση το μοντέλο μηχανής καταστάσεων μιας υλοποίησης. Το μοντέλο που μαθαίνεται μπορεί είτε να επιθεωρηθεί για λογικά σφάλματα και ασυνέπειες με τις προδιαγραφές του προτεινόμενου πρωτοκόλλου είτε να χρησιμοποιηθεί για αυτοματοποιημένο έλεγχο.

Συνεισφορά

Η κύρια συνεισφορά είναι ο σχεδιασμός και η υλοποίηση του εργαλείου EDHOC-Fuzzer, το οποίο μπορεί να χρησιμοποιηθεί για την εκμάθηση μιας (ενδεχομένως προσεγγιστικής) μηχανής καταστάσεων μιας υλοποίησης EDHOC. Ο EDHOC-Fuzzer περιλαμβάνει ως εσωτερικό εξάρτημα ένα γενικό και αφηρημένο εργαλείο, το οποίο επίσης υλοποιήθηκε και ονομάζεται ProtocolState-Fuzzer και θα μπορούσε να εκτεθεί ως αυτόνομο εργαλείο στο μέλλον. Επιπλέον, στην παρούσα διπλωματική, μαθεύτηκαν τα μοντέλα διαφόρων υλοποιήσεων EDHOC και αναλύονται λεπτομερώς.

Περίγραμμα Ενοτήτων

Οι υπόλοιπες ενότητες είναι οργανωμένες ως εξής:

- Η Ενότητα 1.2 παρέχει το απαραίτητο θεωρητικό υπόβαθρο, όσον αφορά τα πρωτόκολλα και τις τεχνικές που χρησιμοποιήθηκαν. Επίσης, αναφέρει συνοπτικά και σχετική δουλειά άλλων πάνω στην τεχνική του protocol state fuzzing.
- Η Ενότητα 1.3 παρέχει λεπτομέρειες υλοποίησης του EDHOC-Fuzzer.
- Η Ενότητα 1.4 παρέχει τα πειράματα που έγιναν και τα μοντέλα που μαθεύτηκαν και αναλύθηκαν.
- Η Ενότητα 1.5 συνοψίζει την διπλωματική και παρέχει προτάσεις για μελλοντικές επεκτάσεις.

1.2 Θεωρητικό Υπόβαθρο

CBOR Μορφή Δεδομένων

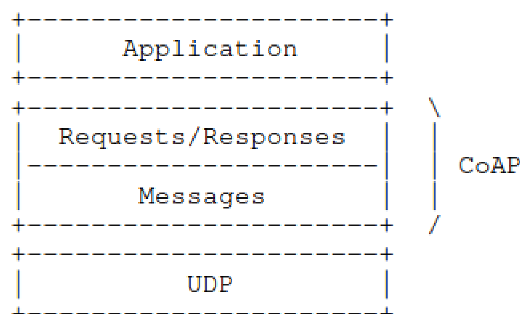
Η Concise Binary Object Representation (CBOR) [RFC8949] είναι μια μορφή κωδικοποίησης δεδομένων για τη δυαδική αναπαράσταση δομημένων δεδομένων (επίσης γνωστή ως μορφή δυαδικής σειριοποίησης). Βασίζεται και επεκτείνει το μοντέλο δεδομένων JSON. Έχει σχεδιαστεί για να επιτυγχάνει πολύ μικρό μέγεθος κώδικα αναλυτή και μικρό μέγεθος μηνύματος. Ένας από τους πρωταρχικούς στόχους του CBOR είναι να μπορεί να κωδικοποιεί με σαφήνεια τις πιο κοινές μορφές δεδομένων που χρησιμοποιούνται στα πρότυπα του διαδικτύου.

COSE Πρότυπο

Το πρότυπο CBOR Object Signing and Encryption (COSE) [RFC9052] καθορίζει τον τρόπο δημιουργίας και επεξεργασίας κωδικών κρυπτογράφησης, υπογραφών και αυθεντικοποίησης μηνυμάτων και τον τρόπο αναπαράστασης κρυπτογραφικών κλειδιών με χρήση του CBOR. Το COSE βασίζεται στο JOSE, αλλά ενσωματώνει τις πρόσθετες δυνατότητες που έχει το CBOR έναντι του JSON.

CoAP Πρωτόκολλο

Το Constrained Application Protocol (CoAP) [RFC7252] είναι ένα εξειδικευμένο διαδικτυακό πρωτόκολλο μεταφοράς για χρήση σε περιορισμένους κόμβους και περιορισμένα δίκτυα, όπως δίκτυα χαμηλής ισχύος. Είναι σχεδιασμένο για εφαρμογές μηχανής προς μηχανή. Το CoAP παρέχει ένα μοντέλο αλληλεπίδρασης αίτησης/απάντησης μεταξύ τελικών σημείων εφαρμογών, υποστηρίζει ενσωματωμένες υπηρεσίες και ανακάλυψη πόρων και περιλαμβάνει βασικές έννοιες του Παγκόσμιου Ιστού, όπως τα URI και τους τύπους μέσων του Διαδικτύου. Επιπλέον, μπορεί να διασυνδεθεί με το HTTP. Λογικά, το CoAP περιέχει δύο επίπεδα που φαίνονται στο Σχήμα 1.1, ένα επίπεδο ανταλλαγής μηνυμάτων CoAP που χρησιμοποιείται για τον χειρισμό του UDP και της ασύγχρονης φύσης των αλληλεπιδράσεων και το επίπεδο αίτησης/απάντησης για τις αλληλεπιδράσεις. Πρακτικά, αυτά τα δύο στρώματα είναι απλώς χαρακτηριστικά της επικεφαλίδας CoAP.

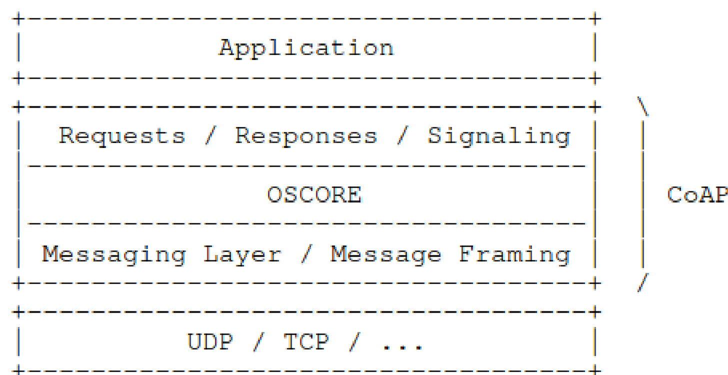


Σχήμα 1.1: Τα δύο αφαιρετικά λογικά επίπεδα του CoAP, το ένα για τα μηνύματα και το άλλο για τα αιτήματα και τις απαντήσεις.

OSCORE Πρωτόκολλο

Το πρωτόκολλο Object Security for Constrained RESTful Environments (OSCORE) [RFC8613] ορίζει μια μέθοδο για την προστασία του πρωτοκόλλου CoAP σε επίπεδο εφαρμογής, χρησιμοποιώντας το COSE. Έχει σχεδιαστεί για να παρέχει ασφάλεια από άκρο σε άκρο μεταξύ δύο τερματικών σημείων CoAP, ενώ παράλληλα εμποδίζει τους ενδιάμεσους να τροποποιήσουν ή να αποκτήσουν πρόσβαση σε οποιοδήποτε πεδίο μηνύματος, το οποίο δεν σχετίζεται με τις προβλεπόμενες λειτουργίες τους. Ουσιαστικά, το OSCORE μετατρέπει ένα μήνυμα CoAP σε προστατευμένο, διασφαλίζοντας όχι μόνο το ωφέλιμο φορτίο, αλλά και όλες τις πλήρως προστατευόμενες επιλογές CoAP, τους αρχικούς κωδικούς REST αίτησης και απάντησης, καθώς και τμήματα του URI των πόρων, στους οποίους στοχεύουν τα μηνύματα. Το αφηρημένο επίπεδο του OSCORE με το πρωτόκολλο CoAP παρουσιάζεται στο Σχήμα 1.2.

Για να χρησιμοποιήσουν το πρωτόκολλο OSCORE, οι συμμετέχοντες πρέπει να δημιουργήσουν ένα κοινό πλαίσιο ασφαλείας για την επεξεργασία των αντικειμένων COSE. Για να συμβεί αυτό, οι απαραίτητες πληροφορίες και τα υλικά για ένα κοινό κλειδί θα πρέπει να ανταλλαχθούν με ασφαλή και πιστοποιημένο τρόπο, τον οποίο παρέχει ένα κατάλληλο πρωτόκολλο ανταλλαγής κλειδιών.



Σχήμα 1.2: Το αφαιρετικό επίπεδο του OSCORE στο CoAP πρωτόκολλο.

EDHOC Πρωτόκολλο

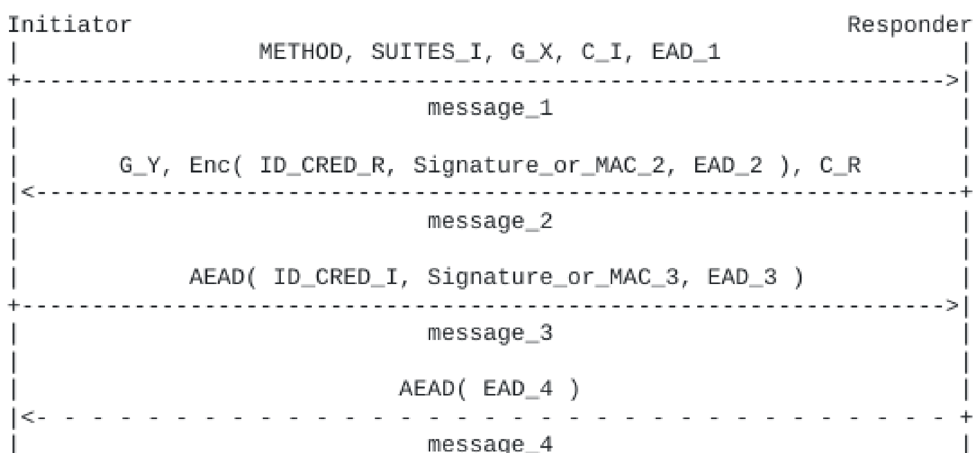
Το πρωτόκολλο Ephemeral Diffie-Hellman Over COSE (EDHOC) [SMP23] είναι ένα συμπαγές και ελαφρύ πρωτόκολλο ανταλλαγής κλειδιών με πιστοποίηση ταυτότητας, το οποίο παρέχει ιδιότητες ασφαλείας όπως προστασία ταυτότητας, διαπραγμάτευση κρυπτογράφησης και μυστικότητα προς τα εμπρός. Μία από τις κύριες περιπτώσεις χρήσης του είναι να αποτελεί τον τρόπο ανταλλαγής κλειδιών για το πρωτόκολλο OSCORE, δηλαδή να παρέχει πιστοποίηση ταυτότητας και δημιουργία κλειδιών συνόδου. Έχει σχεδιαστεί για περιβάλλοντα με υψηλούς περιορισμούς και στοχεύει στην υποδομή του IoT, όπου εμπλέκονται ενσωματωμένοι μικροελεγκτές, αισθητήρες και ενεργοποιητές. Επιπλέον, το πρωτόκολλο EDHOC χρησιμοποιεί τα ίδια πρωτόκολλα με το πρωτόκολλο OSCORE, τα οποία είναι το COSE για την κρυπτογραφία, το CBOR για την κωδικοποίηση και το CoAP για

τη μεταφορά. Τη στιγμή που γράφεται η παρούσα εργασία, το πρωτόκολλο αυτό είναι Internet draft και η τρέχουσα έκδοσή του είναι η 19.

Επειδή η παρούσα διπλωματική αφορά την ανάλυση των υλοποιήσεων του EDHOC, ακολουθεί μια σύντομη και υψηλού επιπέδου επισκόπηση κάποιων στοιχείων του πρωτοκόλλου.

Ρόλοι. Υπάρχουν δύο πιθανοί ρόλοι που μπορεί να έχει ένας κόμβος: Initiator ή Responder. Ο Initiator είναι αυτός που ξεκινά το πρωτόκολλο ανταλλαγής κλειδιών με τον Responder. Αυτοί οι ρόλοι δεν συνδέονται με το χρησιμοποιούμενο διαδικτυακό πρωτόκολλο μεταφοράς. Για παράδειγμα, όταν χρησιμοποιείται το CoAP, ένας CoAP client μπορεί να είναι είτε Initiator είτε Responder για το πρωτόκολλο EDHOC, το ίδιο και ένας CoAP server. Αυτό σημαίνει ότι για το πρωτόκολλο CoAP υπάρχουν τέσσερις πιθανοί κόμβοι: CoAP client Initiator, CoAP client Responder, CoAP server Initiator και CoAP server Responder.

Μηνύματα. Το πρωτόκολλο αποτελείται συνολικά από πέντε μηνύματα. Σε μια επιτυχή ανταλλαγή κλειδιών τα υποχρεωτικά είναι τα *message_1*, *message_2* και *message_3*. Επίσης, υπάρχει το *message_4*, το οποίο είναι προαιρετικό και το *error_message*. Ο Initiator χρησιμοποιεί το *message_1* και το *message_3* και ο Responder χρησιμοποιεί το *message_2* και προαιρετικά το *message_4*. Και οι δύο ρόλοι μπορούν να χρησιμοποιήσουν το *error_message*. Ένα μήνυμα EDHOC κωδικοποιείται ως ακολουθία στοιχείων CBOR. Μια ροή μηνυμάτων παρουσιάζεται στο Σχήμα 1.3.



Σχήμα 1.3: Ροή Μηνυμάτων EDHOC με το προαιρετικό *message_4*.

Ντετερμινιστικό Πεπερασμένο Αυτόματο

Ένα Ντετερμινιστικό Πεπερασμένο Αυτόματο (DFA) είναι μια μηχανή πεπερασμένων καταστάσεων, η οποία αποδέχεται ή απορρίπτει μια δεδομένη συμβολοσειρά. Η μηχανή ξεκινά σε μια αρχική κατάσταση και δεδομένου κάθε συμβόλου ή χαρακτήρα μιας συμβολοσειράς εισόδου, η μηχανή μεταβαίνει από κατάσταση σε κατάσταση σύμφωνα με τη συνάρτηση μετάβασης. Μετά τη μετάβαση που

προκαλείται από το τελευταίο σύμβολο της συμβολοσειράς εισόδου, εάν η μηχανή βρίσκεται σε μία από τις καταστάσεις αποδοχής, τότε η συμβολοσειρά εισόδου γίνεται αποδεκτή, διαφορετικά απορρίπτεται.

Ένα Ντετερμινιστικό Πεπερασμένο Αυτόματο είναι μια πεντάδα $(\Sigma, Q, q_0, \Delta, F)$, όπου:

- Σ είναι το πεπερασμένο αλφάβητο συμβόλων εισόδου
- Q είναι το πεπερασμένο σύνολο καταστάσεων
- $q_0 \in Q$ είναι η αρχική κατάσταση
- $\Delta : Q \times \Sigma \rightarrow Q$ είναι μια συνάρτηση μετάβασης, η οποία δίνει μια επόμενη κατάσταση $\Delta(q, s)$ για κάθε κατάσταση $q \in Q$ και σύμβολο $s \in \Sigma$
- $F \subseteq Q$ είναι το πεπερασμένο σύνολο των καταστάσεων αποδοχής

Μηχανή Mealy

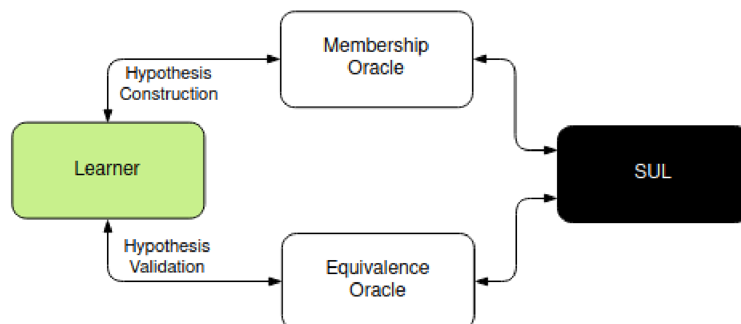
Μια μηχανή Mealy είναι μια μηχανή πεπερασμένων καταστάσεων, της οποίας οι τιμές εξόδου εξαρτώνται από την τρέχουσα κατάσταση και τις εισόδους. Έχουν πεπερασμένο αλφάβητο συμβόλων εισόδου και εξόδου και για κάθε κατάσταση και είσοδο είναι δυνατή το πολύ μία μετάβαση. Από μια αρχική κατάσταση, επεξεργάζονται ένα σύμβολο εισόδου προκαλώντας τη δημιουργία μιας ακολουθίας συμβόλων εξόδου που κάνει τη μηχανή να μεταβεί σε μια νέα κατάσταση. Από αυτή τη νέα κατάσταση μπορεί να επεξεργαστεί το επόμενο σύμβολο εισόδου.

Μια μηχανή Mealy είναι μια εξάδα $(I, O, Q, q_0, \delta, \lambda)$, όπου:

- I είναι το πεπερασμένο αλφάβητο συμβόλων εισόδου
- O είναι το πεπερασμένο αλφάβητο συμβόλων εξόδου
- Q είναι το πεπερασμένο σύνολο καταστάσεων
- $q_0 \in Q$ είναι η αρχική κατάσταση
- $\delta : Q \times I \rightarrow Q$ είναι μια συνάρτηση μετάβασης, η οποία δίνει μια επόμενη κατάσταση $\delta(q, i)$ για κάθε κατάσταση $q \in Q$ και σύμβολο εισόδου $i \in I$
- $\lambda : Q \times I \rightarrow O^*$ είναι μια συνάρτηση εξόδου, η οποία δίνει μια (πιθανώς κενή) ακολουθία συμβόλων εξόδου $\lambda(q, i)$, για κάθε κατάσταση $q \in Q$ και σύμβολο εισόδου $i \in I$

Ενεργητική Εκμάθηση Αυτομάτων

Η ενεργητική εκμάθηση αυτομάτων (AAL) αποτελεί μια ευρεία υποκατηγορία της εκμάθησης μοντέλων [Vaa17]. Πρόκειται για μια αυτοματοποιημένη τεχνική μαύρου κουτιού, η οποία κατασκευάζει (κατά προσέγγιση) μοντέλα μηχανών καταστάσεων συστημάτων λογισμικού και υλικού παρέχοντάς τους εισόδους (ερωτήματα) και παρατηρώντας τις εξόδους τους (απαντήσεις). Συνήθως, οι αλγόριθμοι ενεργητικής εκμάθησης αυτομάτων εξάγουν μια ντετερμινιστική μηχανή Mealy, η οποία αναπαριστά το μοντέλο του υπό εκμάθηση συστήματος (SUL).



Σχήμα 1.4: Οι κύριες συνιστώσες ενός AAL αλγορίθμου. Το Membership Oracle απαντάει membership ερωτήματα αναφορικά με το SUL, από τα οποία ο Learner κατασκευάζει μια υπόθεση. Το Equivalence Oracle επαληθεύει την υπόθεση βάσει του SUL.

Η διαδικασία εκμάθησης είναι μια συνεχής επανάληψη δύο φάσεων: κατασκευή υποθέσεων και επικύρωση υποθέσεων. Οι συνιστώσες της παρουσιάζονται στο Σχήμα 1.4.

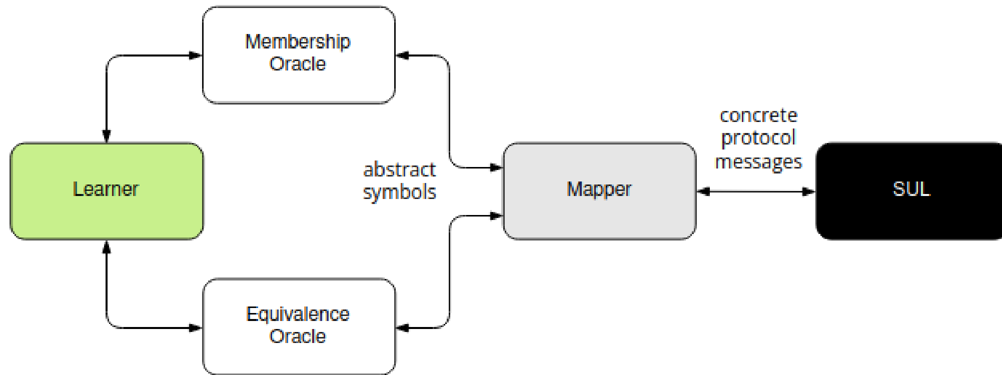
Κατασκευή υποθέσεων. Κατά τη διάρκεια αυτής της φάσης, επιλέγονται ακολουθίες συμβόλων εισόδου και αποστέλλονται στο SUL προκειμένου να παρατηρηθούν οι ακολουθίες συμβόλων εξόδου που θα απαντήσει. Οι επόμενες ακολουθίες εισόδου επιλέγονται με βάση τις παρατηρούμενες αποκρίσεις. Σε αυτή τη φάση τα ερωτήματα εκτελούνται σε ένα Membership Oracle και ονομάζονται membership ερωτήματα. Ο αλγόριθμος εκμάθησης κατασκευάζει μια υπόθεση, όταν πληρούνται ορισμένα κριτήρια σύγκλισης. Η υπόθεση, σε αυτή τη διπλωματική, είναι μια ελάχιστη ντετερμινιστική μηχανή Mealy που συνάδει με τις μέχρι τώρα καταγεγραμμένες παρατηρήσεις. Με άλλα λόγια, για όλες τις ακολουθίες εισόδου, οι οποίες έχουν σταλεί στο SUL, η υπόθεση παράγει τις ίδιες εξόδους με αυτές που παρατηρήθηκαν από το SUL. Όσον αφορά μια ακολουθία εισόδου που δεν έχει σταλεί στο SUL, η υπόθεση προεκτείνει την έξοδό της από τις καταγεγραμμένες παρατηρήσεις, ουσιαστικά υποθέτοντας τες. Προκειμένου να επικυρωθεί ότι αυτές οι υποθέσεις συμμορφώνονται με το SUL, η εκμάθηση μεταφέρεται στη φάση επικύρωσης.

Επικύρωση υποθέσεων. Κατά τη διάρκεια αυτής της φάσης, εκτελείται έλεγχος συμμόρφωσης στο SUL, έτσι ώστε να επικυρωθεί ότι η υπόθεση είναι σύμφωνη με τη συμπεριφορά του SUL. Σε αυτή τη φάση η προς επικύρωση υπόθεση αποστέλλεται σε ένα Equivalence Oracle, το οποίο εκτελεί ερωτήματα, τα λεγόμενα equivalence ερωτήματα στο SUL. Είναι πιθανό αυτός ο έλεγχος συμμόρφωσης να βρει ένα αντιπαράδειγμα, δηλαδή μια ακολουθία εισόδου στην οποία η έξοδος της υπόθεσης και του SUL δεν ταιριάζουν. Σε αυτή την περίπτωση, ο αλγόριθμος εκμάθησης επιστρέφει στη φάση κατασκευής, προσπαθώντας να βελτιώσει την υπόθεση λαμβάνοντας υπόψη το αντιπαράδειγμα που ανακαλύφθηκε. Εάν δεν βρεθεί αντιπαράδειγμα, η εκμάθηση τερματίζεται και επιστρέφει την τρέχουσα υπόθεση ως το μοντέλο του συστήματος. Επειδή ο έλεγχος συμμόρφωσης δεν είναι πλήρης, το μοντέλο που μαθαίνεται περιγράφει κατά προσέγγιση το SUL. Προφανώς, όσο περισσότερα equivalence ερωτήματα εκτελούνται σε αυτή τη φάση,

τόσο μεγαλύτερες είναι οι πιθανότητες εύρεσης αντιπαραδείγματος.

Εάν η αλληλουχία αυτών των δύο φάσεων δεν τερματίσει, τότε είναι πιθανό η συμπεριφορά του SUL να μην μπορεί να αποτυπωθεί από μια μηχανή Mealy της οποίας το μέγεθος και η πολυπλοκότητα είναι προσιτά στον τρέχοντα αλγόριθμο εκμάθησης.

Protocol State Fuzzing



Σχήμα 1.5: Τα τρία εξαρτήματα που χρησιμοποιούνται για το protocol state fuzzing. Προσέξτε την θέση του Mapper μετά τα Oracles και τον χρωματικό κώδικα: πράσινο για τον Learner, γκρι για τον Mapper και μαύρο για το SUL.

Ο όρος Protocol State Fuzzing, που επινοήθηκε από τους de Ruiter και Poll [RP15], είναι μια τεχνική που χρησιμοποιεί την ενεργητική εκμάθηση αυτομάτων, προκειμένου να συμπεράνει τις μηχανές κατάστασης υλοποιήσεων πρωτοκόλλων. Τα μοντέλα που μαθαίνονται στη συνέχεια αναλύονται, είτε χειροκίνητα είτε με τη χρήση μιας τεχνικής ελέγχου μοντέλων, για την αναζήτηση λογικών ατελειών που μπορούν να αποκαλυφθούν από μη τυποποιημένες ή απροσδόκητες ακολουθίες μηνυμάτων. Αντί να γίνεται fuzzing μεμονωμένων μηνυμάτων, γίνεται fuzzing ακολουθιών μηνυμάτων.

Εξαρτήματα εκμάθησης. Με βάση τον αλγόριθμο ενεργητικής εκμάθησης αυτομάτων μέχρι στιγμής, τα απαιτούμενα συστατικά για την εγκατάσταση εκμάθησης είναι δύο: ένας *Learner* και ένα *SUL*. Ο *Learner* είναι υπεύθυνος για την υποβολή ερωτημάτων στο *SUL* και τη λήψη των απαντήσεών του, εκτελώντας με αυτόν τον τρόπο αποτελεσματικά τον αλγόριθμο εκμάθησης. Ωστόσο, ο *Learner* δεν γνωρίζει τίποτα για το εκάστοτε συγκεκριμένο πρωτόκολλο, δηλαδή το αλφάβητο εισόδου περιέχει αφηρημένα σύμβολα εισόδου, τα οποία ο *Learner* δεν γνωρίζει πώς να μετατρέψει σε συγκεκριμένα μηνύματα πρωτοκόλλου. Αυτό το κενό καλύπτεται από ένα ενδιάμεσο εξάρτημα, τον *Mapper* στο Σχήμα 1.5, ο οποίος μετατρέπει τα αφηρημένα σύμβολα εισόδου από το πεπερασμένο αλφάβητο εισόδου σε συγκεκριμένα μηνύματα πρωτοκόλλου που αποστέλλονται στο SUL, συμπληρώνοντας τις απαραίτητες λεπτομέρειες. Αντίστροφα, ο *Mapper* αντιστοιχίζει τα συγκεκριμένα μηνύματα από το SUL σε αφηρημένα σύμβολα του αλφαβήτου εξόδου που είναι γνωστά στον *Learner* αφαιρώντας τις περιττές λεπτομέρειες. Επίσης, διατηρεί

την κατάσταση του πρωτοκόλλου, η οποία απαιτείται για τη συμπλήρωση των παραμέτρων των μηνυμάτων.

Σχετική Βιβλιογραφία

Η τεχνική του protocol state fuzzing και γενικότερα της ενεργητικής εκμάθησης μοντέλων έχει χρησιμοποιηθεί για να αναλυθούν πολλές υλοποιήσεις πρωτοκόλλων. Μια πιο αναλυτική λίστα σχετικής βιβλιογραφίας μπορεί να βρεθεί στο wiki που περιγράφεται στο [Nei+19] και στο [Wen20]. Ακολουθεί μια σχετικά μικρή λίστα σχετικής βιβλιογραφίας.

