On the Adequacy of Statecharts as a Source of Tests for Cryptographic Protocols *

K. R. Jayaram and Aditya P Mathur
Department of Computer Science, Purdue University
W. Lafayette, IN 47907, USA
{jayaram,apm}@purdue.edu

Abstract

The effectiveness of statecharts as a tool to express the desired behavior of security protocols and a source of tests for their implementations was investigated. Specifically, TLS protocol was modeled as a statechart and tests generated from its flattened version. The GnuTLS implementation of the protocol was then tested against the generated tests. The MC/DC coverage of different components of the implementation varied from 51% to 81%. A "what if" analysis revealed that while some defects in the uncovered code will not lead to any security vulnerability due to in-built fault tolerance, others might lead to improper authentication, integrity failure, session hijacking, denial of service, and loss of confidentiality. The analysis suggests that statecharts alone might not be an adequate tool as a source of tests for implementations of security protocols and that tests so generated must be augmented through other formal means such as random testing, stress testing, and code coverage analysis.

Keywords: Statechart, Security protocol, TLS protocol, MC/DC coverage, Security Vulnerability.

1 Introduction

The pervasive use of the internet, mobile devices and wireless networks for electronic commerce, banking, private communication, military command-and-control etc. has increased the extent to which users are forced to rely on cryptographic protocols. Users should be able to justifiably rely on their implementations to process, store, and communicate sensitive information securely.

Information security is constantly endangered by errors in the protocol implementations. Examples of implementation errors leading to security vulnerabilities include: (i) buffer overflows and race conditions in Kerberos implementations which have resulted in an attacker gaining root access to the Key Distribution Server [12]; (ii) IPSec vulnerability [10]; (iii) a man-in-the middle attack on the OpenSSL library forcing the usage of insecure SSL 2.0 protocol even if both the ends support SSL 3.0 [16]. Flaws in the implementations of protocols can lead to undesirable consequences such as the disclosure of private information, identity theft, and fraudulent financial transactions.

While there has been significant research on the effectiveness of Model Based Testing (MBT) techniques for software, there has been little work on the efficacy of these techniques in security testing. The focus of the current study is the effectiveness of tests generated from statecharts [8] in identifying security vulnerabilities in cryptographic protocol implementations. The entire study consists of the following sequence of steps.

- Step 1: Select a suitable security protocol, examine its specification, and model the expected behavior of the implementation as a statechart.
- Step 2: Generate tests from the flattened statechart [1] using the testing tree method [5].
- Step 3: Select an open source implementation that is intended to conform to the selected protocol specification in Step 1, compile it using a suitable coverage measurement tool, execute it against the tests generated in Step 1, and find the MC/DC coverage [13] of the entire test suite.
- Step 4: Select suitable uncovered blocks in the implementation, create erroneous versions by injecting well known errors made by programmers, and analyze possible security failures.
- Step 5: Generate "negative tests" that were ignored in Step 1 due to statechart flattening and execute the original implementation against these. Determine the value of negative tests in terms of the increase in MC/DC coverage.

*This work was supported in part by a grants from the Indiana 21st Century Fund, Arxan Technologies, and the Department of Defense, US Army Research Office.

The following are the main contributions of this work: (1) A quantitative assessment of the adequacy of tests generated from statechart model of a security protocol. (2) An approach to reduce the size of statechart models for security protocols. (3) The impact of errors in uncovered portions of a security protocol.

The subjects, tools, and procedure used in this empirical study are described in Section 2. Data obtained from the experiment are presented in Section 3. Analysis of the results is presented in Section 4. Lastly, we discuss the implications of this work, recommendations and weaknesses of the reported study in Sections 5 and 6.

2 Method

2.1 Step 1: Modeling cryptographic protocols

The formalism chosen should have the necessary expressivity to capture all aspects of a cryptographic protocol, support model-based testing, and be easy to use. Some important characteristics of security protocols follow.

