Evaluating The Stream Control Transmission Protocol Using Uppaal

Shruti Saini
School of Computing, Information and Mathematical Sciences
The University of the South Pacific
Suva, Fiji
shruti.saini@usp.ac.fj

Ansgar Fehnker
Department of Computer Science
University of Twente
Enschede, the Netherlands
ansgar.fehnker@utwente.nl

The Stream Control Transmission Protocol (SCTP) is a Transport Layer protocol that has been proposed as an alternative to the Transmission Control Protocol (TCP) for the Internet of Things (IoT). SCTP, with its four-way handshake mechanism, claims to protect the Server from a Denial-of-Service (DoS) attack by ensuring the legitimacy of the Client, which has been a known issue pertaining to the three-way handshake of TCP. This paper compares the handshakes of TCP and SCTP to discuss its shortcomings and strengths. We present an Uppaal model of the TCP three-way handshake and SCTP four-way handshake and show that SCTP is able to cope with the presence of an Illegitimate Client, while TCP fails. The results confirm that SCTP is better equipped to deal with this type of attack.

1 Introduction

The Internet of Things (IoT) is an emerging field envisioned to connect all physical objects to the Internet, enabling them to communicate with one another and perform tasks autonomously. The technology most commonly referenced to provide such behavior for the objects in IoT is the Radio Frequency Identification Device (RFID), which are low-powered energy-constrained devices. Due to the sheer amount of objects to be connected in IoT, there are many research challenges that need to be tackled. Some of these include its architecture, networks, applications and security. This paper looks at the Transport Layer for IoT.

In the Internet, the protocol most commonly used at the transport layer is the Transmission Control Protocol (TCP). This protocol however, has been found to not meet the requirements for IoT applications due to its high power consumption resulting from the verbose session overhead and requirements for reliability which requires packet acknowledgment [9, 11]. In contrast, the Stream Control Transmission Protocol (SCTP) supports all features of TCP and also claims to provide protection against denial-of-service (DoS) attacks, amongst other advanced features.

We will evaluate SCTP against the SYN flooding DoS attack that is known in TCP [12]. SCTP, with its four-way handshake claims to protect the Server from a DoS attack by ensuring the legitimacy of the Client which has been a known issue pertaining to the three-way handshake of TCP [8]. The SYN flooding attack and several mitigation strategies for TCP are discussed extensively in [15]. This paper, however, uses Uppaal, a model checking tool, to formally model and verify the basic handshake mechanisms employed by TCP and SCTP. Previous analysis of the protocol concentrated on its performance aspects [5] and used network simulator NS-2. Different types of potential attacks were discussed in [2], and the authors undertook a manual review of three implementations to look for vulnerabilities. This paper in contrast, built a formal model of the handshake mechanism in TCP and SCTP, to verify SCTP's resilience to a specific attack.

H. Hermanns, P. Höfner (Eds.): Models for
Formal Analysis of Real Systems (MARS 2017)
EPTCS 244, 2017, pp. 1–13, doi:10.4204/EPTCS.244.1

The next section will introduce the handshake mechanisms of TCP and SCTP. Section 4 and 5 discuss the TCP and SCTP models, respectively. Section 6 will discuss the results for properties that relate to a potential SYN flooding attack.

2 TCP and SCTP

Communication on the Internet is governed by the Internet Protocol (IP) Suite comprised of a set of layered protocols implemented by the host. There are five layers in the Internet Architecture, however, we will only explore protocols from the Transport Layer. The most popular and widely implemented protocol is TCP. This protocol was intended as a highly reliable host-to-host protocol between hosts in packet-switched computer communication networks, and was standardized in 1981 as RFC 793 [7, 10].

SCTP, on the other hand, was developed to support functionalities that neither TCP nor UDP could offer. It was standardized by the Internet Engineering Task Force (IETF) in the year 2000 in the RFC 4960 [6, 13, 14]. The SCTP protocol has become a general purpose transport protocol with most features of TCP and a set of other features for security, multihoming, multistreaming, mobility and partial reliability.

Both SCTP and TCP are connection oriented protocols. This means, prior to any communication between two parties, a setup procedure needs to be executed to establish a communication relationship and state. For TCP, this occurs with a three-way handshake to establish the relationship which is called a connection. For SCTP, a four-way handshake is used, and the relationship is called an association. It encompasses a broader concept than a single connection with its multihoming feature. Both TCP and SCTP use a Transmission Control Block (TCB) to hold their connection or association state information.

2.1 Segment and Packets

Any information exchange in TCP uses segments, while SCTP uses packets. TCP encapsulates the data received from the Application Layer into a TCP segment by adding the TCP Header. The TCP Header follows the IP header, and supplies protocol specific information. Figure 1 depicts the format of the TCP header.

Every packet in SCTP consists first of a common Header and is followed by chunks containing either control information or user data. SCTP allows to bundle multiple chunks into a single packet, with some exceptions. Figure 2 and Figure 3 depict the format of the SCTP header and the SCTP chunk.