- DTLS [FB+20; Fit+22]
- IPSEC [Vel17]
- MQTT [TAB17]
- OpenVPN [DPR18]
- QUIC [Fer+21; RAR19]
- SSH [FB+17]
- TCP [Fer+21; FBJV16]
- TLS [Rui16; RP15]

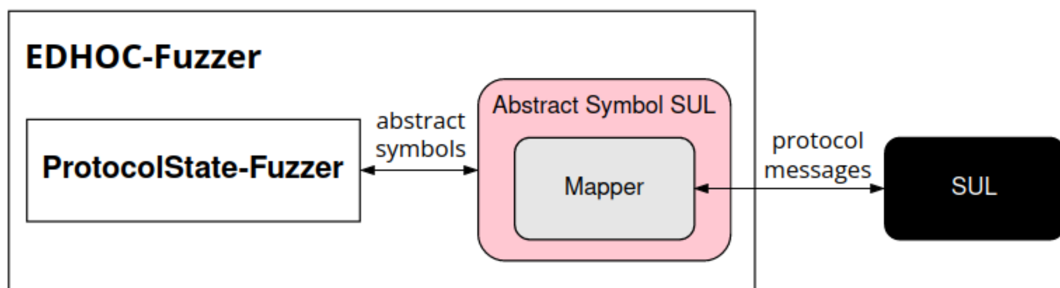
1.3 Υλοποίηση

Μια επισκόπηση της αρχιτεκτονικής του EDHOC-Fuzzer φαίνεται στο Σχήμα 1.6. Αυτό χρησιμοποιεί ως framework ένα άλλο υλοποιημένο εργαλείο που ονομάζεται ProtocolState-Fuzzer και είναι γενικό προσφέροντας την απαραίτητη εγκατάσταση για την τεχνική του ProtocolState-Fuzzing. Για αυτό το λόγο ο EDHOC-Fuzzer υλοποιεί μόνο τα απαραίτητα κομμάτια που λείπουν, τα οποία είναι το Abstract Symbol SUL και ο Mapper.

Η υλοποίηση του ProtocolState-Fuzzer είναι εμπνευσμένη και μοιράζεται τον ίδιο βασικό κώδικα με τον DTLS-Fuzzer [FB+20; Fit+22][1], έναν protocol state fuzzer γραμμένο σε JAVA για το πρωτόκολλο DTLS. Για την ενεργητική εκμάθηση μοντέλων χρησιμοποιείται το LearnLib framework [IHS15][2].

Η χρήση του όρου “SUL” οπουδήποτε αλλού εκτός από το σύστημα υπό εκμάθηση είναι απλώς μια ονομαστική σύμβαση που προέρχεται από το Learnlib framework και δεν σχετίζεται με το σύστημα υπό εκμάθηση που είναι χρωματισμένο με μαύρο χρώμα στο Σχήμα 1.6.

Η ενότητα που ακολουθεί περιγράφει τα κομμάτια που παρέχει ο EDHOC-Fuzzer (χρωματισμένα με ροζ και γκρι στο Σχήμα 1.6) και η επόμενη ενότητα περιγράφει την εσωτερική σύνθεση του ProtocolState-Fuzzer.



Σχήμα 1.6: Η αρχιτεκτονική του EDHOC-Fuzzer αποτελείται από τον ProtocolState-Fuzzer, το Abstract Symbol SUL και τον Mapper. Ο ProtocolState-Fuzzer συνδέεται με το Abstract Symbol SUL, το οποίο χρησιμοποιεί τον Mapper για να επικοινωνήσει με την υπό εκμάθηση υλοποίηση EDHOC που ονομάζεται SUL και αντιμετωπίζεται ως μαύρο κουτί. Ο Mapper και το SUL επικοινωνούν μέσω του δικτύου και συνήθως στο τοπικό δίκτυο.

EDHOC-Fuzzer

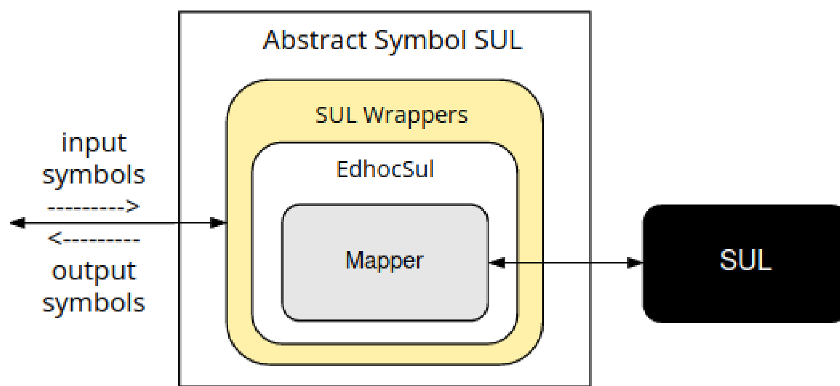
Τα μόνα κομμάτια που χρειάστηκε να υλοποιηθούν είναι ο Mapper και το Abstract Symbol SUL. Ο Mapper πρέπει να γνωρίζει πώς να μετατρέπει αφηρημένα σύμβολα σε μηνύματα EDHOC, OSCORE, CoAP και αντίστροφα. Το Abstract Symbol SUL είναι το απαραίτητο στοιχείο, προκειμένου να χρησιμοποιηθεί ο ProtocolState-Fuzzer και χρησιμοποιεί τον Mapper για να επικοινωνήσει με το SUL, το οποίο

[1] Ο DTLS-Fuzzer είναι διαθέσιμος στο https://github.com/assist-project/dtls-fuzzer
[2] Το LearnLib είναι διαθέσιμο στο https://learnlib.de

αντιμετωπίζεται ως μαύρο κουτί. Επιπλέον, αρκετές χρήσιμες κλάσεις που παρέχει ο ProtocolState-Fuzzer χρησιμοποιήθηκαν για την υλοποίηση και των δύο κομματιών.

Abstract Symbol SUL

Το Abstract Symbol SUL είναι το βασικό δομικό στοιχείο που είναι υπεύθυνο για τη λήψη ενός αφηρημένου συμβόλου εισόδου και την επιστροφή του αντίστοιχου αφηρημένου συμβόλου εξόδου. Τα αφηρημένα σύμβολα είναι τα σύμβολα εισόδου και εξόδου των αντίστοιχων αλφαβήτων. Αυτό το δομικό στοιχείο είναι απαραίτητο για τον EDHOC-Fuzzer, διότι συνδέει τον ProtocolState-Fuzzer με τον Mapper και τέλος με το SUL. Η υλοποίηση του Abstract Symbol SUL παρουσιάζεται στο Σχήμα 1.7.



Σχήμα 1.7: Η υλοποίηση του Abstract Symbol SUL. Στο εσωτερικό είναι η κλάση EdhocSul, η οποία χρησιμοποιεί τον Mapper και είναι τυλιγμένη από ένα κίτρινο στρώμα βοηθητικών κλάσεων που ονομάζονται SUL Wrappers.

SUL Wrappers. Οι SUL Wrappers, χρωματισμένοι με κίτρινο στο Σχήμα 1.7, είναι ένα σύνολο βοηθητικών κλάσεων που προσφέρονται στον ProtocolState-Fuzzer και παρέχουν πρόσθετη λειτουργικότητα στην βασική κλάση, EdhocSul. Ουσιαστικά αυτό το επίπεδο αποτελείται από κλάσεις που τυλίγονται η μία μέσα στην άλλη, οι οποίες αναλύονται στην επόμενη ενότητα. Το σύμβολο εισόδου λαμβάνεται από την πιο εξωτερική κλάση και διαδίδεται στην πιο εσωτερική (ή στον πυρήνα), που είναι το EdhocSul. Όταν η κλάση πυρήνας επιστρέφει ένα σύμβολο εξόδου, αυτό διαδίδεται από την πιο εσωτερική στην πιο εξωτερική κλάση.

EdhocSul. Αυτή η κλάση είναι η κλάση πυρήνας του Abstract Symbol SUL και υλοποιεί τη λογική σχετικά με ένα ερώτημα (ή ένα test), το οποίο είναι μια ακολουθία συμβόλων εισόδου. Από τη σκοπιά του EdhocSul υπάρχουν τρεις φάσεις που συνδέονται με την εκτέλεση ενός ερωτήματος: η φάση πριν, κατά τη διάρκεια και η φάση μετά. Στη φάση πριν από την εκτέλεση, το EdhocSul ρυθμίζει τον Mapper ανάλογα με το αν το SUL αφορά υλοποίηση server ή client και αρχικοποιεί την κατάστασή του. Για παράδειγμα, όταν το SUL αφορά υλοποίηση client, τότε ο Mapper περιμένει το αρχικό μήνυμα EDHOC του client και στη συνέχεια αρχίζει η εκμάθηση. Στη φάση κατά την εκτέλεση του ερωτήματος, όταν το EdhocSul

λαμβάνει όλα τα σύμβολα εισόδου του ερωτήματος, χρησιμοποιεί τον Mapper για να επικοινωνήσει με το SUL και να λάβει το αντίστοιχο σύμβολο εξόδου. Η τελευταία φάση είναι μετά την εκτέλεση του ερωτήματος, κατά την οποία το EdhocSul έχει την ευκαιρία να απελευθερώσει τυχόν δεσμευμένους πόρους. Η επανάληψη αυτών των τριών φάσεων καθοδηγείται από τον ProtocolState-Fuzzer, ο οποίος ενσωματώνει τη λογική του αλγορίθμου εκμάθησης.

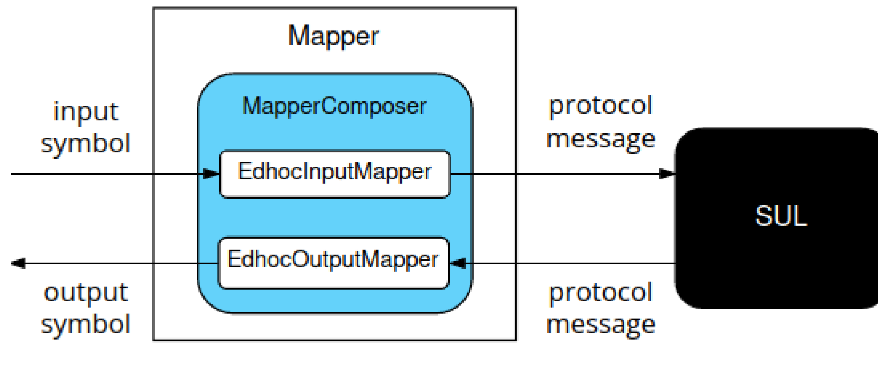
Mapper

Βιβλιοθήκη. Ο Mapper πρέπει να είναι εξειδικευμένος στο EDHOC πρωτόκολλο και για αυτό αποφασίστηκε πως η βιβλιοθήκη που θα παρέχει την εξειδικευμένη για το πρωτόκολλο λειτουργικότητα θα προέρχεται από μια αληθινή υλοποίηση του πρωτοκόλλου[3]. Μετά τις απαραίτητες τροποποιήσεις, η βιβλιοθήκη αυτή παρέχει όλη τη απαιτούμενη λειτουργικότητα του Mapper. Συγκεκριμένα, ο Mapper χειρίζεται την κατάσταση του πρωτοκόλλυ και έχει τη δυνατότητα να διαβάζει και να γράφει CoAP, EDHOC και OSCORE μηνύματα.

Υλοποίηση. Η υλοποίηση του Mapper, η οποία παρουσιάζεται στο Σχήμα 1.8, βασίζεται σε μια κλάση που προσφέρεται στον ProtocolState-Fuzzer και ονομάζεται MapperComposer, η οποία απαιτεί δύο διαφορετικούς sub-Mappers: τον EdhocInputMapper και τον EdhocOutputMapper. Ο EdhocInputMapper μετατρέπει ένα αφηρημένο σύμβολο εισόδου σε ένα συγκεκριμένο μήνυμα πρωτοκόλλου και το αποστέλλει στο SUL, ενώ ο EdhocOutputMapper λαμβάνει το συγκεκριμένο μήνυμα πρωτοκόλλου του SUL και το μετατρέπει σε ένα αφηρημένο σύμβολο εξόδου. Ωστόσο, και οι δύο έχουν εγγενείς δυσκολίες υλοποίησης. Από τη μία πλευρά, ο EdhocInputMapper πρέπει να ενεργεί σαν client για να επικοινωνεί με μια server υλοποίηση και σαν server για να επικοινωνεί με μια client υλοποίηση. Από την άλλη πλευρά, ο EdhocOutputMapper θα πρέπει να προσπαθεί να ταυτοποιήσει τα λαμβανόμενα μηνύματα με όσο το δυνατόν μεγαλύτερη ακρίβεια και να μην γενικεύει. Για παράδειγμα, στην περίπτωση του CoAP, όλα τα λαμβανόμενα μηνύματα είναι μηνύματα CoAP και μόνο ένα υποσύνολο αυτών είναι μηνύματα EDHOC. Αυτό σημαίνει ότι ο EdhocOutputMapper δεν πρέπει να ταυτοποιεί όλα τα μηνύματα ως CoAP, εκτός εάν δεν είναι δυνατή άλλη ταυτοποίηση. Είναι σημαντικό και οι δύο sub-Mappers να συμπεριφέρονται όσο το δυνατόν πιο σωστά, προκειμένου τα αποτελέσματα να είναι ακριβή.

Επιρροή στην εκμάθηση. Από τη σκοπιά του Learner, ο Mapper θα πρέπει να συμπεριφέρεται με διαφάνεια και να αντανακλά τα πραγματικά μηνύματα που μεταδίδονται, αλλά υπάρχουν περιπτώσεις, όπου η συμπεριφορά του Mapper αναπόφευκτα επηρεάζει το μοντέλο που μαθαίνεται. Αυτές σχετίζονται κυρίως με την κατάσταση του πρωτοκόλλου του Mapper. Συγκεκριμένα, υπάρχουν περιπτώσεις όπου ο Learner ζητά να σταλεί ένα συγκεκριμένο σύμβολο εισόδου, αλλά ο Mapper δεν είναι σε θέση να το στείλει, λόγω της κατάστασής του. Τα μηνύματα OSCORE είναι ένα χαρακτηριστικό παράδειγμα, όταν ο Mapper λειτουργεί ως server. Με άλλα λόγια, ο ιδανικός Mapper θα πρέπει να είναι όσο το δυνατόν πιο διαφανής,

[3]Η υλοποίηση που χρησιμοποιήθηκε ως βιβλιοθήκη είναι το cf-edhoc module στο https://github.com/rikard-sics/californium/tree/edhoc



Σχήμα 1.8: Ο Mapper είναι υλοποιημένος ως MapperComposer, ο οποίος παρέχεται στον ProtocolState-Fuzzer, αποτελούμενος από τον EdhocInputMapper για τα σύμβολα εισόδου και τον EdhocOutputMapper για τα σύμβολα εξόδου.

δηλαδή να στέλνει το μήνυμα που του δίνεται εντολή να στείλει και να λαμβάνει την αντίστοιχη έξοδο, χωρίς να παρεμβαίνει στη διαδικασία αυτή.

Αλφάβητα

Υπάρχουν δύο τύποι αλφαβήτων που χρησιμοποιούνται. Το αλφάβητο εισόδου μπορεί να παρέχεται από τον χρήστη, ενώ το αλφάβητο εξόδου είναι εσωτερικό του εργαλείου και εξαρτάται από την ακρίβεια του Mapper. Κάθε σύμβολο εισόδου ή εξόδου έχει μια πλήρη μορφή και μια σύντομη μορφή για το όνομά του. Η σύντομη μορφή χρησιμοποιείται για σκοπούς οπτικοποίησης στα μοντέλα που προκύπτουν.

Αλφάβητο Εισόδου. Όλα τα υποστηριζόμενα σύμβολα εισόδου βρίσκονται στον Πίνακα 1.1. Το προκαθορισμένο αλφάβητο εισόδου περιέχει όλα τα σύμβολα, αλλά ο χρήστης έχει την ευελιξία να παραλείψει κάποια εάν το επιθυμεί.

Πίνακας 1.1: Αλφάβητο Συμβόλων Εισόδου

Πλήρες Όνομα	**Συνοπτικό Όνομα**
EDHOC_MESSAGE_1	M1
EDHOC_MESSAGE_2	M2
EDHOC_MESSAGE_3	M3
EDHOC_MESSAGE_4	M4
EDHOC_ERROR_MESSAGE	ERR_E
EDHOC_MESSAGE_3_OSCORE_APP	$M3APP_O$
OSCORE_APP_MESSAGE	APP_O
COAP_APP_MESSAGE	APP_C
COAP_EMPTY_MESSAGE	EMP_C

Ακολουθεί μια σύντομη επεξήγηση των συμβόλων εισόδου.

- EDHOC_MESSAGE_{1, 2, 3, 4} και EDHOC_ERROR_MESSAGE, είναι τα βασικά μηνύματα EDHOC.

- EDHOC_MESSAGE_3_OSCORE_APP, είναι η συνένωση των δύο μηνυμάτων μετά την παραγωγή ενός νέου πλαισίου OSCORE. Μόνο ένας EDHOC Initiator μπορεί να το στείλει.
- OSCORE_APP_MESSAGE, μπορεί να αποσταλεί αφού ολοκληρωθεί επιτυχώς η ανταλλαγή EDHOC και προκύψει ένα πλαίσιο OSCORE. Αυτό το μήνυμα είναι ουσιαστικά το κρυπτογραφημένο μήνυμα εφαρμογής μεταξύ των δύο κόμβων.
- COAP_APP_MESSAGE, χρησιμοποιείται για να ελεγχθεί εάν ο πόρος του άλλου κόμβου αναμένει πράγματι ένα κρυπτογραφημένο μήνυμα και δεν απαντά σε μη κρυπτογραφημένα. Είναι η μη κρυπτογραφημένη εκδοχή του OSCORE_APP_MESSAGE.
- COAP_EMPTY_MESSAGE, είναι ένα μήνυμα CoAP χωρίς ωφέλιμο φορτίο.

Αλφάβητο Εξόδου. Τα σύμβολα εξόδου που αναγνωρίζει ο Mapper φαίνονται στον Πίνακα 1.2.

Πίνακας 1.2: Αλφάβητο Συμβόλων Εξόδου

Πλήρες Όνομα	**Συνοπτικό Όνομα**
EDHOC_MESSAGE_1	M1
EDHOC_MESSAGE_2	M2
EDHOC_MESSAGE_3	M3
EDHOC_MESSAGE_4	M4
EDHOC_ERROR_MESSAGE	ERR_E
EDHOC_MESSAGE_3_OSCORE_APP	M3APP_O
OSCORE_APP_MESSAGE	APP_O
COAP_APP_MESSAGE	APP_C
COAP_MESSAGE	MSG_C
COAP_ERROR_MESSAGE	ERR_C
COAP_EMPTY_MESSAGE	EMP_C
UNSUPPORTED_MESSAGE	NA
UNSUCCESSFUL_MESSAGE	$\times$
UNKNOWN_MESSAGE	?
SOCKET_CLOSED	$\perp$
TIMEOUT	$\varnothing$

Ακολουθεί μια σύντομη επεξήγηση για τα επιπρόσθετα σύμβολα εξόδου που δεν εμφανίζονται στα σύμβολα εισόδου.

- COAP_MESSAGE, είναι το πιο γενικό μήνυμα εξόδου των μηνυμάτων CoAP.
- COAP_ERROR_MESSAGE, είναι ένα μήνυμα σφάλματος που μεταφέρεται στο CoAP. Δεν πρέπει να συγχέεται με το EDHOC_ERROR_MESSAGE.
- UNSUPPORTED_MESSAGE, είναι ένα ειδικό μήνυμα εξόδου που δείχνει ότι ο Mapper δεν μπορούσε να στείλει το ζητούμενο μήνυμα εισόδου στην κατάσταση στην οποία βρισκόταν. Στις περισσότερες περιπτώσεις εμφανίζεται στο EDHOC_MESSAGE_3_OSCORE_APP, επειδή ο Mapper ως

EDHOC Responder δεν μπορεί να στείλει τέτοια μηνύματα και στο OSCORE_APP_MESSAGE, όταν ο Mapper, ο οποίος δρα ως server, λαμβάνει ένα αίτημα που δεν είναι προστατευμένο με OSCORE, αλλά ο Learner ζητά την αποστολή ενός OSCORE_APP_MESSAGE. Αυτό το μήνυμα εξόδου αποτελεί μέρος του τελικού μοντέλου, αλλά δεν σχετίζεται με τη συμπεριφορά του SUL.

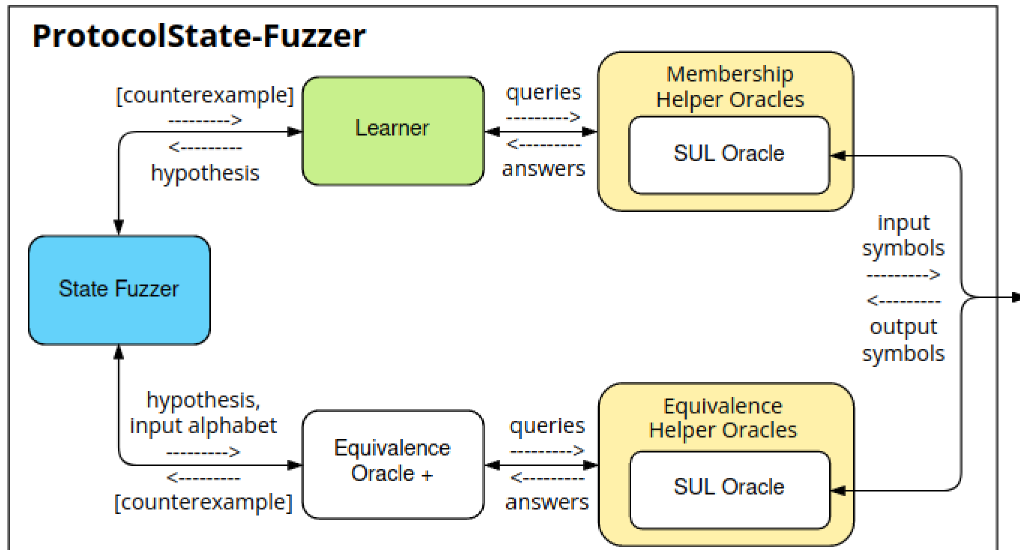
- UNSUCCESSFUL_MESSAGE, είναι ένα ειδικό μήνυμα εξόδου που εμφανίζεται συνήθως όταν το SUL είναι client και ο Mapper δρα ως server. Το μήνυμα υποδεικνύει ότι το πραγματικό μήνυμα εξόδου του SUL ελήφθη, αλλά η επεξεργασία του συνάντησε κάποιο σφάλμα. Για παράδειγμα, εμφανίζεται εάν η υλοποίηση του client στείλει ένα μήνυμα που στοχεύει σε έναν πόρο τον οποίο ο Mapper δεν υποστηρίζει.
- UNKNOWN_MESSAGE, είναι το πιο γενικό μήνυμα εξόδου από όλα και χρησιμοποιείται όταν το μήνυμα που λαμβάνεται δεν μπορεί να αναγνωριστεί με άλλο τρόπο.
- SOCKET_CLOSED, είναι ένα ειδικό μήνυμα εξόδου που δείχνει ότι το SUL έχει σταματήσει να ακούει τη θύρα που άκουγε. Συνήθως αυτό σημαίνει ότι η διεργασία SUL έχει τερματιστεί.
- TIMEOUT, είναι ένα ειδικό μήνυμα εξόδου που δείχνει ότι ο Mapper περίμενε για μια συγκεκριμένη χρονική διάρκεια και δεν έλαβε κανένα μήνυμα εξόδου από το SUL.

ProtocolState-Fuzzer

Ο ProtocolState-Fuzzer προέρχεται από τον επανασχεδιασμό του DTLS-Fuzzer, είναι αρθρωτός και επεκτάσιμος, διατηρώντας παράλληλα την τυπική λειτουργικότητα του DTLS-Fuzzer. Η αρχιτεκτονική του ProtocolState-Fuzzer παρουσιάζεται στο Σχήμα 1.9. Ο State Fuzzer, με μπλε χρώμα, συντονίζει την αλλαγή των φάσεων εκμάθησης. Το επάνω μέρος υλοποιεί τη φάση κατασκευής υποθέσεων, ενώ το κάτω μέρος υλοποιεί τη φάση επικύρωσης υποθέσεων. Ο Learner, με πράσινο χρώμα, προσπαθεί μόνο να κατασκευάσει μια υπόθεση. Λαμβάνοντας υπόψη τον Learner με πράσινο χρώμα στο Σχήμα 1.4 και Σχήμα 1.5, οι αρμοδιότητές του κατανέμονται τώρα μεταξύ του State Fuzzer και του Learner. Παρατηρήστε τα κίτρινα στρώματα που τυλίγουν όπως οι SUL Wrappers στο Σχήμα 1.6. Όλα τα αυτά τα στρώματα, συμπεριλαμβανομένων των SUL Wrappers, περιγράφονται σε αυτή την ενότητα.

Τύποι Δεδομένων

Ερώτημα ή test. Ένα ερώτημα (query) είναι μια σειρά συμβόλων εισόδου που πρέπει να εκτελεστεί ανεξάρτητα από άλλα ερωτήματα. Ονομάζονται επίσης "tests" από την οπτική του software testing. Για να επιτευχθεί η απαιτούμενη ανεξαρτησία, συνήθως μετά από κάθε ερώτημα ζητείται ένα "reset", το οποίο προκαλεί την επανεκκίνηση της διεργασίας του SUL.



Σχήμα 1.9: Η αρχιτεκτονική του ProtocolState-Fuzzer. Ο Learner με πράσινο χρώμα, εκτελεί την κατασκευή της υπόθεσης με το αντίστοιχο Membership SUL Oracle. Τα πιθανώς πολλά Equivalence Oracles που υποδεικνύονται με το σύμβολο συν, εκτελούν την επικύρωση της υπόθεσης με το δικό τους Equivalence SUL Oracle. Ο State Fuzzer με μπλε χρώμα, συντονίζει την εναλλαγή των δύο φάσεων και επιστρέφει το μοντέλο που προκύπτει. Μπορεί επίσης να παράσχει στον Learner ένα αντιπαράδειγμα που βρέθηκε κατά την επικύρωση της υπόθεσης, προκειμένου ο Learner να βελτιώσει την υπόθεσή του. Η ροή δεδομένων είναι επίσης σημειωμένη.

Απάντηση. Μια απάντηση (answer) σε ένα ερώτημα αποτελείται από μια σειρά συμβόλων εξόδου που αντιστοιχούν στο ερώτημα. Είναι δυνατόν ένα σύμβολο εισόδου του ερωτήματος να έχει πολλά αντίστοιχα σύμβολα εξόδου και όχι μόνο ένα. Αυτό εξαρτάται από τη συμπεριφορά του SUL και από το πόσα μηνύματα στέλνει ως απάντηση σε ένα μόνο σύμβολο εισόδου.

State Fuzzer

Ο ρόλος του State Fuzzer είναι να συντονίζει τις φάσεις εκμάθησης, να τις εναλλάσσει και τελικά να επιστρέφει είτε το μοντέλο που έμαθε είτε να σηματοδοτεί ότι κάτι πήγε στραβά και η εκμάθηση δεν μπόρεσε να τερματίσει. Μπορεί επίσης να παρακολουθεί τους γύρους και να τερματίζει τη διαδικασία εκμάθησης, αν έχει οριστεί όριο γύρων. Ένας νέος γύρος θεωρείται, όταν έχει κατασκευαστεί μια υπόθεση, έχει βρεθεί ένα αντιπαράδειγμα και έχει αρχίσει η βελτίωση της υπόθεσης.

Learner

Η μόνη ευθύνη του Learner που χρωματίζεται με πράσινο χρώμα στο Σχήμα 1.9, είναι να μάθει μια νέα υπόθεση ή να βελτιώσει μια υπάρχουσα υπόθεση με βάση ένα παρεχόμενο αντιπαράδειγμα.

Membership SUL Oracle

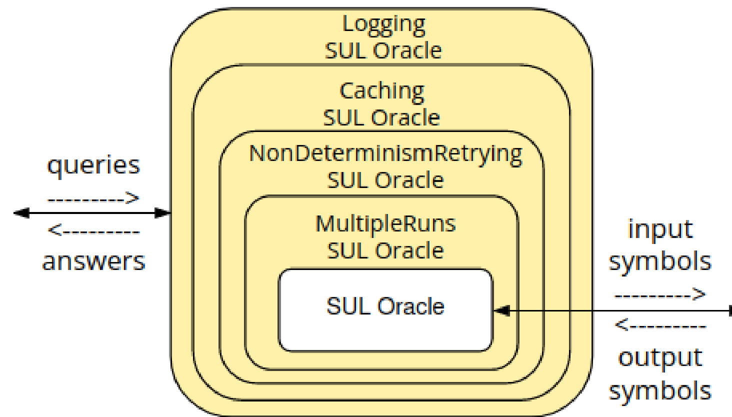
SUL Oracle. Το SUL Oracle είναι το απαραίτητο δομικό στοιχείο στο οικοσύστημα του LearnLib που πρέπει να δέχεται ως είσοδο ένα ερώτημα και να επιστρέφει την αντίστοιχη απάντηση. Συγκεκριμένα, για κάθε σύμβολο εισόδου στο εισερχόμενο ερώτημα, το SUL Oracle θα πρέπει να βρίσκει τα αντίστοιχα σύμβολα εξόδου που επιστρέφει το SUL. Για να επιτευχθεί αυτό, το SUL Oracle θα πρέπει να συνδεθεί με ένα Abstract Symbol SUL (δεν φαίνεται στο Σχήμα 1.9), στο οποίο μπορεί να δώσει κάθε σύμβολο εισόδου και να περιμένει το αντίστοιχο σύμβολο εξόδου. Για το πρωτόκολλο EDHOC ένα τέτοιο Abstract Symbol SUL φαίνεται με ροζ χρώμα στο Σχήμα 1.6. Στο Σχήμα 1.9, το SUL Oracle είναι η κλάση πυρήνας ενός στρώματος από Oracles που παρέχουν πρόσθετη λειτουργικότητα.