1. A security protocol involves concurrent principals (processes), and each principal may in turn have concurrent threads of computation.
2. Each principal is deterministic, i.e. each thread of computation is deterministic.
3. Principals may have memory. For example, a security protocol may involve principal A sending principal B a random number r and expecting $f(r)$ in return (where f is some function).
4. Protocols may involve decision making and looping based on internal memory, i.e. they may use internal variables to loop and branch.
5. Protocols may involve internal computation, e.g. computation of keys from master secrets and random numbers, data compression, etc.

Two visual formalisms that support requirements 1–3 above are statecharts [9] and Finite State Machines (FSM). However, FSMs do not support requirements 4 and 5. Consequently, statecharts appear to be a natural choice to model security protocols. Statecharts are part of UML [2] and its various extensions like UML-RT (UML-Real Time) and UMLSec (UML for Security Protocols).

2.2 Step 2: Generation of tests from the statechart

Generation of tests directly from a set of concurrent statecharts will likely lead to state explosion [1] especially in the case of complex requirements, such as for the TLS protocol. We decided to use the testing tree method for test generation. As described in [5], the testing tree construction is part of the W method. Note that while the W-method described by Chow uses testing tree as well as the characterization set W obtained from an FSM model, we decided not to use the W set in this experiment.

The W method, when applied to statecharts [1] considers all possible interleavings between concurrent states when generating test cases. Hence if there is an AND state in a statechart having m and n states, respectively, in each concurrent thread, the W method uses the product construction to generate *mn* states. This may not be necessary because some interleavings may be infeasible. As an example, consider a simple security protocol in Figure 1 (a) between a client and a server (this statechart is used only to illustrate infeasible states when doing product construction).

In the sample protocol, the server authenticates itself to the client through its certificate. The two then exchange a couple of random numbers encrypted by the server's public-private key pair. At the end of this exchange, both the client and the server agree on a session key as a function of the two random numbers.

Figure 1(b) explains how infeasible paths and states are determined. Product construction proceeds as normal, but once a transition and resulting state S are determined to be infeasible, all further product construction from S is terminated. In Figure 1, the transition $[C1; S1] \rightarrow [C1; S2]$ is feasible, but $[C1; S1] \rightarrow [C2; S2]$ is not. Note that product construction is a rooted tree, and hence if a node is infeasible, all derivations from that node and hence the subtree containing that node are infeasible. $[C1; S1] \rightarrow [C2; S2]$ is infeasible because unless the message from the server is received, $C1 \rightarrow C2$ is infeasible. So the right sequence should be $[C1; S1] \rightarrow [C1; S2] \rightarrow [C2; S2]$. Consequently, all derivations from $[C2; S2]$ are infeasible.

Similarly, the transition $[C2; S2] \rightarrow [C3; S3]$ is not feasible, the proper sequence should be $[C2; S2] \rightarrow [C3; S2] \rightarrow [C3; S3]$. Similarly, $[C4; S3]$ and $[C4; S2]$ are never feasible and hence the corresponding transitions $[C3; S3] \rightarrow [C4; S3]$ and $[C3; S2] \rightarrow [C4; S2]$ are not feasible. Removal of paths that traverse infeasible states results in substantial savings in the number of interleavings to be considered.

This elimination of infeasible paths is possible because (a) in any AND state, if each of the two concurrent threads is a simple statechart, then the guards can ONLY be on events and variables internal to the statechart or on messages exchanged between the threads; cryptographic protocols normally do not share memory between principals and are deterministic, (b) a transition $[S_1; S_2] \rightarrow [S_1'; S_2']$ is infeasible if it is triggered by $recv$ whose corresponding $send$ has not been triggered. These infeasible transitions

