MASTER THESIS

Automatic Verification and Symbolic Analysis for 0-RTT Security in TLS 1.3

Author:
Fadi Abu Farha

Supervisor:
Dr. Ahmad Alsadeh

A thesis submitted in fulfillment of the requirements for the degree of Master of Science

in Computing at Birzeit University, Palestine

December 18, 2018

Abstract

Transport Layer Security (TLS) protocol is one of the most important and widely used cryptographic protocols that is introduced to provide secure communication over Internet. However, over twenty years, attacks against TLS show weaknesses and pitfalls in the protocol design and implementation. Therefore, Internet Engineering Task Force (IETF) is in continuous development to revamp the security of TLS by adding new security features to avoid the weakness of older protocol versions.

Many design goals were proposed in many fields of the TLS protocol to finally produce a new secure, reliable and fast version of this protocol, specifically, reducing the latency of the key exchange (KE) protocols while maintaining the security guarantees represented by forwarding secrecy. To achieve this, zero round trip time (0-RTT) protocol is a candidate solution.

We explore a practical solution to protect the KE process and sends the early data in 0-RTT with full Perfect Forward Secrecy (PFS) and preventing the replay attack. To this end, we analyze the 0-RTT handshake process using Diffie-Hellman and pre-shared keys in TLS1.3; by extending and updating a previous model of 0-RTT protocol in the TLS1.3 protocol specifications (RFC8446). By studying related work which attempted to solve the PFS and replay attack using either puncture mechanism or Google Quick UDP Internet Connection (QUIC) protocol, we found that both solutions have significant performance drawbacks.

We present a new approach to implement 0-RTT based on TLS1.3 without sacrificing PFS. We support the theoretical approach by practical tests using a Tamarin tool for symbolic modeling and analysis of TLS1.3 security protocols. In the practical experiments, we simulate several attempts to break PFS using Tamarin component and we verify that the solution guarantees PFS. Moreover, we deduce that attempting to solve the PFS problem by Diffie-Hellman using more than one key will require a fundamental change to the structure of Diffie-Hellman which ends up with a new protocol that need intensive reviews and studies.

المستخلص

التحقق الآلي والتحليل الرمزي ل 0-RTT في بروتوكول أمان طبقة النقل TLS 1.3

إعداد: فادي أبوفرحة

يعتبر بروتوكول أمان طبقة النقل (TLS) أحد أهم بروتوكولات التشفير الأكثر استخدامًا والذي يوفر اتصال آمن بين طرفين عبر شبكة الإنترنت. ومع ذلك، فقد تعرض البروتوكول إلى العديد من الهجمات المتوالية التي دامت أكثر من عشرين عاما، و التي أظهرت نقاط القوة والضعف في تصميم وتنفيذ هذا البروتوكول. ولذلك، فإن فريق مهام هندسة الإنترنت (IETF) يعمل بجد وبشكل مستمر للحفاظ على تطوير وتجديد أمان طبقة النقل الآمنة (TLS) عن طريق إضافة ميزات أمان جديدة لتجنب ضعف إصدارات البروتوكول القديمة .

تم اقتراح العديد من أهداف التصميم في العديد من مجالات بروتوكول (TLS) لإنتاج إصدار جديد آمن وموثوق وسريع من هذا البروتوكول، وتحديدًا تقليل وقت استجابة بروتوكولات تبادل المفاتيح (KE) مع الحفاظ على الضمانات الأمنية المتمثلة في إعادة توجيه السرية. ولتحقيق ذلك، يعتبر بروتوكول صفر لوقت الرحلة (0-RTT) هو الحل المرشح.

نحن نقوم باستكشاف حلاً عمليا لحماية عملية تبادل المفاتيح (KE) وإرسال البيانات المبكرة باستخدام(0-RTT))مع الحفاظ على السرية التامة (PFS) ومنع عملية إعادة توجيه البيانات (Replay Attack) . تحقيقا لذلك، نقوم بتحليل عملية مصافحة (0-RTT) باستخدام نظرية ((Diffie-Hellman(DHE) وكذلك اعادة استخدام المفاتيح المشتركة التي تم تبادلها مسبقًا (PSK) في البروتوكول(TLS1.3) ؛ عن طريق التحديث والتعديل والاضافة على بعض الابحاث والنماذج السابقة ل (0-RTT) التي قام بوضعها بعض الباحثين في هذا المجال على ما يُعرف بمواصفات بروتوكول طبقة النقل الآمن (TLS1.3). بعد دراسة الأعمال السابقة ذات الصلة والتي قامت بمحاولة حل مشكلتيّ (PFS)ومنع هجوم إعادة توجيه البيانات (Replay Attack)، باستخدام آلية الثقب (Puncture)أو بروتوكول جوجل(QUIC) ، وجدنا أن كلا الحلين المقترحين يتضمنان بعض التحديات والتي تؤثر على سرعة ألاداء.

نحن نقدم مقترح جديد لتنفيذ (0-RTT) في (TLS1.3) دون التضحية بضمان حماية البيانات .(PFS) نحن ندعم هذا النهج النظري من خلال الاختبارات العملية باستخدام أداة تمارين (Tamarin) للنمذجة الرمزية وتحليل بروتوكولات أمان .(TLS) والتي تقوم بمحاكاة عملية الاختراق من خلال عدة محاولات لخرق ضمان حماية البيانات (PFS) باستخدام عناصر هذه الأداة، ولقد قمنا بالفعل بالتحقق من أن الحل الذي اقترحناه يضمن حماية البيانات .(PFS) علاوة على ذلك، نوضح أن محاولة حل مشكلة (PFS) بواسطة (DHE)باستخدام أكثر من جزء من مفتاح واحد (Key-Share) سوف يتطلب تغييرًا جذريًا في بنية (DHE) والتي تنتج بروتوكول جديد يحتاج إلى مراجعات ودراسات مكثفة.

Acknowledgments

الحمد لله كما أمر والصلاة والسلام على حبيبنا الهادي والمبعوث رحمة للبشر، الذي تلقى رسالة العلم (إقرأ) من فوق سابع سماء ونقلها بمحبة إلى البشر، فاللهم إنا نسألك علماً نافعاً وزيادةً في العلم لننتفع بها وننفع اخواننا من البشر.

بادئ ذي بدئ أتوجه بالشكر الجزيل والامتنان والعرفان لحضرة الدكتور الغالي على قلبي د.أحمد السعده والذي شرّفني بأن أكون أحد الطلبة الذين حازوا شرف العمل تحت اشرافه وإتباع نصائحة وتوجيهاته القيمة والتي أعانتني على اتمام العمل على هذه الرسالة. فالعمل على الرسالة ساعدني على صقل مهارات البحث والتحليل والتواصل، ليس فقط على الصعيد الأكاديمي وإنما على الصعيد العملي والاجتماعي والديني والذي جعلني أتنبه أن الحياة هي الامتحان الأصعب والذي يجب أن نُعدّ له بجدية حتى نفوز في دنيانا وأُخرانا.

أريد أيضا أن أعبر عن فائق شكري واحترامي لجميع طاقم الهيئة التدريسية في جامعتنا العريقة بيرزيت بشكل عام وخصوصا اللذين تلقيت منهم المواد التعليمية خلال فترة الماجستير وهم: د.عبد اللطيف أبوعيسى، د.عدنان يحيى، د.إياد طومار، د.أحمد عفانه، د.واصل غانم، د.مأمون نواهضة، د. سامر الزين، والدكتور عبد الكريم عواد.

أريد أيضاً أن أشكر جميع زملاء الدراسة والعمل وجميع من ساهم في دعمي سواءً نفسياً أو معنوياً واللذين كان لهم الأثر الكبير في تجاوز الكثير من العقبات وأخص بالذكر الأخ العزيز والغالي والذي دعمني في جميع مراحل الدراسة أخي الحبيب هيثم الصالحي، ألف ألف شكر لا تكفي، جزاك الله عني كل خير ووفقك إلى ما يحب ويرضى.

وأخيراً وليس آخراً أشكر ومن أعماق قلبي أفراد عائلتي جميعاً واللذين لولاهم لم أخطُ خُطوة واحدة في حياتي، اليك يا والدتي الحبيبة والتي لم تنفك تدعو لي ليلاً نهاراً وتسأل عني قبل وبعد كل امتحان وكل واجب، حفظها الله ورعاها وبارك في عمرها وفي صحتها وفي عافيتها، وكذلك إلى روح والدي الحبيب رحمه الله والذي هو سبب دخولي في برنامج الماجستير، وكان الأب والأخ والصديق والناصح، رحمه الله وأسكنه الفردوس الأعلى، إلى أختي فادية والتي ما زال أثر دعائها يفتح لي كل الصعاب وييسر كل عسير بفضل الله، إلى ريم ومنى ومحمد وعمار وخليل وجميع أبنائهم، أحبكم جميعا من أعماق قلبي.

وفي الختام أشكر من كل قلبي زوجتي الحبيبة هبة والتي كانت بمثابة الأم والأخت والزوجة والتي كانت تساعدني في ترتيب جميع المواعيد وتساعدني حتى في الدراسة وفي الطباعة وتنسيق المشاريع والكثير الكثير مما لا يعد ويحصى فمليون شكر لايكفي، كذلك أبنائي اللذين تحملوني طوال هذه الفترة بحلوها ومرّها إلى رهف ومحمود وتاليا كل الحب والعرفان، أعانني الله أن أعوض ما فاتنا معاً من أيام جميلة.

وآخر دعوانا أن الحمد لله رب العالمين والصلاة والسلام على سيدنا محمد وعلى آله وصحبه أجمعين.

جامعة بيرزيت

BIRZEIT UNIVERSITY

Approved by the thesis committee:

Dr. Ahmad Alsadeh, Birzeit University

[illegible]

Dr. Abdalkarim Awad, Birzeit University

[illegible]

Dr. Iyad Tumar, Birzeit University

[illegible]

Date approved: 17/12/2018

Declaration of Authorship

I, Fadi ABU FARHA, am declare that this thesis titled, “Automatic Verification and Symbolic Analysis for 0-RTT Security in TLS 1.3” and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a master degree at Birzeit University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed my- self.

Signed: Fadi Abu Farha

Date: 17-12-2018

Contents

List of Figures

List of Tables

List of Abbreviations

0-RTT	**Z**ero-**R**ound **T**rip **T**ime
CH	**C**lient **H**ello
CR	**C**ertificate **R**equest
CSRF	**C**ross-**S**ite **R**equest **F**orgery
CTC	**C**lient **C**ertificate
CTS	**S**erver **C**ertificateVerify
CVC	**C**lient **C**ertificate**V**erify
CVS	**S**erver **C**ertificate
DHE	**E**phemeral **D**iffie-**H**ellman
EE	**E**ncrypted **E**xtensions
EF	**E**arly **F**inished
encKey	**e**ncryption **K**ey
FIC	**C**lient **F**inished
FIS	**S**erver **F**inished
HIBE	**H**ierarchical **I**dentity-**B**ased key **E**ncapsulation
HRR	**H**ello **R**etry **R**equest
HSTS	The **HTTP S**trict **T**ransport **S**ecurity
IETF	**I**nternet **E**ngineering **T**ast **F**orce
KE	**K**ey **E**xchange
MAC	**M**essagel **A**uthentication **C**ode
macKey	**m**essage **a**uthentication **K**ey
MD5	**M**essage **D**igest
MITM	**M**an **I**n **T**he **M**iddle
NST	**N**ew **S**ession **T**icket
PFS	**P**erfect **F**orward **S**ecrecy
QUIC	**Q**uick **U**DP **I**nternet **C**onnections
SH	**S**erver **H**ello
SHA	**S**ecure **H**ash **F**unction
TLS	**T**ransport **L**ayer **S**ecurity
trKey	**t**raffic **k**ey
TSH	**T**he handshake up to **S**erver **H**ello

Chapter 1

Introduction

The Transport Layer Security (TLS) protocol is one of the most widely spread cryptographic protocols in practice [1]. It is responsible for providing security over Internet connection to prevent tampering, message forgery and eavesdropping. TLS considered as the core of the Internet security infrastructure. 1995 was the date of birth of this protocol for the Netscape company that named it by Secure Socket Layer (SSL) [1]. This protocol has been growing up with much attention since that time and has been subject to improvements and updates ever since.