Membership Helper Oracles. Το στρώμα Membership Helper Oracles, χρωματισμένο με κίτρινο χρώμα στο Σχήμα 1.9, αποτελείται από πολλά ενδιάμεσα Oracles που προσφέρουν πρόσθετη λειτουργικότητα από την ελάχιστη που προσφέρει το SUL Oracle. Είναι τυλιγμένα το ένα μέσα στο άλλο, πράγμα που σημαίνει ότι το εισερχόμενο ερώτημα διαδίδεται στην εσώτερη κλάση πυρήνα και η απάντηση διαδίδεται από αυτήν προς τα έξω μέχρι να επιστρέψει στον Learner. Αυτά τα ενδιάμεσα Oracles επίσης μοιράζονται ορισμένους πόρους, όπως ένα δέντρο παρατήρησης ή μια κρυφή μνήμη, προκειμένου να επιτύχουν τον συγκεκριμένο σκοπό τους. Η σειρά και ο τύπος των ενδιάμεσων Oracles μπορούν να μεταβληθούν, αλλά τα προκαθορισμένα ορίζονται παρακάτω με τη σειρά από το εξωτερικό προς το εσωτερικό, όπως φαίνεται στο Σχήμα 1.10.

- LoggingSULOracle, εάν είναι ενεργοποιημένο καταγράφει κάθε ερώτημα σε ένα καθορισμένο αρχείο.
- CachingSULOracle, χρησιμοποιεί την κοινή κρυφή μνήμη για την αναζήτηση και αποθήκευση ερωτημάτων, ώστε να μην επαναλαμβάνονται οι εκτελέσεις των ίδιων ερωτημάτων.
- NonDeterminismRetryingSULOracle, χρησιμοποιεί την κοινόχρηστη κρυφή μνήμη για να ελέγχει για μη ντετερμινισμό και εκτελεί εκ νέου ερωτήματα σε περίπτωση εντοπισμού μη ντετερμινισμού.
- MultipleRunsSULOracle, εάν είναι ενεργοποιημένο εκτελεί κάθε ερώτημα πολλές φορές προκειμένου να αντιμετωπιστεί ο μη ντετερμινισμός. Σε περίπτωση που οι εκτελέσεις οδηγήσουν σε διαφορετικές εξόδους, μπορεί να εκτελέσει πιθανολογική εξυγίανση, π.χ. εκτελεί το ερώτημα πολλές φορές και υπολογίζει την απάντηση με τη μεγαλύτερη πιθανότητα.

Equivalence Oracle

Το Equivalence Oracle δέχεται ως είσοδο ένα μοντέλο υπόθεσης και το αλφάβητο εισόδου και χρησιμοποιώντας έναν equivalence αλγόριθμο προσπαθεί να ανακτήσει μια απάντηση από το SUL που δεν είναι συνεπής με την υπόθεση. Εάν βρεθεί αντιπαράδειγμα, αυτό επιστρέφεται στον State Fuzzer. Προκειμένου να απαντηθούν τα ερωτήματά του, το Equivalence Oracle ρωτάει το τυλιγμένο Equivalence SUL Oracle.



Σχήμα 1.10: Το Membership SUL Oracle αποτελείται από βοηθητικά Oracles με κίτρινο χρώμα και από το πιο εσωτερικό, το SUL Oracle.

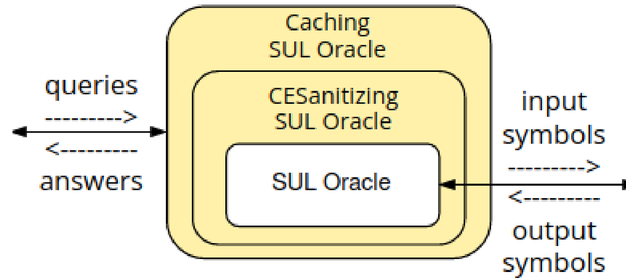
Το Equivalence Oracle εσωτερικά μπορεί να αποτελείται είτε από έναν equivalence αλγόριθμο είτε από μια λίστα equivalence αλγορίθμων, με τρόπο τέτοιο ώστε να επικυρώνουν διαδοχικά την υπόθεση μέχρι κάποιος από αυτούς να βρει ένα αντιπαράδειγμα.

Equivalence SUL Oracle

SUL Oracle. Το SUL Oracle παίζει τον ίδιο ρόλο με το αντίστοιχο του Membership SUL Oracle, δηλαδή δέχεται ως είσοδο ένα ερώτημα και επιστρέφει την αντίστοιχη απάντηση. Ο μόνος λόγος για τον οποίο υπάρχουν δύο SUL Oracles στο Σχήμα 1.9 είναι διότι τυλίγονται με διαφορετικό σύνολο βοηθητικών Oracles με κίτρινο χρώμα.

Equivalence Helper Oracles. Το στρώμα Equivalence Helper Oracles, χρωματισμένο με κίτρινο χρώμα στο Σχήμα 1.9, αποτελείται από πολλά ενδιάμεσα Oracles που προσφέρουν πρόσθετη λειτουργικότητα από αυτή του SUL Oracle και ακολουθεί την ίδια δομή με τα Membership Helper Oracles. Αυτό το στρώμα περιέχει δύο ενδιάμεσα Oracles, αντί για τέσσερα που αποτελούν τα Membership Helper Oracles. Η σειρά και ο τύπος των ενδιάμεσων Oracles μπορούν να τροποποιηθούν, αλλά τα προκαθορισμένα ορίζονται παρακάτω με τη σειρά από το εξωτερικό προς το εσωτερικό, όπως φαίνεται στο Σχήμα 1.11.

- CachingSULOracle, χρησιμοποιεί την κοινή κρυφή μνήμη για την αναζήτηση και την αποθήκευση ερωτημάτων, ώστε να μην επαναλαμβάνονται οι εκτελέσεις των ίδιων ερωτημάτων εισόδου.
- CESanitizingSULOracle, αν είναι ενεργοποιημένο επαναλαμβάνει ένα πιθανό αντιπαράδειγμα πολλές φορές, ελέγχοντας την έξοδό του. Σε περίπτωση λανθασμένων αντιπαραδειγμάτων, αυτό το Oracle εξοικονομεί χρόνο, μη επιτρέποντας στη φάση επικύρωσης υποθέσεων να τερματίσει και να επιστρέψει αυτό το λανθασμένο αντιπαράδειγμα. Αντ΄ αυτού, η φάση επικύρωσης υποθέσεων συνεχίζει την αναζήτηση άλλου αντιπαραδείγματος.



Σχήμα 1.11: Το Equivalence SUL Oracle αποτελείται από βοηθητικά Oracles με κίτρινο χρώμα και από το πιο εσωτερικό, το SUL Oracle.

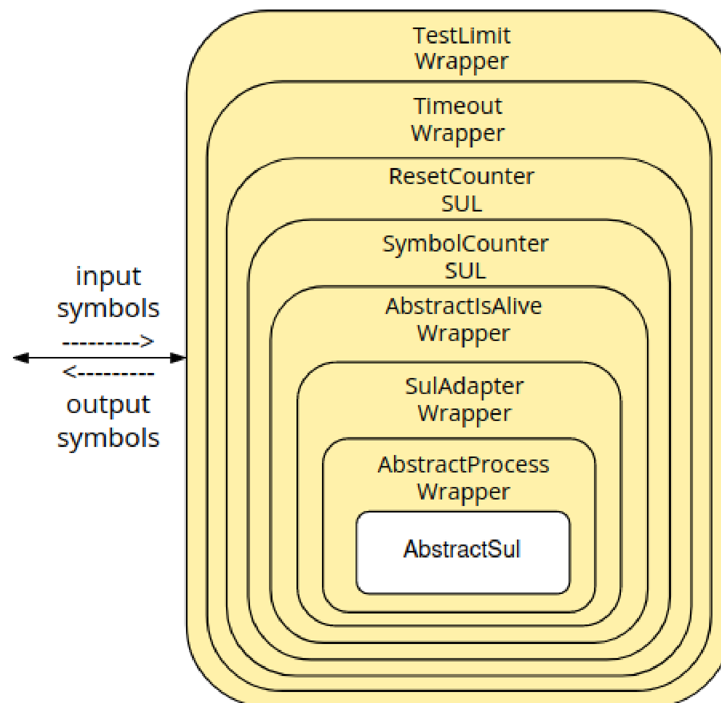
SUL Wrappers

Με τον ίδιο τρόπο που υπάρχουν ενδιάμεσα βοηθητικά Oracles που περιβάλλουν τον πυρήνα SUL Oracle, ο ProtocolState-Fuzzer παρέχει τους λεγόμενους SUL Wrappers, οι οποίοι περιβάλλουν την αφηρημένη κλάση του πυρήνα, που ονομάζεται AbstractSul. Η χρήση αυτών των wrappers είναι προαιρετική, αλλά παρέχουν πρόσθετη λειτουργικότητα, είτε για σκοπούς εκμάθησης είτε για τη διαχείριση και παρακολούθηση της διεργασίας SUL. Ο πιο εξωτερικός wrapper λαμβάνει ένα σύμβολο εισόδου και το διαδίδει σε όλη τη διαδρομή προς την πιο εσωτερική ή κλάση πυρήνα, η οποία θα είναι μια υλοποιημένη υποκλάση της AbstractSul, όπως η EdhocSul στο Σχήμα 1.7. Η υλοποιημένη υποκλάση της AbstractSul μπορεί να συνδεθεί με έναν υλοποιημένο Mapper ώστε να επικοινωνεί με το SUL. Όταν επιστρέφονται τα αντίστοιχα σύμβολα εξόδου, αυτά διαδίδονται από την κλάση πυρήνα προς τον πιο εξωτερικό wrapper. Είναι επίσης δυνατό ένας wrapper να επιλέξει να μην διαδώσει ένα εισερχόμενο σύμβολο εισόδου, σύμφωνα με κάποια ελεγχόμενη συνθήκη, και να επιστρέψει αμέσως ένα σύμβολο εξόδου. Η σειρά και ο τύπος των wrappers μπορούν να τροποποιηθούν, αλλά τα προκαθορισμένα καθορίζονται παρακάτω με τη σειρά από το εξωτερικό προς το εσωτερικό, όπως φαίνεται στο Σχήμα 1.12.

- TestLimitWrapper, παρακολουθεί τα tests (ή ερωτήματα) που έχουν ολοκληρωθεί και δημιουργεί μια εξαίρεση όταν επιτευχθεί το παρεχόμενο όριο tests, οπότε η διαδικασία εκμάθησης σταματά.
- TimeoutWrapper, παρακολουθεί τη διάρκεια εκμάθησης και δημιουργεί μια εξαίρεση όταν επιτευχθεί το παρεχόμενο χρονικό όριο, οπότε η διαδικασία εκμάθησης σταματά.
- ResetCounterSUL, ενημερώνει έναν μετρητή σε κάθε εκτέλεση ενός test κατά τη διάρκεια και των δύο φάσεων εκμάθησης. Επειδή οι δοκιμές είναι ανεξάρτητες, διαχωρίζονται με "resets", ο αριθμός των οποίων μετράται από αυτόν τον wrapper.
- SymbolCounterSUL, ενημερώνει έναν μετρητή σε κάθε εκτέλεση εισόδου κατά τη διάρκεια και των δύο φάσεων εκμάθησης.
- AbstractIsAliveWrapper, ελέγχει αν ένα σύμβολο εξόδου που μεταφέρει πληροφορίες σχετικά με τη ζωντάνια της διεργασίας SUL, δείχνει ότι η διεργαίσα

έχει τερματίσει. Σε αυτή την περίπτωση, οι ακόλουθες είσοδοι δεν διαδίδονται περαιτέρω από αυτόν τον wrapper και επιστρέφεται μια κατάλληλη ειδική έξοδος ως απάντησή τους.

- SulAdapterWrapper, χρησιμοποιείται σε περίπτωση που ένας launcher server πρέπει να εκκινήσει μια νέα διεργασία του SUL.
- AbstractProcessWrapper, είναι υπεύθυνο για την εκκίνηση ή τον τερματισμό της διεργασίας του SUL. Η εκκίνηση μπορεί να γίνει σε δύο διαφορετικά σημεία ενεργοποίησης: (1) μία φορά κατά την έναρξη, με τερματισμό στο τέλος της εκμάθησης ή (2) πριν από την εκτέλεση κάθε test, με τερματισμό μετά την εκτέλεση του test. Επιπλέον, αυτός ο wrapper προσθέτει στην απάντηση κάθε SUL την πληροφορία εάν η διεργασία SUL εξακολουθεί να είναι ζωντανή ή όχι.



Σχήμα 1.12: Οι SUL Wrappers με κίτρινο χρώμα προσφέρουν επιπρόσθετη λειτουργικότητα στην κλάση πυρήνα, την AbstractSul.

1.4 Πειράματα

Όλες οι υλοποιήσεις που αναλύονται σε αυτή την ενότητα είναι ανοικτού κώδικα και διατίθενται στο διαδίκτυο. Κάθε μία από αυτές διεθέτει και EDHOC client και EDHOC server. Δεδομένου ότι οι υλοποιήσεις εξελίσσονται συνεχώς, τα μοντέλα που παρατίθενται παρακάτω μπορεί να μην αντικατοπτρίζουν τις τελευταίες εκδόσεις τους, ωστόσο εξακολουθούν να είναι χρήσιμα.

Στις επόμενες ενότητες, τα μοντέλα παρέχονται μαζί με μια αναλυτική περιγραφή. Τα ονόματα συμβόλων που περιέχονται στα μοντέλα θα είναι στη σύντομη μορφή τους, όπως βρίσκονται στον Πίνακα 1.1 και στον Πίνακα 1.2 και οι ίδιες μεταβάσεις συγχωνεύονται σε μία. Για παράδειγμα, εάν υπάρχει μια κατάσταση με πέντε μεταβάσεις προς μια διαφορετική κατάσταση, αυτό αντικατοπτρίζεται στα μοντέλα ως μία μετάβαση με πέντε ετικέτες, μια για κάθε μετάβαση. Οι μεταβάσεις σε ένα μοντέλο ακολουθούν τη μορφή "I / O", που σημαίνει ότι ο EDHOC-Fuzzer έστειλε το I και το SUL απάντησε με το O ή κάποιος SUL Wrapper απάντησε με ένα ειδικό σύμβολο εξόδου O.

Οι μεταβάσεις που έχουν το ειδικό μήνυμα εξόδου NA σημαίνουν ότι ο Mapper δεν μπόρεσε να στείλει το ζητούμενο μήνυμα εισόδου. Αυτό συμβαίνει κυρίως όταν ο EDHOC-Fuzzer, ο οποίος λειτουργεί ως CoAP server EDHOC Responder, προσπαθεί να στείλει M3APP_O και APP_O. Ανατρέξτε στη σύντομη επεξήγηση του NA στο Ενότητα 1.3. Σημειώστε, επίσης, ότι το ειδικό μήνυμα εξόδου $\perp$ (socket closed) επιστρέφεται ως απάντηση σε ένα μήνυμα εισόδου από ένα ενδιάμεσο SUL Wrapper, ο οποίος ανιχνεύει ότι η διεργασία SUL έχει σταματήσει να ακούει στην καθορισμένη θύρα της.

Σε όλα τα παρακάτω μοντέλα ο client είναι ο EDHOC Initiator και ο server είναι ο EDHOC Responder. Όταν το SUL είναι client, ο EDHOC-Fuzzer λειτουργεί ως server και όταν το SUL είναι server, ο EDHOC-Fuzzer λειτουργεί ως client.

EDHOC-RS

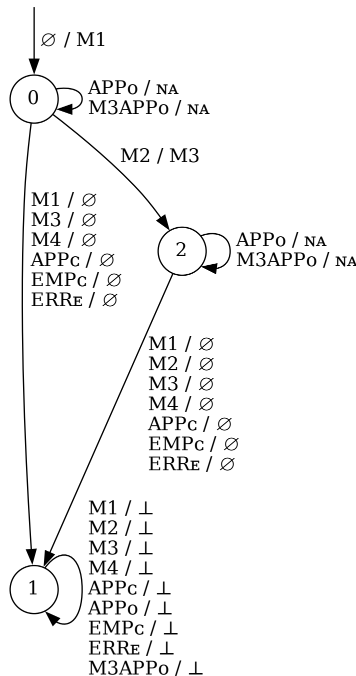
Το project βρίσκεται στο https://github.com/openwsn-berkeley/edhoc-rs.
Το commit hash 09aa2a822a9aa278146808b8498bea06f9103d59 υποδηλώνει το version που χρησιμοποιήθηκε.

Default Client

Το μοντέλο μηχανής καταστάσεων του default client φαίνεται στο Σχήμα 1.13.

Αρχική μετάβαση. Η μετάβαση στην αρχική κατάσταση 0 είναι απλώς μια υπενθύμιση ότι η διαδικασία εκμάθησης ξεκινά μόνο αφού ο client στείλει το αρχικό M1 χωρίς ο EDHOC-Fuzzer να στείλει τίποτα. Ουσιαστικά, ο client ξεκινάει την αλληλεπίδραση.

Ανταλλαγή EDHOC. Η ανταλλαγή EDHOC ξεκινά με τον client να στέλνει το M1 και το μοντέλο να μεταβαίνει στην κατάσταση 0. Ο EDHOC-Fuzzer το επεξεργάζεται και στέλνει M2, στο οποίο ο client απαντά με M3 κάνοντας το μοντέλο να μεταβεί στην κατάσταση 2. Έτσι ολοκληρώνεται το πρωτόκολλο από την πλευρά του client, κάτι που φαίνεται από την έξοδο $\varnothing$ σε κάθε μήνυμα που ακολουθεί από τον EDHOC-Fuzzer και την τελική έξοδο $\perp$ στην κατάσταση 1. Η μετάβαση του μοντέλου στην κατάσταση 1 σημαίνει ότι η διεργασία SUL έχει τερματίσει.



Σχήμα 1.13: Το μοντέλο του default EDHOC-RS client.

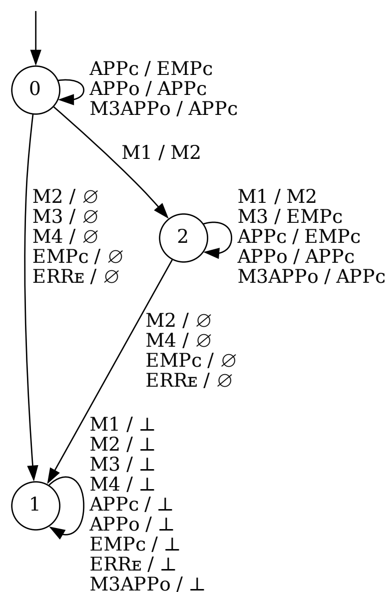
Default Server

Το μοντέλο μηχανής καταστάσεων του default server φαίνεται στο Σχήμα 1.14.

Ανταλλαγή EDHOC. Ο EDHOC-Fuzzer στέλνει το αρχικό M1 και λαμβάνει από τον server το M2, ενώ το μοντέλο μεταβαίνει από την κατάσταση 0 στην κατάσταση 2. Στη συνέχεια ο EDHOC-Fuzzer στέλνει M3 και λαμβάνει από τον server EMP_C, ενώ το μοντέλο παραμένει στην κατάσταση 2. Εάν ο EDHOC-Fuzzer στείλει M1 τότε μια νέα ανταλλαγή EDHOC ξεκινά από την αρχή και αντικατοπτρίζεται στη self-loop μετάβαση της κατάστασης 2.

M3 στην κατάσταση 2. Η ενδιαφέρουσα συμπεριφορά που παρουσιάζει η κατάσταση 2 είναι η self-loop μετάβαση όταν ο EDHOC-Fuzzer στέλνει M3 και λαμβάνει EMP_C. Αυτό σημαίνει ότι μετά τον τερματισμό μιας επιτυχημένης ανταλλαγής EDHOC (το μοντέλο βρίσκεται στην κατάσταση 2), εάν ο EDHOC-Fuzzer στείλει ξανά M3, τότε ο server δεν θα απαντήσει με σφάλμα.

Απαντήσεις στις καταστάσεις 0 και 2. Στις καταστάσεις 0 και 2, ο server απαντά στα APP_O, APP_C και $M3APP_O$. Αυτά τα μηνύματα αποστέλλονται σε ένα CoAP resource που ο server δεν έχει ρυθμιστεί να υποστηρίζει. Αυτό σημαίνει ότι οι κανονικές απαντήσεις σε τέτοια μηνύματα θα έπρεπε να είναι σφάλματα, αλλά αντ΄ αυτού είναι είτε ένα κενό μήνυμα CoAP (EMP_C) είτε μηνύματα CoAP με κάποιο τυχαίο ωφέλιμο φορτίο (APP_C).



Σχήμα 1.14: Το μοντέλο του default EDHOC-RS server.

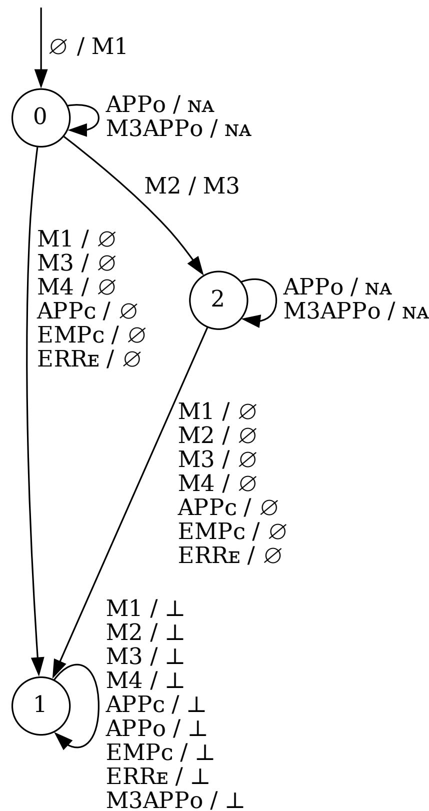
RISE

Το project βρίσκεται στο https://github.com/rikard-sics/californium/tree/edhoc. Το commit hash f994359a0bc04d62df5b3706a64ce857f1d3dfb7 υποδηλώνει το version που χρησιμοποιήθηκε.

Default Client

Το μοντέλο μηχανής καταστάσεων του default client φαίνεται στο Σχήμα 1.15.

Ανταλλαγή EDHOC. Ο EDHOC-Fuzzer, ο οποίος δρα ως server, περιμένει το αρχικό M1 από τον client. Στη συνέχεια στέλνει M2 και λαμβάνει πίσω M3. Το μοντέλο του client μεταβαίνει από την κατάσταση 0 στην κατάσταση 2. Έτσι ολοκληρώνεται η ανταλλαγή και για τα δύο μέλη, κάτι που είναι προφανές στις μεταβάσεις από την κατάσταση 2 στην κατάσταση 1.



Σχήμα 1.15: Το μοντέλο του default RISE client.

Default Server

Το μοντέλο μηχανής καταστάσεων του default server φαίνεται στο Σχήμα 1.16.

Ανταλλαγή EDHOC. Ο EDHOC-Fuzzer, ο οποίος δρα ως client, στέλνει το αρχικό M1 και λαμβάνει από τον server M2. Στη συνέχεια, ο EDHOC-Fuzzer μπορεί να στείλει είτε M3 και να λάβει EMP_C ή να στείλει M3APP_O και να λάβει APP_O. Σε αυτό το μοντέλο υπάρχουν 5 διαφορετικές επιτυχείς ανταλλαγές EDHOC. Συγκεκριμένα, (1) ακολουθία καταστάσεων $0 \to 1 \to 2$, (2) ακολουθία καταστάσεων $1 \to 1 \to 2$, (3) ακολουθία καταστάσεων $2 \to 3 \to 2$, (4) ακολουθία καταστάσεων $3 \to 3 \to 2$, (5) ακολουθία καταστάσεων $4 \to 3 \to 2$. Μια νέα ανταλλαγή EDHOC μπορεί να ξεκινήσει στη μέση της προηγούμενης. Για παράδειγμα, εάν το μοντέλο έχει μεταβεί από την κατάσταση 0 στην κατάσταση 1 και ο EDHOC-Fuzzer στείλει ένα M1 τότε ξεκινά μια νέα ανταλλαγή.

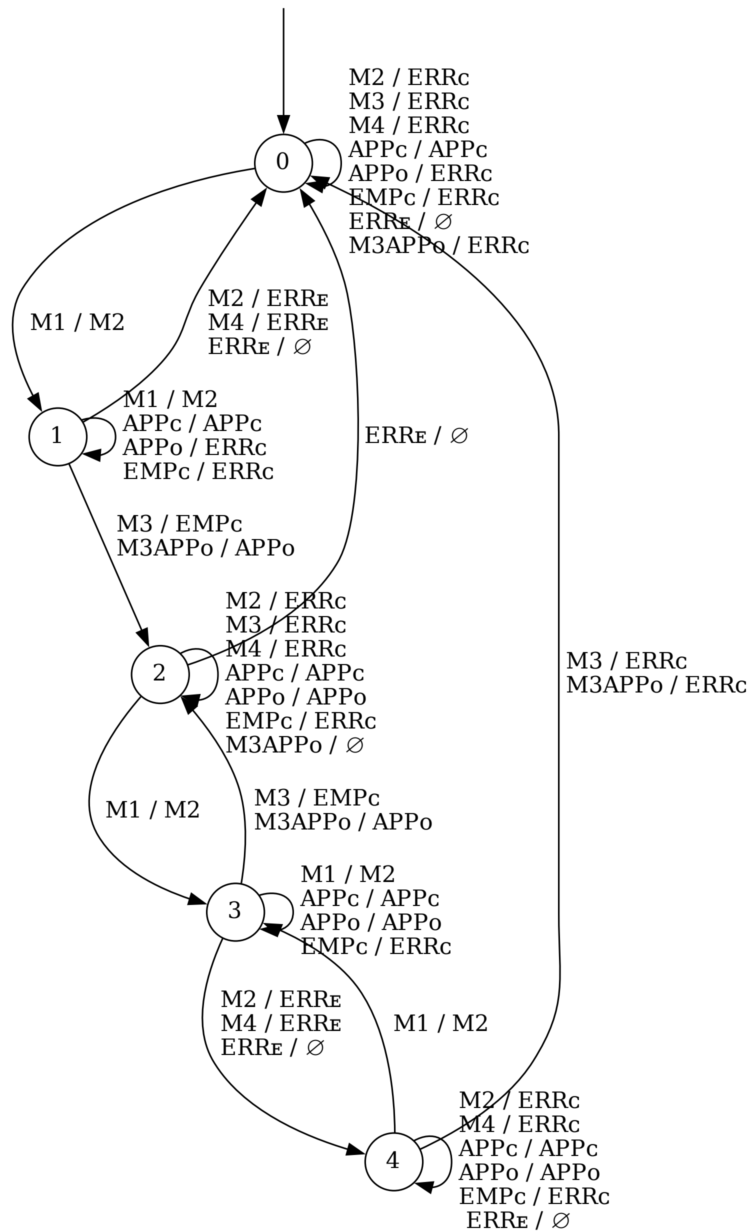
Plaintext Application Message. Παρατηρήστε ότι σε κάθε κατάσταση εάν ο EDHOC-Fuzzer στείλει APP_C στον server, τότε ο server απαντά με APP_C. Αυτό συμβαίνει, επειδή ο server δίχως να ελέγχει αν το μήνυμα είναι προστατευμένο με OSCORE, απαντάει κανονικά.

Ακύρωση δεύτερης ανταλλαγής EDHOC. Μια επιτυχής ανταλλαγή EDHOC έχει ολοκληρωθεί, έχει δημιουργηθεί ένα πλαίσιο ασφαλείας OSCORE και το μοντέλο βρίσκεται στην κατάσταση 2. Ο EDHOC-Fuzzer στέλνει το M1 και λαμβάνει πίσω M2, κάνοντας το μοντέλο του server να μεταβεί στην κατάσταση 3. Ο EDHOC-Fuzzer καθυστερεί το επόμενο μήνυμα που αναμένει ο server, το οποίο είναι M3 ή M3APP_O, και στέλνει αντί αυτού M2. Ο server απαντά με ERR_E, κάνοντας το μοντέλο να μεταβεί στην κατάσταση 4 και τερματίζει την τρέχουσα σύνοδο ανταλλαγής. Εάν τώρα ο EDHOC-Fuzzer προσπαθήσει να στείλει τα προηγουμένως αναμενόμενα μηνύματα (M3 ή M3APP_O), τότε ο server απαντά με ERR_C, υποδεικνύοντας ότι πράγματι η τρέχουσα σύνοδος ανταλλαγής EDHOC έχει τερματίσει.