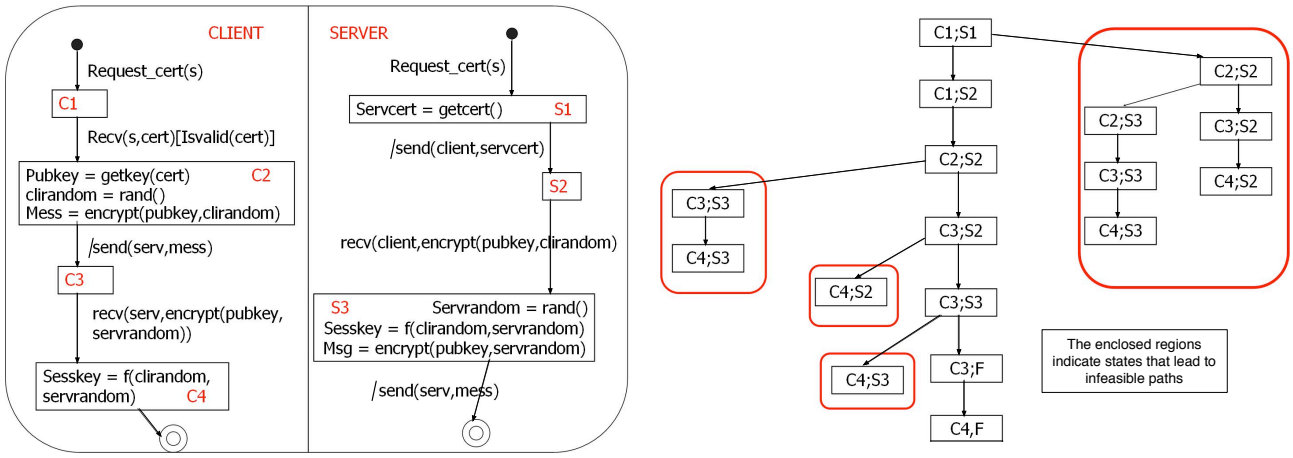

Figure 1. (a) A sample authentication protocol. (b) Infeasible paths and states in product construction.

can be eliminated, with time complexity O(mn), by a depth first or breadth first traversal of the product automaton. This notion of infeasible paths can be extended to shared memory as well. We define interaction points as a matching $send - recv$ pair. In shared memory, an interaction point is a read/write to a memory location.

2.3 Step 3: Testing a TLS implementation

The GnuTLS implementation [15] of the TLS protocol was selected for this study primarily because it is available freely and is known to be of high quality. The BullsEye tool [3] was used to measure the MC/DC coverage [7, 13] of the generated tests. All statecharts are available elsewhere [11].

In a distributed application with n peers, a test case is defined as an input which leads to a sequence of interactions between any subset of the peers. Any communication protocol, and hence a cryptographic protocol, is a distributed program with a well defined sequence of interactions. In testing a cryptographic protocol, we would like to exercise as many types of protocol runs as possible. Hence a test case is essentially an input inducing a particular protocol run. Each test case (run) has an initiator and a responder. A run can happen between peers or between clients and a server. Note also that in a cryptographic protocol, an input inducing a protocol run is given by the upper layer. So input is a request from the upper layer (application layer in the case of TLS).

Formally, a test case is a triple $(S, R, < i_1, i_2, \ldots, i_n >), n \geq 0$, where S is a protocol state, R is application-layer-request and $< i_1, i_2, \ldots, i_n >$ is the expected sequence of interactions. Also, as described earlier, a test case causes the program under test to traverse a path through the product statechart. We note that (1) The initiator is always the client in the TLS system (2) Any path through the final product statechart is a valid run, and (3) If different initiation parameters are used by the client, there is a clear difference in runs.

2.4 Step 4: Selection of uncovered blocks and "what if" analysis

We are not aware of any errors in the uncovered portions of the GnuTLS 1.4.1 code. Hence, even if tests were generated to cover the uncovered blocks, there is a good possibility that the implementation would behave correctly. However, we asked: *What if there is an error in the uncovered block ?* To answer this question 10 uncovered blocks were selected with the objective of obtaining variety in their function. Of these 10, six blocks are in the Handshake component and four in Transmit.