Most networks all over the world are not secure by itself. To securely communicate over these widespread networks, we need secure algorithms, namely protocols, which achieve security goals, such as authentication and secrecy.

TLS is the most important protocol that critically affects business infrastructures and modern networks. It's important to have security protocols, but most importantly is to verify these security protocols. Therefore, unverified protocols may cause the loss of money or seriously companies' damages.

1.1 Motivation

TLS has been repeatedly suffering from security weaknesses and deficiencies. Either on its cryptographic primitive or on the design of the TLS protocol itself. Accordingly, many modifications have been conducted to this protocol. Specifically, after the emergence of RC4 and BEAST attacks in 2011, which led to intense analysis and academic study that produces a vast work in less than five years [2]–[8], that is almost equal to the entire studies that have been done during the past two decades [9]–[13]. Many of these studies have revealed weaknesses and proposed great solutions in both, manual and automatic analysis [1].

According to TLS security weaknesses, occasional updates and recurrent modifications, TLS becomes more complicated, as well as the implementation and deployment process becomes hard to achieve. Specifically achieving low latency overhead for the key exchange protocol. Marc et al. [14] have targeted the 0-RTT protocol in order to raise up the speed of key exchange. On the other hand, Felix et al. [15] have shown that it is impossible to achieve PFS and to get secure against the replay attack in TLS 1.3 using 0-RTT. However, Google QUIC protocol have prevented replay attack in key exchange process but it fails to prevent replay attack for the data exchange, which remarks as a logical replay attack [14]. This motivates us to seriously do our best to find an approach that achieve full forward secrecy and get rid of replay attack problem.

1.2 Approaches to verify TLS security properties - 0-RTT

We divide our approaches into; general approach and specific approach as follows:

1.2.1 General Approach: Verifying TLS security properties in general

This approach shows the differences between manual and automatic verification of TLS security properties alongside with the needed tools.

Many approaches have been proposed in the last few years in order to check and verify the specifications of the TLS protocol. The first approach depends on manual verification of security protocols, using successful methods, for instance, exploration [16], induction [17], and belief logic [18] methods. However, this approach does not support automation, which makes the manual verification methods hard to achieve in practice according to the unbounded number of sessions.

Other proposed approach based on automatic tools; AVISPA tool set [19], FDR [20], ProVerif [21], Scyther [22], and Tamarin prover [23]. The main difference between these automatic tools is using the explore state space that is exploring all possible behaviors, or exploring strict subsets, namely scenario [24].

One of these automatic analysis approaches used label transition system and knowledge reasoning to sufficiently and correctly specify and verify security protocols using SeVe tool [25], which support anonymity and privacy along with security properties like authentication and secrecy.

Another two powerful approaches are ProVerif [21] and Scyther [22]. ProVerif is a protocol-specific abstractions that reduce the existence of attacks on a security protocol to the standard problem in logic; namely Horn clauses. The Scyther tool does not use any abstraction like ProVerif does, but both of them deal with a linear sequence of send and receive steps called roles. In addition, they both have almost the same performance [26].

The last approach is adapted from Scyther-proof's verification theory as a supportive model for non-monotonic state modeling and includes Horn-theories as a special case [26]. This model is practically implemented in the Tamarin prover tool, because it offers many unique features; it supports unbounded verifications of security properties, flexible properties, equational theories, global state, and falsification of the protocols, it also automatically construct many parallel interleaved protocol roles automatically, it supports loops and branches well and it supports Diffie-Hellman (DHE) as a built-in symbolic protocol inside Tamarin [27]. So, we built our approach using Tamarin prover tool to analyze and update 0-RTT model in TLS 1.3 specifications.

1.2.2 Specific Approaches: Verifying 0-RTT security properties

We show the most recent two approaches that aimed to solve the PFS and replay attacks pitfalls:

The first approach has been invented by Google. QUIC protocol uses a third party server to store the server key share, which allows the client to directly send its payload data at the first flight by combining its own key share with one of the server key shares. The second approach is done by Gunther et al. [15], they took advantage of using a puncturing algorithm, which immediately deletes the key used to decrypts the cipher text only once, then developing the session key rather than modifying the related public key. In order to achieve this, they have used a secure Hierarchical Identity-Based key Encapsulation (HIBE [28]) scheme to create many keys out of only one key, more details for these two approaches can be found in 2.1.1 and 5.0.3 sections.

1.3 Problem Statement and Scope

After viewing QUIC, Puncture algorithm and comparing manual vs. automatic ver- ification tools, we went through the proactive development process of TLS 1.3 speci- fication protocol, which aims to improve protocol efficiency using the analysis-prior- to-deployment process to address the weaknesses and to prevent attacks of TLS1.3 protocol, specifically achieving 0-RTT with PFS and preventing replay attacks before its first release instead of using the post-deployment-analysis process after releasing the TLS1.3 protocol (note: the proactive process used to be on draft-28 that we were working on, whereas the RFC8446 has released recently after defending this thesis). In this thesis, we look into analyzing and updating the model of 0-RTT protocol in the TLS 1.3 using an automatic analysis of a security protocol tool called; Tamarin prover [23].

1.4 Research Objectives and Methodology

The main objective of this study is to analyze the core security properties of the TLS 1.3 specifications, during the development phase; using the Tamarin prover tool [23]. Mainly, our focus on proving the PFS and preventing the replay attack on 0-RTT KE protocol. Proving/disproving the security properties of PFS and replay attack for 0-RTT KE protocol.

In this study, we follow a scientific approach of research; to analyze and update the model for 0-RTT in the TLS 1.3 using Tamarin prover tool. We accomplish our main objective through the following these steps:

1. Selecting the modeling tool: TLS 1.3 has its own security properties and KE protocols. We are going to use the Tamarin tool to analyze these security properties and create our own models or update existing models after testing and verifying the recent models; to finally achieve our target of proving 0-RTT KE protocol secrecy in order to make the entire protocol clean and safe.

2. Experimental design: Our experimental process focus on key exchange methods, either to exchange a list of DHE key shares; between parties, to be used one at a time, then deleting the used keys; or using a combination of PSK keys to achieves PFS. Moreover, we need to find a way to prevent the replay attack problem benefiting from using the ephemeral keys and fresh nonces or message authentication code (MAC).

3. Running experiments: In the first phase, we have tested, verified and confirmed the TLS protocol model that was created by Stettler [29] using Tamarin tool then, we have updated the 0-RTT model by adding many DHE key-share instead of one. Moreover, we did a combination of pre-shared keys to be used in the key exchange process. We noticed that we got a positive results, which converts the state of falsified (that denotes attack) in Stettler 's results to verified (which denotes a proof - secure protocol properties) as shown in Table 5.7.

4. Evaluating the results: We compared our results with Stettler 's results then, we made sure that our results are compatible with the TLS security protocol properties and if they are acceptable and achieve the TLS constraints.

1.5 Contributions

The main two folds contributions in this thesis are proving the PFS and preventing replay attacks for 0-RTT KE protocol, by proposing two methods, firstly, we used DHE hand shake KE protocol to theoretically prevent replay attacks by creating a list of ephemeral keys-shares for the client and the server, (clientshare : $x_1 = g^{a_1}, x_2 = g^{a_2}, x_3 = g^{a_3}$...etc., servershare : $y_1 = g^{b_1}, y_2 = g^{b_2}, y_3 = g^{b_3}$, ...etc.) these key-shares are exchanged between the parties in order to use them once per a session resumption, then vanishes, and regenerate another list before consuming the last key. Secondly, we used a PSK method to achieves PFS by combining multiple pre-shared keys ($psk = HKDF(< psk1, psk2 >)$), which allows the client to send early data with 0-RTT in the first flight of KE process alongside with keeping the PFS. To this end, we have tested, verified, confirmed, and partially updated some of security properties of TLS 1.3 protocol that done by Stettler [29]. Specifically, PSK using the Tamarin prover tool to model the protocol handshake process and the flow of messages between parties.

1.6 Outline

This thesis contains six chapters, which structured as follows:
Chapter 2 the foundation; including the history of Transport Layer Security protocol, through the development process of this protocol including threats, flaws and modification, showing the weaknesses and strengths of TLS protocol, and introducing cryptography basics. Chapter 3 presents more about the TLS attacks. We introduce the Tamarin prover tool in chapter 4. Chapter 5 includes modeling, analyzing and updating the TLS1.3 besides 0-RTT models using Tamarin tool. Then conclude and results take place in Chapter 6.

Chapter 2

Transport Layer Security (TLS) Foundation

Internet connection becomes one of the most important part of our lives, even more, the Internet has changed our lives. The world around us becoming more connected day after another, especially with the massive widespread of computers and smartphones, which we use to communicate, pay bills, shop online, travel, bank transactions and much more. In order to do these operations, we need a secure protocol in common, to protect our transmitted data across these interconnected devices [30].

2.1 TLS Protocol

The Internet connection was initially limited to number of users. For instance, universities have used it in scientific research. Therefore, no need for a high level of protection. Meanwhile, the communication protocols were implicitly insecure [30]. However, with the widespread of Internet connection and the emergence of e-commerce websites, the need for a strong and secure protocol has emerged. Consequently, In 1995, Netscape released the first version of TLS protocol, namely SSL protocol[1], which is one of the most important and widely used cryptography protocols that was introduced to provide secure communication over insecure infrastructure [1].

TLS protocol is one of the most important and widely used cryptography protocol that provides secure communication between two parties to exchange messages over TCP protocol.

The properly developed SSL/TLS protocol provides the client with a secure communication channel to the intended server, the information will arrive correctly and safely with no tampering or content change by others, protecting the transport layer link, which is the reason for naming the protocol as TLS. Besides the TLS security goals that are represented as follows:

- Cryptographic security: Enable exchanges the information between two parties in a secure communication by selecting the cryptographic algorithms.

[1]SSL is the predecessor protocol for TLS: The same protocol with two names (SSL/TLS).

- Interoperability: Developers have the ability to use the common cryptographic parameters under the same framework to build their libraries, code, and programs that easily intersects with each other without errors (Briefly: write once, run everywhere).
- Extensibility: TLS effectively deals with actual cryptographic protocols as a reference framework for deployment and development, aiming at allowing migration between the existing primitives instead of creating new protocols and to be independent of the actual cryptographic primitives used.
- Efficiency: Achieving an acceptable performance cost, providing session cache scheme and minimizing the costly cryptographic to insure achieving the above three goals.

2.1.1 TLS Protocol Brief History

The SSL/TLS protocol has a long history, back to November 1994 when Netscape released the second version, called SSL2.0 after the first version had cryptographic flaws and never saw the light. The second version didn't last longer, after the first deployment of SSL2.0 using Netscape Navigator version1.1 in March 1995, it had cryptographic and practical flaws, because Netscape did not ask for any expert consultation from outside. Few months later, Netscape comes out with a stronger, more secure and new design version namely SSL3.0 protocol.

In May 1996, after political issues and a real competition between Netscape and Microsoft to control the Web, both companies have agreed to support the IETF taking over the protocol, which leads to release TLS1.0 in January 1999, as RFC2246. The TLS1.0 is the same as SSL3.1 that is an adapted version of SSL3.0, done by IETF.

In April 2006, TLS1.1 was released including only security fixes. Taking in consideration the incorporation of TLS extensions that were released in June 2003 as a major change to TLS1.0.

In August 2008, TLS1.2 was released, all hard-coded security primitives were removed from the specification and additional support for authenticated encryption. These modifications increased the elasticity of the protocol.

A few months ago, while working on this thesis, in August 2018, TLS1.3 was lastly released after a hard work by the IETF working group to make it stronger, less complex, and optionally faster if using 0-RTT, taking into account the trade-off between speed versus weaker anti-replay attacks. But, we are still able to use 0-RTT in some cases where the impact of replay attack is less severe or the level of security offered by TLS1.3 is not required. However, we propose a solution to the 0-RTT problem to prevent replay attacks and the PFS problem as well.

Zero Round-Trip Time (0-RTT)

The time between sending a message back and forth between two parties is called round-trip time (RTT) during the key exchange process (KE). Reducing the complexity of this round-trip time was the major concern of the KE protocols' designers. Many low-latency designs for KE has been proposed in several researches [31], [32].