OSCORE Context. Στην κατάσταση 2 και τα δύο μέλη έχουν παραγάγει το πλαίσιο OSCORE και είναι σε θέση να ανταλλάσσουν προστατευμένα μηνύματα. Το συγκεκριμένο πλαίσιο OSCORE θα παραμείνει, τυπικά, ζωντανό, μέχρι να ολοκληρωθεί μια νέα ανταλλαγή EDHOC και να προκύψει ένα νέο πλαίσιο OSCORE. Ο EDHOC-Fuzzer ξεκινά μια νέα ανταλλαγή EDHOC, στέλνει M1 και λαμβάνει M2, ενώ το μοντέλο μεταβαίνει στην κατάσταση 3. Ο EDHOC-Fuzzer στέλνει τώρα ένα APP_O και λαμβάνει πίσω ένα APP_O ως απάντηση με το πλαίσιο OSCORE να είναι το ίδιο με το προηγούμενο, επειδή η τρέχουσα ανταλλαγή EDHOC δεν έχει ολοκληρωθεί. Εάν ο EDHOC-Fuzzer στείλει ERR_E, ο server δεν απαντά και το μοντέλο μεταβαίνει στην κατάσταση 4. Και πάλι στην κατάσταση 4, οι ανταλλαγές APP_O αναφέρονται στο ίδιο πλαίσιο OSCORE όπως και πριν. Εάν, στην κατάσταση 4, ο EDHOC-Fuzzer παραγάγει ένα νέο πλαίσιο OSCORE (αντικαθιστώντας το προηγούμενο ενεργό) μετά το M3 ή πριν το M3APP_O και στείλει είτε M3 είτε M3APP_O στον server, ο οποίος δεν έχει παραγάγει αυτό το νέο πλαίσιο, τότε ο server θα απαντήσει με ERR_C. Εν τω μεταξύ, ο EDHOC-Fuzzer έχει αντικαταστήσει το προηγούμενο ενεργό πλαίσιο OSCORE και κάθε περαιτέρω APP_O που στέλνει οδηγεί σε ERR_C, το οποίο σημαίνει ότι το μοντέλο έχει μεταβεί στην κατάσταση 0.

Σημείωση για τον Mapper. Η προηγούμενη παράγραφος περιγράφει τη μετάβαση από την κατάσταση 4 στην κατάσταση 0. Αυτή είναι μια περίπτωση, όπου η συμπεριφορά του Mapper επηρεάζει αναπόφευκτα το μοντέλο προς μάθηση. Συγκεκριμένα, η συμπεριφορά του είναι να παράγει ένα νέο πλαίσιο OSCORE αμέσως μετά την επεξεργασία ενός M3. Εάν ο Mapper δεν είχε αντικαταστήσει το προηγούμενο πλαίσιο OSCORE, τότε είναι πιθανό αυτή η μετάβαση να ήταν ένα self-loop στην κατάσταση 4. Ωστόσο, εάν η μετάβαση από την κατάσταση 4 στην κατάσταση 0 είχε τις κανονικές εξόδους του server, δηλαδή EMP_C και M3APP_O αντίστοιχα, τότε αυτό θα σήμαινε ότι ο server δεν είχε τερματίσει τη συνεδρία.

Μετάβαση από την κατάσταση 2 στην κατάσταση 0. Μια ενδιαφέρουσα συμπεριφορά του server είναι η μετάβαση από την κατάσταση 2 στην κατάσταση 0. Η κατάσταση 2 είναι το σημείο όπου έχει ολοκληρωθεί μια επιτυχημένη ανταλλαγή EDHOC και τα δύο μέλη έχουν παραγάγει ένα πλαίσιο OSCORE, με το οποίο μπορεί να έχουν ήδη ανταλλάξει μηνύματα. Εάν ο EDHOC-Fuzzer στείλει τώρα ένα ERR_E, ο server δεν απαντάει τίποτα. Το ενδιαφέρον είναι ότι ο server, επεξεργάζεται το ERR_E, δεν απαντάει τίποτα αλλά διαγράφει το παραγόμενο πλαίσιο OSCORE, κάτι που φαίνεται από τις ERR_C απαντήσεις του στα APP_O του EDHOC-Fuzzer στην κατάσταση 0. Με άλλα λόγια, ο server επεξεργάζεται ένα ERR_E που δεν περιέχει καμία άλλη συγκεκριμένη πληροφορία για την προηγούμενη ολοκληρωμένη συνεδρία εκτός από το αναγνωριστικό σύνδεσης στην αρχή του μηνύματος. Ουσιαστικά αυτό το ERR_E κάνει τον server να τερματίσει την κρυπτογραφημένη σύνδεση μεταξύ των δύο κόμβων.



Σχήμα 1.16: Το μοντέλο του default RISE server.

SIFIS-HOME

Το project βρίσκεται στο https://github.com/sifis-home/wp3-solutions.
Το commit hash 9956c8cf9a6f8cb3ab09e48842ceafeb9d2a790e υποδηλώνει το version που χρησιμοποιήθηκε.

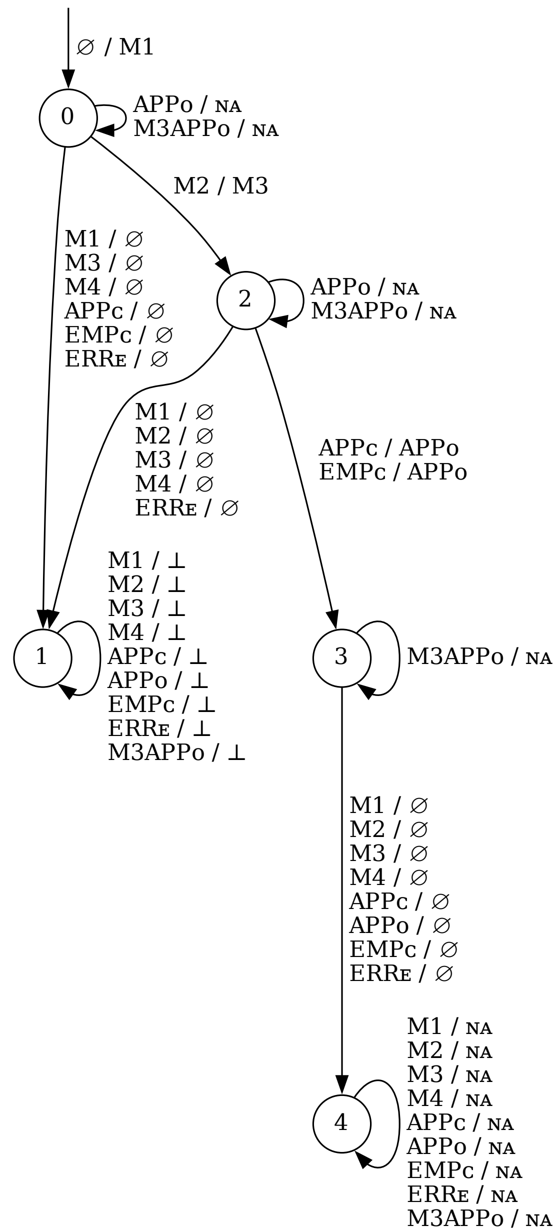
Αυτό το project αναπτύσσεται από τους ίδιους ανθρώπους που έχουν αναλάβει και το RISE. Για αυτό τα μοντέλα είναι παρόμοια με αυτά του RISE.

Default Client Phases 1, 2

Το μοντέλο μηχανής καταστάσεων των default clients των phases 1 και 2 είναι ίδιο και φαίνεται στο Σχήμα 1.17.

Ανταλλαγή EDHOC. O EDHOC-Fuzzer, ο οποίος δρα ως server, περιμένει το αρχικό M1 και στη συνέχεια στέλνει M2 στο οποίο ο client απαντά με M3 και το μοντέλο του μεταβαίνει στην κατάσταση 2. Η απάντηση του EDHOC-Fuzzer στο M3 είναι είτε APP_C είτε EMP_C. Η ανταλλαγή EDHOC ολοκληρώνεται, τα δύο μέλη έχουν παραγάγει ένα πλαίσιο OSCORE και ο client απαντά στον EDHOC-Fuzzer με APP_O. Το μοντέλο του μεταβαίνει στην κατάσταση 3, στην οποία ο client περιμένει την απάντηση του EDHOC-Fuzzer στο APP_O, και όταν τη λάβει δεν απαντάει τίποτα (φαίνεται με την έξοδο $\varnothing$) και το μοντέλο μεταβαίνει στην κατάσταση 4.

Έξοδος NA στην κατάσταση 4. Όταν το μοντέλο μεταβαίνει στην κατάσταση 4, ο EDHOC-Fuzzer δεν μπορεί να στείλει κανένα μήνυμα πίσω. Αυτό συμβαίνει, επειδή ο client στέλνει CoAP requests στα οποία ο EDHOC-Fuzzer, δρώντας ως server, απαντά με CoAP responses. Η τελευταία έξοδος που έλαβε ο EDHOC-Fuzzer, σύμφωνα με τη μετάβαση από την κατάσταση 3 στην 4, είναι το $\varnothing$, το οποίο σημαίνει ότι δεν έχει ληφθεί κανένα νέο CoAP request, επομένως ο EDHOC-Fuzzer δεν είναι σε θέση να απαντήσει και επιστρέφει NA μηνύματα εξόδου. Για αυτό τέτοιου τύπου μεταβάσεις δεν σχετίζονται με τη συμπεριφορά του SUL, αλλά προκαλούνται από τον EDHOC-Fuzzer και αντικατοπτρίζονται στο μοντέλο.

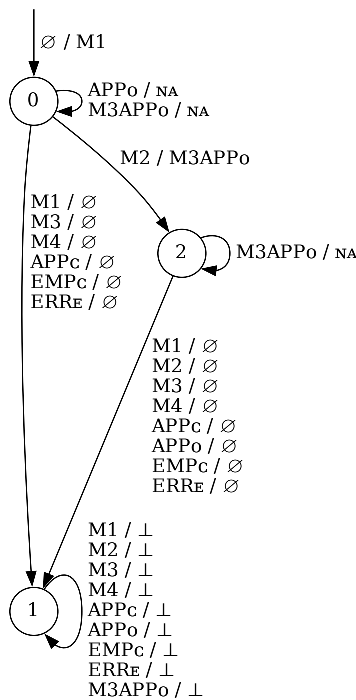


Σχήμα 1.17: Το μοντέλο των default SIFIS-HOME clients των phases 1 και 2.

Default Client Phases 3, 4

Το μοντέλο μηχανής καταστάσεων των default clients των phases 3 και 4 είναι ίδιο και φαίνεται στο Σχήμα 1.18.

Ανταλλαγή EDHOC. Ο EDHOC-Fuzzer, ο οποίος δρα ως server, περιμένει το αρχικό M1 και στη συνέχεια στέλνει M2 στο οποίο ο client απαντάει με $M3APP_O$ και το μοντέλο του μεταβαίνει στην κατάσταση 2. Η ανταλλαγή EDHOC έχει ολοκληρωθεί και τα δύο μέλη έχουν παραγάγει ένα πλαίσιο OSCORE. Ο client έχει ήδη στείλει ένα μήνυμα OSCORE στο $M3APP_O$ και περιμένει την απάντηση του EDHOC-Fuzzer, στην οποία δεν απαντά. Το μοντέλο μεταβαίνει στην κατάσταση 1 και η διεργασία του client τερματίζει.



Σχήμα 1.18: Το μονέλο των default SIFIS-HOME clients των phases 3 και 4.

Default Server Phases 1, 2, 3, 4

Το μοντέλο μηχανής καταστάσεων των default servers των phases 1, 2, 3 και 4 είναι παρόμοιο με αυτό του RISE στο Σχήμα 1.16. Η μόνη διαφορά είναι ότι τώρα οι απαντήσεις του server στα APP_C μηνύματα του EDHOC-Fuzzer είναι ERR_C, το οποίο σημαίνει πως ο server ελέγχει αν τα CoAP requests που έρχονται είναι προστατευμένα με OSCORE ή όχι. Μόνο αν είναι προστατευμένα, ο server απαντάει κανονικά, αλλιώς απαντάει με ERR_C.

uOSCORE-uEDHOC

Το project βρίσκεται στο https://github.com/eriptic/uoscore-uedhoc.
Το commit hash fbaa96caa1a2028d369c70e24a173caa60a0ce15 υποδηλώνει το version που χρησιμοποιήθηκε.

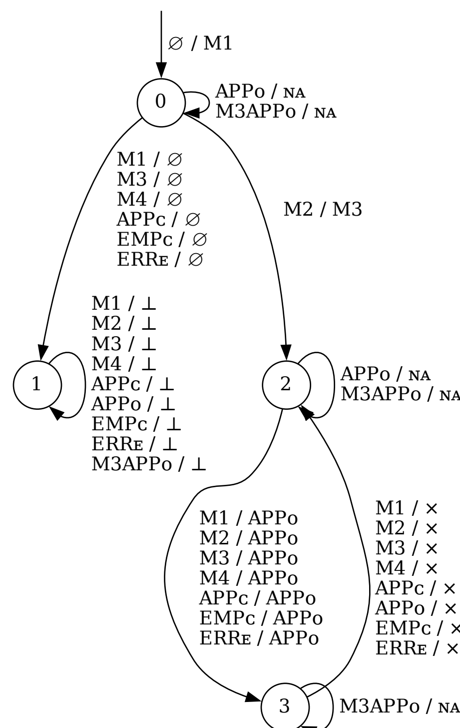
Default Client EDHOC and OSCORE

Το μοντέλο μηχανής καταστάσεων του default client EDHOC and OSCORE φαίνεται στο Σχήμα 1.19.

Ανταλλαγή EDHOC. Ο EDHOC-Fuzzer, ο οποίος δρα ως server, περιμένει το αρχικό M1 από τον client, μετά στέλνει M2 και λαμβάνει M3. Το μοντέλο μεταβαίνει από την κατάσταση 0 στην 2 και έτσι ολοκληρώνεται η ανταλλαγή, με τα δύο μέλη να έχουν παραγάγει επίσης ένα πλαίσιο OSCORE.

Βρόγχος στις καταστάσεις 2 και 3. Οι μεταβάσεις από την κατάσταση 2 στην 3 και αντίστροφα δημιουργούν ένα βρόγχο στο μοντέλο και υποδεικνύουν ότι και ο ίδιος client εκτελεί έναν ατέρμονα βρόγχο.

Έξοδος ×. Η μετάβαση από την κατάσταση 3 στην 2 έχει ως έξοδο ένα ×, επειδή ο client κάνει ένα CoAP request σε ένα μη διαθέσιμο CoAP Resource του EDHOC-Fuzzer, ο οποίος απορρίπτει το CoAP request.



Σχήμα 1.19: Το μοντέλο του default uOSCORE-uEDHOC EDHOC and OSCORE client.

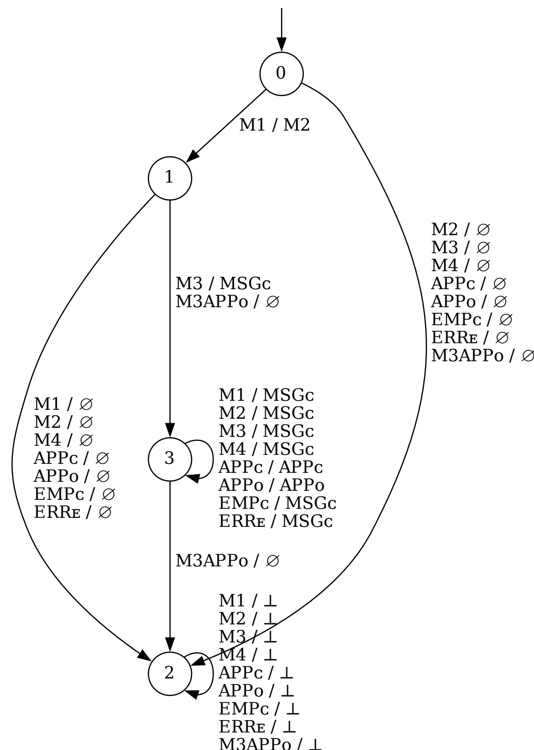
Default Server EDHOC and OSCORE

Το μοντέλο μηχανής καταστάσεων του default server EDHOC and OSCORE φαίνεται στο Σχήμα 1.20.

Ανταλλαγή EDHOC. Ο EDHOC-Fuzzer, ο οποίος δρα ως client, στέλνει το αρχικό M1, λαμβάνει M2 και το μοντέλο μεταβαίνει στην κατάσταση 1. Εάν ο EDHOC-Fuzzer, στείλει M3 θα πάρει MSG_C ως απάντηση και εάν στείλει $M3APP_O$, ο server δεν θα απαντήσει τίποτα και το μοντέλο του θα μεταβεί στην κατάσταση 3, ολοκληρώνοντας την ανταλλαγή EDHOC.

Application Messages. Στην κατάσταση 3, και τα δύο μέλη έχουν παραγάγει ένα πλαίσιο OSCORE και ανταλλάσσουν κρυπτογραφημένα μηνύματα (APP_O). Ο server δέχεται και μη κρυπτογραφημένα μηνύματα (APP_C). Σε οποιαδήποτε άλλα μηνύματα, ο server απαντά με MSG_C. Στην πραγματικότητα αυτό το MSG_C είναι ίδιο με το APP_C, αλλά αυτή η περίπτωση κάνει εμφανή τη δυσκολία του Mapper να ταυτοποιήσει αυτά τα δύο μηνύματα, η οποία συζητήθηκε στην Ενότητα 1.3.

Τερματισμός. Στην κατάσταση 3, εάν ο EDHOC-Fuzzer στείλει ένα $M3APP_O$ τότε ο server δεν απαντάει, η έξοδος είναι ∅ και στη συνέχεια τερματίζει. Είναι πιθανό το μέγεθος του $M3APP_O$ να κάνει τον server να τερματίσει ανεπιτυχώς λόγω σφάλματος.



Σχήμα 1.20: Το μοντέλο του default uOSCORE-uEDHOC EDHOC and OSCORE server.

Αυτοματοποιημένη Ανίχνευση Σφαλμάτων

Ακόμα και αν αυτά τα μοντέλα που μαθαίνονται περιέχουν μικρό αριθμό καταστάσεων και μεταβάσεων, η αυτοματοποίηση της ανίχνευσης σφαλμάτων είναι εξίσου σημαντική. Προς την κατεύθυνση αυτή, θα χρησιμοποιηθεί το εργαλείο SMBugFinder [Fit+23; FB+22].

Το εργαλείο αυτό υλοποιεί μια αυτόματη τεχνική μαύρου κουτιού για την ανίχνευση σφαλμάτων μηχανών καταστάσεων. Αυτή η τεχνική χρειάζεται να εφοδιαστεί με ένα (κατά προσέγγιση) μοντέλο της υλοποίησης και έναν κατάλογο των σφαλμάτων μηχανής καταστάσεων του πρωτοκόλλου (ή bug patterns) με τη μορφή DFAs.

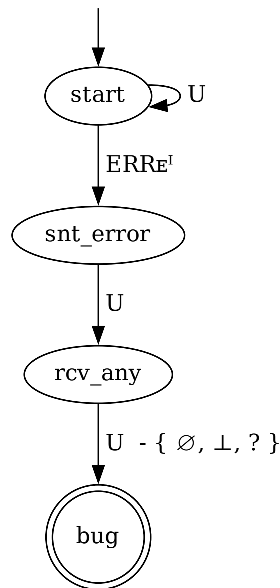
Το μοντέλο που παρέχεται είναι αυτό που παράγει ο EDHOC-Fuzzer πριν από οποιαδήποτε οπτική βελτίωση, όπως η συγχώνευση μεταβάσεων και η συντόμευση ονομάτων. Τα bug patterns είναι DFAs εμπλουτισμένα με κάποια συγκεκριμένα σύμβολα που αναγνωρίζονται από το εργαλείο. Ο SMBugFinder μετατρέπει το παρεχόμενο μοντέλο, το οποίο είναι μια μηχανή Mealy, σε ένα DFA και για κάθε bug pattern δημιουργεί ένα νέο πλήρες DFA μετά τις απαραίτητες τροποποιήσεις που επιβάλλουν αυτά τα σύμβολα. Στη συνέχεια, ο SMBugFinder χρησιμοποιεί DFA intersection στα δύο νέα DFAs προκειμένου να ψάξει για το bug pattern στο DFA του παρεχόμενου μοντέλου.

Σε αυτή την ενότητα θα παρατεθεί μόνο ένα bug pattern, η σύνταξη του οποίου δεν είναι αυτή που χρησιμοποιεί το εργαλείο, υπάρχει όμως κάποια αντιστοιχία. Αυτό γίνεται για να δοθεί έμφαση στην κατανόηση του DFA σε υψηλότερο επίπεδο και για τον ίδιο λόγο τα ονόματα των μηνυμάτων είναι στη σύντομη μορφή τους. Το ειδικό σύμβολο U σε μια μετάβαση από μια κατάσταση s σε μια άλλη κατάσταση αντιπροσωπεύει όλα τα διαθέσιμα μηνύματα που δεν βρίσκονται σε μια εξερχόμενη μετάβαση της κατάστασης s. Επιπλέον, χρησιμοποιούνται οι άνω δείκτες I, R όταν είναι απαραίτητο να ξεκαθαριστεί αν το μήνυμα προέρχεται από τον Initiator ή τον Responder αντίστοιχα.

Bug Pattern

Ο client Initiator διατηρεί τη σύνοδο ζωντανή μετά την αποστολή μηνύματος EDHOC Error. Αυτό το DFA μπορεί να εφαρμοστεί σε έναν client Initiator και αποτυπώνει το σφάλμα ότι ένας client διατηρεί τη σύνοδο EDHOC ζωντανή μετά την αποστολή ενός μηνύματος EDHOC error. Παρουσιάζεται στο Σχήμα 1.21.

Το self-loop με τη μετάβαση U στην αρχική κατάσταση, παρακάμπτει όλες τις μεταβάσεις μέχρι να βρεθεί ένα (εξερχόμενο) ERR_E του Initiator, το οποίο παρουσιάζεται με τον δείκτη I. Τότε η μετάβαση U αντιστοιχεί σε οποιοδήποτε σύμβολο εισόδου που είναι η απάντηση του άλλου κόμβου στο ERR_E. Ο client θα έπρεπε να έχει τερματίσει τη σύνοδο, αλλά αν η έξοδος του client είναι άλλη από $\emptyset$, $\perp$ ή ? τότε πρόκειται για σφάλμα, το οποίο σημαίνει ότι ο client απαντά στην ίδια σύνοδο.



Σχήμα 1.21: Το bug pattern που αναγνωρίζει όταν ένας client Initiator, διατηρεί τη σύνοδο ζωντανή μετά την αποστολή μηνύματος EDHOC Error.

1.5 Συμπέρασμα

Η παρούσα διπλωματική αρχίζει με μια επισκόπηση του απαραίτητου θεωρητικού υπόβαθρου. Στη συνέχεια, ακολουθούν οι λεπτομέρειες σχετικά με τον EDHOC-Fuzzer, ο οποίος αποτελεί την κύρια συνεισφορά αυτής της διπλωματικής. Αυτές οι λεπτομέρειες αφορούν κυρίως τα κομμάτια που λείπουν και ολοκληρώνουν ένα εργαλείο ικανό να εκτελεί protocol state fuzzing σε υλοποιήσεις EDHOC. Στη συνέχεια, περιγράφονται οι λεπτομέρειες υλοποίησης και η εσωτερική σύνθεση του ProtocolState-Fuzzer. Τέλος, παρουσιάζονται τα πειράματα που πραγματοποιήθηκαν σε διάφορες υλοποιήσεις. Αυτά αποτελούνται από τα μοντέλα που μπόρεσε να δημιουργήσει ο EDHOC-Fuzzer μαζί με μια ενδελεχή ανάλυση τους. Επιπλέον, παρουσιάζεται και ένα υποσύνολο καθορισμένων bug patterns που μπορούν να συμπληρώσουν την οπτική επισκόπηση των μοντέλων αυτοματοποιώντας την αναζήτηση ορισμένων κατηγοριών σφαλμάτων στα μοντέλα.

Παρόλο που, κατά μέσο όρο, τα μοντέλα αποτελούνται από λίγες καταστάσεις και μεταβάσεις, είναι πολύ ενδιαφέρον το γεγονός ότι παρατηρείται μια ποικιλία συμπεριφορών. Αυτό προκύπτει φυσικά από το γεγονός ότι διαφορετικές ομάδες ανθρώπων λαμβάνουν ορισμένες σχεδιαστικές αποφάσεις και αναπτύσσουν τις υλοποιήσεις τους με διαφορετικό τρόπο. Η πρακτική πρόκληση τέτοιων εργαλείων όπως ο EDHOC-Fuzzer είναι η ικανότητα να αλληλεπιδρούν επιτυχώς με πολλές υλοποιήσεις και να μην περιορίζονται σε ορισμένες. Επιπλέον, το γεγονός ότι η εκμάθηση είναι επιτυχής και η ανάλυση των μοντέλων μπορεί να αποκαλύψει καλά κρυμμένες ακραίες περιπτώσεις και λογικά σφάλματα, υποδηλώνει τη σημασία αυτών των εργαλείων, όπως ο EDHOC-Fuzzer, ειδικά για τους ανθρώπους που έχουν αναλάβει την υλοποίηση αυτών των πρωτοκόλλων.

Μελλοντικές Επεκτάσεις

Όσον αφορά τον EDHOC-Fuzzer, υπάρχουν διάφορες επιλογές που χρήζουν διερεύνησης. Η πρώτη είναι αρκετά προφανής και αφορά τη συνεχή βελτίωση του εργαλείου και την υποστήριξη άλλων υλοποιήσεων καθώς αυτές θα αρχίσουν να εμφανίζονται. Αυτού του είδους ο έλεγχος συμβατότητας με νέες υλοποιήσεις όχι μόνο θα αυξήσει την ακρίβεια και την ευρωστία του ίδιου του εργαλείου, αλλά και θα βελτιώσει τις πιθανότητες να χρησιμοποιηθεί από άλλους ως ένα αποτελεσματικό εργαλείο για τις δικές τους υλοποιήσεις. Η δεύτερη επιλογή που θα μπορούσε να διερευνηθεί είναι η προσθήκη ορισμένων συγκεκριμένων λεπτομερειών στα σύμβολα εισόδου και εξόδου. Αυτές μπορούν να διευκολύνουν την ανάλυση της συμπεριφοράς των υλοποιήσεων. Μια τρίτη επιλογή είναι η επέκταση των δυνατοτήτων του EDHOC-Fuzzer προς το fuzzing γενικά, το οποίο μπορεί να αποκαλύψει ορισμένες κατηγορίες σφαλμάτων που το protocol state fuzzing δεν είναι σε θέση να το κάνει.

Όσον αφορά τον ProtocolState-Fuzzer, μια βιώσιμη μελλοντική κατεύθυνση θα ήταν να κυκλοφορήσει ως ένα αυτόνομο εργαλείο που θα λειτουργεί ως framework για πολλούς διαφορετικούς protocol state fuzzers. Αυτό θα μείωνε το κόστος της υλοποίησης ενός μεγάλου μέρους της εγκατάστασης εκμάθησης, προσφέροντας παράλληλα μεγάλο βαθμό επεκτασιμότητας και προσαρμογής. Μια άλλη πιθανή, αλλά πιο ριζοσπαστική, κατεύθυνση θα ήταν η εκμάθηση του μοντέλου

να πραγματοποιείται και με άλλες μεθόδους, όπως η παθητική εκμάθηση ή άλλες προτεινόμενες τεχνικές που θα χρειάζονταν τη δική τους εγκατάσταση εκμάθησης. Ωστόσο, είναι αμφισβητήσιμο αν αυτά τα χαρακτηριστικά θα ταίριαζαν καλά στον ProtocolState-Fuzzer ή θα ήταν προτιμότερο να προσφέρονται σε δικό τους αυτόνομο εργαλείο.