2.5 Step 5: Negative testing

Flattening the statechart and eliminating infeasible paths leads to the omission of certain tests. Negative tests are tests that would likely traverse a path, that is infeasible in the statechart, but might be feasible in the implementation (perhaps due to a programmer oversight). To check whether or not negative tests lead to any additional coverage, a few negative tests were generated manually and GnuTLS executed against these. Generating all infeasible paths is not useful because it leads to state space explosion as described in section 2.2. To generate negative tests, we consider each feasible state in the product automaton (in section 2.2), and randomly select one infeasible path from

that state. For example, in figure 1 (b), for state $[C1; S1]$, we select $[C1; S1] \rightarrow [C2; S2] \rightarrow [C3; S2] \rightarrow [C4; S2]$, out of the two infeasible paths from $[C1; S1]$. Selecting one random path per feasible state distributes the negative tests uniformly across the statechart.

3 Results

3.1 Protocol selection and modeling

The GnuTLS implementation [15] of the TLS protocol [6] contains the following three components.

1. **Handshake Component** responsible for initiating the session between the client and server, authenticating them to each other and negotiating session parameters (master secret, session key, keys used in computing the MAC). It can also be used to re-negotiate these parameters anytime during the session.

2. **Transmit Component** responsible for compression, encryption and transmission of messages on the sender side and reception, decryption and de-compression of the messages on the receiver side. The Transmit component corresponds to the Record component in RFC 2246 [6].

3. **Alert Component** responsible for handling erroneous data and exceptions and accordingly notifying the peers. Alerts can occur anywhere in the protocol.

The client and the server, each have a handshake component and a record component running concurrently (consequently four concurrent threads). Even-though the client runs the handshake protocol fewer times than the server, it also has two concurrent threads because it may have to periodically re-negotiate cryptographic parameters with the server to prevent the theft of some session parameters from compromising the entire session. The concurrent statecharts and their flattened equivalent, with infeasible paths removed, are not included here due to space constraints and are available elsewhere [11].

3.2 Statechart flattening and test generation

The statechart model of the TLS protocol has 13 and 15 states, respectively, in the client and server handshake threads. It has four states in the record protocol. Hence, in total a statechart product construction would result in 13*15*4*4 = 3120 states. An *estimate* of the number of paths (and hence the number of tests) is 1600 (based on the fact that there are 40 paths in each of the the client and server threads).

After elimination of infeasible paths, 32 states and 41 paths remained. This led to a total of 58 test cases. Such a substantial reduction was possible because, as explained in section 2.2 cryptographic protocols involve substantial communication between participants and are synchronous. A participant waits for messages from the other participant and cannot proceed otherwise. Since participants proceed in lockstep with each other, consideration of all possible interleavings becomes unnecessary. Also, cryptographic protocols may have several independent (non-interfering) concurrent threads. In TLS, the handshake thread and the record thread do not interact at every step, and instead interact at the end of the handshake.

3.3 MC/DC coverage [7, 13]

The MC/DC coverage of Handshake, Transmit and Algorithms are, respectively, 64.7%, 72.6% and 73.9%. Table 1 lists the MC/DC coverage with respect to some key GnuTLS (version 1.4.1) files. A count of the total conditions/decisions to be covered is also listed in the table the fourth column from the left. We noticed that the GnuTLS implementation has more features than required by RFC 2246 [6]. It would thus be inappropriate to include all conditions/decisions in the computation of coverage of the tests generated from the statechart which is derived from the RFC 2246 [6]. Hence the implementation was examined manually[1] and conditions/decisions that correspond to features not required by the RFC 2246 [6] were removed. This led to the Adjusted Coverage in the third column from the left.