Since every connection and association in TCP and SCTP is distinct, certain data about each needs to be maintained separately. Both TCP and SCTP, for this purpose, utilize a special data structure called the TCB that records the state of a connection and association, respectively.

For TCP, the variables stored in the TCB are the local and remote socket numbers, the state information, the security and precedence of the connection, pointers to the users send and receive buffers, pointers to the retransmit queue and to the current segment and several variables relating to the send and receive sequence numbers, amongst others.

SCTP maintains similar information about its associations like the local and remote socket numbers, the state information, a list of all local and remote transport addresses bound to the association, several variables relating to the send and receive sequence numbers and an array of structures to track the inbound and outbound streams, amongst others.

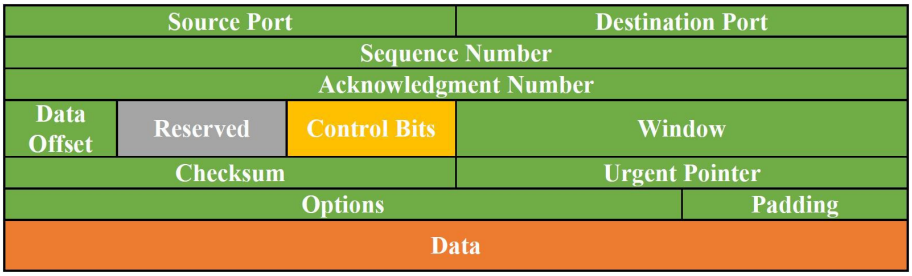

Figure 1: TCP Segment format

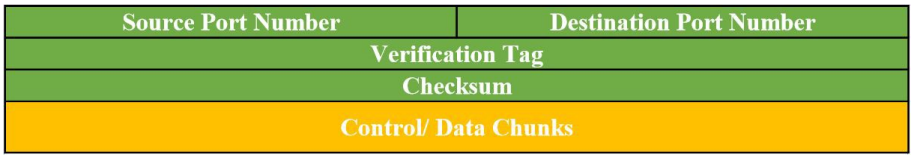

Figure 2: SCTP Packet format

Figure 3: SCTP Chunk format

2.2 Connections in TCP

Communication in TCP is dictated by the establishment of a successful connection between two endpoints through the three-way handshake. The handshake includes the following steps:

1. All Endpoints begin from the CLOSED state. When Endpoint A wishes to start communicating with another it performs an Active OPEN whereby it creates a TCB to store the necessary information and sends the Synchronize (SYN) segment before moving to the SYN-SENT state.
2. An Endpoint B that wishes to receive any incoming connection request performs a Passive OPEN whereby it creates the TCB which is partially filled with unspecified foreign sockets and enters the LISTEN state. Endpoint B receives the incoming SYN segment and fills the parameters of the partially completed TCB before replying with a SYN segment itself along with an Acknowledgment (ACK) segment, collectively called the SYN+ACK. In doing so, Endpoint B allocates resources to the unestablished connection and updates the TCB to the SYN-RECEIVED state in wait for an ACK segment.
3. Endpoint A receiving the SYN+ACK replies with an ACK segment as its final reply to establish the connection and moves to the ESTABLISHED state.
4. Endpoint B upon receiving the final ACK segment also updates its TCB to the ESTABLISHED state and successfully establishes the connection.

There are only two segments, SYN and ACK, involved in the three-way handshake for TCP as illustrated in Figure 4.

2.3 Associations in SCTP

Prior to any communication that can occur between two Endpoints in SCTP, they must first establish an association. The handshake includes the following steps:

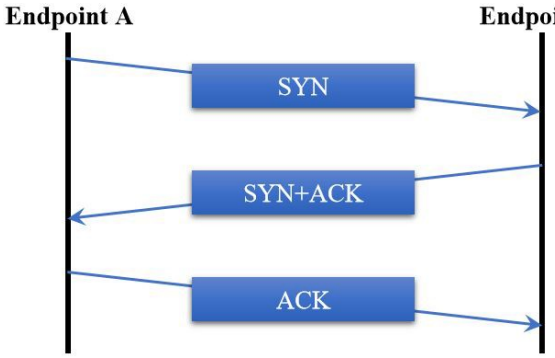

Figure 4: TCP three-way handshake

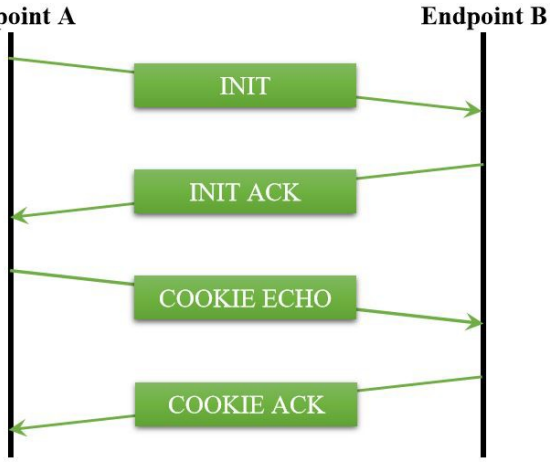

Figure 5: SCTP four-way handshake

1. Initially, all Endpoints begin from the CLOSED state. When Endpoint A wishes to start communicating with another it creates a TCB to store the necessary information and sends the Initiation (INIT) chunk before moving to the COOKIE-WAIT state.
2. Endpoint B receiving the incoming INIT chunk creates a temporary TCB to extract a subset of information that would help recreate the TCB along with a Message Authentication Code (MAC) and a secret key which are then used to generate a cookie. This cookie is sent as a reply with the Initiation Acknowledgment (INIT ACK) chunk. The temporary TCB is then deleted and Endpoint B remains in the CLOSED state preventing the allocation of resources for an unestablished connection.
3. Endpoint A upon receiving the INIT ACK chunk replies by echoing the cookie back with a Cookie Echo (COOKIE ECHO) chunk and enters the COOKIE-ECHOED state.
4. Endpoint B upon receiving the cookie back with the COOKIE ECHO chunk validates the TCB with the MAC to confirm the authenticity of the cookie. The TCB is then recreated from the information present in the cookie and a final reply is sent with the Cookie Acknowledgment (COOKIE ACK) chunk to establish the association and updates its TCB to the ESTABLISHED state. In doing so, resources are finally assigned to the association.
5. Endpoint A receives the final COOKIE ACK chunk and moves to the ESTABLISHED state and successfully establishes the association.

There are four chunks involved in the four-way handshake for SCTP as illustrated in Figure 5.

3 UPPAAL Model Checker

Distributed systems are difficult to understand, design, and reason about due to their complexity and non-deterministic nature. They usually involve subtle interactions of a number of components and a high level of parallelism. This is why the correctness of these systems is difficult to ensure. Several systems and protocols have been proven not to succeed in satisfying their intended goals after they have been

published [1]. One promising solution to this problem is the use of formal verification techniques such as model checking [1].

Developed in conjunction by the Department of Computer Systems at Uppsala University, Sweden, and BRICS at Aalborg University, Denmark, UPPAAL is a tool for the modeling, simulation and verification of real-time systems [1, 3]. The tool is appropriate for systems that can be modeled as a collection of non-deterministic processes with finite control structure and real-valued clocks, communicating through channels or shared variables [3].

The following Section discusses our UPPAAL implementation of the TCP and SCTP handshakes. The models implemented consider all requests to be valid and authentic. This assumption allows us to create a simulated version of the handshakes which enables the receiver to non-deterministically chose a reply to an incoming request regardless of its content and authenticity. The key focus, however, is to the handling of the first incoming request which may be susceptible to a SYN flooding attack.

4 TCP Model

According to the connection establishment process of TCP, there are five states in the three-way handshake. In the model, these are represented by constant integer variables declared as `CLOSED`, `LISTEN`, `SYN_SENT`, `SYN_RECEIVED` and `ESTABLISHED` states as shown in Table 1. Typical application areas include real-time controllers and communication protocols. UPPAAL has been applied successfully in case studies ranging from communication protocols to multimedia applications [4].

State	Description
`CLOSED`	No connection
`LISTEN`	Waiting for any connection request
`SYN_SENT`	Waiting for a matching connection request after having sent a connection request
`SYN_RECEIVED`	Waiting for a confirming connection request acknowledgment after having both received and sent a connection request
`ESTABLISHED`	An open active connection

Table 1: Summary of States in the three-way handshake

4.1 The Client Template

The *Legitimate Client* template for TCP as modelled in Uppaal is shown in Figure 6. The TCP model contain a number of channels. All of these channels are broadcast channels, meaning that messages can be dropped, if the intended recipient is not able to receive the message. The descriptions for the edges of the template are as follows:

1. Location LC0 is the initial location of the automaton, representing the `CLOSED` state. The *Client* can perform an Active Open on the `syn` channel in order to send a connection request, set the `counter`, start the `timer` and move to location LC1.