Chapter 2

Introduction

The last decade has witnessed an immense increase in the number of low-powered devices connected to the internet. These devices are referred to as Internet of Things (IoT) and have the ability to collect and exchange data in real time using mainly embedded sensors. Some primary examples include lights, thermostats, refrigerators, smart watches and even sensors in cars. The environment that those devices operate in is usually constrained in terms of network access and available resources, which limits the security offered by those devices. Hence the anecdotal saying that "the S in IoT stands for Security". To tackle this challenge, a lot of network and security protocols have been proposed, in order to enable reliable and safe data exchange between those devices operating in constrained environments.

The restrictions posed by the environment make the use of the UDP transport protocol suitable for the communication of IoT devices. Built on top of that is CoAP [RFC7252], a web transfer protocol specialized for constrained environments that makes the communication possible. The security of this communication is provided by OSCORE [RFC8613], a protocol used for encrypting and thus protecting the privacy of the exchanged data. However, the OSCORE protocol requires symmetrical secret parameters to be established by the communicating peers, a fact that indicates the need for a secure and reliable key exchange protocol. This gap is filled by EDHOC [SMP23], a newly proposed protocol currently in draft status, which is a lightweight key exchange protocol ideal for constrained environments.

On the one hand in theory, protocols are proposed incrementally, in order to address the arising problems. On the other hand in practice, the implementations of those protocols are the ones encountered daily. For this reason it is crucial that these implementations are precise, robust and compliant with their specification. This is the problem that this thesis is trying to tackle by trying to develop techniques and tools for the effective testing of EDHOC protocol implementations.

The main technique used is called protocol state fuzzing, which essentially uses active automata learning, a technique that seeks to learn a close approximation of the underlying state machine model of the implementation under learning. The learned model can either be inspected for logical flaws and inconsistencies with the proposed protocol's specification or used for model-based testing.

2.1 Contribution

The main contribution is the design and implementation of the EDHOC-Fuzzer tool, which can be used for learning a (possibly approximate) state machine of an EDHOC implementation. EDHOC-Fuzzer includes as a component a general and abstract tool, which is also implemented and called ProtocolState-Fuzzer, which could be exposed as a standalone tool in the future. Moreover, in this thesis, the models of several EDHOC server and client implementations are learned and analyzed in detail.

2.2 Outline

The remaining chapters are organized as follows:

- Chapter 3 introduces the necessary theoretical background concerning the protocols and the techniques used. It also briefly reviews related work.
- Chapter 4 provides implementation details of the EDHOC-Fuzzer.
- Chapter 5 provides the experiments and the models that have been learned and analyzed.
- Chapter 6 concludes this thesis and provides suggestions for future work.

Chapter 3

Background

3.1 CBOR Data Format

The Concise Binary Object Representation (CBOR) [RFC8949] is a data encoding format for binary representation of structured data (also known as binary serialization format). It is built on and extends the JSON data model. It is designed to achieve very small parser code size and small message size. One of CBOR's primary objectives is to be able to unambiguously encode the most common data formats that are used in Internet standards.

3.2 COSE Standard

The CBOR Object Signing and Encryption (COSE) [RFC9052] specifies how to create and process encryption, signatures, and message authentication codes and how to represent cryptographic keys using CBOR. COSE builds on JOSE, but incorporates the additional capabilities that CBOR has over JSON, like a direct encoding method of binary data without converting them first into a base64-encoded text string. Essentially this standatd specifies basic application-layer security services that can be efficiently encoded in CBOR.

3.3 CoAP Protocol

The Constrained Application Protocol (CoAP) [RFC7252] is a specialized web transfer protocol for use with constrained nodes and constrained networks, such as low-power or lossy networks. It is designed for machine-to-machine (M2M) applications, such as smart energy and building automation. CoAP provides a request/response interaction model between application endpoints, supports built-in services and resources discovery and includes key concepts of the Web, such as URIs and Internet media types. Furthermore, it can be interfaced with HTTP and it meets specialized requirements of constrained environments, such as simplicity, multicast support and very low overhead. Logically CoAP contains two layers shown in Fig. 3.1, a CoAP messaging layer used to deal with UDP and the asynchronous nature of the interactions and the request/response layer for the interactions. However, these two layers are just features of the CoAP header.

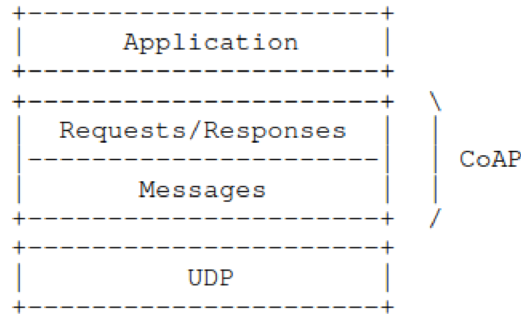


Figure 3.1: The two abstract CoAP layers, one for messages and one for the requests and responses.

A CoAP message contains a header and a payload, as shown in Fig. 3.2. The CoAP header consist of four bytes. Next, a token used to bind a request with a response follows. Its size can vary between 0 and 8 bytes, as indicated in the previous part of the CoAP header. Also, the header can include a number of options, following a Type-Length-Value scheme, whose value has variable length depending on their type. These options are used to control various functions of the protocol, such as indicate message fragmentation at the application layer. Finally, payload can be present, in which case a single byte with value 0xFF acts as payload delimiter and is followed by the actual CoAP payload.

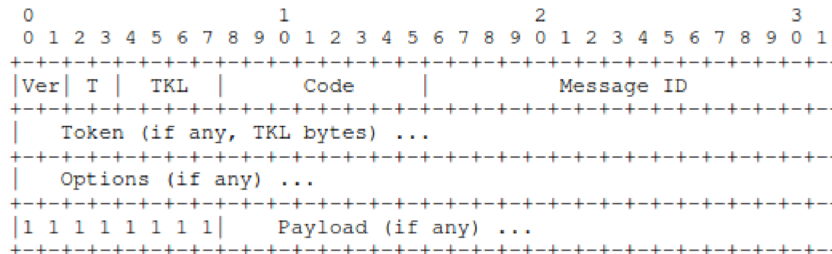


Figure 3.2: The message format of CoAP.

3.4 OSCORE Protocol

The Object Security for Constrained RESTful Environments (OSCORE) [RFC8613] defines a method for application-layer protection of the CoAP Protocol, using COSE. It is designed to provide end-to-end security between two CoAP endpoints, while preventing intermediates to alter or access any message field, which is not related to their intended operations. Essentially, OSCORE transforms an unprotected CoAP message into a protected one by securing not only the payload, but also all the fully protected CoAP options, the original request and response REST codes, as well as parts of the URI to resources targeted in request messages. The abstract layer of OSCORE with the CoAP protocol is shown in Fig. 3.3.

In order to use the OSCORE protocol, the participants need to derive and establish a shared security context to process the COSE objects. In order for this to happen, the necessary information and keying material should be exchanged in a secure and authenticated way, provided by a suitable key exchange protocol.

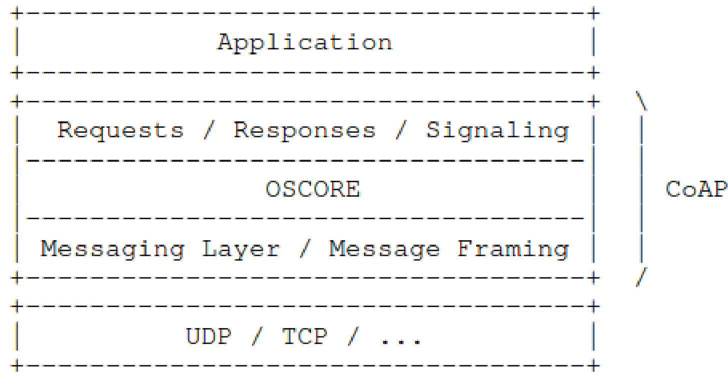


FIGURE 3.3: The abstract layer of OSCORE within the CoAP protocol.

3.5 EDHOC Protocol

The Ephemeral Diffie-Hellman Over COSE (EDHOC) protocol [SMP23] is a compact and lightweight authenticated key exchange protocol, which provides security properties like identity protection, cipher-suite negotiation and forward secrecy. One of its main use cases is to be a key exchange for the OSCORE protocol, which means to provide authentication and establishment of session keys. It is designed for highly constrained environments and it targets the IoT infrastructure, where embedded microcontrollers, sensors and actuators are involved. Moreover, the EDHOC protocol uses the same primitives as the OSCORE protocol, which are COSE for cryptography, CBOR for encoding and CoAP for transport. At the time of writing, this protocol is an Internet Draft and its current version is 19.

Because this thesis concerns the analysis of EDHOC implementations, a rather short and high-level overview of the protocol elements follows.

3.5.1 Roles and Messages

Roles. There are two possible EDHOC roles that a peer can have, namely Initiator or Responder. These roles are not tied to the web transfer protocol used. For example, when CoAP is used, a CoAP client can be either Initiator or Responder for the EDHOC protocol, so can a CoAP server. This means that for the CoAP protocol there are four possible nodes: CoAP client Initiator, CoAP client Responder, CoAP server Initiator and CoAP server Responder.

Messages. The protocol consists of five messages in total. In a successful key exchange the mandatory ones are *message_1*, *message_2* and *message_3*. There is *message_4*, which is optional and the *error_message*. Initiator uses *message_1* and *message_3* and Responder uses *message_2* and optionally *message_4*. Both roles can use

the *error_message*. An EDHOC message is encoded as a sequence of CBOR data items. A message flow can be seen in Fig. 3.4.

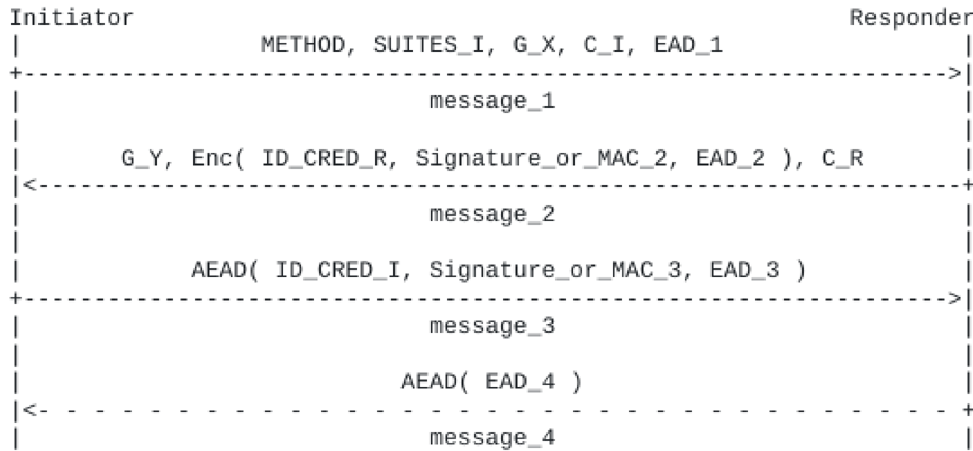


FIGURE 3.4: EDHOC message flow with optional *message_4*.

3.5.2 Authentication Method

EDHOC supports authentication with signature or with static Diffie-Hellman keys. The authentication method, or *METHOD*, is an integer out of { 0, 1, 2, 3 } and specifies the authentication method of both parties. When a static Diffie-Hellman key is used, the authentication is done by computing the Message Authentication Code (MAC) of an ephemeral-static ECDH shared secret. The different authentication methods are shown in Fig. 3.5.

3.5.3 Connection Identifiers

Connection Identifiers are selected by each peer to identify the current session. Initiator selects its identifier, *C_I* and sends it in *message_1* to the Responder, which then selects its identifier, *C_R* and sends it in *message_2* to the Initiator. The connection identifiers can be used for the correlation of messages and the retrieval of the protocol state during the session. For example the CoAP client, regardless of its EDHOC role, prepends the CoAP server's connection identifier before its messages

Method Type Value	Initiator Authentication Key	Responder Authentication Key
0	Signature Key	Signature Key
1	Signature Key	Static DH Key
2	Static DH Key	Signature Key
3	Static DH Key	Static DH Key

FIGURE 3.5: Authentication Keys for Method Types

to the server. Additionally, they can be used for an application protocol after the successful key exchange, such as OSCORE. In this case, the selection of an identifier by each peer should follow the restrictions of that application protocol. For instance, when these connection identifiers are used for the OSCORE protocol, they should not have the same value. Two EDHOC exchanges with prepended identifiers are shown in Fig. 3.6 and Fig. 3.7.

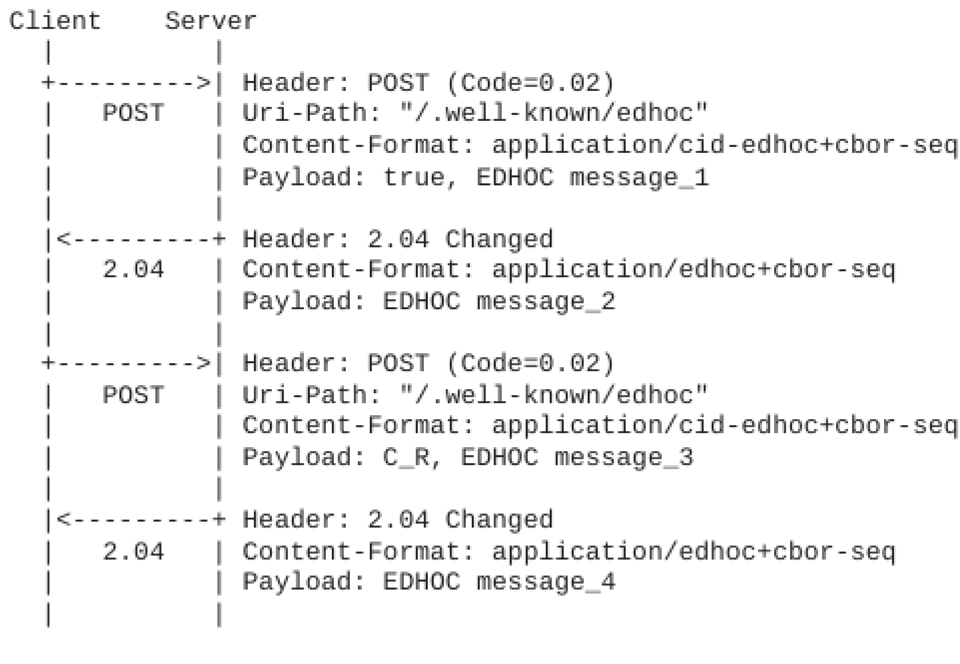


FIGURE 3.6: EDHOC exchange of CoAP Client Initiator and CoAP Server Responder. The prepended connection identifiers and the optional *message_4* are included.

3.5.4 Authentication Parameters

There are four types of authentication parameters, authentication keys, *authentication credentials*, *authentication credential identifiers* and *external authorization data*.

Authentication Keys. The authentication key is the public key (of the other peer) that is used for authentication. It must be either a signature key or a static Diffie-Hellman key, according to the selected authentication method.

Authentication Credentials. The authentication credentials of the Initiator and the Responder contain their public authentication key and are defined as *CRED_I* and *CRED_R*. The authentication credentials can be used to verify the integrity of the other peer and to verify proof-of-possession of the private key. They are not usually transported in EDHOC, although it is possible, but they are provisioned otherwise. This means that both Initiator and Responder have their and their peer's authentication credentials stored.

Authentication Credential Identifiers. They are used to identify the corresponding stored authentication credentials. Their size is smaller than the credentials that

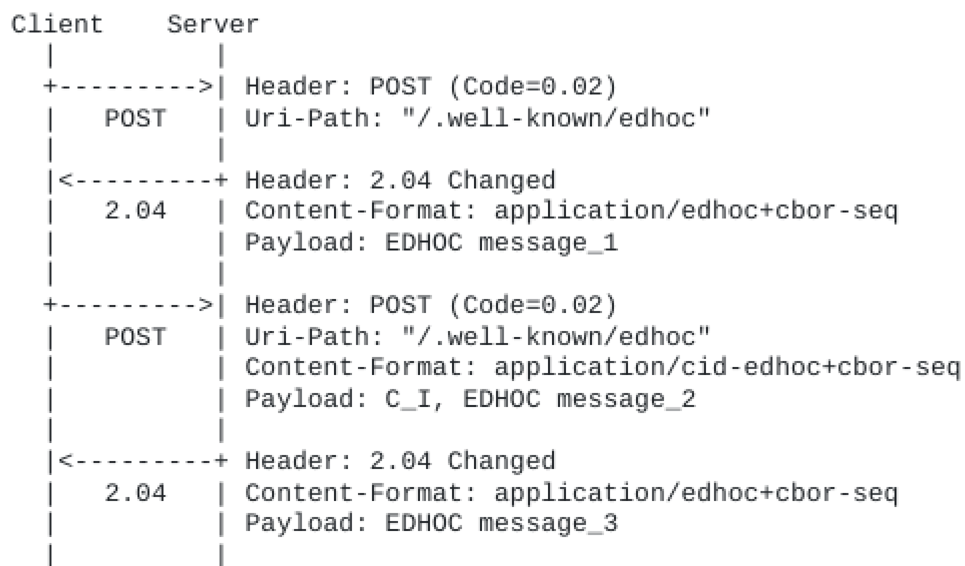


FIGURE 3.7: EDHOC exchange of CoAP Client Responder and CoAP Server Initiator. The prepended connection identifiers are included.

they identify, so they are the ones that are transported during EDHOC. The identifier of the Responder, *ID_CRED_R*, is transported in *message_2* and the identifier of the Initiator, *ID_CRED_I*, is transported in *message_3*. Optionally they can contain their respective authentication credential, but it is not necessary for many settings as they are otherwise acquired. Essentially, both peers have somehow stored in their memory the mapping of (possibly many) *ID_CRED_Is* to (possibly many) *CRED_Is* and the mapping of (possibly many) *ID_CRED_Rs* to (possibly many) *CRED_Rs*.

External Authorization Data (EAD). These are external application data that can be integrated in each message of the EDHOC protocol. The format of each EDHOC message has a dedicated field for these data, namely *EAD_1*, *EAD_2*, *EAD_3* and *EAD_4*. These are integrated in an attempt to reduce round trips and the number of exchanged messages and simplify processing.

3.5.5 Cipher Suites

An EDHOC cipher suite contains in that order: an *EDHOC AEAD algorithm*, an *EDHOC hash algorithm*, an *EDHOC MAC length* in bytes for static authentication, an *EDHOC key exchange algorithm*, an *EDHOC signature algorithm* for signed authentication, an *Application AEAD algorithm* and an *Application hash algorithm*. A cipher suite is identified by an integer label, the value of which can be found in a new registry titled "EDHOC Cipher Suites" under the new registry group "Ephemeral Diffie-Hellman Over COSE (EDHOC)" of IANA.

The EDHOC protocol also specifies how a cipher suite negotiation can take place between two peers. The important part of the negotiation is that it does not happen on the same session as the exchange. Specifically the Initiator sends its preferred

cipher suites in *message_1* and the Responder reply with an *error_message* that contains its supported cipher suites, while discontinuing the current session. In this way, the Initiator is responsible for storing and resolving this disagreement in the next *message_1* sent to the Responder. In other words, a successful EDHOC exchange cannot contain a cipher suite negotiation.

3.5.6 Ephemeral Public Keys

The ephemeral public keys are represented as *G_X* and *G_Y* for the Initiator and the Responder and they are exchanged in *message_1* and *message_2*, respectively. They are used as source of randomness for each session. That is the reason why each peer should generate fresh and random ephemeral key pairs.

3.5.7 Application Profile

The application profile is stored in each peer and contains information that enable the relevant processing and verification to be made. Some examples of data that it contains are the authentication method, if *message_4* should be sent or expected, the use and type of external authorization data.

3.5.8 OSCORE Context Derivation

The two peers can derive OSCORE security context parameters after they have successfully processed *message_3*. Specifically, the OSCORE Master Secret and Master Salt can be derived using a specific interface called EDHOC-Exporter interface. Additionally, the selected cipher suite of the EDHOC session contains an application AEAD algorithm, which could be also the OSCORE AEAD Algorithm and contains an application hash algorithm, on which the OSCORE HKDF Algorithm can be based. Lastly, a peer can have the other peer's EDHOC connection identifier as the OSCORE Sender ID and its own EDHOC connection identifier as the OSCORE Recipient ID.

3.6 Deterministic Finite Automaton

A Deterministic Finite Automaton (DFA) is a finite-state machine, which accepts or rejects a given symbol string. The machine starts in an initial state and given each symbol or character of an input string, the machine transitions from state to state according to its transition function. After the transition caused by the last symbol of the input string, if the machine is in one of the accepting states then the input string is accepted, otherwise it is rejected.

A Deterministic Finite Automaton is a 5-tuple $(\Sigma, Q, q_0, \Delta, F)$, where:

- Σ is the finite alphabet of input symbols
- Q is the finite set of states
- $q_0 \in Q$ is the initial state

- $\Delta : Q \times \Sigma \rightarrow Q$ is a transition function, which gives a next state $\Delta(q, s)$ for each state $q \in Q$ and symbol $s \in \Sigma$
- $F \subseteq Q$ is a finite set of accepting states

3.7 Mealy Machine

A Mealy Machine is a finite-state machine, whose output values depend on the current state and inputs. They have finite alphabets of input and output symbols and for each state and input at most one transition is possible. From an initial state, they process an input symbol triggering the generation of an output symbol sequence that makes the machine transition to a new state. From this new state the next input symbol can be processed.

A Mealy machine is a 6-tuple $(I, O, Q, q_0, \delta, \lambda)$, where:

- I is the finite alphabet of input symbols
- O is the finite alphabet of output symbols
- Q is the finite set of states
- $q_0 \in Q$ is the initial state
- $\delta : Q \times I \rightarrow Q$ is a transition function, which gives a next state $\delta(q, i)$ for each state $q \in Q$ and input symbol $i \in I$
- $\lambda : Q \times I \rightarrow O^*$ is an output function, which gives a (possibly empty) sequence of output symbols $\lambda(q, i)$, for each state $q \in Q$ and input symbol $i \in I$

3.8 Active Automata Learning

Active Automata Learning (AAL) constitutes a broad category of model learning [Vaa17]. It is an automated black-box technique, which constructs (approximate) state machine models of software and hardware systems by providing them with inputs (queries) and observing their outputs (answers). Usually in this thesis, the active automata learning algorithms output a deterministic Mealy machine, which represents the learned model of the System Under Learning (SUL). The learning procedure is a constant iteration of two phases: hypothesis construction and hypothesis validation. Its components are shown in Fig. 3.8.

Hypothesis Construction. During this phase, sequences of input symbols are selected and sent to the SUL in order to observe the responded sequences of output symbols. Next input sequences are selected based on the observed responses to previous ones. In this phase the queries are performed on a membership oracle and are called membership queries. The learning algorithm constructs a hypothesis, when certain convergence criteria are met. The hypothesis is a minimal deterministic Mealy machine consistent with the so far recorded observations. In other words, for all input sequences, which have been sent to the SUL, the hypothesis produces the same outputs as those observed from the SUL. Regarding an input sequence not

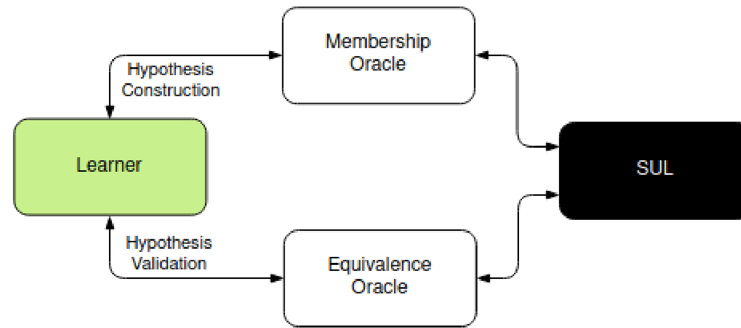


FIGURE 3.8: The main components of an AAL algorithm. The Membership Oracle answers membership queries regarding the SUL, from which the Learner creates a hypothesis. The Equivalence Oracle validates the hypothesis according to the SUL.

sent to the SUL, the hypothesis extrapolates its output from the recorded observations, essentially assuming it. In order to validate that these assumptions conform with the SUL the learning shifts to the hypothesis validation phase.

Hypothesis Validation. During this phase, conformance testing is performed on the SUL, so as to validate that the hypothesis agrees with the behavior of the SUL. In this phase the hypothesis to be validated is sent to an equivalence oracle, which performs queries, called equivalence queries on the SUL. It is possible that this conformance testing would find a counterexample, which is an input sequence on which the output of the hypothesis and the SUL do not match. In this case, the learning algorithm shifts back to the hypothesis construction phase, trying to refine the hypothesis by considering the discovered counterexample. If no counterexample could be found, learning terminates and returns the current hypothesis as the learned model. Because conformance testing is not complete, the learned model approximately describes the SUL. Obviously, the more equivalence queries are performed in this phase, the higher are the chances of finding a counterexample.

If the alteration of those two phases does not terminate, then it is possible that the SUL's behavior cannot be captured by a Mealy machine whose size and complexity is within reach of the current learning algorithm.

3.9 Protocol State Fuzzing

Protocol State Fuzzing, a term coined by de Ruiter and Poll [RP15], is a technique that uses active automata learning, in order to infer the state machines of protocol implementations. The learned models are then analyzed, either manually or using a model checking technique, in search for logical flaws that can be exposed by non-standard or unexpected message sequences. Instead of fuzzing individual messages, sequences of messages are fuzzed.

Learning Components. Based on the active automata learning algorithm so far, the needed components for the learning setup are two: a *Learner* and a *SUL*. The

Learner is responsible for asking queries to the *SUL* and receiving its answers, running this way effectively the learning algorithm. However, the *Learner* is protocol-agnostic meaning that the input alphabet contains abstract input symbols, which the *Learner* does not know how to turn into concrete protocol messages. This gap is filled by an intermediate component, the *Mapper* in Fig. 3.9, which acts as a test harness that transforms abstract input symbols from the finite input alphabet to concrete protocol messages sent to the SUL. The *Mapper* transforms the abstract input messages to concrete messages by filling the necessary details. Conversely, the *Mapper* maps the concrete messages from the SUL to abstract symbols of the output alphabet known to the *Learner* by removing protocol-specific unnecessary details. The *Mapper* also maintains a state hidden from the *Learner* needed for filling the message parameters.

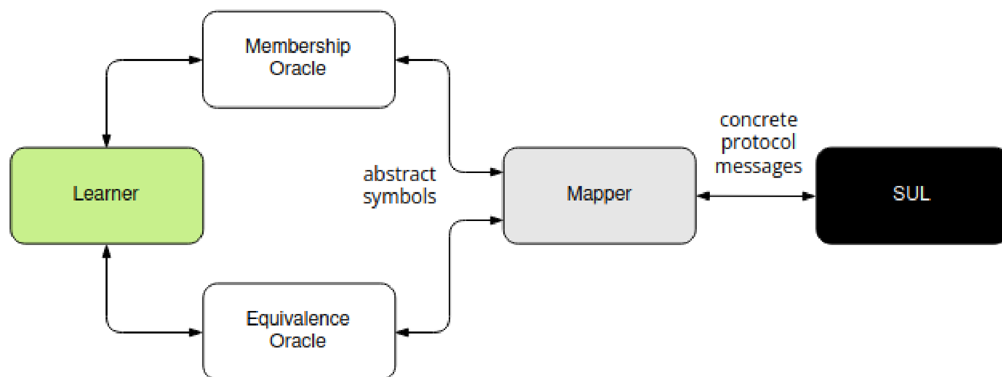


FIGURE 3.9: The three components used for the protocol state fuzzing. Notice the position of the Mapper after the Oracles and the color coding: green for the Learner, gray for the Mapper and black for the SUL.

3.10 Related Work

Protocol state fuzzing and active automata learning in general have been used to analyze many protocol implementations. A more detailed list of related work can be found in the wiki described in [Nei+19] and in [Wen20]. A rather incomplete list of works follows.

- DTLS [FB+20; Fit+22]
- IPSEC [Vel17]
- MQTT [TAB17]
- OpenVPN [DPR18]
- QUIC [Fer+21; RAR19]
- SSH [FB+17]
- TCP [Fer+21; FBJV16]
- TLS [Rui16; RP15]

Chapter 4

Implementation

An overview of the EDHOC-Fuzzer's architecture is shown in Fig. 4.1. It uses as a framework another implemented tool, called ProtocolState-Fuzzer, which is generic, protocol agnostic and offers the necessary setup for the protocol state fuzzing. For this reason the EDHOC-Fuzzer implements only the necessary parts, which are the Abstract Symbol SUL and the Mapper.