Google's QUIC protocol and TLS 1.3 protocol are practical examples that establish an initial key in zero round-trip time 0-RTT, which allows the client to send his key share message alongside with early data to a pre-visited server. It's known that the client key-share without the server key-share contribution does not guarantee the same strong security as standard key exchange protocols (classical KE protocol that needs a full round-trip time). Particularly, the forward secrecy cannot be provided by the initial key-share since no shared state between sessions. Moreover, most of the keying material is compromised after running the KE protocol except the ephemeral keying material. The protocol achieves forward secrecy in the second step; after the server contributes its key-share.

2.2 TLS Location in Network Layers

IP and TCP protocols are considered major pillars of Internet construction, which are responsible for good, solid, and reliable communication between two entities on the network over the familiar Internet Protocol (IP) address, in order to send/receive data packets across many computer nodes (hops) for a long or short path. This insecure path makes the transferred data vulnerable to be stolen or hijacked, this is because the IP and the TCP protocols do not provide security by themselves. Many other vulnerable protocols, such as BGP can be exploited by the attackers. Moreover, the attacker may replace these routing protocols, and redirect the connection to himself.

Even though the attackers might be able to hijack the encrypted data, they still cannot decrypt it. To guarantee message authenticity, TLS uses the Public-Key Infrastructure (PKI) algorithm.

As shown in Table 2.1, the known Open Systems Interconnection (OSI) model, which includes seven layers, starting from the physical layer up to application layer. The SSL/TLS lays between TCP and application layer, we can still work directly with TCP and remove SSL/TLS from our model -if the encryption process is not necessary- without affecting other protocols. When there is a necessity for encryption, we just use SSL/TLS to encrypt HTTP. Moreover, we can encrypt other important protocols, such as SMTP, IMAP and any other TCP protocol.

TABLE 2.1: OSI Model Layers

Layer No.	OSI Layer	Function/Description	Examples Protocol
Layer7	Application	Application data	HTTP, SMTP, IMAP
Layer6	Presentation	Data representation, conversion, encryption	SSL/TLS
Layer5	Session	Management of multiple connections	Net-BIOS, Sockets
Layer4	Transport	Reliable delivery of packets and streams	TCP, UDP
Layer3	Network	Routing and delivery of datagrams between network nodes	IP, IPsec
Layer2	Data link	Reliable local data connection (LAN)	Ethernet
Layer1	Physical	Direct physical data connection (cables)	CAT5

2.3 TLS Cryptography

Cryptography was almost limited to military, diplomatic and government applications till the 1970s, then some financial and telecommunication industries start using it during the 1980s. Nowadays, cryptography has become one of the most important topics in our daily life. For instance, shopping using credit card, voice-over-IP phone calls, e-health applications and the evolution of smart cities will make cryptography even more ubiquitous.

Today, cryptography algorithms are much stronger, security definitions are better understood, and new algorithms have replaced the old broken ones. A lot of intersection between computer science, math, and electrical engineering areas to produce a secure cryptography system, which means, we need to combine more than one scientific field to finally get secure cryptographic methods/systems.

Paar et al. [33] defines Cryptography as " The science of secret writing with the goal of hiding the meaning of a message". Which was the first branch of cryptology. The second branch is cryptanalysis, which is a way of breaking cryptosystems to get the plaintext without a need to know the encryption details. The cryptanalysis is a way of securing the system. Since, without cryptanalysis, we will never know if our system is secure or not. So, many researchers use this technique to check the security level of the system.

Cryptography includes the main branches:
Symmetric Algorithms: Is the way for two parties having an encryption and decryption methods to communicate and exchange the data securely with each other, using the same shared secret key. Symmetric methods were solely the base of cryptography from antiquity until 1976. Data encryption and messages integrity check are still done by symmetric ciphers.

Asymmetric Algorithms (Public key): Whitfield Diffie, Martin Hellman, and Ralph Merkle have introduced different types of encryptions methods in 1976 rather than the symmetric key, the main difference is that the user has another key (Public-key) and the private key that introduced in the symmetric key cryptography. We can use asymmetric algorithm in digital signature, key establishment, and data encryption.

Cryptographic Protocols: The secure Internet communications applications can be accomplished through the symmetric and asymmetric algorithms that considered to be a building blocks. TLS is an example of cryptographic protocols that deals with cryptographic algorithms [33].

Cryptographic Hash Function:
A hash function is a compressed fixed length output of a numerical value for an arbitrary length of the same numerical input value.

The properties of cryptographic hash function is summarized in Pre-Image Resistance, which prevents reversing the hash function process. Second Pre-Image Resistance, which prevents to find the same hash result for different inputs -each input value has its unique hash result. Collision Resistance, which makes it impossible to have the same hash function result for two different inputs. Many hash functions can be used to protect the password storage and data integrity check, such as message Digest family (MD5), Secure Hash Function family (SHA) and Message Authentication Code (MAC) that provides authentication using a symmetric key cryptographic technique, and many other hash functions [30].

2.3.1 Symmetric Encryption

The symmetric encryption has been used thousands of years ago. The case with most early ciphers is to keep the method itself secure. For instance, the substitution cipher method for encryption is replacing each alphabet letter with another one (replace A to k, B to d, C to w). We reverse the process for decryption. Many approaches were adopted over time. One approach in the 19th century for a cryptographer Auguste Kerckhoffs: "A cryptosystem should be secure even if the attacker knows everything about the system, except the secret key". Kerckhoffś principle makes sense considering the following:

- To have useful encryption algorithm, others must share it. The greater the number of people with access to the algorithm, the probability of falling the algorithm in the enemy’s hands will increase too.
- It’s inconvenient to use a none-key single algorithm in large groups; since the communication decryption is allowed by everyone.
- The design of a good encryption algorithm is very hard. The more inspection, observation, and examination for the algorithm, the more secure it can be. Usually, cryptographers need many years of breaking attempts to make

sure their cipher is secure. So, they recommend a conservative approach when adopting new algorithms [30].

When the attacker couldn't analyze or retrieve the plaintext, in this case, we could say the encryption algorithm is good (secure) especially when the ciphertext is randomly produced. For instance, the attacker could easily reveal the substitution cipher by inspecting the frequent letters, which as known in English language that the frequent of some letters repeated more often than others, which leads the attacker to reveal the plaintext by observing these frequent letters. So, obviously, the simple letter substitution cipher is not a good algorithm. Otherwise, if we have a good cipher, the attacker has to try all possible decryption keys, which know as Brute-Force or exhaustive key search.

We conclude that, the key is the main factor of the ciphertext security. So, if the selection of the key from a large keyspace and many iterations have been done to break the encryption through large number of possible keys, then the cipher is computationally secure.

2.3.2 Stream Ciphers

A keystream is producing an infinite stream of random data from a stream cipher. For encryption, one byte of keystream is combined with one byte of plaintext using XOR operation. Vice-versa for decryption process, which done XORing the ciphertext with the same keystream byte as shown in FIGURE 2.1.

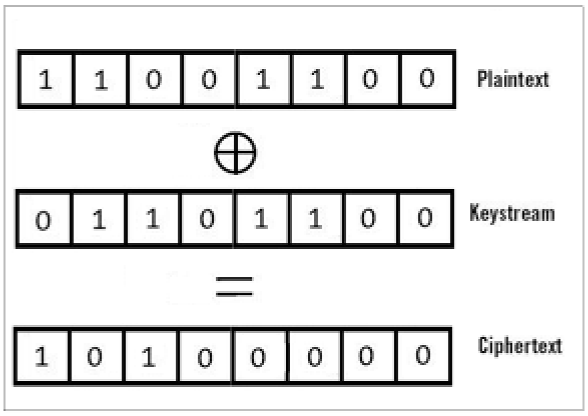


FIGURE 2.1: Stream Cipher process [34]

If the attacker couldn't predict the position of each key-stream bytes, which key-stream bytes are at which position?, then we say that the encryption process is secure. So, it's recommended to use the stream ciphers just once with the same key, and that's because the attacker can predict the plaintext at certain locations, practically; when we encrypt HTTP connection, all requests (e.g., protocol version, header names) will be the same during the same connection.
Knowing the plaintext and having access to the correspondent ciphertext, will give the attacker opportunity to reveal parts of key-stream, using the same information will reveal other parts of ciphertext in the future; if the same key reused. To get rid

of this issue, we could derive and use one time keys from long-term keys in stream algorithms.

2.3.3 Block Ciphers

Block ciphers encrypt the whole blocks of data at a time. Most block ciphers nowadays use 16 bytes block size (128 bits). A block cipher takes some input data and transforms it to random output. Using the same key will produces exactly the same output for the same input combination. A small variation of input produces a large variation of output.

Block cipher have some limitations. For instance, the produced output is always the same for the same input (deterministic problem), which makes it vulnerable to attacks. Also, the encryption of data has a limited length equal to the block size length [30].

In practice, block cipher modes are encryption schemes that use the block cipher to decrease the limitations and to add authentication for the process. Some cryptographic primitives (e.g., MAC, pseudo-random generators, and hash functions) also uses block ciphers as a base for the encryption process.

Advanced Encryption Standard (AES) that is available in different strengths (128, 192, and 256 bit) is the most popular block cipher [30].

2.3.4 Padding

Padding is the extra data to be added/appended to the plaintext when the block size is less than 16-byte when using 128-bit AES, which is one of the approaches for handling the encryption of data lengths that are smaller than the encryption block size.

The padding must consist of a distinct data and the number of bytes to be discarded must be known to the receiver. For instance, the padding length of TLS can be found in the last byte of an encryption block, which determines the number of padding bytes. All padding bytes shares the same value as the padding length byte. Finally, the receiver has the ability to check the correctness of the padding.

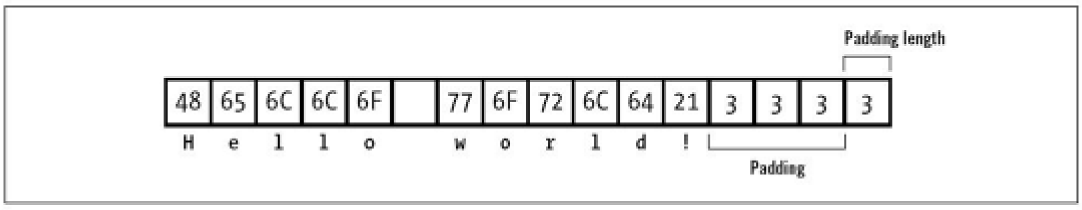


FIGURE 2.2: TLS padding illustration [30]

The last byte of the data block contains the number of padding to be removed, the receiver removes it first, then the indicated number of bytes will be removed too, as shown in FIGURE 2.2.

2.4 Protocols Overview and Structure

SSL/TLS includes two layers of protocols as shown in FIGURE 2.3; TLS Handshake protocol produces a cryptographic parameters that used by the secure channel. This Handshake protocol takes place at a very first step of communication between the client and server. This allows the negotiation about the protocol version between peers, choose cryptographic algorithms and establishing shared secret keying material. After completing the handshake process, the established keys are used to protect the application layer traffic. Any error or failure of the Handshake protocol causing the termination of the connection and sometimes an alert message precedes this termination. The Handshake protocol includes cipher specs, SSL Alert, and HTTP/FTP to provide security for the second upper layer protocol; namely the Record Protocol, which provides a secure session between two or more parties.

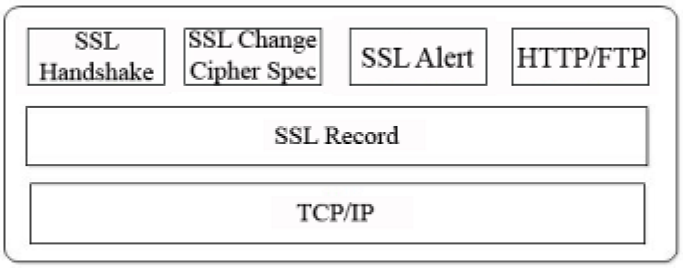


FIGURE 2.3: Structure of SSL/TLS Protocol [35]

TLS record protocol is a layered protocol. Each layer consists of messages that include fields for description, length, and content. The record protocol (sender) is responsible for data fragmentation (by segmenting it into a number of chunks), message transition, besides optional data compression, applying/computing the MAC, data encryption using corresponding MAC, and transmission of the results. On the other side (receiver); the backward operation is needed for decrypt, verify, decompress, reassemble the received data, and finally deliver the data to higher-level clients. Compression is disabled in SSLv3.0 and above [35] since it is vulnerable to one of the brute force attack (CRIME attack). SSL/TLS record protocol creation operations shown in FIGURE 2.4.