2. Location LC1 represents the `SYN_SENT` state. Here the *Client* has the following options:

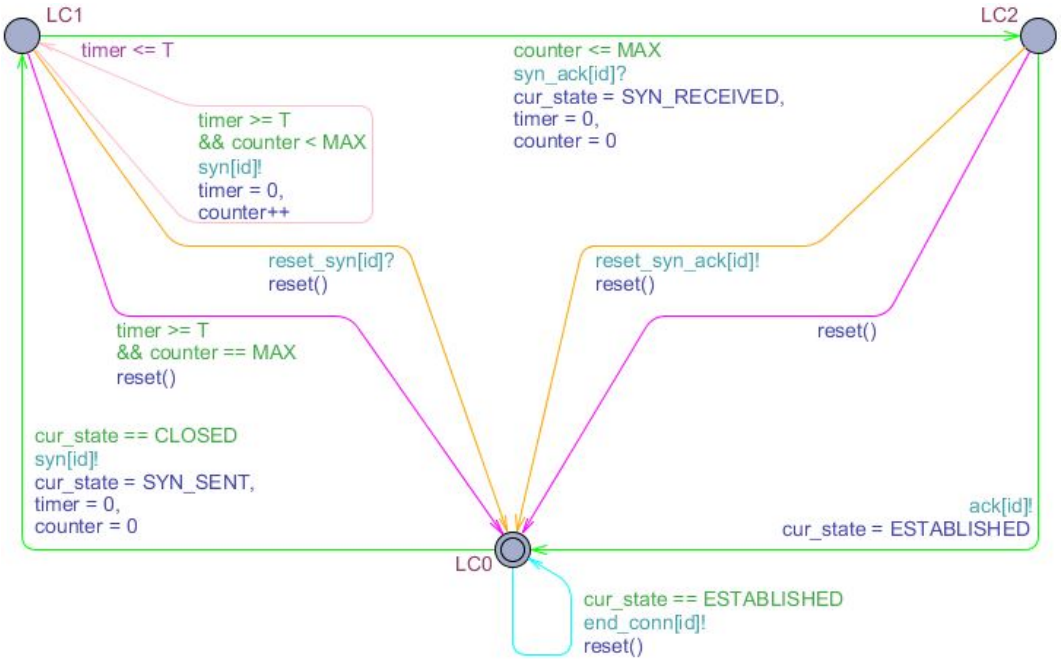

Figure 6: TCP Legitimate Client template

(a) *Receive Acknowledgment* The *Client* can receive an acknowledgment on the `syn_ack` channel, move to location LC2, and update its state to `SYN_RECEIVED` as long as the reply comes before the maximum number of retransmits are made and before the `timer` expires.

(b) *Reset* The *Client* can receive a request to reset the connection on the `reset_syn` channel, move back to location LC0 and reset all state information.

(c) *Retransmit* The *Client* can retransmit the connection request on the `syn` channel if no reply is received in a certain time `T` and if retransmits are still allowed. The `counter` of retransmits is incremented and the `timer` restarted.

(d) *Discard* The *Client* can silently discard the connection request made, move back to location LC0 and reset all state information.

3. Location LC2 is an intermediate state, where a `syn_ack` has been received. Here the *Client* has the following options:

   (a) *Send Acknowledgment* The *Client* can send the acknowledgment on the `ack` channel, move back to location LC0, and change the state to `ESTABLISHED`.

   (b) *Reset* The *Client* can ask the *Server* to reset the connection on the `reset_syn_ack` channel, move back to location LC0 and reset all state information.

   (c) *Discard* The *Client* can silently discard the connection request, move back to location LC0 and reset all state information.

4. If the *Client* is in location L0 with current state `ESTABLISHED`, it can request to end the connection on the `end_conn` channel, return to location LC0 and reset all state information.

The *Illegitimate Client* template for TCP as modelled in Uppaal has a single Location IC0 with a single self loop that keeps transmitting a connection request on the `syn` channel in an attempt to occupy all *Server* resources.

4.2 The Server Template

Both the TCP and SCTP protocol include a TCB block. The Uppaal model of these protocols uses the same basic data structure to model the TCB block. It does not incorporate all the fields from the segments and packets or the TCB that are used during the handshake as described in [10, 13]. Only the fields that are necessary for simple verification and identification of the segments and packets are kept.

The TCB is maintained for all connections and associations by the *Server* Endpoint. Listing 1 shows the declaration. The constant `RESOURCES` refers to the number of possible Endpoints a connection or association can be established with, which is one less that the total number of Endpoints created.

```
typedef struct {
    ids peer;
    int [CLOSED, ESTABLISHED]cur_state;
} TCB;

TCB tcb[RESOURCES];
```

Listing 1: Local Declarations of the TCB at the Server Endpoint

The *Server* template includes several functions to maintain the TCB, most notably `update_TCB` to add new connections or associations information. A resource `i` is available if the current state `tcb[i].cur_state` is `LISTEN`. Initially all resources are available. When the ACK segment is received for an active connection request with an Endpoint the current state will be `SYN_RECEIVED`.

The *Server* template for TCP as modelled in Uppaal is shown in Figure 7. The descriptions for the edges of the template are as follows:

1. Location S0 is the initial committed location. The *Server* can perform a Passive Open, create and partially initialize the TCB, move to location S1, and update its state to `LISTEN` in order to start receiving connection requests from the *Client*.
2. In location S1 the *Server* Endpoint receives incoming connection requests on the `syn` channel and moves to location S2.
3. Location S2 is an intermediate committed state. Here the *Server* has the following options:
   (a) *Send Acknowledgment* If a resource is available, the *Server* can send the acknowledgment on the `syn_ack` channel, move to location S1, and update its state to `SYN_RECEIVED`.
   (b) *Reset* The *Server* can ask the *Client* to reset the connection on the `reset_syn` channel, move back to location S1 and reset all state information.
   (c) *Discard* The *Server* can silently discard the connection request, move back to location S1 and reset all state information.
4. In location S1 new connection requests from *Clients* can be received continuously as well as requests to further or end the half-open and fully established connections:
   (a) *Receive Acknowledgment* The *Server* can receive an acknowledgment on the `ack` channel, move back to location S1, and change the state to `ESTABLISHED`.