The implementation of ProtocolState-Fuzzer not only draws inspiration from, but also shares the same backbone code with DTLS-Fuzzer [FB+20; Fit+22][1], a protocol state fuzzer for the DTLS protocol written in JAVA. The LearnLib framework [IHS15][2] is used for active automata learning.

The use of the term "SUL" anywhere else than the SUL implementation is just a naming convention originating from the LearnLib framework and is not to be mistaken with the implementation under learning colored in black in Fig. 4.1.

The following section describes the parts that EDHOC-Fuzzer provides (colored in pink and gray in Fig. 4.1) and the next section describes the internal composition of ProtocolState-Fuzzer.

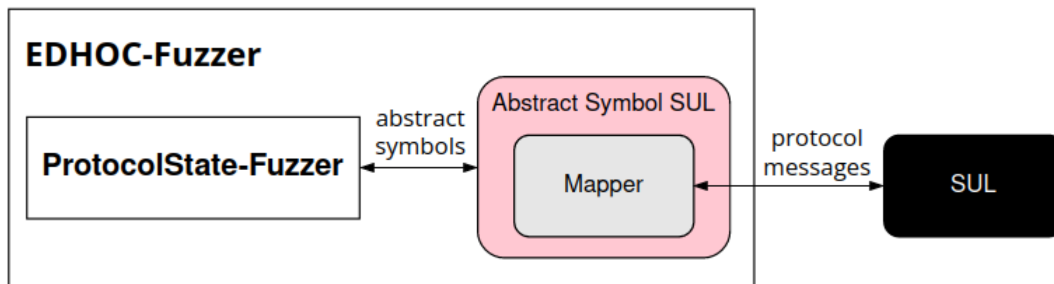


FIGURE 4.1: The architecture of the EDHOC-Fuzzer consists of the ProtocolState-Fuzzer, the Abstract Symbol SUL and the Mapper. The ProtocolState-Fuzzer connects with the Abstract Symbol SUL, which uses the Mapper to communicate with the black-box EDHOC implementation under learning called SUL. The Mapper and the SUL communicate through the network and usually in localhost.

[1]DTLS-Fuzzer is available at https://github.com/assist-project/dtls-fuzzer
[2]LearnLib is available at https://learnlib.de

4.1 EDHOC-Fuzzer

The only parts that needed to be specified are the Mapper and the Abstract Symbol SUL. The Mapper should know how to convert abstract symbols to EDHOC, OSCORE and CoAP messages and vice versa. The Abstract Symbol SUL is the necessary component, in order to utilize the ProtocolState-Fuzzer and uses the Mapper to communicate with the SUL, which is treated as a black-box. Additionally, several useful classes that the ProtocolState-Fuzzer provides were used for the implementation of both parts.

4.1.1 Abstract Symbol SUL

The Abstract Symbol SUL is the basic building block that is responsible for taking an abstract input symbol and returning the corresponding abstract output symbol. The abstract symbols are the input and output symbols of the corresponding alphabets. This building block is essential to the EDHOC-Fuzzer, because it connects the ProtocolState-Fuzzer with the Mapper and finally the SUL. The implementation of the Abstract Symbol SUL is shown in Fig. 4.2.

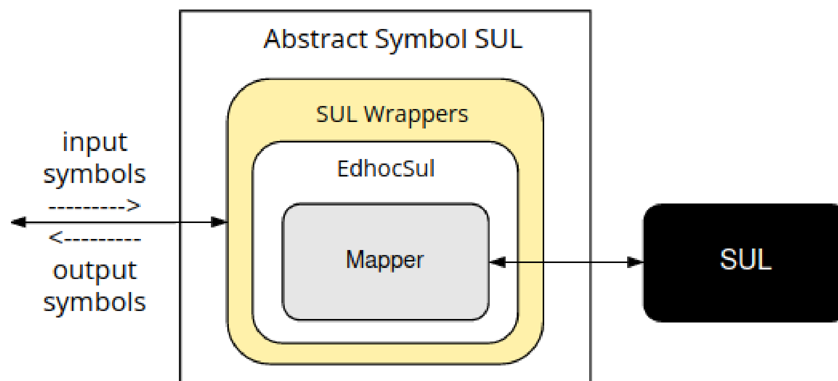


FIGURE 4.2: The implementation of the Abstract Symbol SUL. Innermost is the EdhocSul class, which uses the Mapper and is wrapped with a yellow layer of helper classes, called SUL Wrappers.

SUL Wrappers. The SUL Wrappers, colored yellow in Fig. 4.2, is a set of helper classes offered in ProtocolState-Fuzzer and provide additional functionality to the core class, EdhocSul. Essentially this layer consists of classes wrapped in one another, which are detailed in the next section. The input symbol is received from the outermost class and propagated to the innermost one (or core), which is the EdhocSul. When the core class returns an output symbol, this propagates from the innermost to the outermost class.

EdhocSul. This class is the core class of the Abstract Symbol SUL and implements the logic regarding a query (or a test), which is a sequence of input symbols. From the perspective of the EdhocSul there are three phases tied to the execution of a query; the prior, during and the after phase. In the phase prior to the execution, the EdhocSul sets up the Mapper according to whether the SUL concerns a server or a client implementation and initializes its state. For example, when the SUL is a

client implementation then the Mapper waits for the client's initial EDHOC message and then the learning starts. In the phase during the query execution, when the EdhocSul is provided with all the input symbols in the query, it uses the Mapper to communicate with the SUL and to receive the corresponding output symbol. The last phase is after the query execution, in which the EdhocSul has the chance to release any held resources. The repetition of those three phases is instructed by the ProtocolState-Fuzzer, which incorporates the learning algorithm logic.

4.1.2 Mapper

The Mapper should be specific to the EDHOC protocol and therefore three choices were available for its implementation in order of decreasing implementation difficulty:

- Implement the EDHOC protocol from scratch
- Find an EDHOC implementation and modify it, in order to use it as a library
- Find an API that allows sending on demand and receiving EDHOC messages

Library. The second option was selected, thus the basic library regarding the EDHOC protocol stems from an actual protocol implementation[3]. The code used as the library provides all the necessary pieces for the Mapper to function (after some necessary modifications). For example the Mapper can keep the protocol's state and has the ability to read and write CoAP, EDHOC and OSCORE messages.

Implementation. The implementation of the Mapper, shown in Fig. 4.3, is based on a class offered in ProtocolState-Fuzzer and called MapperComposer, which requires two different sub-mappers; the EdhocInputMapper and the EdhocOutputMapper. The EdhocInputMapper converts an abstract input symbol to a concrete protocol message and sends it to the SUL, while the EdhocOutputMapper receives the SUL's concrete protocol message and converts it to an abstract output symbol. However, both of them have inherent implementation difficulties. On the one hand, the EdhocInputMapper should act like a client to communicate with a server implementation and like a server in order to communicate with a client implementation. On the other hand, the EdhocOutputMapper should try to identify the received messages as accurate as possible and not generalize. For instance, in case CoAP is concerned, all received messages are CoAP messages and only a subset of them are EDHOC messages. This means that the EdhocOutputMapper should not identify all of the messages as CoAP ones unless no other specific identification is possible. It is crucial that both of the sub-mappers behave as correctly as possible, in order for the results to be accurate.

Impact on Learning. From the learner's perspective, the Mapper should behave transparently and reflect the actual messages that are being transmitted, but there are instances, where the Mapper's behavior inevitably affects the learned model. These are mostly related to the Mapper's protocol state. Specifically, there are cases when the Learner requests a specific input symbol to be sent, but the Mapper is

[3]The implementation used as library is the cf-edhoc module at https://github.com/rikard-sics/californium/tree/edhoc

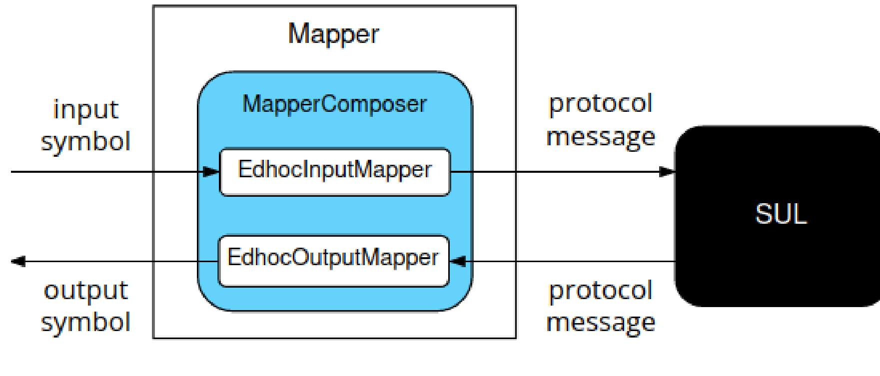


FIGURE 4.3: The Mapper is implemented as a MapperComposer, which is provided in ProtocolState-Fuzzer, consisting of an EdhocInputMapper for the input symbols and an EdhocOutputMapper for the output symbols.

unable to send it, because of its state. The OSCORE messages are a characteristic example of this, when the Mapper acts as a server. In other words, the ideal Mapper should be as transparent as possible, meaning that it sends the message it is instructed to send and receives the corresponding output, without interfering in this process.

4.1.3 Alphabets

There are two types of alphabets that are used. The input alphabet can be provided from the user, while the output alphabet is internal to the tool and depends on the accuracy of the Mapper. Each input or output symbol has a full-form and a short-form for its name. The short-form is used for visualization purposes in the resulting models.

Input Alphabet. All the input symbols that are supported are shown in Table 4.1. The default input alphabet includes all input symbols, but the user has the flexibility to omit some of them, according to their needs.

TABLE 4.1: The Input Alphabet Symbols

Full Name	**Short Name**
EDHOC_MESSAGE_1	M1
EDHOC_MESSAGE_2	M2
EDHOC_MESSAGE_3	M3
EDHOC_MESSAGE_4	M4
EDHOC_ERROR_MESSAGE	ERR_E
EDHOC_MESSAGE_3_OSCORE_APP	$M3APP_O$
OSCORE_APP_MESSAGE	APP_O
COAP_APP_MESSAGE	APP_C
COAP_EMPTY_MESSAGE	EMP_C

A short explanation for the input symbols follows.

- EDHOC_MESSAGE_{1, 2, 3, 4} and EDHOC_ERROR_MESSAGE, are the basic EDHOC messages.
- EDHOC_MESSAGE_3_OSCORE_APP, is the concatenation of the two messages after a new OSCORE context has been derived. No prepended connection identifier is used and only an EDHOC Initiator can send this.
- OSCORE_APP_MESSAGE, can be sent after the EDHOC exchange has been successfully completed and an OSCORE context has been derived. This message is essentially the encrypted application message between the two peers.
- COAP_APP_MESSAGE, is the plaintext version of OSCORE_APP_MESSAGE that is used to check if the other peer's resource actually expects an encrypted message and does not reply to unencrypted ones.
- COAP_EMPTY_MESSAGE, is a CoAP message with no payload.

Output Alphabet. The output symbols that the Mapper identifies are shown in Table 4.2.

TABLE 4.2: The Output Alphabet Symbols

Full Name	Short Name
EDHOC_MESSAGE_1	M1
EDHOC_MESSAGE_2	M2
EDHOC_MESSAGE_3	M3
EDHOC_MESSAGE_4	M4
EDHOC_ERROR_MESSAGE	ERR_E
EDHOC_MESSAGE_3_OSCORE_APP	M3APP_O
OSCORE_APP_MESSAGE	APP_O
COAP_APP_MESSAGE	APP_C
COAP_MESSAGE	MSG_C
COAP_ERROR_MESSAGE	ERR_C
COAP_EMPTY_MESSAGE	EMP_C
UNSUPPORTED_MESSAGE	NA
UNSUCCESSFUL_MESSAGE	$\times$
UNKNOWN_MESSAGE	?
SOCKET_CLOSED	$\perp$
TIMEOUT	$\varnothing$

A short explanation follows for the additional output symbols not shown in the input symbols.

- COAP_MESSAGE, is the most general output message of the CoAP messages.
- COAP_ERROR_MESSAGE, is an error message transferred in CoAP. Not to be confused with EDHOC_ERROR_MESSAGE.
- UNSUPPORTED_MESSAGE, is a special output message indicating that the Mapper could not send the requested input message at the state it was in. In most cases it occurs to EDHOC_MESSAGE_3_OSCORE_APP, because the

Mapper as an EDHOC Responder cannot send such messages and to OSCORE_APP_MESSAGE, when the Mapper, which acts as a server, receives a request which is not OSCORE-protected, but the Learner requests an OSCORE_APP_MESSAGE to be sent. This output message is part of the learned model, but it is not related to the behavior of the SUL.

- UNSUCCESSFUL_MESSAGE, is a special output message that usually occurs when the SUL is a client and the Mapper acts as a server. The message indicates that the actual output message of the SUL was received, yet its processing encountered an error. For example, if the client implementation sends a message targeting a resource that the Mapper does not support.
- UNKNOWN_MESSAGE, is the most general output message of all and is used when the message received cannot be identified otherwise.
- SOCKET_CLOSED, is a special output message indicating that the SUL has stopped listening to the port that was supposed to listen. Usually this means that the SUL process has terminated.
- TIMEOUT, is a special output message indicating that the Mapper waited for a specific time duration and has not received any output message from the SUL.

4.1.4 Remarks

The identification of output symbols, implemented in the Mapper, is quite difficult. It depends on the way and the order that the Mapper tries to identify the SUL's responses. Some output symbols are closely related from the Mapper's perspective. The COAP_APP_MESSAGE and the COAP_MESSAGE are a characteristic example. Both are CoAP messages, but they possibly differ in the payload content. A logical approach for the Mapper is to expect a COAP_APP_MESSAGE right after sending an OSCORE_APP_MESSAGE or a COAP_APP_MESSAGE. However, this leads sometimes a COAP_APP_MESSAGE to be identified as a COAP_MESSAGE. Overall, the more detailed input and output symbols the Mapper supports, the better an SUL can be analyzed.

4.1.5 Outcome

The outcome is the EDHOC-Fuzzer that accomplishes two goals. Firstly, it acts as a proof-of-concept for the ProtocolState-Fuzzer, since it effectively uses it in order to avoid a lot of implementation details regarding the learning setup and secondly it proves that the protocol state fuzzing of EDHOC implementations can be done successfully, in order to help in their analysis. Its development has not stopped, instead the tool is continuously being improved. The current functionality that EDHOC-Fuzzer provides is that it can either learn the approximate state machine model of an EDHOC implementation or it can perform explicit tests (or queries) on an EDHOC implementation.

4.2 ProtocolState-Fuzzer

The ProtocolState-Fuzzer stems from the abstraction and redesign of DTLS-Fuzzer, it is modular and extensible, while keeping the DTLS-Fuzzer's standard functionality. The architecture of the ProtocolState-Fuzzer is shown in Fig. 4.4. The State Fuzzer, colored in blue, coordinates the alteration of the learning phases. The top part realizes the hypothesis construction phase, while the bottom part realizes the hypothesis validation phase. The Learner, colored in green, only tries to construct a hypothesis. Considering the green colored Learner in Fig. 3.8 and Fig. 3.9, its responsibilities are now split between the State Fuzzer and the Learner. Notice the presence of wrapping yellow colored components like the SUL Wrappers in Fig. 4.1. All components, including the SUL Wrappers are described in this section.

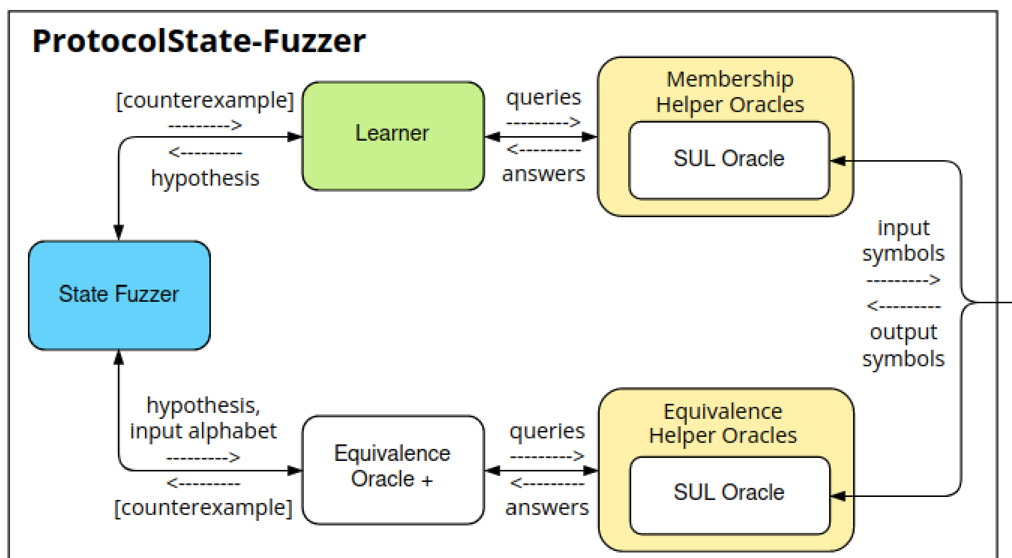


FIGURE 4.4: The architecture of ProtocolState-Fuzzer. The Learner, in green, performs the hypothesis construction with its corresponding Membership wrapped SUL Oracle. The possibly many Equivalence Oracles, indicated by the plus sign, perform the hypothesis validation with their Equivalence wrapped SUL Oracle. The State Fuzzer, in blue, coordinates the alteration of the two phases and returns the resulting model. It can also provide the Learner with a counterexample, found during hypothesis validation, in order for the Learner to refine its hypothesis. The data flow is also annotated.

4.2.1 Data Flow

Query or Test. A query is a series of input symbols that has to be run independently from other queries. They are also called "tests" from a software testing perspective. In order to achieve the needed independence, usually after each query (or test) a "reset" is requested, which causes the process of the SUL to be restarted.

Answer. An answer to a query or a test consists of a series of output symbols, corresponding to the query. It is possible that an input symbol of the query has many

corresponding output symbols and not only one. This depends on the behavior of the SUL and on how many messages it sends in response to a single input symbol.

Hypothesis Construction Data Flow. When a new hypothesis needs to be generated, the State Fuzzer colored in blue in Fig. 4.4, asks the Learner. Aside from the first time, the State Fuzzer can provide the Learner with a counterexample, found during a previous hypothesis validation phase, in order for the current hypothesis to be refined. During the hypothesis construction phase, the Learner, which implements a learning algorithm, creates and sends queries to the wrapped SUL Oracle. This Oracle connects to an Abstract Symbol Oracle (it is not shown in Fig. 4.4) and propagates to it every input symbol in the query, in order to receive the corresponding output symbols. Finally, the answer to the query is created and returned to the Learner.

Hypothesis Validation Data Flow. When a hypothesis needs to be validated, the State Fuzzer provides the learned hypothesis alongside with the input alphabet to the Equivalence Oracle. The Equivalence Oracle is responsible for creating queries and sending them to its own wrapped SUL Oracle. This wrapped SUL Oracle connects to an Abstract Symbol SUL, in order to get the output symbols from the black-box SUL and finally create the answers, which are returned to the Equivalence Oracle. If an inconsistent answer of a query between the provided hypothesis and the black-box SUL is found, then the pair (query, answer) is a counterexample and is returned to the State Fuzzer. The State Fuzzer, then, can provide this counterexample to the Learner, in order to refine its current hypothesis. If no counterexample can be found, then the last hypothesis is returned from the State Fuzzer as the learned model.

4.2.2 State Fuzzer

As it has already been mentioned earlier, the role of the State Fuzzer is to coordinate the learning phases, alternate them and finally return either the learned model or signal that something went wrong and the learning could not terminate. It can also keep track of rounds and terminate the learning process, if a round limit has been set. A new round is considered, when a hypothesis has been constructed, a counterexample has been found and hypothesis refinement has begun.

4.2.3 Learner

The only responsibility of the Learner, colored in green in Fig. 4.4, is to learn a new hypothesis or refine an existing hypothesis based on a provided counterexample. The Learner implements the following learning algorithms that are used for the hypothesis construction phase:

- L^*
- Rivest-Schapire
- TTT
- Kearns-Vazirani

4.2.4 Membership SUL Oracle

SUL Oracle. The SUL Oracle is the necessary building block in LearnLib's ecosystem that should take as input a query and return its corresponding answer. Specifically, for every input symbol in the incoming query, the SUL Oracle should find the corresponding output symbols that the black-box SUL would return. In order to achieve this, the SUL Oracle should be connected with an Abstract Symbol SUL (not shown in Fig. 4.4), to which can propagate every input symbol and expect the corresponding output symbol. For the EDHOC protocol one such Abstract Symbol SUL is shown in pink in Fig. 4.1. In Fig. 4.4, the SUL Oracle is the core class of a layer of Oracles that provide additional functionality.

Membership Helper Oracles. The Membership Helper Oracles layer, colored in yellow in Fig. 4.4, consists of many intermediate oracles that offer additional functionality than the minimum one of the SUL Oracle. They are wrapped in one another, meaning that the incoming query is propagated to the innermost core class and the answer is propagated from the innermost to the outermost Oracle and then returned to the Learner. Those intermediate Oracles can also share some resources, like an observation tree or cache, in order to achieve their specific purpose. The order and the type of intermediate Oracles can be altered, but the predefined ones are specified below in the order of outermost to innermost as shown in Fig. 4.5.

- LoggingSULOracle, if enabled logs every query to a specified file.
- CachingSULOracle, uses the shared cache to lookup and store queries so that executions of the same query inputs are not repeated.
- NonDeterminismRetryingSULOracle, uses the shared cache in order to check for non-determinism and re-runs queries in case non-determinism is detected.
- MultipleRunsSULOracle, if enabled executes each query multiple times in order to handle non-determinism. In case the runs result in different outputs, it can perform probabilistic sanitization, e.g. runs the query many times and computes the answer with the highest likelihood.

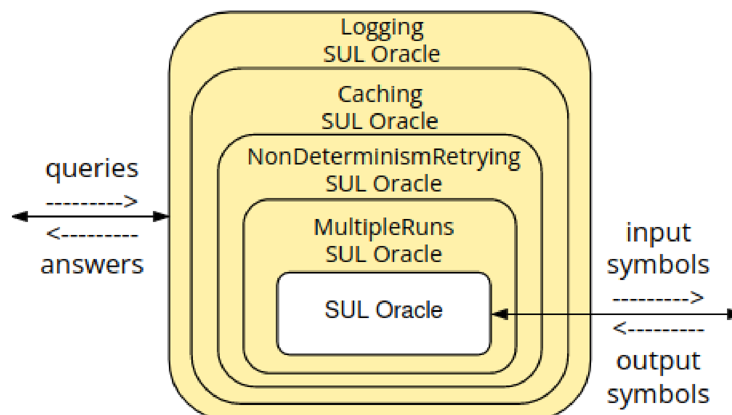


FIGURE 4.5: The Membership SUL Oracle consists of the helper oracles in yellow and the core innermost SUL Oracle.

4.2.5 Equivalence Oracle

The Equivalence Oracle takes as input a hypothesis model and the input alphabet and using an Equivalence Algorithm tries to retrieve an answer from the black-box SUL that is not consistent with the hypothesis. If a counterexample is found, it is returned to the State Fuzzer. In order to have its queries answered, the Equivalence Oracle asks its wrapped Equivalence SUL Oracle.

The Equivalence Oracle internally can consist either of one equivalence algorithm or a list of equivalence algorithms, in a way that they validate the hypothesis consecutively until one of them finds a counterexample. The available equivalence algorithms are:

- Random Walk
- W Method
- WP Method
- Random WP Method
- Sampled Tests
- WP Sampled Tests

4.2.6 Equivalence SUL Oracle

SUL Oracle. The SUL Oracle plays the same role as the one in the Membership SUL Oracle, which is to take as input a query and return its corresponding answer. It is more detailed in the Membership SUL Oracle description. The only reason that there are two SUL Oracles in Fig. 4.4 is that they are wrapped with different set of helper oracles colored in yellow.

Equivalence Helper Oracles. The Equivalence Helper Oracles layer, colored in yellow in Fig. 4.4, consists of many intermediate oracles that offer additional functionality than that of the SUL Oracle and follows the same structure as the Membership Helper Oracles. In fact this layer contains two intermediate oracles, instead of four that consist the Membership Helper Oracles. The order and the type of intermediate Oracles can be altered, but the predefined ones are specified below in the order of outermost to innermost as shown in Fig. 4.6.

- CachingSULOracle, uses the shared cache to lookup and store queries so that executions of the same query inputs are not repeated.
- CESanitizingSULOracle, if enabled reruns a potential counterexample multiple times, checking its output. In case of spurious counterexamples this oracle saves time by not allowing the hypothesis validation phase to terminate and return that erroneous counterexample. Instead the hypothesis validation phase continues searching for another counterexample.

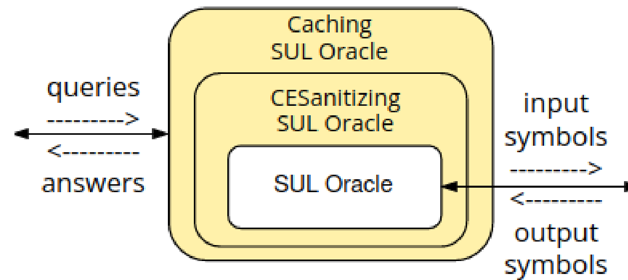


FIGURE 4.6: The Equivalence SUL Oracle consists of the helper oracles in yellow and the core innermost SUL Oracle.

4.2.7 SUL Wrappers

In the same way that are intermediate helper oracles that wrap the core SUL Oracle, ProtocolState-Fuzzer provides the so-called SUL Wrappers, which wrap the core abstract class, called AbstractSul. These wrappers are optional to use, but provide additional functionality, either for Learning purposes or for managing and monitoring the black-box SUL process. The outermost wrapper receives an input symbol and propagates it all the way to the innermost or core class, which would be an implemented subclass of AbstractSul, just like the EdhocSul in Fig. 4.2. The implemented subclass of AbstractSul can connect to an implemented Mapper so as to communicate with the black-box SUL. When the corresponding output symbols are returned, they are propagated from the core class to the outermost wrapper. It is also possible that one wrapper can choose not to propagate an incoming input symbol, according to some monitored condition, and return an output symbol immediately. The order and the type of wrappers can be altered, but the predefined ones are specified below in the order of outermost to innermost as shown in Fig. 4.7.

- TestLimitWrapper, keeps track of the tests (or queries) that have finished and raises an exception when the provided test limit is reached, in which case the learning process is stopped.
- TimeoutWrapper, keeps track of the learning duration and raises an exception when the provided time limit is reached, in which case the learning process is stopped.
- ResetCounterSUL, carries and updates a counter on every test run during both learning phases. Because tests are independent they are separated by “resets”, the number of which is counted by this wrapper.
- SymbolCounterSUL, carries and updates a counter on every input run during both learning phases.
- AbstractIsAliveWrapper, checks whether an output, which carries information about the aliveness of the process, indicates that the process has terminated. In this case, the following inputs do not propagate further from this wrapper and an appropriate special output is returned as their response.

- SulAdapterWrapper, is used in case a launch server should launch a new instance of the SUL.
- AbstractProcessWrapper, is responsible for launching or terminating the process that starts up the SUL. Launches can be made at two distinct trigger points: (1) once at the start, with termination taking place at the end of learning/testing or (2) before executing each test, with termination done after the test has been executed. Additionally this wrapper adds to each SUL's response the information whether the SUL process is still running or not.

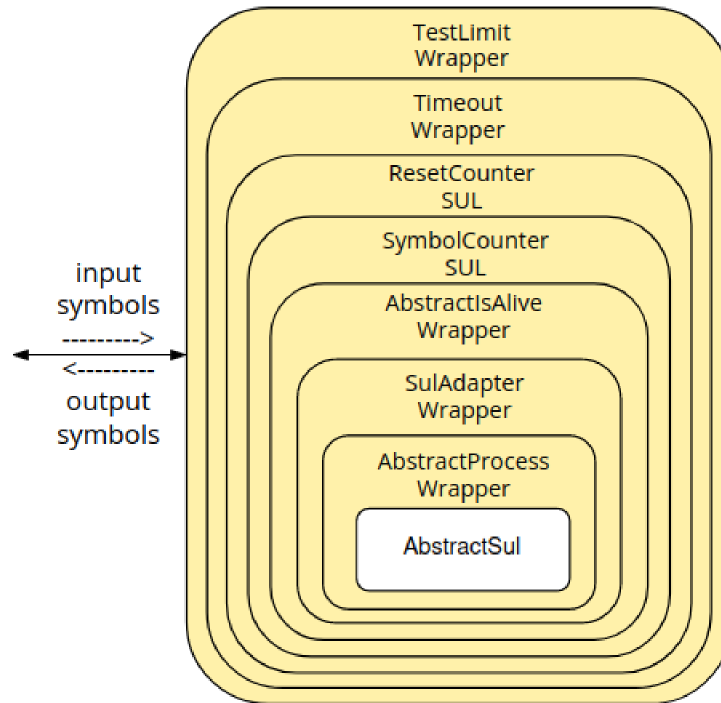


FIGURE 4.7: The SUL Wrappers that are colored in yellow provide additional functionality to the core AbstractSul class.

