Applying Symbolic Execution to Test Implementations of a Network Protocol Against its Specification

Hooman Asadian, Paul Fiterau-Brostean, Bengt Jonsson, Konstantinos Sagonas

UPPSALA UNIVERSITET

Introduction

- Testing correctness of network protocol implementations is essential
- A successful software testing technique is **symbolic execution**
  - ⇒ However, it is not so effective at testing stateful systems

This work:

- Presents a methodology that makes symbolic execution effective in
  - Testing network protocol implementations, and
  - Exposing requirement violations using assumptions and assertions
- Applies this methodology to implementations of the DTLS protocol
  - Revealing numerous new security vulnerabilities and bugs in them

UPPSALA
UNIVERSITET

Methodology

1. Extract Specification Requirements
   - Represent the requirements by logical formulas
2. Augment the SUT with assumptions and assertions
   - Assume inputs under which a requirement can be violated
   - Assert that no forbidden action is performed
3. Symbolic Execution
   - Explores the paths in the augmented SUT
4. Test Case Construction and Validation
   - Confirm the bug on the unmodified SUT

1- Extract Specification Requirements

- Requirements from the protocol RFC are identified by particular keywords:
  - MUST, MUST NOT, SHOULD, SHOULD NOT, …
- Two types of requirements are extracted:
  - Input validity requirements
  - Input-output requirements
- Represent the requirements by logical formulas

Input Validity Requirements

- E.g., the DTLS 1.2 RFC states:

"For each received record, the receiver MUST verify that the record contains a sequence number that does not duplicate the sequence number of any other record received during the life of this session."

- For a set of Records R, received during a DTLS session:

$$\forall\, r, r' \in R:\ r \neq r' \Longrightarrow r.sequence\_number\ \neq\ r'.sequence\_number$$

2- Augment the SUT with Assumptions

$$\forall\, r, r' \in R\colon\ r \neq r' \Longrightarrow r.sequence\_number \neq r'.sequence\_number$$

$$R = \{CH2, CKE, CCS\}$$

Pair-wise conjunction

Negation

$$\begin{aligned} assume\ (!\,(\ & CH2.sequence\_number\ != CKE.sequence\_number\ \&\\ & CH2.sequence\_number\ != CCS.sequence\_number\ \&\\ & CKE.sequence\_number\ != CCS.sequence\_number\ )) \end{aligned}$$

2- Augment the SUT with Assertions

- Add an assert statement to check if the implementation of the protocol uses invalid input in some forbidden way

- E.g., the DTLS 1.2 RFC:

“Invalid records SHOULD be silently discarded …”

- Check whether progress occurs after reception of invalid records
  - Approximate this by successful completion of protocol interaction
  - Add failing assertion

3- Symbolic Execution

- Exploring the paths in the augmented SUT looking for assertion violation, crashes, memory errors, etc

- To achieve scalability:
  - Only make symbolic the relevant fields in a requirement
  - Other fields are given concrete values from a pre-captured session
  - Check one requirement at a time

- To ensure deterministic execution of the SUT:
  - De-randomize the SUT

4- Test Case Construction and Validation

- For each path, the tool returns:
  - A tuple of values for the symbolic fields

- For the sequence number experiment, we will have concrete values for $sequence\_number$ in the participating records

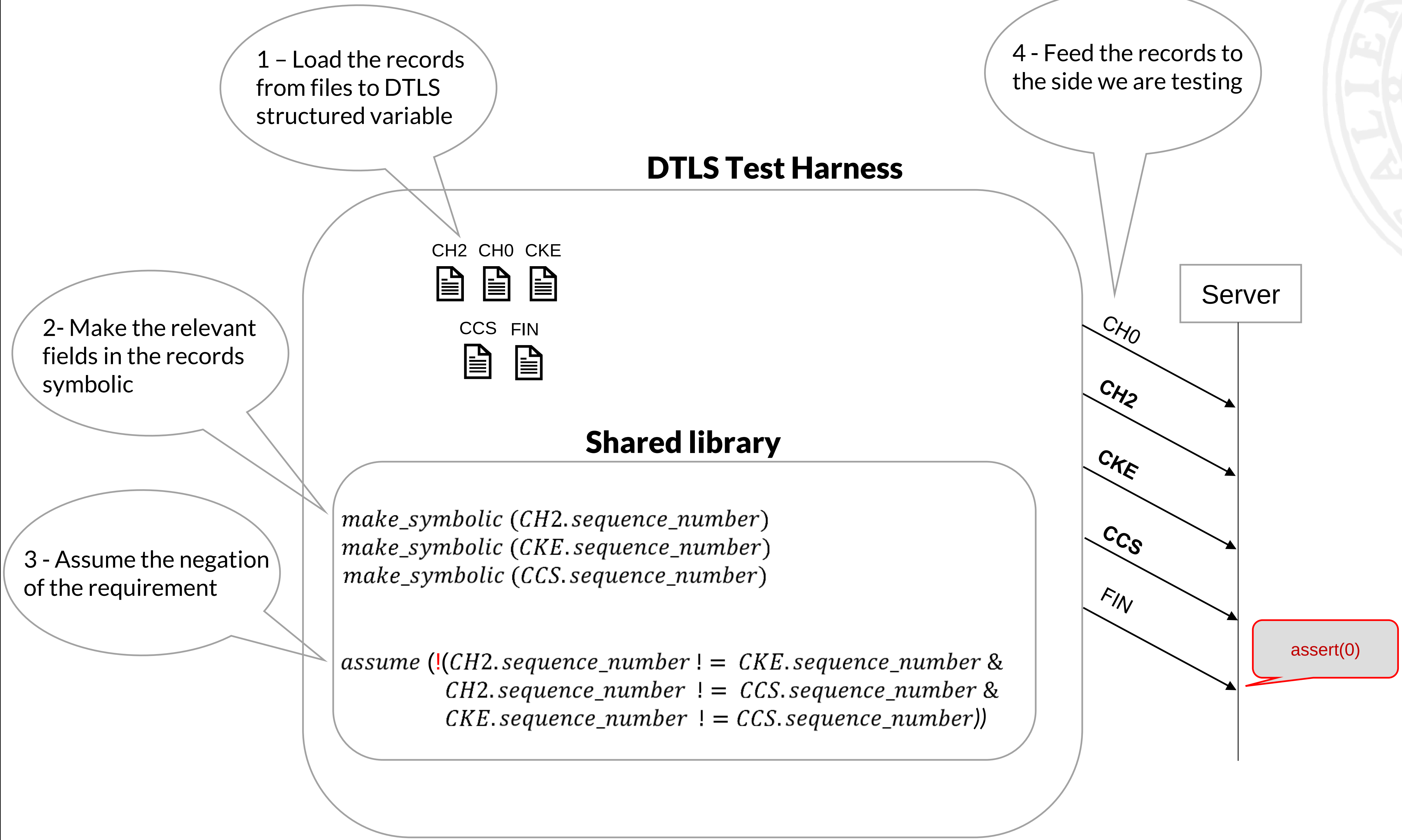
- For concrete values that cause bugs:
  - Assign concrete values to relevant fields
  - Validate the bug by running the resulting test cases on the unmodified SUT