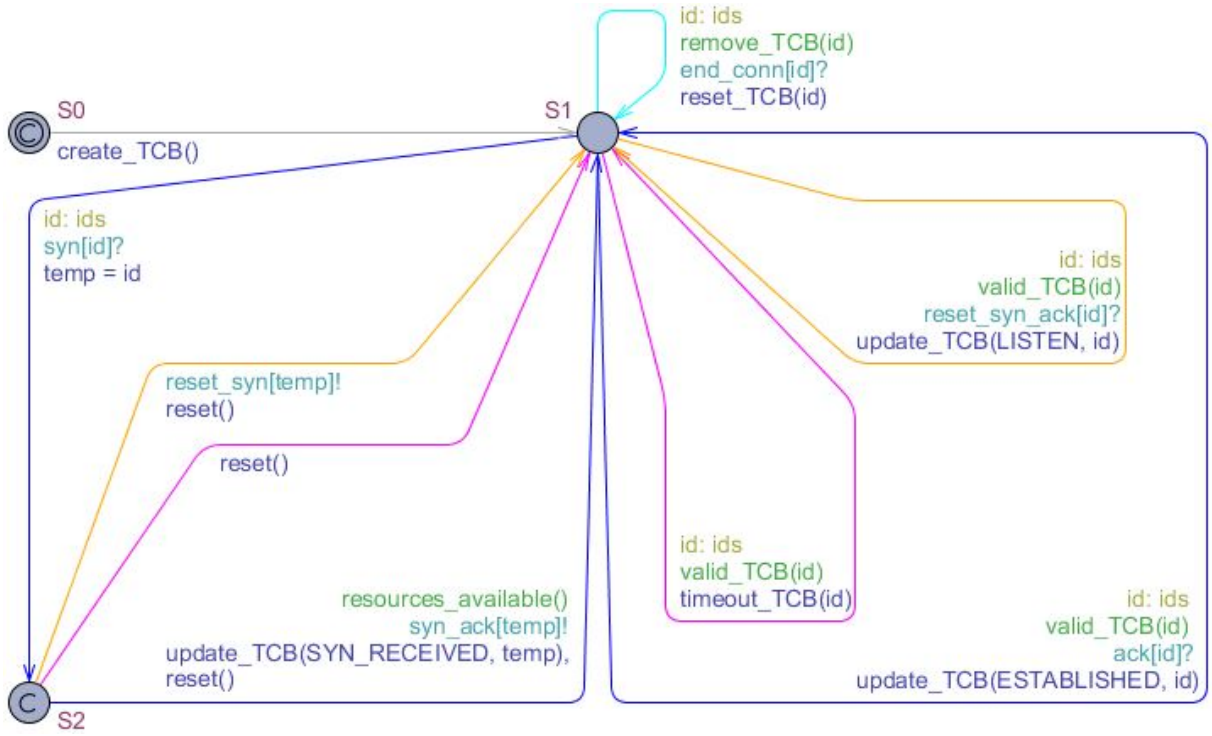

Figure 7: TCP Server template

(b) *Reset* The *Server* can receive a request to reset the connection on the `reset_syn_ack` channel, move back to location S1 and reset all state information.

(c) *Time-out* The *Server* can remove half-open connection requests which have timed out and reset all state information. This is a condensed representation of the time-wait and retransmission attempts to fully establish the connection before they are ceased.

(d) *End Connection* The *Server* can receive a request to end the connection on the `end_conn` channel and reset all state information.

5 SCTP Model

According to the association establishment process of SCTP, there are four states in the four-way handshake. In the model, these are represented by integer constants `CLOSED`, `COOKIE_WAIT`, `COOKIE_ECHOED` and `ESTABLISHED`, as shown in Table 2.

5.1 The Client Template

The *Legitimate Client* template for SCTP as modelled in Uppaal is shown in Figure 8. The descriptions for the edges of the template are as follows:

1. Location LC0 is the initial location of the automaton, representing the `CLOSED` state. The *Client* can send an association request over the `initiation` channel, set the `counter`, start the `timer` and move to location LC1.

State	Description
CLOSED	No association
COOKIE_WAIT	Waiting for a confirming association request acknowledgment with a cookie after having sent an association request
COOKIE_ECHOED	Waiting for a confirming association request acknowledgment after having both received and sent the cookie back
ESTABLISHED	An open active association

Table 2: Summary of States in the four-way handshake

2. Location LC1 represents the COOKIE_WAIT state. Here the *Client* has the following options:
   (a) *Receive Acknowledgment* The *Client* can receive an acknowledgment on the init_ack channel, move to location LC2, as long as the reply comes before the maximum number of retransmits are made and before the timer expires.
   (b) *Abort* The *Client* can receive a request to abort the association over the abort_init channel, move back to location LC0 and reset all state information.
   (c) *Retransmit* The *Client* can retransmit the association request over the initiation channel if no reply is received in a certain time T and if retransmits are still allowed. The counter of retransmits is incremented and the timer restarted.
   (d) *Discard* The *Client* can silently discard the association request, move back to location LC0 and reset all state information.