4.2.8 Outcome

The outcome of the refactoring and redesigning process of DTLS-Fuzzer was a tool, called ProtocolState-Fuzzer, which is currently internal to this project. It is a protocol-agnostic, modular and extensible framework that requires, in the default case, only the essential parts of the learning setup to be implemented.

Chapter 5

Experiments

All of the implementations in this chapter are open-source, actively maintained and updated. Each one of them offered both client and server implementations of the EDHOC protocol, whose (approximate) state machine would be provided in this chapter. Notice that, since the implementations keep evolving, the learned models may not reflect their latest versions, yet they are still useful.

In the following sections, the learned models will be provided alongside a description. The symbol names contained in the models would be in their short form, as they are found in Table 4.1 and Table 4.2 and the same transitions are merged into one. For example if there is a state with seven labelled transitions to a different state this is reflected in the learned models with one transition with seven stacked labels. The transitions in a learned model follow the format "I / O", which means that the EDHOC-Fuzzer sent the input I and the implementation replied with the output O or some SUL Wrapper replied with a special output O.

The transitions that have the special output message NA mean that the Mapper was unable to send their corresponding inputs and returned this output message. This occurs mainly when the EDHOC-Fuzzer, which acts as a CoAP server EDHOC Responder, tries to send $M3APP_O$ and APP_O. See the short explanation of the NA in Chapter 4. Note, also that the special output message $\perp$ (socket closed) is returned to an input message by an intermediate wrapper, which detects that the SUL process has stopped listening to its specified port.

In all the following models the client is the EDHOC Initiator and the server is the EDHOC Responder. When the SUL is a client, the EDHOC-Fuzzer acts like a server and when the SUL is a server, the EDHOC-Fuzzer acts like a client.

5.1 EDHOC-RS

The project is available at https://github.com/openwsn-berkeley/edhoc-rs.
The commit hash 09aa2a822a9aa278146808b8498bea06f9103d59 identifies the version.

5.1.1 Default Client

The learned state machine model of the default client is shown in Fig. 5.1.

Initial Transition. The transition to the initial state 0 is just a reminder that the learning process starts only after the client sends the initial M1 without the EDHOC-Fuzzer sending anything. Essentially, the client initializes the interaction.

EDHOC Exchange. The EDHOC exchange starts with the client sending M1 and the model transitioning to state 0. The EDHOC-Fuzzer then processes it and sends M2, which is processed by the client and M3 is replied making the model transition to state 2. This concludes the protocol from the client's side, which is seen from the output ∅ to any message followed by the EDHOC-Fuzzer and the final output ⊥ in state 1. Having the model transitioned to state 1 means that the client process has terminated.

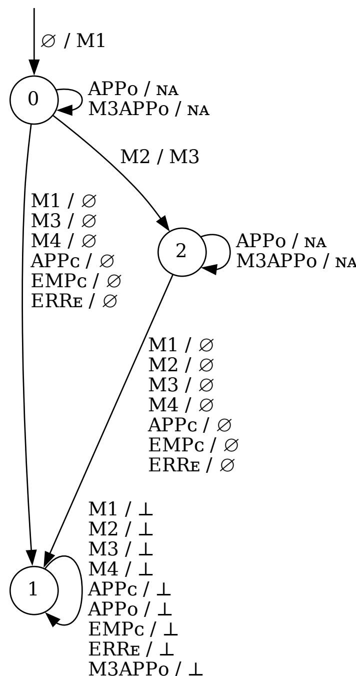


FIGURE 5.1: The learned model of the default EDHOC-RS client.

5.1.2 Default Server

The learned state machine model of the default server is shown in Fig. 5.2.

EDHOC Exchange. The EDHOC-Fuzzer sends the initial M1 and receives M2 from the server, while the model transitions from state 0 to state 2. Then the EDHOC-Fuzzer processes M2 and sends M3 that the server receives and replies with EMP_C, while the model stays in state 2. If the EDHOC-Fuzzer sends M1 then a new EDHOC exchange starts afresh and is reflected in the self-loop transition in state 2.

M3 in State 2. The interesting behaviour that state 2 exhibits is the self-loop transition when the EDHOC-Fuzzer sends M3 and receives EMP_C. This means that after a successful EDHOC exchange has been terminated and the model is in state 2, if the EDHOC-Fuzzer sends M3 again, then the server would not reply with an error.

Responses in States 0 and 2. In states 0 and 2 the server responds to APP_O, APP_C and M3APP_O. These messages are sent to a CoAP resource that the server is not configured to support. This means that the normal responses to such messages should be errors, but instead they are either an empty CoAP message (EMP_C) or CoAP messages with some random payload (APP_C).

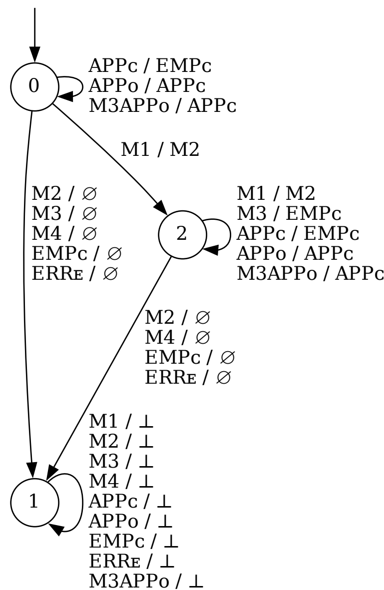


Figure 5.2: The learned model of the default EDHOC-RS server.

5.2 RISE

The project is available at https://github.com/rikard-sics/californium/tree/edhoc. The commit hash f994359a0bc04d62df5b3706a64ce857f1d3dfb7 identifies the version.

A patch has been applied to the downloaded source code that provides additional functionality to the server and client implementations and allows for altering some parameters regarding the EDHOC exchange. For example, it allows to specify if the EDHOC exchange would include the message_4 or not.

5.2.1 Default Client

The learned state machine model of the default client is shown in Fig. 5.3.

EDHOC Exchange. The EDHOC-Fuzzer, which acts as a server, waits for the initial M1 from the client. Then sends M2 to the client and receives back M3. The model transitions from state 0 to state 2. This concludes the exchange for both peers, which is obvious from the transitions from state 2 to state 1.

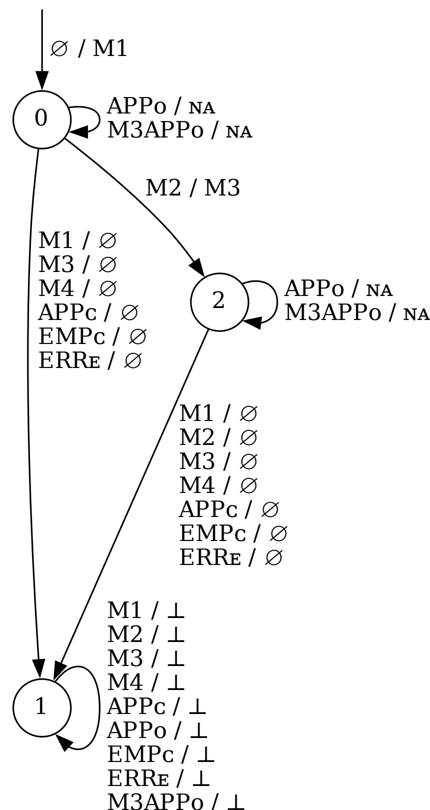


FIGURE 5.3: The learned model of the default RISE client.

5.2.2 Patched Client

Utilizing the additional functionality of the patch, the model of the RISE client was learned when the EDHOC exchange had message_4 enabled and after the exchange OSCORE messages were also sent. Note that the patch did not alter the logic incorporated in the client. The learned state machine model is shown in Fig. 5.4.

Normal EDHOC Exchange. The EDHOC-Fuzzer, which acts as a server, waits for the initial M1 from the client. Then sends M2 to the client and receives back M3. The model transitions from state 0 to state 2. The EDHOC-Fuzzer then sends M4, which the client receives. The EDHOC exchange is finished here, but the client is configured to send an OSCORE message, so it prepares it after having derived the necessary OSCORE context and replies with APP_{O}. This is the direct transition from state 2 to state 4. In state 4, the client waits for a reply to its OSCORE message (without replying back, thus the output is $\varnothing$ in the transition from state 4 to state 1) and then terminates in state 1.

Alternative EDHOC Exchange. The EDHOC-Fuzzer has received M1, sent M2 and received back M3. The model is at state 2. Notice the alternative path from state 2 to state 4 through state 3. In state 2 the client expects M4, in order to successfully complete the EDHOC exchange. Instead, if the EDHOC-Fuzzer sends either M2 or M3 then correctly the client responds with ERR_{E} (EDHOC error message) and the model transitions to state 3. Having responded with ERR_{E}, the client should have terminated the exchange session, which is not the case. Regardless of the EDHOC-Fuzzer's message, the client receives it and continues with deriving the OSCORE context and sending back an APP_{O}, which makes the model transition from state 3 to state 4. Note that an OSCORE context can be derived by both peers after receiving M3. Then the client waits for a reply to its OSCORE message and terminates in state 1. This alternative EDHOC exchange has been reported to the developers, who acknowledged the problem and promptly fixed it by making the client terminate the session after sending an ERR_{E} and waiting for a response. In the model this means that the transition from state 3 to state 4 changed to a transition from state 3 to state 1.

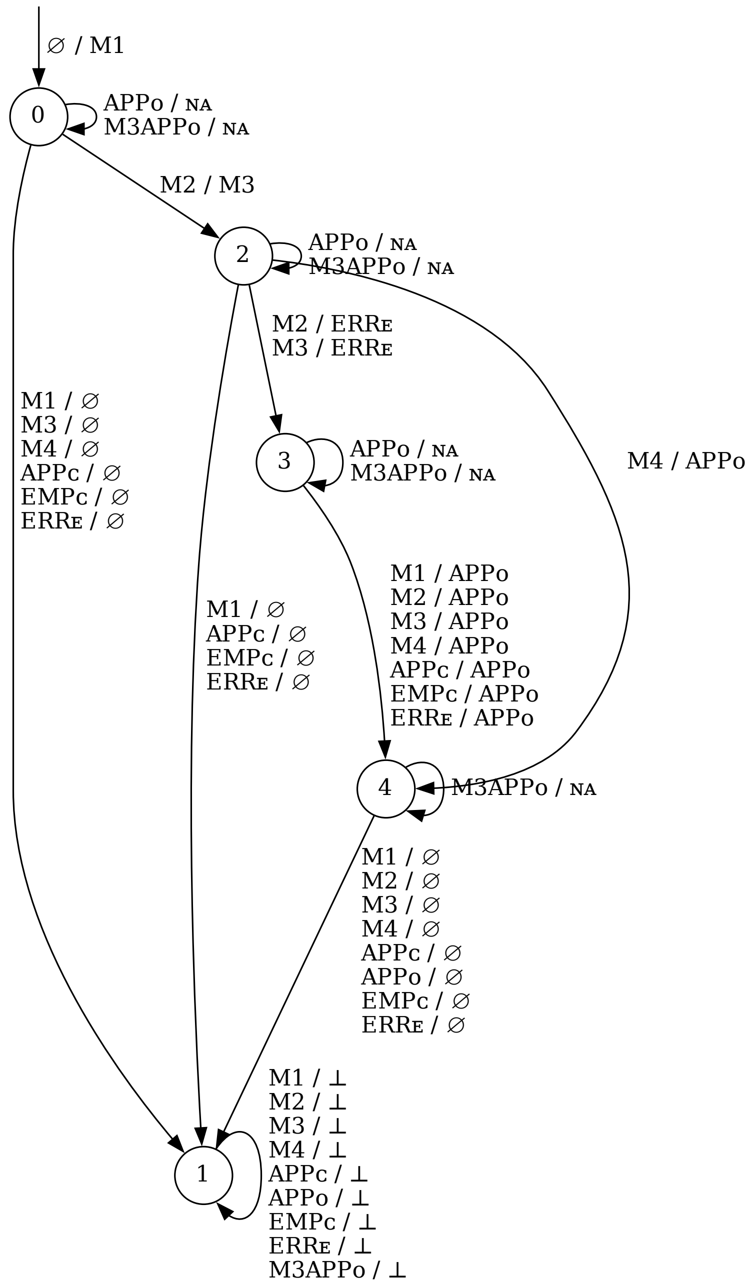


FIGURE 5.4: The learned model of the patched RISE client.

5.2.3 Default Server

The learned state machine model of the default server is shown in Fig. 5.5.

EDHOC Exchange. The EDHOC-Fuzzer, which acts as a client, sends the initial M1 and receives M2 from the server. Then the EDHOC-Fuzzer can send M3 and the server would reply with EMP_C or it can send M3APP_O and the server would reply with APP_O. In this model there are 5 different successful EDHOC exchanges. Specifically, (1) state sequence $0 \to 1 \to 2$, (2) state sequence $1 \to 1 \to 2$, (3) state sequence $2 \to 3 \to 2$, (4) state sequence $3 \to 3 \to 2$, (5) state sequence $4 \to 3 \to 2$. Additionally, a new EDHOC exchange can be started in the middle of a previous one. For example if the model has transitioned from state 0 to state 1 and the EDHOC-Fuzzer sends a M1 then a new exchange starts.

Plaintext Application Message. Notice that in every state if the EDHOC-Fuzzer sends a plaintext (non OSCORE-protected) message APP_C to the server's resource, then the server replies with APP_C. This happens, because the server's resource does not check if the received request is OSCORE protected.

Cancel Second EDHOC Exchange. A successful EDHOC exchange has completed, an OSCORE context has been derived and the model is in state 2. The EDHOC-Fuzzer sends M1 and receives back M2, and the model transitions to state 3. The EDHOC-Fuzzer delays the next message that the server expects, which is M3 or M3APP_O, and sends a M2 instead. The server replies with ERR_E, making the model transition to state 4 and terminates the current EDHOC exchange session. If now the EDHOC-Fuzzer tries to send the previously expected messages (M3 or M3APP_O) then the server replies with ERR_C, indicating that indeed the ongoing EDHOC exchange session has been terminated.

OSCORE Context. In state 2 both peers have derived the OSCORE context and are able to exchange OSCORE-protected messages. This OSCORE context will, typically, remain alive unless a new EDHOC exchange is completed and a new OSCORE context will be derived. The EDHOC-Fuzzer starts a new EDHOC exchange and sends M1 and receives M2, while the model transitions to state 3. The EDHOC-Fuzzer now sends an APP_O and receives back an APP_O as response; this OSCORE context is the same as before, because the current EDHOC exchange has not finished. If the EDHOC-Fuzzer sends ERR_E, the server does not reply and the model transitions to state 4. Again in state 4, the APP_O exchanges refer to the same OSCORE context as before. If, in state 4, the EDHOC-Fuzzer derives a new OSCORE context (overwriting the previous active one) after M3 or before M3APP_O and sends either of M3 or M3APP_O to the server, who has not derived this new OSCORE context, then the server would reply with ERR_C. Meanwhile the EDHOC-Fuzzer has overwritten its previously active OSCORE context and any further APP_O sent, results in a ERR_C, which means that the model has transitioned to state 0.

Note on the Mapper. The previous paragraph describes the transition from state 4 to state 0. This is an instance, where the Mapper's behavior inevitably affects the learned model. Specifically its behavior is to derive a new OSCORE context immediately after an EDHOC message_3 has been processed. Had the Mapper not overwritten its previous OSCORE context, then it is possible that this transition

would be a self-loop in state 4. However, if the transition from state 4 to state 0 had the normal server outputs, EMP_C and M3APP_O respectively, then this would mean that the server had not terminated its session.

Transition from state 2 to state 0. One interesting behavior of the server is the transition from state 2 to state 0. State 2 is the point where a successful EDHOC exchange has been completed and both peers have derived an OSCORE context, with which they may have already exchanged messages. If the EDHOC-Fuzzer now sends an ERR_E, the server does not reply anything back. Interestingly, the server processes this ERR_E, replies nothing but deletes the derived OSCORE context, which is shown from the ERR_C replies to the APP_O in state 0. The EDHOC-Fuzzer's ERR_E contains no other specific information to the previous session, than the prepended connection identifier, which makes the server terminate the encrypted connection between the two peers.

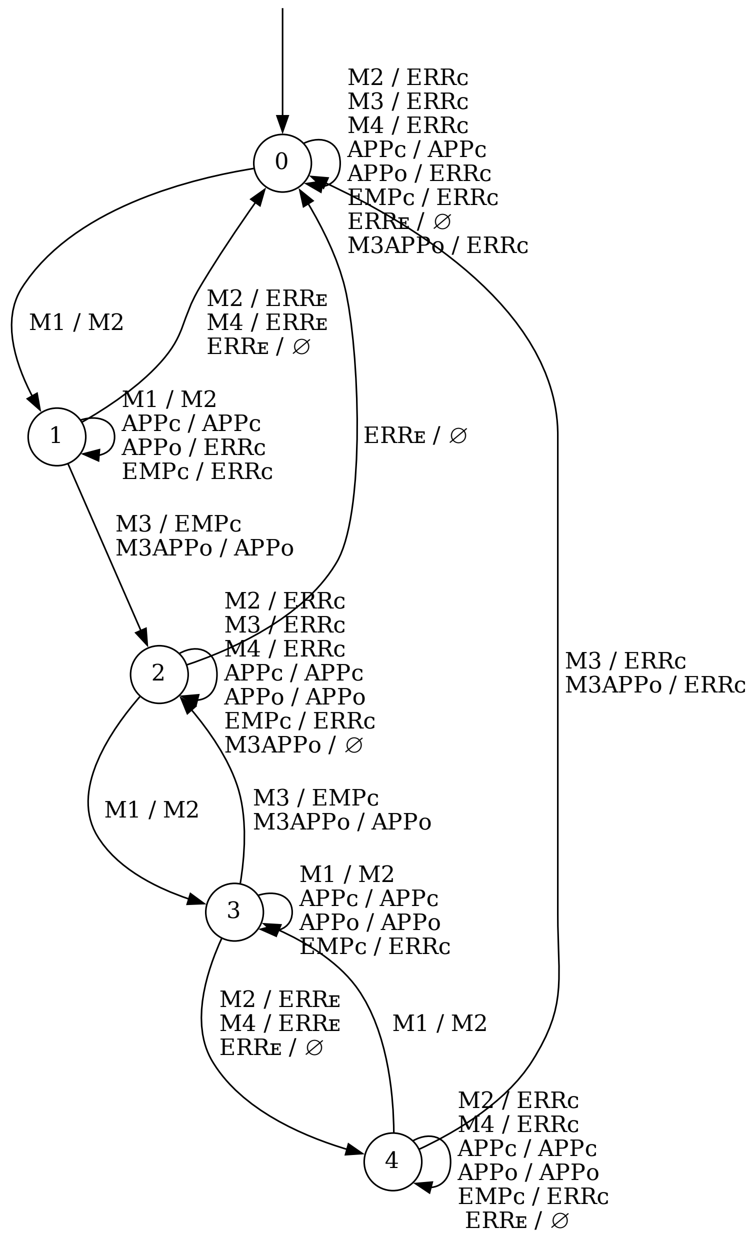


FIGURE 5.5: The learned model of the default RISE server.

5.2.4 Patched Server

Utilizing the additional functionality of the patch, the RISE server's model was learned when the EDHOC exchange had message_4 enabled and after the exchange OSCORE messages were also sent. Note that the patch did not alter the logic incorporated in the server. The learned state machine model of the patched server is shown in Fig. 5.6.

Resemblance to the Default Server Learned Model. This model is similar to the one in the default case, except that the EDHOC exchange is completed after the server sends M4 in response to the EDHOC-Fuzzer's M3 and now the server does not accept M3APP$_O$. All the other transitions are the same.

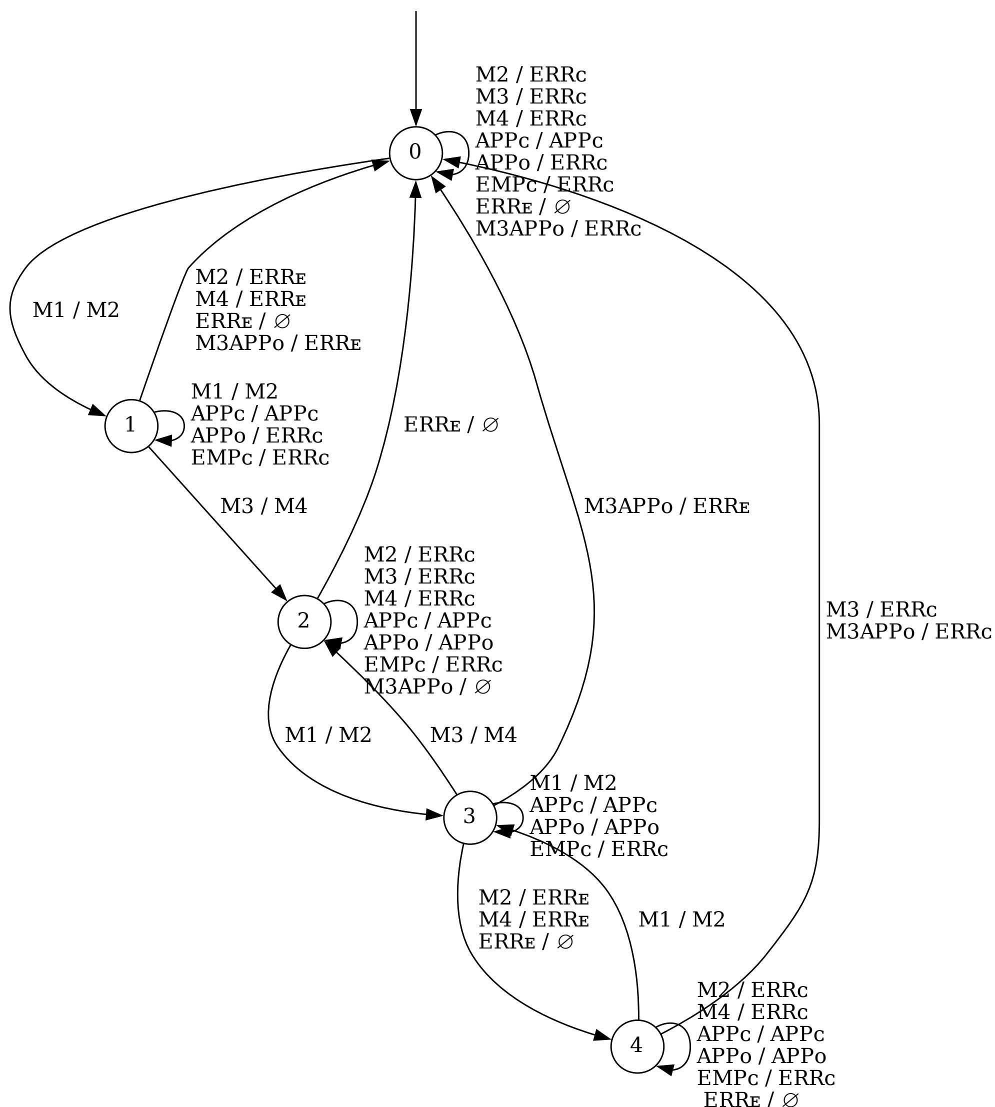


FIGURE 5.6: The learned model of the patched RISE server.

5.3 SIFIS-HOME

The project is available at https://github.com/sifis-home/wp3-solutions.
The commit hash 9956c8cf9a6f8cb3ab09e48842ceafeb9d2a790e identifies the version. The directory *edhoc-applications* contains the client and server implementations.

A patch has been applied to the clients, because they were interactive in nature and were waiting for user input. The patch disabled this interaction and made the clients send one OSCORE message after the EDHOC exchange.

This project is being developed by the same people as the previous RISE project. For this reason the EDHOC logic is the same and some learned models are very similar.

5.3.1 Default Client Phases 1, 2

The learned state machine model of the default clients of phases 1 and 2 are identical and are shown in Fig. 5.7.

EDHOC Exchange. The EDHOC-Fuzzer, which acts as a server, waits for the initial M1, sends M2 and receives M3. The client's model transitions to state 2. The EDHOC-Fuzzer's now sends either APP_C or EMP_C. The EDHOC exchange is completed, both peers have derived an OSCORE context and the client replies to the EDHOC-Fuzzer with an APP_O. The model transitions to state 3, in which the client waits for the EDHOC-Fuzzer's response to their APP_O. When they receive it they reply nothing back (annotated with $\varnothing$) and the model transitions to state 4.

Output NA in State 4. When the model transitions to state 4, the EDHOC-Fuzzer is unable to send any message. This happens, because under the hood the client sends CoAP requests to which the EDHOC-Fuzzer, acting as a server, replies with CoAP responses. The last output that the EDHOC-Fuzzer received, according to the transition from state 3 to state 4, is $\varnothing$, which means that no new client's request has been received, thus the EDHOC-Fuzzer is unable to respond and deems all further output messages as NA. This is why this type of transitions are not related to the SUL's behavior, but they are caused by the EDHOC-Fuzzer and are reflected in the learned models.

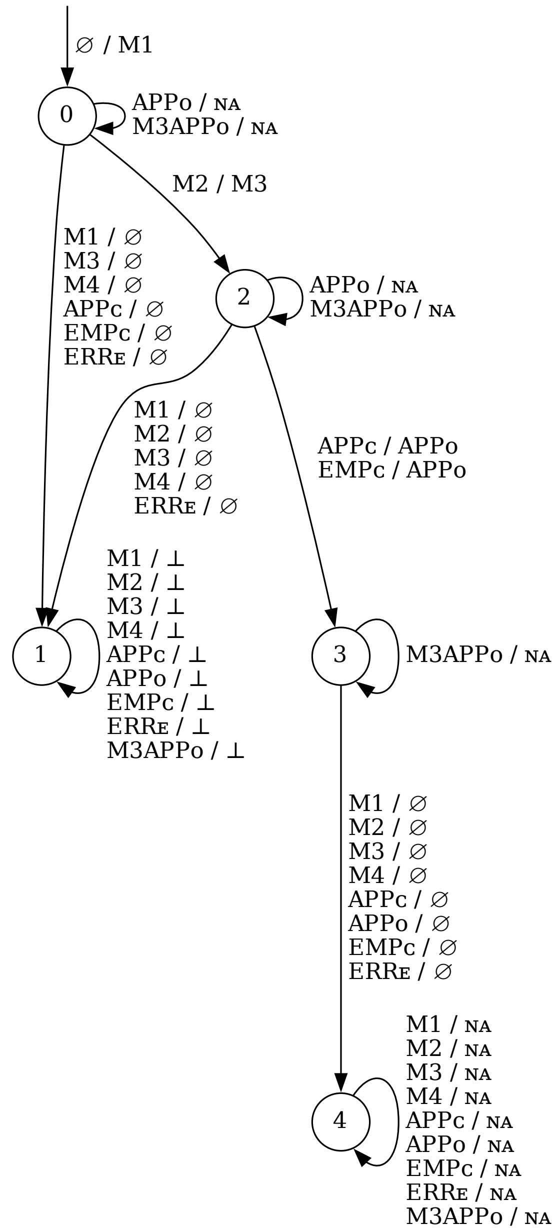


FIGURE 5.7: The learned model of the default SIFIS-HOME clients for phases 1 and 2.

5.3.2 Default Client Phases 3, 4

The learned state machine model of the default clients of phases 3 and 4 are identical and are shown in Fig. 5.8.

EDHOC Exchange. The EDHOC-Fuzzer, which acts as a server, waits for the initial M1 and then sends M2 to which the client replies with $M3APP_O$. The model transitions to state 2. The EDHOC exchange is completed and both peers have derived an OSCORE context. The client has already sent an OSCORE message (combined with the $M3APP_O$) and waits for the EDHOC-Fuzzer's reply to which it does not respond. The model transitions to state 1 and the client process terminates.