Fragmentation done to every received messages, the maximum allowed chunk size is 2^{14} [35] bytes. But when applying compression; the length of chunk is not more than 1024 [35] bytes.

Compression Algorithms: Compression techniques used to decrease data size without any loss of data.(e.g., LZ77, GZIP etc.)

Hash Algorithms: Hash functions provide integrity to data chunks (e.g., MD5, SHA-1, SHA-256)

Encryption Algorithms: Data could be encrypted using symmetric stream or block cipher techniques to create SSL payload. In the stream cipher encryption; chunks and MAC are encrypted together. Before block cipher encryption padding

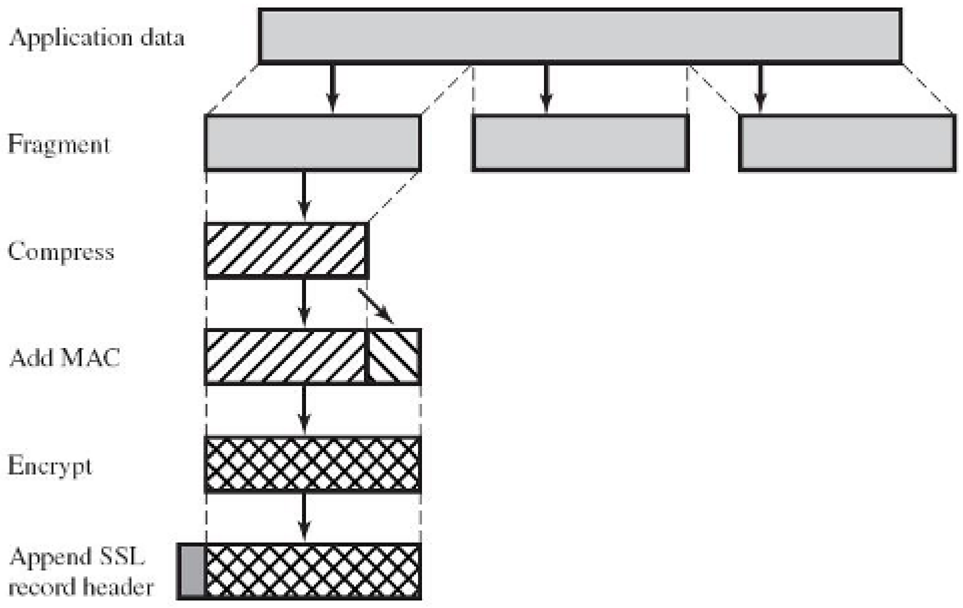


FIGURE 2.4: Operations of SSL/TLS Record Protocol [36]

bits are added to both MAC and chunk. Table 2.2 shows the encryption algorithms with their key sizes.

TABLE 2.2: (Stream & Block) Cipher Encryption [35]

Stream Cipher	
Algorithms	Key Sizes (bits)
RC4	40, 128
Block Cipher	
Algorithms	Key Sizes (bits)
3DES	168
AES	128, 192, 256
DES	56
Fortezza	80
IDEA	128
RC2	40

The Change Cipher Spec Protocol is the simplest protocol with one byte single message holding the value 1, used to update the current state by copying the pending state on it, to finally change the used cipher suite.

The Alert Protocol is used to announce the compressed and encrypted alert messages related to SSL protocol negotiation faults; to peer devices. Alert protocol message is two bytes long; one byte includes two values: (one 1) for warning and (two 2) for fatal, the fatal terminates the specific connection directly. However, the other connections during the same session continue working; but no new connections will be established. The other byte contains specific alerts represented by specific code that indicates the severity degree, as shown in Table 2.3. Three fields included in the handshake protocol (type: represented in 1 byte, length: represented by 3 bytes,

TABLE 2.3: Alert Protocol Messages [35]

Code	Alert	Representations	Type
0	close_notify	No more messages on this link to receiver	1
10	unexpected_message	Inappropriate message to receiver	2
20	bad_record_mac	Incorrect MAC record to receiver	2
21	decryption_failed	Invalid decryption due to improper chunk size	2
30	decompression_failure	Decompression fail due to improper input	2
40	handshake_failure	Negotiation fail due to improper security parameters set	2
41	no_certificate	Reply to no proper certificate is available	1
42	bad_certificate	Corrupted certificate or contains invalid signature	1
43	unsupported_certificate	Sender certificate is unsupported	1
44	certificate_revoked	Certificate was withdrawn by signer	1
45	certificate_expired	Issued certificate is no longer valid	1
46	certificate_unknown	An uncertain problem causes certificate to be inappropriate while handling	1
47	illegal_parameter	Security parameter are inconsistent with respect to their field in handshake	2

and content: greater than or equal to 0 bytes for associated parameter with the message) code and security parameters for handshake messages shown in Table 2.5 and Table 2.4. SSL Handshake Protocol: Before the connection is established by the record

TABLE 2.4: Cipher Suites for SSL [35]

Parameters	Values
Key exchange algorithms	RSA, Diffie-Hellman, Fortezza
Cipher algorithm	RC4, RC2, DES, 3DES or IDEA,Fortezza
MAC algorithm	MD5 or SHA
Cipher type	Stream or Block
MAC size	MD5(0 or 16 bytes) or SHA (20 bytes)
IV size	Initialization vector size used in CBC

protocol; here comes the handshake protocol to allow client and server to exchange the needed parameters (e.g., cipher suite, Identities, nonces) before start exchange the application data as shown in FIGURE 2.5.

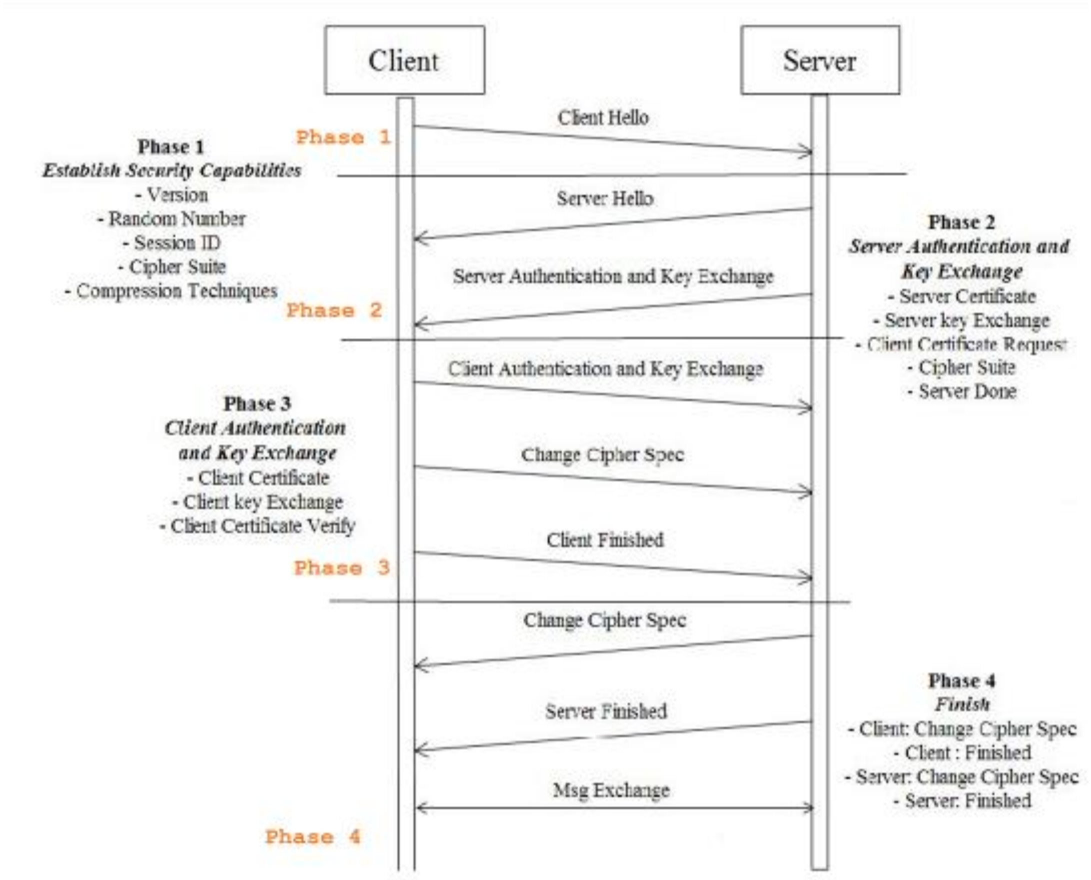


FIGURE 2.5: Operations of Handshake Protocol [35]

2.5 TLS1.3 New Features

TLS1.3 came with major changes to TLS1.2, some of these key changes are:

- Adding 0-RTT mode to save round trip connection for application data, sacrificing some security properties.
- Version negotiation was removed to increase compatibility for servers which fail to implement version negotiation.
- Single new PSK exchange has replaced session resumption with and without server-side state and PSK based cipher suites.

TLS1.3 Exchange Modes Diffie-Hellman (DHE), pre-shared key (PSK) exchange and a combination of both; (DHE and PSK) are the three key exchange modes with different properties that offers an elastic security guarantees for the TLS1.3. This allows session resumption and early data transmission.

TLS1.3 offers three post-handshake mechanisms, post handshake client authentication, and sending new session ticket (NST) by a PSK for subsequent resumption, to cover the traffic key updates. The communication between parties must be documented and both parties need to agree on.

TABLE 2.5: SSL Handshake Messages [35]

Codes	Messages Type	Parameters
0	hello_request	Void
1	client_hello	version, random_no, session_id, cipher_suite, compression_tech
2	sever_hello	version, random_no, session_id, cipher_suite, compression_tech
11	certificate	X.509 certificates chain
12	server_key_exchange	msg_signature, public_parameters
13	certificate_request	cert_authorities, cert_type
14	server_done	Void
	client_key_exchange	msg_signature, public_parameters
15	certificate_verify	cert_signature
20	finished	MD5_hash, SHA_hash

Chapter 3

Attacks on TLS

TLS has been vulnerable to several majors attacks, especially on the most common used ciphers and its modes of operations. For instance, RC4 and AES-CBC suffered from serious attacks; since a combination of both is widely used in TLS context. More about the major attacks on TLS can be found in [37].

According to an old saying to the US National Security Agency "Attacks always get better; they never get worse", therefore the attacks will never stop.

As there are many attacks on TLS protocol, there are a lot of security solutions recommendation that have been proposed by IETF [38], the FIGURE 3.1 shows the attacks and analysis line for TLS protocol followed by number of TLS attacks:

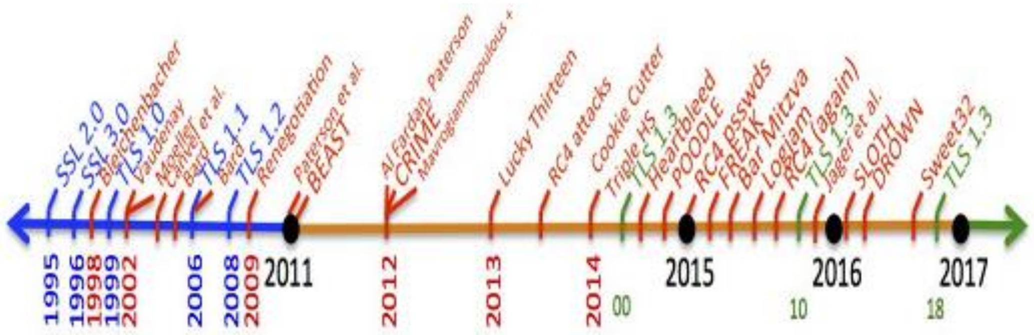


FIGURE 3.1: Attacks and Analysis Line on TLS protocol [39]

3.0.1 SSL Stripping

Stripping attack introduced by Moxie Marlinspike in 2009 [40]. This attack prevents the use of SSL/TLS, by exploiting and modifying unencrypted protocols, such as HTML and HTTP that requests the use of TLS.