Implementation and Application to DTLS

- Used KLEE as the symbolic execution engine

- Built a test harness that:
  - Captures the records a client and server exchange during a session
  - Is used to symbolically execute the SUT in order to check each requirement

- We implemented a shared library to facilitate test harness construction. It contains:
  - Helper functions
  - DTLS packet parser
  - Functions to make specific fields of records symbolic and to form Boolean expression in *assumes* and *asserts*


1 – Load the records from files to DTLS structured variable
4 - Feed the records to the side we are testing
DTLS Test Harness
CH2 CH0 CKE
CCS FIN
Server
2- Make the relevant fields in the records symbolic
Shared library
CH0
CH2
CKE
CCS
FIN
$make\_symbolic\ (CH2.sequence\_number)$
$make\_symbolic\ (CKE.sequence\_number)$
$make\_symbolic\ (CCS.sequence\_number)$
3 - Assume the negation of the requirement
$assume\ (!(CH2.sequence\_number\ != CKE.sequence\_number\ \&$
$CH2.sequence\_number\ != CCS.sequence\_number\ \&$
$CKE.sequence\_number\ != CCS.sequence\_number))$
assert(0)

Evaluation

- We tested 4 DTLS libraries against 16 requirements:
  - 36 unique bugs
  - 7 vulnerabilities of which 6 are new

	OpenSSL		Mbed TLS	TinyDTLS^E		TinyDTLS^C
	1.0.1f	3.0.0-alpha12	2.22.0	7068882	94205ff	53a0d97
Vulnerability	1	1	–	3	3	2
Other	–	–	–	3	4	1
Non-conformance	2	2	3	9	10	10

TinyDTLS Reassembly Bug

- The DTLS 1.2 RFC specifies:

> *"When a DTLS implementation receives a handshake message fragment, it MUST buffer it until it has the entire message"*

- Memory over-read when client/server reassemble a fragmented message
  - Occurs if the fragment length field is greater than the size of the actual fragment

- Three pull request attempts before the bug was fixed

KLEE Experiences

- Protocol implementations define incoming/outgoing buffers sizes with respect to the Maximum Transmission Unit (MTU)
  - Memory over-read/over-write bugs can be missed by KLEE
  - Our solution: Allocate memory dynamically with respect to the size of the actual packets

- Significant interpretation slowdown when functions in cryptographic libraries are executed
  - Even in the absence of symbolic variables
  - Provided a benchmark in issue *#1255* (700% slowdown)
  - (Partial) solution: Execute the functions as an external call

Conclusion

Methodology

1. Extract Specification Requirements
   - Represent the requirements by formulas
2. Augment the SUT with assumptions and assertions
   - Assume inputs under which a requirement can be violated
   - Assert that no forbidden action is performed
3. Symbolic Execution
   - Explores the paths in the augmented SUT
4. Test Case Construction and Validation
   - Confirm the bug on the unmodified SUT

UPPSALA UNIVERSITET

Evaluation

- We tested 4 DTLS libraries against 16 requirements:
  - 36 unique bugs have been found
  - 7 vulnerabilities of which 6 are new

	OpenSSL		Mbed TLS	TinyDTLS^E		TinyDTLS^C
	1.0.1f	3.0.0-alpha12	2.22.0	7068882	94205ff	53a0d97
Vulnerability	1	1	–	3	3	2
Other	–	–	–	3	4	1
Non-conformance	2	2	3	9	10	10

UPPSALA UNIVERSITET

Thank You for Listening

Replication materials available at:
https://zenodo.org/record/5929867#.YkS3HSjMJaT

UPPSALA UNIVERSITET