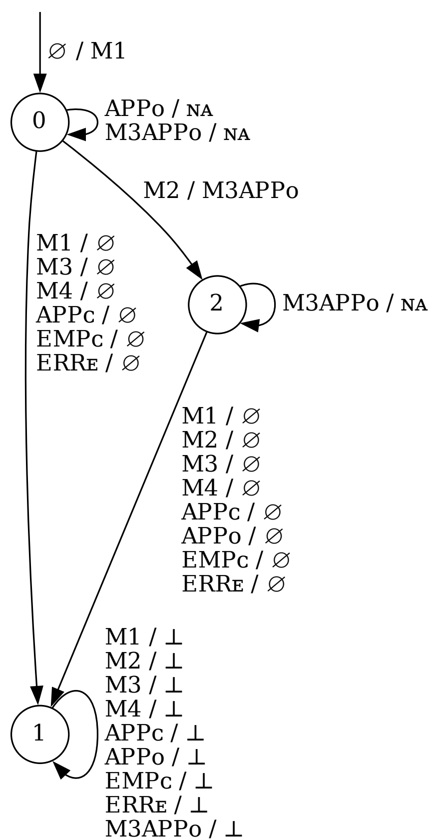


FIGURE 5.8: The learned model of the default SIFIS-HOME clients for phases 3 and 4.

5.3.3 Default Server Phases 1, 2, 3, 4

The learned state machine model of the default server implementations of phases 1, 2, 3 and 4 are identical and are shown in Fig. 5.9.

Resemblance to the RISE Default Server Learned Model. This model is similar to the one of the RISE default server in Fig. 5.5, except that the server's responses to a APP_C is ERR_C, which shows that the server's resource now rejects an incoming request if it is not OSCORE protected. All the other transitions are the same.

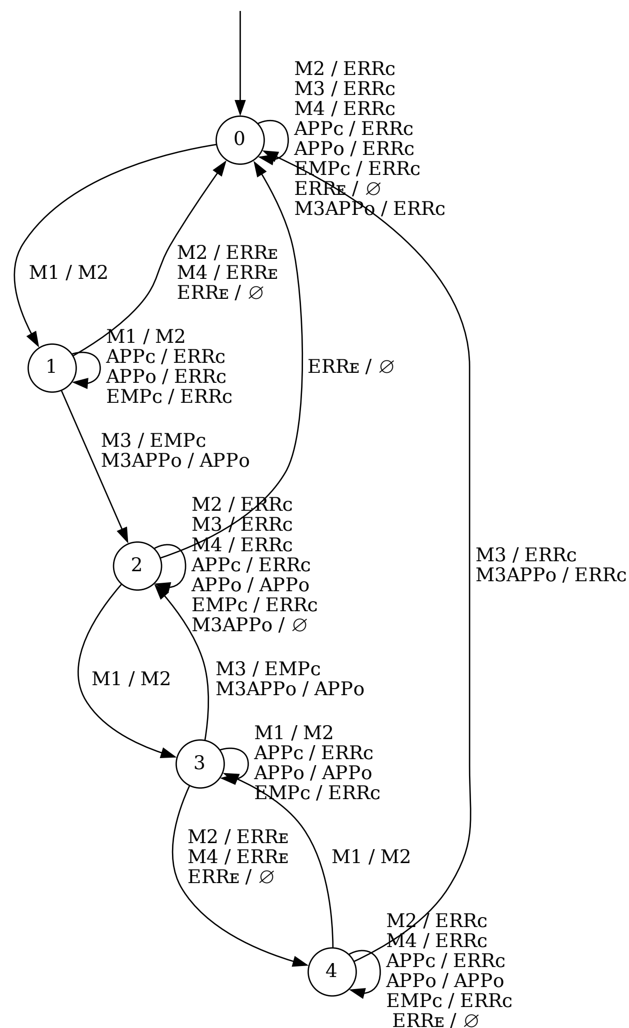


FIGURE 5.9: The learned model of the default SIFIS-HOME servers for phases 1,2, 3 and 4.

5.4 uOSCORE-uEDHOC

The project is available at https://github.com/eriptic/uoscore-uedhoc.
The commit hash fbaa96caa1a2028d369c70e24a173caa60a0ce15 identifies the version. The directory *samples* contains the client and server implementations.

In this project the prepended connection identifiers are not used.

5.4.1 Default Client EDHOC

The learned state machine model of the default client EDHOC is shown in Fig. 5.10.

EDHOC Exchange. The EDHOC-Fuzzer, which acts as a server, waits for the initial M1 from the client. Then sends M2 to the client and receives back M3. The model transitions from state 0 to state 1. This concludes the exchange for both peers. In this model there are only two states, because the client does not wait for a response to the M3. In other words, the client sends M3 and terminates.

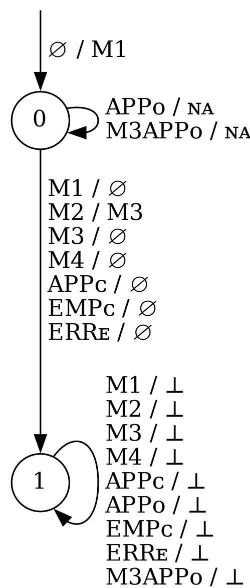


FIGURE 5.10: The learned model of the default uOSCORE-uEDHOC EDHOC client.

5.4.2 Default Client EDHOC and OSCORE

The learned state machine model of the default client EDHOC and OSCORE is shown in Fig. 5.11.

EDHOC Exchange. The EDHOC-Fuzzer, which acts as a server, waits for the initial M1 from the client. Then sends M2 to the client and receives back M3. The model transitions from state 0 to state 2. This concludes the exchange for both peers, who have also derived an OSCORE context.

State 2 and State 3 Loop. In state 2 both peers have derived an OSCORE context. Regardless of the EDHOC-Fuzzer's reply to client's M3, the client responds with APP_O and the model transitions to state 3. Then regardless of the EDHOC-Fuzzer's reply, the client responds with $\times$. The transitions from state 2 to state 3 and vice versa, indicates that the client is executing an infinite loop.

Output $\times$. The transition from state 3 to state 2 has an output message of $\times$. This happens, because the client makes a CoAP request to an EDHOC-Fuzzer's unavailable resource, which makes the EDHOC-Fuzzer reject that request. The rejection, for any reason, of a client's request is deemed as an $\times$ message.

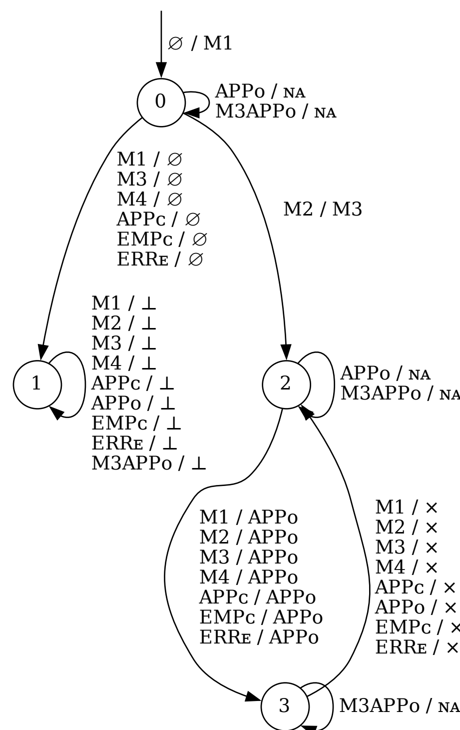


FIGURE 5.11: The learned model of the default uOSCORE-uEDHOC EDHOC and OSCORE client.

5.4.3 Default Server EDHOC

The learned state machine model of the default server EDHOC is shown in Fig. 5.12.

EDHOC Exchange. The EDHOC-Fuzzer, which acts as a client, sends the initial M1 and receives M2 from the server and the model transitions to state 1. The EDHOC-Fuzzer, now, sends either M3 or M3APP_O and the server does not reply anything, finishing the current EDHOC exchange.

Problem with M3APP_O. Since this server is used only for the EDHOC protocol, without deriving any OSCORE context, a combined message should not be accepted. However, this is not the case for this server, which accepts the message, without replying anything back. Note that M3APP_O does not have any prepended connection identifier and this server generally does not make use of them, so it is possible that the server processes only the EDHOC message_3 part of the M3APP_O and completes the protocol. This behavior raises the question of whether or not the APP_O part of the message is deliberately not processed by the server or is unintentionally skipped without knowing its presence.

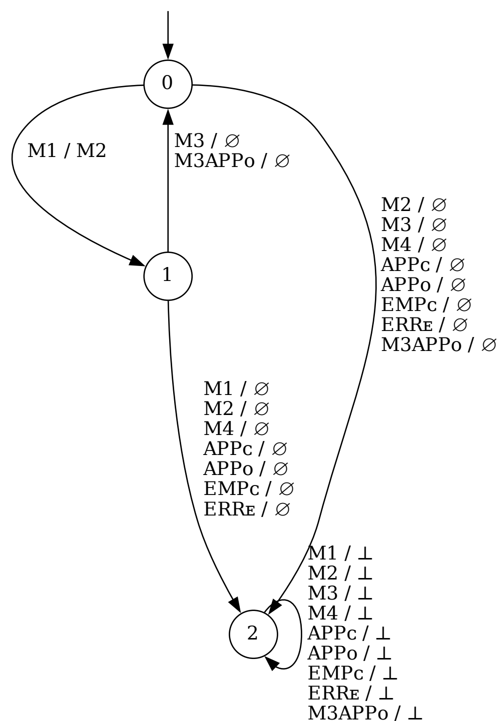


FIGURE 5.12: The learned model of the default uOSCORE-uEDHOC EDHOC server.

5.4.4 Default Server EDHOC and OSCORE

The learned state machine model of the default server EDHOC and OSCORE is shown in Fig. 5.13.

EDHOC Exchange. The EDHOC-Fuzzer, which acts as a client, sends the initial M1, receives M2 and the model transitions to state 1. If the EDHOC-Fuzzer sends M3 gets a MSG_C as response and if it sends $M3APP_O$ the server does not reply with anything and the model transitions to state 3, completing the EDHOC exchange.

Application Messages. In state 3, both peers have derived an OSCORE context and can exchange encrypted messages (APP_O). The server also accepts non encrypted messages (APP_C). To any other EDHOC-Fuzzer's messages, the server responds with a MSG_C. Actually this MSG_C is the same as the APP_C, but this instance makes apparent the difficulty of the Mapper to identify these two messages, discussed in Chapter 4.

Termination. In state 3, if the EDHOC-Fuzzer sends an $M3APP_O$ then the server does not reply back (∅) and it terminates afterwards. It is possible that the size of the $M3APP_O$ makes the server terminate unsuccessfully or crash due to an error.

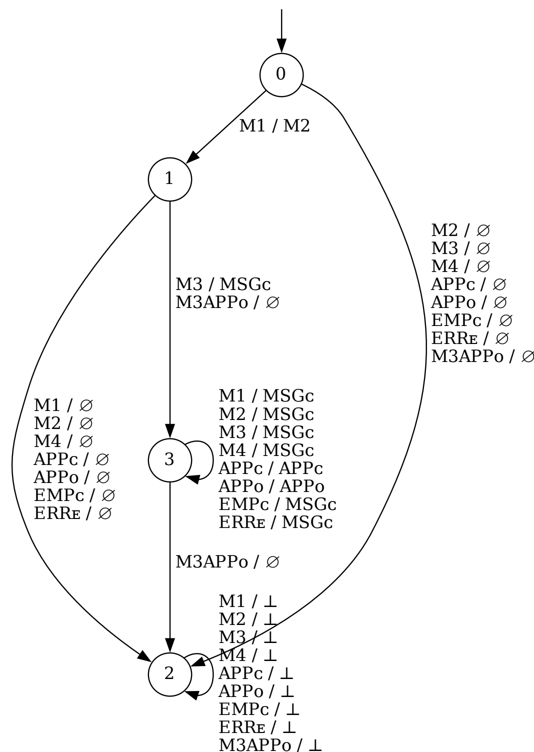


FIGURE 5.13: The learned model of the default uOSCORE-uEDHOC EDHOC and OSCORE server.

5.5 Automated Bug Detection

Even if those learned models contain a small number of states and transitions, automating the bug detection is equally important. Towards this goal, the tool SMBugFinder will be used [Fit+23; FB+22].

This tool implements a fully automatic black-box technique for detecting state machine bugs in implementations of stateful network protocols. This technique needs to be supplied with an (approximate) model of the implementation and a catalogue of the protocol's state machine bugs (or bug patterns) in the form of DFAs.

The supplied learned model is the one that the EDHOC-Fuzzer produces prior to any visual enhancement like transition merging and name shortening. The supplied bug patterns are DFAs augmented with some specific symbols recognized by the tool. SMBugFinder turns the supplied learned model, which is a Mealy Machine, into a DFA and for each bug pattern, represented as a DFA with special symbols, it creates a new complete DFA after the necessary modifications imposed by these symbols. Then SMBugFinder uses DFA intersection on the two DFAs (one of the learned model and the new one of the bug pattern) in order to search for the presence of the bug pattern in the learned model.

The syntax of the following bug patterns is not the one that the tool uses, there is some correspondence however. This is done so as to emphasize on the understanding of the DFAs in a higher-level and for the same reason the names of the messages are in their short form. The special symbol U in a transition from a state s to another state resembles all the available messages that are not in an outgoing transition of the state s. In addition, there are used some superscripts I, R when it is necessary to disambiguate whether the message comes from the Initiator or the Responder respectively.

5.5.1 Bug Patterns

Client Initiator keeps session alive after sending EDHOC error message. This DFA can be applied to a client Initiator and captures the bug that a client keeps its EDHOC session alive after sending an EDHOC error message. It is shown in Fig. 5.14.

The self-loop with the transition U in the start state, skips over all transitions until an Initiator's (outgoing) ERR_E is found, which is shown with the I superscript. Then the transition U refers to any input symbol being the response from the other peer to the ERR_E. The client should have terminated the session, but if the client's output is other than $\varnothing$, $\perp$ or $?$ then this is a bug, which means that the client responds in the same session.

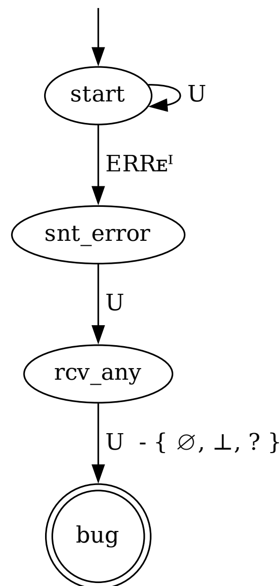


FIGURE 5.14: Bug pattern that captures a client Initiator, when it keeps the session alive after sending an EDHOC error message.

Server exchanges OSCORE messages and is forced to terminate. It is usual from a server (regardless of being EDHOC Initiator or Responder), when a successful EDHOC exchange has been completed and an OSCORE context has been derived, that it remains active and replies to further OSCORE requests. It is quite unusual for a server to terminate after an unrelated input has been received, unless some type of crash has occured. This bug intends to capture the DFA in Fig. 5.15.

In the DFA all transitions (if any) are skipped until an input APP_O is received by the server, to which it replies back with an APP_O. Then if any input that the server receives causes it (after a series of transitions) to terminate, is captured by this DFA.

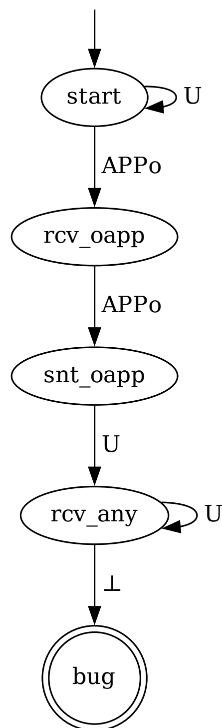


FIGURE 5.15: Bug pattern that captures a server Initiator or Responder, when after an OSCORE exchange some sequence of inputs causes it to terminate.

Server's OSCORE resource responds to plaintext messages. Usually a server (regardless of being EDHOC Initiator or Responder) has some CoAP resources that are OSCORE protected, meaning that only an OSCORE protected message gets processed, otherwise an error message is returned. However, some implementations may not return an error message, but treat the received message in any other way. The DFA shown in Fig. 5.16, captures the behavior of a server that returns some response other than error.

In the DFA all transitions (if any) are skipped until an input APP_C to the implementation is found. This APP_C is sent to a specified resource to which OSCORE messages are also sent. Then if the server's output is something other than error message (or other special output), the desired bug is found.

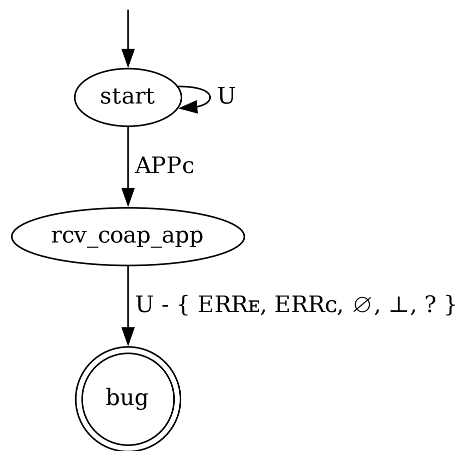


FIGURE 5.16: Bug pattern that captures a server Initiator or Responder, when its resource intended for OSCORE messages replies also to plaintext messages.

Initiator without the required message_4 sends OSCORE message. This pattern can be applied to models, where the SUL is an EDHOC Initiator, a message_4 is required during EDHOC and an OSCORE message is exchanged afterwards from the SUL. This OSCORE message is used to signal that the EDHOC exchange has been completed and an OSCORE context has been derived. The DFA is shown in Fig. 5.17.

In the DFA all transitions (if any) are skipped until an input M2 to the Initiator is found. The Initiator then responds with M3. Afterwards, the DFA captures the behavior of the Initiator that without receiving the required M4, it manages to derive an OSCORE context and send an APP_O. Essentially, the Initiator ignores the fact that no M4 is received and completes successfully the EDHOC exchange.

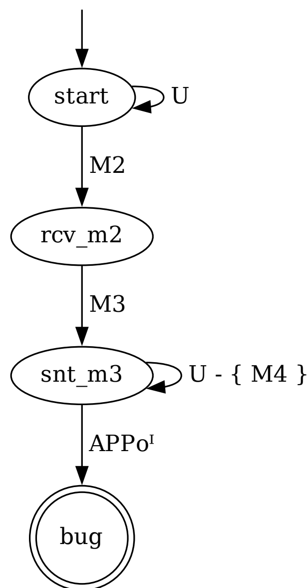


FIGURE 5.17: Bug pattern that captures an EDHOC Initiator, when it completes the exchange without the required message_4, derives OSCORE context and sends back an OSCORE message.

Receiving EDHOC error message harms the derived OSCORE context. This pattern can be applied to both EDHOC Initiator and Responder models and is shown in Fig. 5.18.

In the DFA all transitions (if any) are skipped until the SUL receives an input APP_O, and replies back with an APP_O. This means that an EDHOC exchange has been successfully completed and OSCORE messages are exchanged. If the SUL receives an ERR_E, normally this should not cause any harm to the previously established OSCORE context, which means that if the SUL receives an APP_O it should respond to it. However the bug is found, when the response to the received APP_O is other than an APP_O.

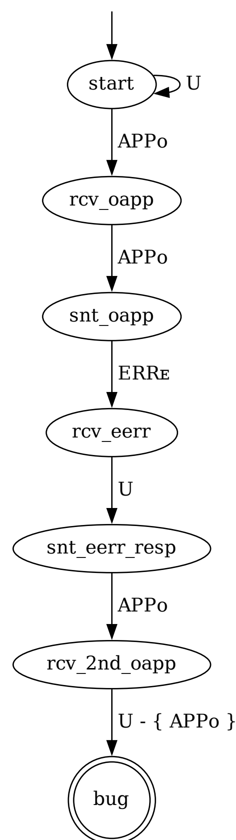


FIGURE 5.18: Bug pattern that captures an SUL, when receiving an EDHOC error message harms the active OSCORE context.

Chapter 6

Conclusion

This thesis begins with an overview of the necessary theoretical background. Afterwards, follow the details regarding the EDHOC-Fuzzer, which is the main contribution of this thesis. These details mainly concern the missing pieces that complete a tool capable of performing protocol state fuzzing on EDHOC implementations. Next, the ProtocolState-Fuzzer's implementation details and internal composition are described. Finally, the experiments that have been conducted in several implementations are presented. These consist of the learned models that EDHOC-Fuzzer was able to generate alongside with a thorough analysis of those models. Additionally, a subset of defined bug patterns has been introduced that can complement the visual inspection of the models by automating the search of certain classes of bugs in the learned models.

Although, in average the learned models consist of few states and transitions, it is very interesting that a variety of behaviors are observed. This naturally stems from the fact that different groups of people take certain design decision and develop their implementations differently. The practical challenge of such tools as EDHOC-Fuzzer is the ability to successfully interact with many implementations and not be limited to certain ones. Moreover, the fact that the learning is successful and the analysis of the learned models can uncover well-hidden edge cases and logical flaws, indicates the importance of these tools, such as EDHOC-Fuzzer, especially for the people that are tasked with implementing those protocols.

6.1 Future Work

Regarding EDHOC-Fuzzer, there are several options in need of exploration. The first one is quite obvious and concerns the continuous improvement of the tool and support of other implementations as they will start to emerge. This type of compatibility testing with new implementations not only would increase the preciseness and robustness of the tool itself, but also would improve the chances of being used by others as an effective tool for their previously unseen implementation. The second option that could be explored is the addition of certain concrete details in input and output symbols. These could provide better insights in the observed behavior of the implementations. A third option is to extend the capabilities of EDHOC-Fuzzer towards fuzzing in general, which can uncover certain classes of bugs that protocol state fuzzing is unable to do so.

Regarding ProtocolState-Fuzzer, a viable future direction would be to release it as a standalone tool that will act as a framework for many different protocol state fuzzers. This would reduce the implementation burden of a large part of the learning setup, while offering a large degree of extensibility and customization. Another possible, yet more radical, direction would be the model learning to be performed also by other methods, such as passive learning or other proposed techniques that would need their own learning setup. However, it is debatable if these features would fit well in ProtocolState-Fuzzer or they would be better offered in their own standalone tool.

Bibliography

[DPR18] Lesly-Ann Daniel, Erik Poll, and Joeri de Ruiter. “Inferring OpenVPN State Machines Using Protocol State Fuzzing”. In: *2018 IEEE European Symposium on Security and Privacy Workshops (Euros&PW)*. London, U.K.: IEEE, Apr. 2018, pp. 11–19. DOI: 10.1109/EuroSPW.2018.00009. URL: https://doi.org/10.1109/EuroSPW.2018.00009.

[FB+17] Paul Fiterău-Broştean et al. “Model Learning and Model Checking of SSH Implementations”. In: *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*. SPIN 2017. New York, NY, USA: ACM, 2017, 142–151. DOI: 10.1145/3092282.3092289. URL: https://doi.org/10.1145/3092282.3092289.

[FB+20] Paul Fiterau-Brostean 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.

[FB+22] Paul Fiterău-Broştean et al. *Artifact for the Paper “Automata-Based Automated Detection of State Machine Bugs in Protocol Implementations”*. Version 1.0.0. Sept. 2022. DOI: 10.5281/zenodo.7129240. URL: https://doi.org/10.5281/zenodo.7129240.

[FBJV16] Paul Fiterău-Broştean, Ramon Janssen, and Frits Vaandrager. “Combining Model Learning and Model Checking to Analyze TCP Implementations”. In: *International Conference on Computer Aided Verification*. Vol. 9780. LNCS. Springer, 2016, pp. 454–471. DOI: 10.1007/978-3-319-41540-6_25. URL: https://doi.org/10.1007/978-3-319-41540-6_25.

[Fer+21] Tiago Ferreira et al. “Prognosis: Closed-Box Analysis of Network Protocol Implementations”. In: *Proceedings of the 2021 ACM SIGCOMM Conference*. SIGCOMM ’21. New York, NY, USA: ACM, 2021, pp. 762–774. DOI: 10.1145/3452296.3472938. URL: https://doi.org/10.1145/3452296.3472938.

[Fit+22] 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.

[Fit+23] Paul Fiterau-Brostean et al. “Automata-Based Automated Detection of State Machine Bugs in Protocol Implementations”. In: *Network and Distributed System Security Symposium*. NDSS 2023. San Diego, CA, USA: The Internet Society, Feb. 2023. URL: https://www.ndss-symposium.org/wp-content/uploads/2023/02/ndss2023_s68_paper.pdf.

[IHS15] Malte Isberner, Falk Howar, and Bernhard Steffen. "The Open-Source LearnLib - A Framework for Active Automata Learning". In: *Computer Aided Verification - 27th International Conference, CAV*. Vol. 9206. LNCS. Springer, 2015, pp. 487–495. DOI: 10.1007/978-3-319-21690-4_32. URL: https://dx.doi.org/10.1007/978-3-319-21690-4_32.

[Nei+19] Daniel Neider et al. "Benchmarks for Automata Learning and Conformance Testing". In: *Models, Mindsets, Meta: The What, the How, and the Why Not? Essays Dedicated to Bernhard Steffen on the Occasion of His 60th Birthday*. Ed. by Tiziana Margaria, Susanne Graf, and Kim G. Larsen. Cham: Springer International Publishing, 2019, pp. 390–416. ISBN: 978-3-030-22348-9. DOI: 10.1007/978-3-030-22348-9_23. URL: https://doi.org/10.1007/978-3-030-22348-9_23.

[RAR19] Abdullah Rasool, Greg Alpár, and Joeri de Ruiter. "State machine inference of QUIC". In: *arXiv preprint arXiv:1903.04384* (2019).

[RFC7252] Zach Shelby, Klaus Hartke, and Carsten Bormann. *The Constrained Application Protocol (CoAP)*. Tech. rep. 7252. June 2014. 112 pp. DOI: 10.17487/RFC7252. URL: https://www.rfc-editor.org/info/rfc7252.

[RFC8613] Göran Selander et al. *Object Security for Constrained RESTful Environments (OSCORE)*. Tech. rep. 8613. July 2019. 94 pp. DOI: 10.17487/RFC8613. URL: https://www.rfc-editor.org/info/rfc8613.

[RFC8949] Carsten Bormann and Paul E. Hoffman. *Concise Binary Object Representation (CBOR)*. Tech. rep. 8949. Dec. 2020. 66 pp. DOI: 10.17487/RFC8949. URL: https://www.rfc-editor.org/info/rfc8949.

[RFC9052] Jim Schaad. *CBOR Object Signing and Encryption (COSE): Structures and Process*. Tech. rep. 9052. Aug. 2022. 66 pp. DOI: 10.17487/RFC9052. URL: https://www.rfc-editor.org/info/rfc9052.

[RP15] Joeri de Ruiter and Erik Poll. "Protocol State Fuzzing of TLS Implementations". In: *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C., USA: USENIX Association, Aug. 2015, pp. 193–206. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/de-ruiter.

[Rui16] Joeri de Ruiter. "A tale of the OpenSSL state machine: A large-scale black-box analysis". In: *Nordic Conference on Secure IT Systems*. Vol. 10014. Cham: Springer, 2016, pp. 169–184. DOI: 10.1007/978-3-319-47560-8_11. URL: https://doi.org/10.1007/978-3-319-47560-8_11.

[SMP23] Göran Selander, John Preuß Mattsson, and Francesca Palombini. *Ephemeral Diffie-Hellman Over COSE (EDHOC)*. Internet-Draft draft-ietf-lake-edhoc-19. Work in Progress. Internet Engineering Task Force, Feb. 2023. 108 pp. URL: https://datatracker.ietf.org/doc/draft-ietf-lake-edhoc/19/.

[TAB17] Martin Tappler, Bernhard K Aichernig, and Roderick Bloem. "Model-Based Testing IoT Communication via Active Automata Learning". In: *2017 IEEE International conference on software testing, verification and validation (ICST)*. Tokyo, Japan: IEEE Computer Society, Mar. 2017, pp. 276–287. DOI: 10.1109/ICST.2017.32. URL: https://doi.org/10.1109/ICST.2017.32.

[Vaa17] Frits Vaandrager. “Model Learning”. In: *Commun. ACM* 60.2 (Jan. 2017), 86–95. ISSN: 0001-0782. DOI: 10.1145/2967606. URL: https://doi.org/10.1145/2967606.

[Vel17] Bart Veldhuizen. *Automated state machine learning of IPsec implementations*. Bachelor Thesis, Computer Science, Radboud University, The Netherlands. Aug. 2017. URL: https://www.cs.ru.nl/bachelors-theses/2017/Bart_Veldhuizen___4492765___Automated_state_machine_learning_of_IPsec_implementations.pdf.

[Wen20] Cheng Wen. *A Quick Survey of Active Automata Learning*. Accessed: 14 January 2023. Mar. 2020. URL: https://wcventure.github.io/Active-Automata-Learning/.