3.4 Error injection

Note that the errors introduced are hypothetical and do not exist in the implementation. We introduced several common programming errors. Original code segments and their error-injected counterparts are available elsewhere [11] though two examples are reproduced here. Note that there are several such uncovered blocks in the code. These components were selected because we wanted to investigate whether or not there exists a potential of possible errors in uncovered code leading to security vulnerabilities, especially serious ones such as session hijack and authentication failure. Table 2 summarizes the type of errors introduced that eventually led to a security vulnerability. We do not claim that every programming error in any uncovered code fragment leads to a serious security failure, we only illustrate the fact that programming errors in uncovered fragments lead to security vulnerabilities. Sample

[1] We do understand that this task is error prone and hence the Adjusted Coverage might turn out to be different, albeit slightly, if the implementor of the code were identify the "inessential" conditions/decisions.

Table 1. MC/DC coverage for selected GnuTLS files.

Component	File	Adjusted Coverage	Conditions/ Decisions	Number Covered	Un-adjusted Coverage
Transmit	gnutls_record.c	51%	237	121	51%
Alert	gnutls_priority.c	81%	37	30	81%
Algorithms	gnutls_cipher.c	65%	132	86	65%
Algorithms	gnutls_algorithms.c	69%	137	82	60%
Handshake	gnutls_auth.c	64%	61	39	64%
Handshake	gnutls_handshake.c	65%	538	269	50%
Alert	gnutls_datum.c	62%	24	15	62%
Transmit	gnutls_session.c	65%	26	12	46%
Handshake	auth_cert.c	62%	353	183	52%

Table 2. Errors injected in uncovered code.

Type	Location		Security Failure
	In code	In statechart	
Missing function call and Incorrect Statement	auth_cert.c; 406,523	ComputeCryptoParameters	Authentication error
Missing function call and Incorrect Statement Incorrect Statement	gnutls_record.c gnutls_kx.c; 651	Renegotiate start ComputeCryptoParameters	Session hijack Integrity failure Authentication failure
Incorrect Statement	gnutls_session.c;	Renegotiate start	Invalid session
Incorrect Statement	gnutls_kx.c; 608	ComputeCryptoParameters	Authentication failure
Missing function calls	gnutls_record.c; 976	Simple_state 13	Session Hijack integrity failure
Missing function calls Missing Statement Incorrect Statement	gnutls_sig.c; 351	Simple_state 13	Session Hijack integrity failure
Incorrect Statement	gnutls_handshake.c; 1460	Simple_State 10,11	Loss of Confidentiality
Incorrect Statement	gnutls_handshake.c; 1141	Simple_State 11	Denial of Service
Incorrect Statement	gnutls_handshake.c; 696	ComputeCryptoParameters	Authentication failure

code segments and their error-injected counterparts are reproduced next.

Example 1:

Uncovered Code:

```
cred = (gnutls_certificate_credentials_t)
       _gnutls_get_cred (session->key,
             GNUTLS_CRD_CERTIFICATE, NULL);
if (cred == NULL) {
      gnutls_assert ();
      return GNUTLS_E_INSUFFICIENT_CREDENTIALS;
}
```

Implementation error:

```
cred = (gnutls_certificate_credentials_t)
       _gnutls_get_cred (session->key,
             GNUTLS_CRD_CERTIFICATE, NULL);
if (cred == NULL) { return OK; }
```

Example 2:

Uncovered Code:

```
else if ( session_is_valid (session) != 0 ||
           session->internals.may_not_read != 0) {
             gnutls_assert ();
      return GNUTLS_E_INVALID_SESSION;
   }
```

Implementation error:

```
else if ( session_is_valid (session) != 0 ||
    session->internals.may_not_read != 0)
{ return OK; }
```

3.5 Negative testing

A total of 70 negative tests were generated. The GnuTLS implementation was executed against each of these tests.

There was no increase in MC/DC coverage [7, 13] over and above the coverage already obtained using tests generated from the flattened statechart. This may be because the infeasible paths generated may be infeasible in the implementation, and the error handling code executed as a result has already been covered by the tests generated in section 3.2.

4 Code-coverage Analysis

The MC/DC coverage [7, 13] obtained from tests generated from the flattened statechart appears to be much lower than what one might expect of a test suite for testing an implementation of a security protocol. Obviously, one expects to at least cover all code blocks in such an implementation which is not nearly the case as is evident from Table 1. The impact of each of the ten errors injected as was analyzed manually as well as through the execution of the error-injected code. Table 2 summarizes the impact of ten errors, injected one at a time in the GnuTLS code.