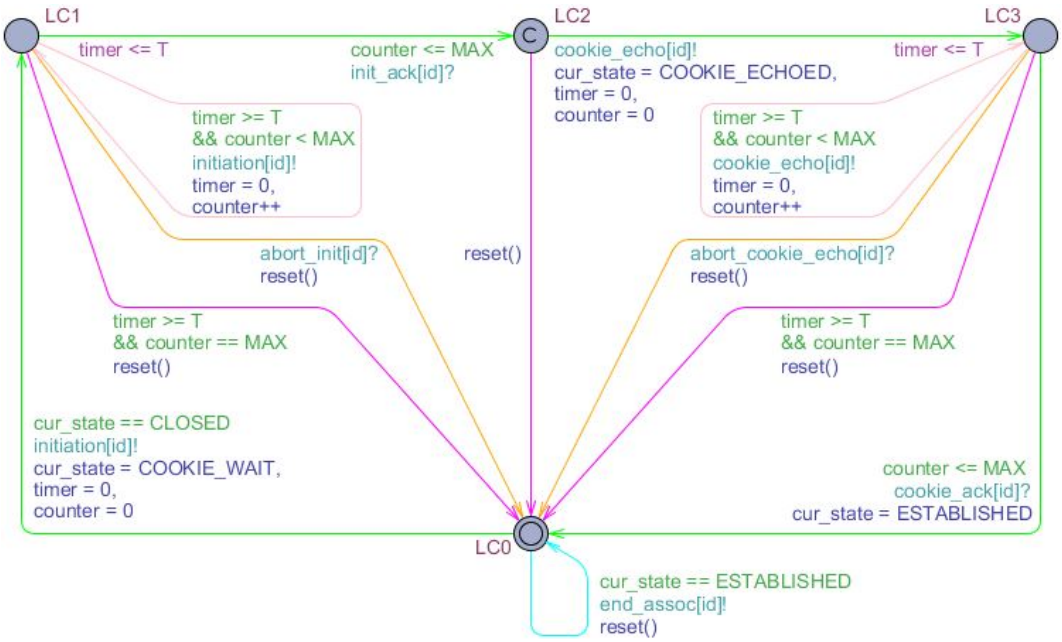

Figure 8: SCTP Legitimate Client template

3. Location LC2 is an intermediate committed state, where a `init_ack` has been received. Here the *Client* has the following options:
   (a) *Send Acknowledgment* The *Client* can send the acknowledgment on the `cookie_echo` channel, set the `counter`, start the `timer`, and move to location LC3.
   (b) *Discard* The *Client* can silently discard the association request, move back to location LC0 and reset all state information.
4. Location LC3 represents the `COOKIE_ECHOED` state where the cookie has been echoed back by the *Client*. The transitions available here are similar to those in location LC1.
5. If the *Client* is in location LC0 with current state `ESTABLISHED`, it can request to end the association over the `end_assoc` channel, return to location LC0 and reset all state information.

The *Illegitimate Client* template for SCTP is similar to the model for TCP. It has only one location, and a self loop that keeps transmitting an association request over the `initiation` channel in an attempt to occupy all *Server* resources.

5.2 The Server Template

The *Server* template uses the same TCB block as the TCP Server template. The Uppaal model is shown in Figure 9. The descriptions for the edges of the template are as follows:

1. Location S0 is the initial location of the *Server* Endpoint which receives incoming association requests on the `initiation` channel and moves to location S1.
2. Location S1 is an intermediate committed state. Here the *Server* has the following options:

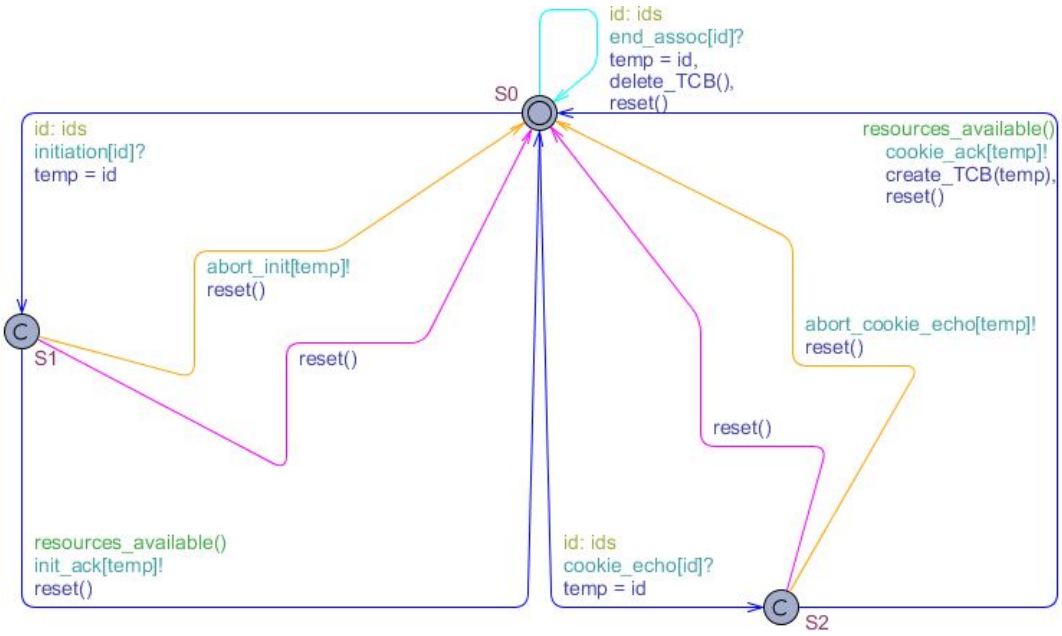

Figure 9: SCTP Server template

(a) *Send Acknowledgment* If a resource is available, the *Server* can send the acknowledgment on the `init_ack`, move back to location S0 and keep no state information.

(b) *Abort* The *Server* can ask the *Client* to abort the association on the `abort_init` channel, move back to location S0 and keep no state information.

(c) *Discard* The *Server* can silently discard the association request, move back to location S0 and keep no state information.

3. In location S0 new association requests from *Clients* can be received continuously as well as requests to further or end the associations:

(a) *Receive Acknowledgment* The *Server* can receive an acknowledgment on the `cookie_echo` channel and move to location S2 which is an intermediate committed state. The transitions available here are similar to those in location S1.

(b) *End Association* The *Server* can receive a request to end the association over the `end_assoc` channel and reset all state information.

6 Verification Results

We considered a number of properties to explore the correctness of TCP and SCTP. This section discusses two properties that highlight differences between TCP and SCTP, especially weaknesses in TCP.

The property in Listing 2 checks if in all states along all paths it holds that if a *Legitimate Client* is in the `ESTABLISHED` state, then a corresponding *Server* `tcb` resource is also in the `ESTABLISHED` state. If a *Legitimate Client* has an active connection or association then there must be a corresponding *Server* `tcb` resource in the fully-established state.

However, it appears that the TCP model does not produce the desired results, and examination confirms that this a known problem with TCP. The TCP model allows for half-open connections where the `ack` is not received by the *Server*. Failure to satisfy the property in Listing 2 shows that the TCP model allows a *Legitimate Client* to reach the `ESTABLISHED` state while the *Server* remains in some other arbitrary state.

```
A[] forall (i:ids) (
  Legit_Client(i).cur_state == ESTABLISHED imply
    exists (j: int[0,(RESOURCES-1)])(
      Server.tcb[j].peer == i and Server.tcb[j].cur_state == ESTABLISHED
    )
)
```

Listing 2: For any active connection should involve a server in the fully-established state.

Half-open connections are a known problem of TCP and they occurs due to a number of reasons in a real world application. A slow or lossy network, for example, can lead to the `ack` not being received by the *Server* in time, resulting in the *Client* assuming the connection was successfully established while the *Server* may remain in an arbitrary state.

This behavior of TCP is known to be exploited for SYN flooding attack. The attacker merely attempts to send enough `syn` requests, engaging its resources. Once a backlog of bogus half-open connections are established, the *Server* is not able to process requests from *Legitimate Clients*.

SCTP, in comparison, does not allow these half-open associations. Every association request in SCTP is replied to without allocating it any `tcb` resources. One can argue that the cookie may fall susceptible to the SYN flooding attack, however the purpose of the model was not to authenticate received chunks but to simulate all potential replies. The model provides the *Server*, as per its specification, the ability to *Send Acknowledgment*, *Abort* or *Discard* incoming cookies same as the first association request. The authentication of the cookie is assumed but through the implementation we are able to observe the cookie mechanism of SCTP successfully keep its resources free until the received cookie is authenticated by the *Server*.

The property in Listing 3 checks if any resource has been allocated to the same *Client*. For SCTP it is defined as follows:

```
E<> exists (i:ids) (
   forall (j: int[0,(RESOURCES-1)])(
      Server.tcb[j].peer == i and Server.tcb[j].cur_state != CLOSED
   )
)
```

Listing 3: For SCTP we check if the current state is in `SYN_RECEIVED`, instead of not equal to `Closed`.

If the model satisfies this property, it means that a resource has been successfully hogged by an *Illegitimate Client*. TCP satisfies this property, and thus fails to prevent hogging of resources. It allows the *Illegitimate Clients* to successfully occupy all *Server* `tcb` resources while attempting to establish a connection. In contrast, all SCTP models promptly replied to the *Illegitimate Client* association request without allocating it any *Server* `tcb` resource. This keeps the *Server* `tcb` resources free which prevents it from a DoS since the resources are not tied up with *Illegitimate Client* requests.

Although, the *Illegitimate Clients* are not able to establish a connection in TCP, the backlog of these half-open connections allows for a DoS like the SYN flooding attack. From this, we are able to formally verify and observe how SCTP using its cookie authentication is able to successfully prevent DoS attacks, as claimed.