Stripping attack has derived from downgrade attack, which is more general. These attacks affects; only if the user starts accessing web servers using HTTP. The HTTP Strict Transport Security (HSTS) specification was subsequently developed to mitigate these attacks.

3.0.2 STARTTLS Command Injection Attack

This attack targets the transmitted packets between unprotected and TLS protected traffic. Many application level commands (e.g., STARTTLS) used by IETF application protocols to upgrade the cleartext connection in order to use TLS. The attackers exploited a flaw in STARTTLS, which is retaining a pipelined STARTTLS command with an application layer input buffer. These commands received prior to TLS negotiation and executed after TLS negotiation. To solve this problem, it's required to keep the application level command input buffer to be empty before TLS negotiation. Well, this flaw does not affect the TLS protocol directly since it's considered to reside in the application layer code.

As STARTTLS is vulnerable to downgrade attacks, other similar mechanisms are vulnerable too. It is a simple mission for the attackers; they have just to remove the STARTTLS indication from the HTTP/unprotected request. Adding HSTS-like solutions will mitigate the attack [41].

3.0.3 BEAST

The BEAST attack targets TLS1.0 and earlier versions. BEAST has violated origin policy constraints for the cipher block chaining (CBC), which is the predictable Initialization Vector (IV). To this end, Duong and Rizzo exploit this known weakness in IV construction in 2011. They predict the IV to decrypt small parts of a packet (HTTP cookies) when it's run over the TLS protocol. The problem was solved in TLS1.1 [41], [42]. Duong and Rizzo, have proved that the attacks get better with time, and we have to seriously deal with any small weaknesses (e.g., IV weaknesses) and do not ignore them since they could grow big eventually.

To make BEAST attack works, the attackers have effectively reduced the CBC mode to Electronic Code Book (ECB) mode. ECB splits input data into blocks and individually encrypts each block. The problem with this approach is that; the output data of a block is always the same for the same encrypted block. This facilitates the attacker's job and makes it possible to guess the plaintext, as follows:

1. The attacker monitors the size of encrypted blocks, which depends on the encrypted algorithm, for example, 16 bytes for AES-128.

2. The attacker submits 16 bytes of plaintext for encryption since he could guess the whole block at once, in addition, any difference in any bit of input will affect all output bits.

3. The attacker monitors the encrypted block and compare it to the ciphertext in the first step if there is no difference, then we have the first correct guess, or the attacker goes back to the second step.

Note: The attacker can guess one block at a time. So, the attacker needs to make 2^{127} guesses on average in order to guess 16 bytes only.

In order to hide patterns in the ciphertext. The CBC masks every message before encryption using IV, which is differentiate it from ECB. Moreover, the ciphertext output is not always the same for the same input block. Therefore, the attacker couldn’t guess the plaintext like ECB.

To have an effective IV, we need to make it unpredictable for each message. Many unfeasible solutions have been proposed, but the practical one in CBC is to use only one block of random data at the beginning, then the output of the current block used as input for the next block, which also known by chaining. The chaining approach is safe, if and only if the attacker is not able to monitor the encrypted data. else, if the attacker could reach one encrypted block, so he will have the IV for the next. TLS 1.0 and earlier deals with the entire connection as a single message and use the random IV for the first TLS record. All following records use the last encryption block as their IV. The attacker could see all the encrypted data; so he knows the IVs from the second one and above. TLS 1.1 and 1.2 do not suffer from this weakness since they use per-record IVs.

Finally, the protocol is still vulnerable to a blockwise chosen plaintext attack. CBC effectively downgrade to EBC when the IV is predictable. FIGURE 3.2 shows the attack against CBC with predictable IV. The figure includes three encrypted blocks; the browser sent two of these blocks, while the third one has been sent by the attacker through the browser.

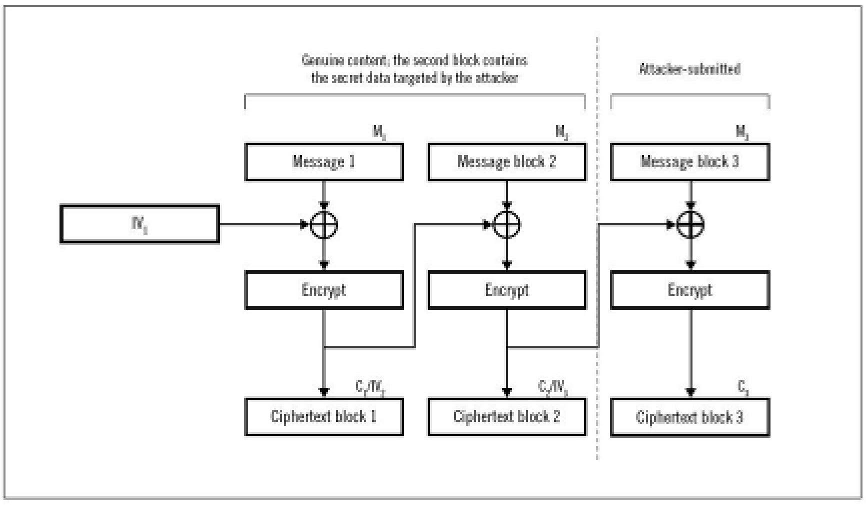


FIGURE 3.2: BEAST attack against CBC with predictable IV [30]

The IV of the first block is unknown, so the attacker targets the second block to reveal its content. The attacker knows the IV_2 after seeing the first block. The same goes with IV_3 after seeing the second block. Moreover, the second block C_2 is also known to the attacker.

The attacker now has seen the first two blocks, the attacker keeps observing the encrypted version on the wire. The IVs are all known to the attacker, so the effect of IVs is eliminated from attacker guesses. When the attacker complete guessing is successful, then the C_3 (an encrypted version of the guess) will be the same as C_2 (the encrypted version of the secret).

3.0.4 Padding Oracle Attacks

The MAC-then-encrypt design is used in all versions of the TLS protocol, which leads to the padding oracle attacks [43]. The Lucky Thirteen attack [3] and a timing side-channel attack that helps the attacker to decrypt arbitrary ciphertext are samples of padding oracle attacks.

We can mitigate the Lucky Thirteen attack using authenticated encryption, such as AES-GCM [44] or using encrypt-then-MAC [45] instead of MAC-then-encrypt. The latest version of the padding oracle attack is POODLE attack [46] on SSL 3.0, no timing information used in this attack.

3.0.5 Attacks on RC4

The RC4 algorithm [47] has been firstly used with SSL then TLS for years. RC4 suffered from different kinds of cryptographic weaknesses for a long time (e.g., [48], [49], [50]). The biases property in RC4 keystream is one of the weak points that attackers exploit to retrieve the plaintexts that have been encrypted many times [41], as shown in cryptanalysis results [51].

According to the result of 2014, which inform us that most of the above attacks are practically exploitable. We conclude that RC4 doesn't provide an adequate level of security for TLS, The link[1] includes more details.

3.0.6 Compression Attacks: CRIME, TIME, and BREACH

Active attacker is able to decrypt ciphertext (HTTP cookies) using CRIME attack [52]. This attack happens when using compression with TLS level. Both TIME [53] and BREACH [54] attacks decrypt the secret data passed in the HTTP response using HTTP level compression. Most attackers prefer the use of HTTP message body compression more than compression at TLS level.

In order to mitigate TIME attack, we just need to disable TLS compression. No information available about BREACH attack mitigations at TLS protocol level, therefore, application level mitigations are needed [54]. For instance, HTTP implementations that use Cross-Site Request Forgery (CSRF) tokens will need to randomize them.

3.0.7 Certificate and RSA-Related Attacks

When using TLS with RSA certificate many practical attacks have shown up, the most used attacks are Bleichenbacher [10] and Klima [13]. Bleichenbacher has been mitigated in TLS 1.0, while Klima that relies on a version-check oracle in TLS 1.1.

[1] **https://tools.ietf.org/html/draft-ietf-tls-prohibiting-rc4-01**.

The exploitable timing issues, such as Brumley [55] are often involved when using RSA certificates. Unless they are explicitly eliminated by the implementation. Many vulnerabilities have been uncovered in different TLS libraries related to certificate validation; using the certificate fuzzing tool namely Brubaker [56].

3.0.8 Theft of RSA Private Keys

Any encrypted sessions that were initiated by any server using TLS with most non-Diffie-Hellman cipher suites can be easily decrypted by obtaining and using the private key for that server. The popular Wireshark network sniffer uses this technique in order to inspect TLS-protected connection.

A large-scale monitoring [57] for certain servers, including stealing private keys, active or passive wiretaps (eavesdrop), and traffic analysis were used as part of pervasive monitoring. The mitigations for such attacks through better protecting the private key, for instance, using hardware solutions or operating system protections. Moreover, using cipher suites for forward secrecy, which prevent the passive attackers of exposing the past or future sessions, even if they could reveal the private key.

3.0.9 Diffie-Hellman Parameters

The Cross-Protocol attack [58] exploits the interactions between the different cipher suites. Specifically, when the client incorrectly interprets the signed Elliptic Curve Diffie-Hellman ECDH key parameters as valid plain Diffie-Hellman.

The adversary could impersonate the server and use it as an oracle that provides signed parameters, then start sending these signed valid parameters to the victim clients. In order to mitigate Cross-Protocol attack, we can use predefined DHE groups [59].

Additionally, the client has to properly verify any received parameters, or he will be vulnerable to MITM attack. It is unfortunate that, the TLS protocol don't offer this verification, more information about analogous information for IPsec in RFC6989 [60].

3.0.10 Renegotiation

SSL and TLS renegotiation are vulnerable to an attack in which the attacker creates a TLS connection with the server, sends some content data to the server, at the same time; the attacker creates a new TLS connection with the client, who sends his initial TLS handshake to the server, which treats it as a renegotiation, considering the initially transmitted data by the attacker as a subsequent from the real client. The attacks and it's resolved found in RFC5746 [61].

Chapter 4

Tamarin Prover for Security Protocols Modeling

There are many ways to verify security protocols. In this thesis, we focus on au- tomatic verification of security protocols in symbolic models of cryptography. Ex- tended the scope of automatic verification; to cover more practical security problems has been subjected to much research. However, it is impossible to verify all security protocols. Since (in general) there are many requests from the server at the same time, therefore verifying all security properties needs more time, so we decided to focus on verifying PFS and preventing the replay attacks in our thesis.