5 Discussion

Only one implementation was considered and one model generated. Techniques proposed by other researchers might certainly generate smaller test suites [4, 1, 14] than the testing tree method but will not increase coverage because the smaller test suite is a subset of the that obtained through the testing tree method. The lack of coverage increase as a consequence of executing the implementation against the negative tests is surprising. While this could be due to the possible coverage of error checking code, more analysis is needed to find out the cause.

This empirical study raises the following question: *What is a sufficient level of granularity at which the statechart must be constructed so that while not explicit as the actual code, it generates tests that provide desired code coverage ?* An example of granularity is contracts between functions: Function A may call Function B which is expected to return a value or -1(SYSERR). Function A may check for the return of SYSERR. But since this check is not modeled in the statechart, it is not guaranteed to be covered. Such conditions will be covered if the SYSERR corresponds to an error condition modeled in the statechart.

In the"what-if" scenarios constructed in Section 2.4, the selection of uncovered blocks were made from modules responsible for authentication, integrity and confidentiality. While the effects of all possible programming errors in the selected blocks was not analyzed, sufficient analysis was done to check for various possible security failures.

6 Conclusion and Recommendations

The key observation arising out of this study is that the statechart model is inadequate for testing security protocols. This observation leads us to recommend that statechart alone is not a suitable device for test generation for security protocols. Further, we are not critiquing the use of statecharts as a means to model application behavior and as a source of tests; we are merely suggesting that other sources of tests must also be used. While this might appear to be a trivial outcome of this work, the low coverage of tests seems to indicate otherwise.

References

[1] K. Bogdanov. *Automated Testing of Harel's Statecharts*. PhD thesis, University of Sheffield, January 2000.

[2] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language Guide*. Addison-Wesley, 1999.

[3] Bullseye Testing Technology Inc. Bullseye Coverage Measurement Tool. www.bullseye.com.

[4] S. Burton. *Automated Generation of High Integrity Test Suites from Graphical Specifications*. PhD thesis, University of York, Department of Computer Science, March 2002.

[5] T. S. Chow. Testing software design modelled by finite state machines. *IEEE Transactions on Software Engineering*, SE-4(3):178–187, May 1978.

[6] T. Dierks and C. Allen. RFC 2246: The TLS Protocol. http://www.ietf.org/rfc/rfc2246.txt.

[7] J. B. Goodenough and S. L. Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, SE-1(2):156–173, June 1975.

[8] D. Harel. A visual formalism for complex systems. *Science of Programming*, 8:231–274, 1987.

[9] D. Harel and M. Politi. *Modeling Reactive Systems With Statecharts : The Statemate Approach*. McGraw-Hill Companies, October 1998.

[10] NISCC IPSEC vulnerability advisory. http://www.niscc.gov.uk/niscc/docs/al-20050509-00386.html?lang=en.

[11] K. R. Jayaram and Aditya P. Mathur. Adequacy of Statecharts as a Source of Tests for Implementations of Cryptographic Protocols. http://www.serc.net/report/tr288.pdf, 2007.

[12] Security advisories for the Kerberos protocol. http://web.mit.edu/kerberos/advisories/.

[13] N. Leveson and A. Dupuy. An empirical evaluation of the MC/DC coverage criterion on the hete-2 satellite software. In *Proceedings of the Digital Aviation Systems Conference (DASC), Philadelphia*, October 2000.

[14] L. Liuying and Q. Zhichang. Test selection from UML statecharts. In *Proceedings Technology of Object-Oriented Languages and Systems, TOOLS 31*, pages 273 –279, September 1999.

[15] S. Josefsson et al. The GNUTLS project. http://www.gnu.org/software/gnutls/.

[16] OpenSSL vulnerability advisory. `http://www.openssl.org/news/secadv_20051011.txt`.