7 Conclusion

This paper analysed network issues at the Transport Layer and confirmed that the TCP protocol does not provide basic security against DoS attacks on the IoT enabled devices. To clearly understand the differences between the two protocols, we took a detailed look into their handshake mechanisms which can be vulnerable to a DoS attack like the SYN flooding attack in TCP. A model-checker Uppaal was used to formally test the protocol's handshake mechanisms and to test SCTP's claims and TCP's vulnerability. We were able to confirm TCP's susceptibility to DoS attacks, as well as SCTP's ability in preventing it.

In conclusion, we were able to successfully evaluate SCTP to check for its applicability to IoT in comparison with TCP. We can conclude that the handshake mechanism of SCTP does in fact provide protection against DoS attacks, fulfilling a security requirement of IoT.

References

[1] Omar Al-Bataineh, Tim French & Terry Woodings (2012): *Formal Modeling and Analysis of a Distributed Transaction Protocol in UPPAAL*. In: *2012 19th International Symposium on Temporal Representation and Reasoning (TIME)*, IEEE, pp. 65–72, doi:10.1109/TIME.2012.12. Available at `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6311116`.

[2] Tuomas Aura, Pekka Nikander & Gonzalo Camarillo (2004): *Effects of mobility and multihoming on transport-protocol security*. In: *Security and Privacy, 2004. Proceedings. 2004 IEEE Symposium on*, IEEE, pp. 12–26, doi:10.1109/SECPRI.2004.1301312.

[3] Gerd Behrmann, Johan Bengtsson, Alexandre David, Kim G Larsen, Paul Pettersson & Wang Yi (2002): *UPPAAL Implementation Secrets*. *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pp. 3–22, doi:10.1007/3-540-45739-9_1.

[4] Gerd Behrmann, Alexandre David & Kim G Larsen (2004): *A Tutorial on Uppaal*. *Formal methods for the design of real-time systems* (November), pp. 200–236, doi:10.1007/978-3-540-30080-9_7.

[5] Sinda Boussen, Nabil Tabbane & Sami Tabbane (2009): *Performance analysis of SCTP protocol in WiFi network*. In: *Computer Sciences and Convergence Information Technology, 2009. ICCIT'09. Fourth International Conference on*, IEEE, pp. 178–182, doi:10.1109/ICCIT.2009.30.

[6] Thomas Dreibholz, Erwin P Rathgeb, Irene Rungeler, Robin Seggelmann, Michael Tuxen & Randall R Stewart (2011): *Stream control transmission protocol: Past, current, and future standardization activities*. *Communications Magazine, IEEE* 49(4), pp. 82–88, doi:10.1109/MCOM.2011.5741151.

[7] Charles M. Kozierok (2005): *The TCP/IP Guide: A Comprehensive, Illustrated Internet Protocols Reference*, 1 edition. No Starch Press, San Francisco. Available at `http://www.tcpipguide.com`.

[8] Preethi Natarajan, Fred Baker, Paul D Amer & Jonathan T Leighton (2009): *SCTP: What, why, and how*. *Internet Computing, IEEE* 13(5), pp. 81–85, doi:10.1109/MIC.2009.114.

[9] Maria Rita Palattella, Nicola Accettura, Xavier Vilajosana, Thomas Watteyne, Luigi Alfredo Grieco, Gennaro Boggia & Mischa Dohler (2013): *Standardized protocol stack for the internet of (important) things*. *Communications Surveys & Tutorials, IEEE* 15(3), pp. 1389–1406, doi:10.1109/SURV.2012.111412.00158.

[10] Jon Postel (1981): *Transmission Control Protocol*. Technical Report, DARPA Information Processing Techniques Office. Available at `http://www.ietf.org/rfc/rfc793.txt`.

[11] Anuj Sehgal, Vladislav Perelman, Siarhei Kuryla & Jurgen Schonwalder (2012): *Management of resource constrained devices in the internet of things*. *Communications Magazine, IEEE* 50(12), pp. 144–149, doi:10.1109/MCOM.2012.6384464.

[12] Paul Stalvig (2014): *Introduction to the Stream Control Transmission Protocol (SCTP): The next generation of the Transmission Control Protocol (TCP)*. Technical Report. Available at `https://f5.com/resources/white-papers/introduction-to-the-stream-control-transmission-protocol`.

[13] Randall Stewart (2007): *Stream Control Transmission Protocol*. Technical Report, Network Working Group. Available at `http://tools.ietf.org/html/rfc4960`.

[14] T Daniel Wallace & Abdallah Shami (2012): *A review of multihoming issues using the stream control transmission protocol*. *Communications Surveys & Tutorials, IEEE* 14(2), pp. 565–578, doi:10.1109/SURV.2011.051111.00096. Available at `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5875919`.

[15] Eddy Wesley M. (2007): *TCP SYN Flooding Attacks and Common Mitigations*. Technical Report, Network Working Group. Available at `https://tools.ietf.org/html/rfc4987`.