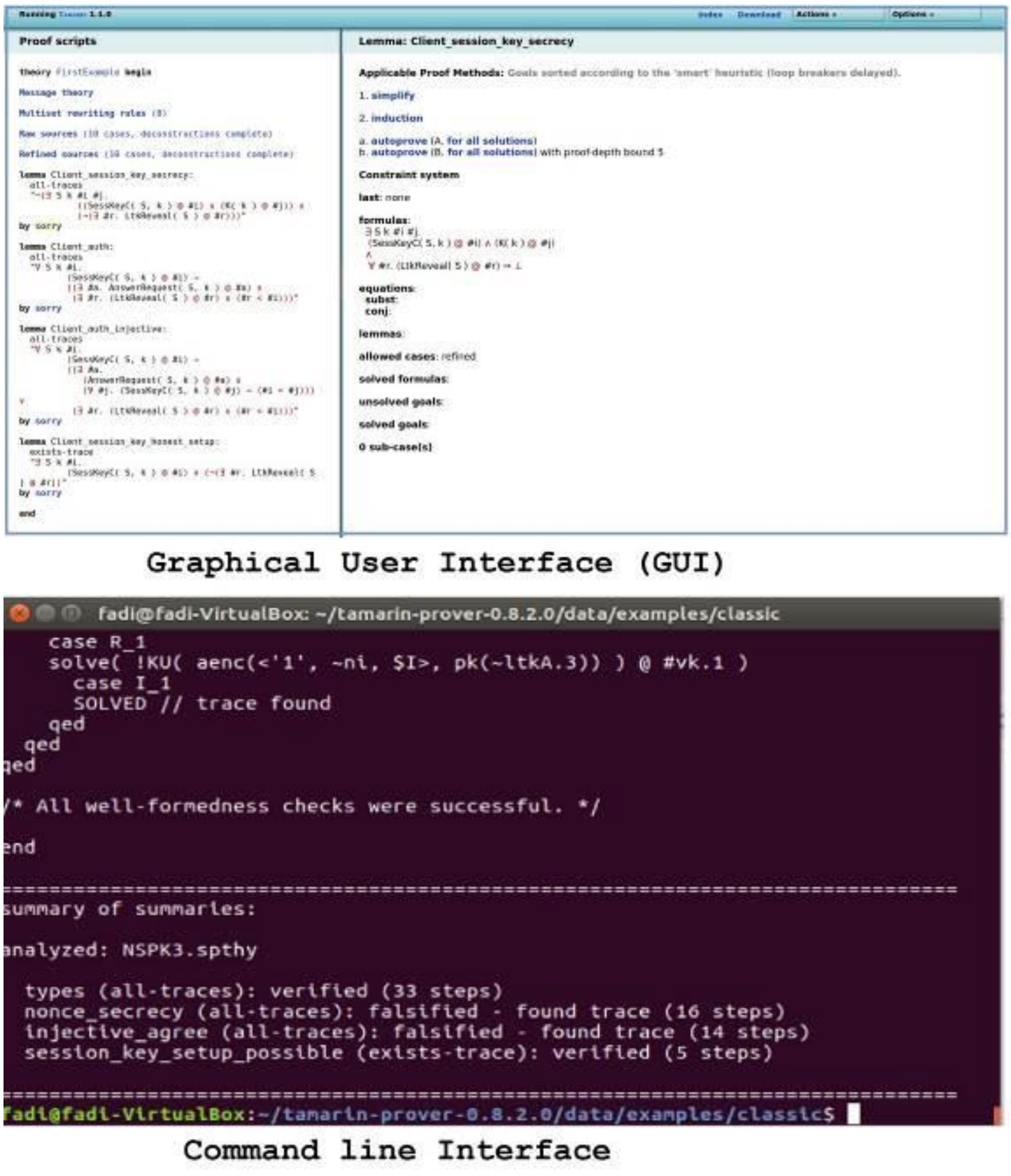
We use Tamarin prover tool [62] for automatic verification. Tamarin supports automatic falsification and verification of protocols, benefit from loops and non-monotonic states, and of protocols that use Diffie-Hellman exponentiation to fulfill resilience against adversaries.

This chapter explains the automated analysis for some security protocols using constraint-reduction rules [29].

4.1 Automatic Protocol Analysis

The constraint solving algorithms is the main idea that automated analysis is based on. It consists of two components, on a high level. Firstly, a constraint-reduction strategy, which leads to a possibly infinite search tree. Secondly, a search strategy that used to search the tree for a solved constraint system.

Tamarin prover tool is used to implement the constraint solving algorithms based on constraint-reduction rules. Tamarin provides automatic and interactive interface modes. Iterative deepening search strategy is used in the automatic mode to se- lect the next constraint-reduction rule to be applied (command line interface). The Graphical User Interface (GUI) is used in the interactive mode, which allows the user to determine interactively both the search and the constraint-reduction strategy as shown in FIGURE 4.1.



```
Proof scripts

theory FirstExample begin

Message theory

Multiset rewriting rules (8)

Raw sources (10 cases, deconstructions complete)

Refined sources (10 cases, deconstructions complete)

lemma Client_session_key_secrecy:
  all-traces
  "¬(∃ S k #i #j.
          ((SessKeyC( S, k ) @ #i) ∧ (K( k ) @ #j)) ∧
          (¬(∃ #r. LtkReveal( S ) @ #r)))"
by sorry

lemma Client_auth:
  all-traces
  "∀ S k #i.
        (SessKeyC( S, k ) @ #i) ⇒
        ((∃ #a. AnswerRequest( S, k ) @ #a) ∨
         (∃ #r. (LtkReveal( S ) @ #r) ∧ (#r < #i)))"
by sorry

lemma Client_auth_injective:
  all-traces
  "∀ S k #i.
        (SessKeyC( S, k ) @ #i) ⇒
        ((∃ #a.
           (AnswerRequest( S, k ) @ #a) ∧
           (∀ #j. (SessKeyC( S, k ) @ #j) ⇒ (#i = #j)))
∨
         (∃ #r. (LtkReveal( S ) @ #r) ∧ (#r < #i)))"
by sorry

lemma Client_session_key_honest_setup:
  exists-trace
  "∃ S k #i.
        (SessKeyC( S, k ) @ #i) ∧ (¬(∃ #r. LtkReveal( S
) @ #r))"
by sorry

end
```

```
Lemma: Client_session_key_secrecy

Applicable Proof Methods: Goals sorted according to the 'smart' heuristic (loop breakers delayed).

1. simplify

2. induction

a. autoprove (A. for all solutions)
b. autoprove (B. for all solutions) with proof-depth bound 5

Constraint system

last: none

formulas:
 ∃ S k #i #j.
  (SessKeyC( S, k ) @ #i) ∧ (K( k ) @ #j)
 ∧
 ∀ #r. (LtkReveal( S ) @ #r) ⇒ ⊥

equations:
 subst:
 conj:

lemmas:

allowed cases: refined

solved formulas:

unsolved goals:

solved goals:

0 sub-case(s)
```

```
fadi@fadi-VirtualBox: ~/tamarin-prover-0.8.2.0/data/examples/classic
    case R_1
    solve( !KU( aenc(<'1', ~ni, $I>, pk(~ltkA.3)) ) @ #vk.1 )
      case I_1
      SOLVED // trace found
    qed
  qed
qed

/* All well-formedness checks were successful. */

end

==============================================================================
summary of summaries:

analyzed: NSPK3.spthy

  types (all-traces): verified (33 steps)
  nonce_secrecy (all-traces): falsified - found trace (16 steps)
  injective_agree (all-traces): falsified - found trace (14 steps)
  session_key_setup_possible (exists-trace): verified (5 steps)

==============================================================================
fadi@fadi-VirtualBox:~/tamarin-prover-0.8.2.0/data/examples/classic$
```

FIGURE 4.1: Graphical user interface and Command line interface

4.2 Constraint Solving Algorithms

A constraint-reduction strategy is a partial function (**r**) from constraint systems Γ (where Γ is a finite set of constraints, **r** is a partial function that represents the constraint-reduction strategy) to finite sets of constraint systems that meet the following conditions:

1. The constraint-reduction relation $\{(\Gamma, \mathbf{r}\,(\Gamma)) | \Gamma \in \mathrm{dom}(\mathbf{r})\}$ is correct, complete, and well-formed.
2. Every well-formed constraint system not in the domain of **r** has a non-empty set of solutions.

Certainly, the function (**r**) minimizes constraint systems in its domain to fixed/finite sets of constraint systems that cover the same set of solutions, while any constraint systems out of its domain will be marked as solved [63].

4.3 The Tamarin Prover

The Tamarin prover is one of the automatic verification tools, it supports equational theories, which takes into account the cryptographic operators, alongside with their properties and the adversary's ability to deduce messages, in order to modeling Diffie-Hellman, multisets [63] and bilinear pairings.

To avoid undecidable automatic verification that comes out from theoretical results, we need to bind at least two out of the following three quantities:

- Number of messages.
- Number of sessions.
- Number of nonces.

Thus, Tamarin prover could also use automatic verification without bounding any of the above three quantities, which known as a symbolic backward implementation with complete proofs. But in this case, the implementation process may not be terminated because of the infinite number of messages, sessions, and nonces.

4.3.1 Formalism

This section includes the underlying formalism for Tamarin prover tool. We use the equational theory to represent cryptographic messages using Tamarin, then we model protocol execution by formalizing the labeled multiset rewriting system. Lastly, we specify the security properties by Tamarin [29].

4.3.1.1 Messages

An order-sorted term algebra used by Tamarin in modeling cryptographic messages. Tamarin defines two sub-sort messages; fresh messages (nonces) that represent freshly generated values, and pub messages that represent publicly known values. A signature defines the term algebra, which specifies the function symbols used in the definition of terms:

Definition 1 (Signature): A signature Σ is a set of function symbols, each having an arity $n \geq 0$. The arity 0 function symbols $(n{=}0)$ are called constants, where arity means the number of variables in the function, for instance, $\mathsf{fst(pair(x,y))=x}$ has an arity of two variables x and y, which means $(n{=}2)$.

Follows an example of the operators used for modeling of symmetric encryption/decryption:

Example: The signature $\Sigma ex = \{\mathsf{senc},\ \mathsf{sdec}\}$ defines the binary function symbols used for modeling symmetric encryption and decryption. Together with a set of variables, we can now inductively define the term algebra, the set of all possible terms over a signature Σ.

Definition 2 (Term Algebra): Let $\mathcal{X}$ be a set of variables (disjoint from Σ). The term algebra over Σ, denoted as $T_\Sigma(\mathcal{X})$, is the least set such that:

- $\mathcal{X} \subseteq T_\Sigma(\mathcal{X})$
- $t_1, t_2, ..., t_n \in T_\Sigma(\mathcal{X})$ and $f \in \Sigma$ with arity $n \Rightarrow f(t_1, t_2, ..., t_n) \in T_\Sigma(\mathcal{X})$

Cryptographic messages of Tamarin can be defined as follows (i.e., Σmsg is concrete signature):

Definition 3 (Message): A message is a term in $T_{\Sigma}msg(\mathcal{X})$ where the signature is defined as $\Sigma msg = A \cup F \cup Func \cup \{pair, fst, snd\}$ [29].

- $\mathcal{X}$: set of variables
- A: set of agent names ($\in$ pub)
- F: set of fresh values ($\in$ fresh)
- Func: set of user-defined functions (e.g., hashing)
- pair(t_1, t_2): pairing; $t_1, t_2, ..., t_n$ are terms
- fst: first element of a pair
- snd: second element of a pair

By default, the function symbols for pairing can be found in the signature Σmsg. While any other functions used by the protocol are elements of Func.

Example: Modeling symmetric encryption includes adding the corresponding function symbols to the set of user-defined functions:

senc, sdec $\in$ Func. With this definition, the terms t_1 := sdec(senc(x,y), y) and t_2 := x are messages in $T_{\Sigma}msg(\mathcal{X})$ (sdec and senc represents symmetric decryption and symmetric encryption respectively).

All algebra in the above definitions, are called free algebra, which means; each term is interpreted syntactically. Briefly, t_1 and t_2 are syntactically different and would not be considered the same. then an equational theory is used to clarify the semantic equivalence of messages.

Definition 4 (Equational Theory): An equational theory is a set of equations, where an equation is a pair of terms, t, t' $\in$ $T_\Sigma(\mathcal{X})$, written as t=t'.

The equational theory defines an equivalence relation between the terms in $T_\Sigma(\mathcal{X})$, the term algebra then partitioned into equivalence classes. The resulting quotient algebra $T_\Sigma(\mathcal{X})|_{=E}$ interprets each term t by its equivalence class $[t]_E$.

Example: The equational theory of pairing consists of the following equations:

- fst(pair(x,y)) = x, fst means, we need to take the first value of the pair(x,y) since x is the first value and y is the second value.
- snd(pair(x,y)) = y

The equational theory defines the meaning of the functions, while the term algebra defines the message structure in terms of function symbols.
Example: In the equational theory, containing the equation sdec(senc(x,y),y) = x, the messages t1 and t2 would be in the same equivalence class, and considered semantically equivalent. Much more of Tamarin's cryptographic messages can be found in Meier 's PhD thesis [26].

4.3.1.2 Supported Verification Problems

Tamarin verifies and analyzes the validity claims of trace formulas for protocols that use the public networks, which have active and passive attackers. Specifically, a Dolev-Yao style adversary who has the ability to control these networks. This analysis is performed modulo an equational theory, modeling the semantics of the employed cryptographic algorithms.

The supported equational theories combined from:

- An arbitrary subterm-convergent rewriting theory.
- Modeling Diffie-Hellman exponentiation.
- The equations modeling multiset union.
- The equations modeling bilinear pairing.

In order to simplify the verification problems, Tamarin allows restricting the set of considered traces using axioms that implements rules with inequality checks, such as: Firstly, add InEq(t,s) to the rules, which needs to have two different terms t and s. Secondly, filtering the traces where t and s are instantiated to the same message, like $\forall$ x i. InEq(x, x) @i $\Rightarrow$ $\perp$. (where InEq(x,x) represents two
different values for each x; they are not equal. $\perp$ means
false)

The protocol, the considered equational theory, and the expected properties jointly as security protocol theory; will be specified according to the input given to the Tamarin prover. Formally, a security protocol theory is a six-tuple
$T = (\Sigma, E, P, \alpha, \phi, \chi)$ the description follows:

- Σ: Specifies the functions for constructing cryptographic messages.
- E (equational theory): specifies the semantics of the functions in Σ.
- P: specifies a set of protocol rules.
- α (the axioms of T): Specifies sequences of closed trace formulas α.
- ϕ (the validity claims of T): Specifies sequences of closed trace formula ϕ.
- χ (the satisfiability claims of T): Specifies sequences of closed trace formula χ [26].

As shown in the equations below, the security protocol theory T is true if its validity and satisfiability claims are achieved for the traces of $P \cup MD_{\Sigma}$ satisfying the axioms:

$$P \cup MD_{\Sigma} \; \forall_{E} \; (\wedge_{\alpha \in set(\alpha)} \alpha) \Rightarrow \qquad for \;\; each \;\; \phi \in set(\phi)$$

In this equation: A set of protocol rules (P) alongside with message deduction rule MD_{Σ} achieves the validity claims ϕ of the security protocol theory (T).

$$P \cup MD_{\Sigma} \; \exists_{E} \; (\wedge_{\alpha \in set(\alpha)} \alpha) \wedge \chi \quad for \;\; each \;\; \chi \in set(\tilde{\wedge})$$

In this equation: A set of protocol roles (P) alongside with message deduction rule MD_{Σ} achieves the satisfiability claims χ of the security protocol theory (T).

4.3.1.3 Execution and State

Tamarin uses a labeled multiset rewriting system for protocol execution. Moreover, the state transitions are modeled by multiset rewriting rules. The state of that transition system are multisets of facts.

Definition 5 (Fact). All arguments of facts are terms in $T_{\Sigma}(X)$. Facts are the elements of the multisets which represent the state of the transition system.

Tamarin have three types of facts In, Out and Fr, we use Out facts to model the adversary knowledge and messages on the network. In facts used to model a party receiving a message that controlled by Dolev-Yao from the untrusted network, and the fresh (Fr) facts used to model nonces, which are very important facts to keep the security of information.

Example: The fact K(x) means that the adversary knows the term x. The fact Out(x) means that the term x was sent across the network by a protocol participant, ready to be learned by the adversary. The fact In(x) indicates that x has been seen by the adversary and is ready to be received by an agent.

Facts also describe the protocol participant's state. The arguments of the state facts can be considered as the knowledge of the participant. A common notation is used for the names of state facts, as shown in the following example:

Example: The fact St_A_3(a, sk) denotes that agent a, executing the protocol in role A, is in it's third internal state and knows the key sk.

A statement can be seen as one fact. Depicting just a small part of the state, while multiset specifies the whole state of such facts.

Linear versus persistent facts:

linear facts appear in just one state, and not the other. They could consumed by rules as well as they are produced by rules.

Persistent facts denoted by the prefix (!) and it never removed from the state. **Definition 6 (Multiset and State)**. A multiset is a set where elements may occur more than once (Elements have a multiplicity). The denoted operations on multisets with a # sign next to the usual set operator. A state is a multiset of facts, meaning that the same fact may occur multiple times in a state.

Four types of facts consist of the state of a protocol execution:

- State facts name (St_A_4(a, sk))
- Adversary knowledge facts (K(sk)); which means the adversary knows the session key (sk).
- Messages on the network facts (Out(sk)); which means sending the (sk) to an unsecured network.
- Fresh facts (Fr(na)); fresh nonces.

Example: A state fact describing two agents executing a protocol, each in the first state of the corresponding role, defined as follows:

S_i = [St_A_1(a,sk,b), St_B_1(b,sk,a), K(sk), Fr(na), K(n)]

Each agent knows the name and the shared key sk of the other agent. There is also a fresh fact in the state, and the adversary knows sk as well as the value of a unique nonce n, which is the same as na but it is not fresh anymore when it is revealed by the adversary.

Each step in the protocol execution corresponds to a change of the state, which called state transition that represented by a multiset rewriting rule, which consists of a left-hand side (l), namely premise, and a right-hand side (r), namely conclusion, and a label (a) as well.

Definition 7 (Labeled Multiset Rewriting). A labeled multiset rewriting rule is a triple (l), (a), (r), each of which is a multiset of facts. We write such a rule using either the short, in-line notation l − [a] → r, or in the form of a deduction rule:

$$\frac{l_1 \quad l_2 \quad \dots \quad l_k}{r_1 \quad r_2 \quad \dots \quad r_m}\, a$$

where $l_1, l_2, ..., l_k$ are the facts of the premise, and $r_1, r_2, ..., r_m$ are the facts of the conclusion, and (a) is the action fact [29]. We can also rewrite the above equation in a form: $l_1, l_2, ..., l_k$ − [a] → $r_1, r_2, ..., r_m$.

4.3.2 Protocols Modeling

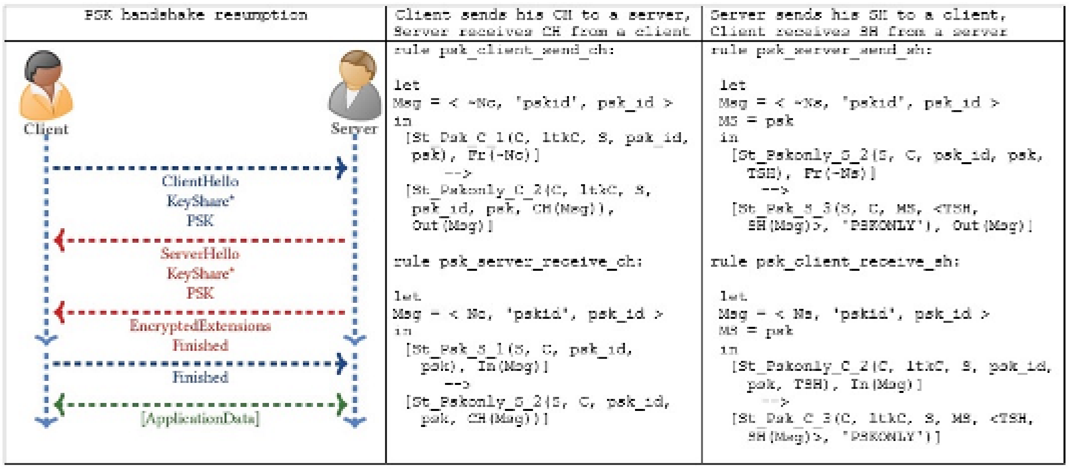
The execution of security protocols could be defined in many ways, according to the used tool. Using Tamarin, the user has no limitations of modeling the security protocols in the way he/she choose, since there is no pre-defined protocol concept.

In this section, we present some of the previous Tamarin models of TLS security protocol. We have examined and verified these models, which helps us in updating and building our own models accordingly, to achieve our goal of proving PFS and preventing replay attacks.

To start our protocol Modeling we need to consider the following:

- Our models includes three main factors; the client, server and the adversary that is able to modify, inject and delete the messages on the network represented by Dolev-Yao adversary.

- We write our security properties models using Tamarin tool with a standard format, which needs to choose a name for the theory in the first line for every security properties file in Tamarin, preceded by the keyword **theory** (e.g. theory TLSsecurity) after that we use **begin** keyword to indicate the start of Tamarin security code, then we translate the security properties by constructing **rules** for initializing the client, the server and the handshake process including **client hello**, **server hello** and **Finished** messages ...etc. Then we close our Tamrin file using end keyword as shown in FIGURE 4.2.
- We declare the cryptographic primitives used by the protocol, the multiset rewriting rules that models the protocol, then, writing the security properties lemmas to be proven or refuted/disproven.
- Finally, we execute the security properties theory using Tamarin tool to get our results.



PSK handshake resumption	Client sends his CH to a server, Server receives CH from a client	Server sends his SH to a client, Client receives SH from a server

```
rule psk_client_send_ch:

let
Msg = < ~Nc, 'pskid', psk_id >
in
  [St_Psk_C_1(C, ltkC, S, psk_id,
   psk), Fr(~Nc)]
        -->
  [St_Pskonly_C_2(C, ltkC, S,
   psk_id, psk, CH(Msg)),
   Out(Msg)]

rule psk_server_receive_ch:

let
Msg = < Nc, 'pskid', psk_id >
in
  [St_Psk_S_1(S, C, psk_id,
    psk), In(Msg)]
        -->
  [St_Pskonly_S_2(S, C, psk_id,
    psk, CH(Msg))]
```

```
rule psk_server_send_sh:

 let
 Msg = < ~Ns, 'pskid', psk_id >
 MS = psk
 in
  [St_Pskonly_S_2(S, C, psk_id, psk,
    TSH), Fr(~Ns)]
      -->
  [St_Psk_S_3(S, C, MS, <TSH,
    SH(Msg)>, 'PSKONLY'), Out(Msg)]

rule psk_client_receive_sh:

 let
 Msg = < Ns, 'pskid', psk_id >
 MS = psk
 in
  [St_Pskonly_C_2(C, ltkC, S, psk_id,
    psk, TSH), In(Msg)]
      -->
  [St_Psk_C_3(C, ltkC, S, MS, <TSH,
    SH(Msg)>, 'PSKONLY')]
```

FIGURE 4.2: Part of PSK resumption Handshake

A set of traces is defined in Tamarin using first-order logic formulas over time-points and action facts. The syntax for specifying security properties is defined as follows:

- **All** for universal quantification, temporal variables are prefixed with #
- **Ex** for existential quantification, temporal variables are prefixed with #
- ==> for implication, & for conjunction
- **|** for disjunction
- **not** for negation
- **f @ i** for action constraints, the sort prefix for the temporal variable 'i' is optional
- **i < j** for temporal ordering, the sort prefix for the temporal variables 'i' and 'j' is optional (i and j for temporal variables - could be any variable)

- **#i = #j** for an equality between temporal variables ‘i’ and ‘j’
- **x = y** for an equality between message variables ‘x’ and ‘y’ (x and y for message variables - could be any variable)

Example of Protocol Modeling - NAXOS

NAXOS models the DHE handshake protocol that shown in FIGURE 4.3

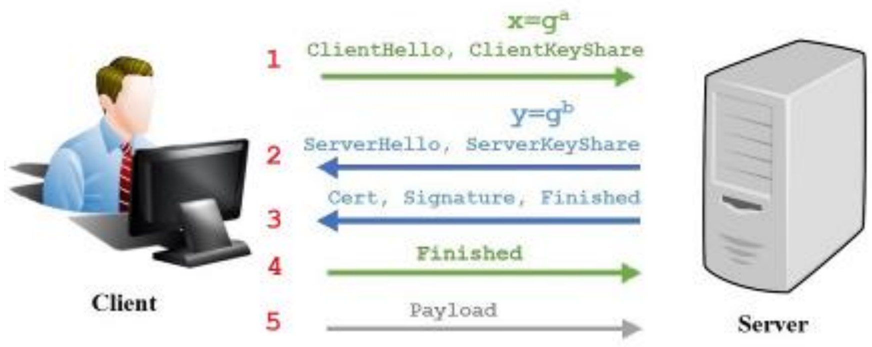

x=g^a
1
ClientHello, ClientKeyShare
y=g^b
2
ServerHello, ServerKeyShare
Cert, Signature, Finished
3
Finished
4
Client
5
Payload
Server

FIGURE 4.3: Diffie-Hellman handshake protocol

Starting of modeling a Public Key Infrastructure (PKI), no pre-defined notation in Tamarin for the PKI. Accordingly, we can model a pre-distributed PKI with an asymmetric key for each party using a single rule to generate a key for a party. The identity, the private and public keys are stored in the state as facts, allowing protocol rules to restore them. (Pk fact denotes public key, Ltk fact denotes long-term private key, ~x: denotes a fresh value of variable x, $A: denotes the identity of an agent A). The rule Generate_key_pair: To generate client’s public and private keys, the client generates his key-share, then sends it to the server as FIGURE 4.3 show in step (1).

```
rule  Generate_key_pair:
[ Fr(~x) ]
  ⇒
   [ !Pk($A,pk(~x))
   , Out(pk(~x))
   , !Ltk($A,~x)
   ]
```

Some protocols depend on algebraic properties of the key pairs. In many DHE-based protocols; g^x: is a public key, x: is a private key, it enables exploiting the commutativity of the exponents to establish the keys, as shown in the following rule:

```
rule  Generate_DH_key_pair:
[ Fr(~x) ]
  ⇒      [ !Pk($A,’g’^~x) , Out(’g’^~x) , !Ltk($A,~x) ]
```

4.3.3 Modeling of Protocol steps

The protocol steps that user/agent able to perform are, receives a message, responds by sending a message, or starting a session.
Modeling the responder role is simpler than the initiator role and can be done in one rule. It uses a DHE exponentiation and two hash functions (h_1, h_2 that must be declared by the user), the model for the Naxos responder role looks like:

```
rule  NaxosR_attempt1:
  [ In(X),
  Fr(~eskR),
  !Ltk($R,  lkR)
  ]
    ⇒
     [
     Out(  'g'^h1(<  ~eskR,  lkR  >)  )
     ]
```

~eskR: fresh value, $R: Responder , kR: a session Key, lkR:Long-term private key.
The responder also computes a session key kR into the action state:

```
rule  NaxosR_attempt2:
  [ In(X),
  Fr(~eskR),
  !Ltk($R,  lkR)
  ]
    -[  SessionKey($R,  kR  )  ]->
       [ Out( 'g'^h1(< ~eskR, lkR >) ) ]
```

To specify the communication of kR without decreasing the reliability and to avoid duplication and mismatches; we use binding to have the following:

```
rule  NaxosR_attempt3:
let
  exR  =  h1(<  ~eskR,  lkR  >)
  hkr  =  'g'^exR
  kR  =  h2(<  pkI^exR,   X^lkR,  X^exR,  $I,  $R  >)
in
    [ In(X),
    Fr(  ~eskR ),
    Fr(  ~tid  ),
    !Ltk($R, lkR),
    !Pk($I,  pkI) ]
     -[  SessionKey(  ~tid,   $R, $I,  kR  )  ]->
       [ Out( hkr  ) ]
```

~**tid**: Thread identifier, $pkI: The public key of the communication partner, **I**: Communication partner.

The model for Naxos initiator role sends a message and wait for the response. While waiting, other agents might also perform steps. Accordingly, modeling the initiator using two rules.
First rule; sending a message:

```
rule  NaxosI_1_attempt1:
let
 exI = h1(<~eskI, ~lkI >)
 hkI = ’g’^exI
in
   [ Fr( ~eskI ),
   !Ltk( $I, ~lkI )
   ]
     –>
      [  Out( hkI )  ]
```

While the initiator is waiting for the response; we need to consider receiving this message by the responder, which can be represented by storing the initiator’s thread (sending a message at the first rule) in the state face transition. This allows us to return back and complete the steps where we have left off. The optimization of the sending a message rule including the state fact thread must be as follows:

```
rule  NaxosI_1:
let
 exI = h1(<~eskI, ~lkI >)
 hkI = ’g’^exI
in
   [ Fr( ~eskI ),
   !Ltk( $I, ~lkI )
   ]
     –>
      [ Init_1( ~tid, $I, $R, ~lkI, ~eskI ),
      Out( hkI )]
```

The second initiator rule:

```
rule  NaxosI_2:
let
 exI = h1(<~eskI, ~lkI >)
 kI = h2(< Y^ lkI, pkR^exI, Y^exI, $I, $R >)
in
   [ Init_1( ~tid, $I, $R,~lkI ,  eskI),
   !Pk( $R, pkR ),
   In( Y ) ]
     -[ SessionKey( ~tid, $I, $R, kI ) ]->
      [ ]
```

Y: a message to be received from the network.

The output is not needed in this case, since there are no further steps in the protocol. The same agent identities and the exponent for an initiator exI computed in the first step will be used in this step. At last, the complete example; including initiator and responder will look like [62]:

```
theory Naxos
begin
builtins: diffie-hellman
functions: h1/1
functions: h2/1
rule Generate_DH_key_pair:
 [ Fr(~x) ]
   ⇒
    [!Pk($A,'g'^~x)
    , Out('g'^~x)
    , !Ltk($A,~x)
    ]
rule NaxosR:
let
 exR = h1(< ~eskR, ~lkR >)
 hkr = 'g'^exR
 kR = h2(< pkI^exR, X^~lkR, X^exR, $I, $R >)
in
   [ In(X),
   Fr( ~eskR ),
   Fr( ~tid ),
   !Ltk($R, ~lkR),
   !Pk($I, pkI) ]
    -[ SessionKey( ~tid, $R, $I, kR ) ]->
     [Out( hkr )]

rule NaxosI_1:
let
 exI = h1(<~eskI, ~lkI >)
 hkI = 'g'^exI
in
   [ Fr( ~eskI ),
   Fr( ~tid ),
   !Ltk( $I, ~lkI ) ]
    –>
     [ Init_1( ~tid, $I, $R, ~lkI, ~eskI ), Out( hkI ) ]
rule NaxosI_2:
let
continue ...
```

```
...continue
  exI = h1(<~eskI, ~lkI >)
  kI = h2(< Y^~lkI, pkR^exI, Y^exI, $I, $R >)
in
   [Init_1( ~tid, $I, $R,~lkI , ~eskI),
   !Pk( $R, pkR ), In( Y )]
    -[ SessionKey( ~tid, $I, $R, kI ) ]->
      [ ]
end.
```

4.3.4 Property Specification

This section presents how to specify trace and observational equivalence properties of the protocol, depending on action facts in the model.

4.3.4.1 Trace Properties

The system state of Tamarin is a multiset of facts with an empty multiset as an initial system state. The rules define how the system can make a transition to a new state. The protocol's behavior is explained by the action facts.

Each rule contains three parts, left and right-hand sides, and the action facts. We replace the left-hand side by the right-hand side when it is already contained in the current state, in this case; the rule can be applied to a state fact. Appending the action facts to the trace in order to record the application of the rule in the trace. The explanation is in the following example.

```
rule fictitious:
  [ Pre(x), Fr(~n) ]
   -[ Act1(~n), Act2(x) ]->
     [ Out(<x,~n>) ]
```

The rule consumes the facts Pre(x) and Fr(~n) and produces the fact Out(<x,~n>). It is labeled with the actions Act1(~n) and Act2(x). We can apply the rule if we found two arguments matching the x and ~n variables in the Pre and Fr facts. When applying this rule, x and ~n are instantiated with the matched values and the state transition is labeled with the instantiations of Act1(~n) and Act2(x). The time of occurring for these two instantiations is at the same time-point.

We can use action fact symbols in formulas. There are limited terms of these facts and allowed to be built only from quantified variables, free function symbols and public constants. Excluding equation's function symbols. The most importantly, is to guard all variables or the Tamarin tool will produce an error.

Guardedness: Check of occurring the universally quantified variables directly after a quantifier in an active constraint, the same goes with existentially quantified variables. For outermost logical operator inside the quantifier are an implication universally quantified variables and conjunction with the existentially quantified variable. Using parentheses is recommended, especially with a precedence of logical connectives, but keeping the standard precedence. The highest priority is for negation, then conjunction, then disjunction and then implication.

To verify a specific property for the protocol, we use Lemma followed by the property name and a guarded first-order formula. This means that the property must hold for all traces of the protocol. For example, the freshness of the value (~n) is distinct in all applications of the rule, or we identify the same instance by timepoint for the same fresh value that appears twice, we write:

```
lemma distinct_nonces:
 "All n #i #j.  Act1(n)@i & Act1(n)@j ==> #i=#j"
  or equivalently
   lemma distinct_nonces:
   all-traces
    ” All n #i #j.  Act1(n)@i & Act1(n)@j ==> #i=#j"
```

To express that there exists a trace for which the property hold, we add (exists-trace word) after the name and before the property. For example, the following lemma is true if and only if the preceding lemma is false [62]:

```
lemma distinct_nonces:
 exists-trace
  "not All n #i #j.  Act1(n)@i & Act1(n)@j ==> #i=#j".
```

4.3.5 Security Properties

In this section, we are including some security properties from Tamarin manual to focus on and to adapt them according to our model. Secrecy and authentication are our interesting properties.

4.3.5.1 Secrecy

Secrecy is the term property that unknown to the adversary. For instance, secrecy of a term t is satisfied for agent A, if the term t is not compromised by the adversary or its communicated party, or it's even unknown to the adversary. The conventional secrecy claim is an action fact includes the agent name and the term that is claimed to be secret. Using this structure, any role can prove secrecy of any claimed term. We use lemmas to prove secrecy claims made by a role as shown in the below example:

```
lemma example_role_secrecy:"All T tag x
 #i. Example_Role_Claim_Secret(T, tag, x)
 @i
   ==> (not (Ex #j.  K(x) @j)
   |(Ex A#j.  Rev(A)@j & Honest(A) @i))"
```

4.3.5.2 Perfect Forward Secrecy PFS

Perfect Forward Secrecy (PFS) is a strong security property that keeps the established session keys between parties secure even if the long-term secret keys are exposed [31]. This prevents the eavesdropper from revealing the secret data of past communications [64].
To differentiate between secrecy and perfect forward secrecy we need to consider the following two examples:

```
lemma secrecy:
 "All x #i.
 Secret(x) @i ==>
  not (Ex #j. K(x)@j)
   | (Ex B #r. Reveal(B)@r & Honest(B) @i)"
```

This lemma indicates that the agents are supposed to be honest whenever the message x is kept secret and not been compromised. We also can read it as follows: The lemma states that whenever a secret action Secret(x) occurs at timepoint i, the adversary does not know x or an agent claimed to be honest at time point i has been compromised at a timepoint r.

At a timepoint i the occurring of a secret action Secret(x) prevents the adversary of knowing x, or the compromisation is done at time point r where it should be honest at time point i according to agent's claim.

A PFS is a stronger secrecy property, which requires that a Secret action labeled messages remain secret before compromisation.

```
lemma secrecy_PFS:
 "All x #i.
  Secret(x) @i ==>
   not (Ex #j. K(x)@j)
    | (Ex B #r. Reveal(B)@r & Honest(B) @i & r < i)"
```

The following example (one message protocol) distinguish between secrecy and PFS. To send an encrypted message between two agents A and B. An agent A encrypts and sends a message to an agent B; using agent B public key. The secrecy are claimed by both agents but only agent A claims the secrecy of the message. Two action facts roles are applied Role ('A') and Role ('B') for A and B agent's roles.

The PFS claim is not applied to agent A. This can be reflected by negating the PFS property using an exists-trace lemma [62].

```
theory secrecy_template
begin
builtins: asymmetric-encryption
/* Protocol formalization for:
1. A -> B: A,napk(B)
*/
continue ...
continue ...
// Public key infrastructure
rule Register_pk:
[ Fr(~ltkA) ]
-->
[ ... ... !Ltk($A, ~ltkA)
, !Pk($A, pk(~ltkA))
, Out(pk(~ltkA))
] rule Reveal_ltk:
[ !Ltk(A, ltkA) ] – [ Reveal(A) ]-> [
Out(ltkA) ]

// Initialize Role A
rule Init_A:
[ Fr(~id)
, !Ltk(A, ltkA)
, !Pk(B, pkB)
]
–[ Create(A, ~id), Role('A') ]->
[ St_A_1(A, ~id, ltkA, pkB, B)
]

// Initialize Role B
rule Init_B:
[ Fr(~id)
, !Ltk(B, ltkB)
, !Pk(A, pkA)
]
–[ Create(B, ~id), Role('B') ]->
[ St_B_1(B, ~id, ltkB, pkA, A)
]

// Role A sends first message
rule A_1_send:
[ St_A_1(A, ~id, ltkA, pkB, B)
, Fr(~na)
]
–[
Send(A, aenc{A, ~na}pkB)
, Secret(~na), Honest(A), Honest(B), Role('A')
]->
[ St_A_2(A, ~id, ltkA, pkB, B, ~na)
, Out(aenc{A, ~na}pkB) ]
continue ...
```

```
...continue
// Role B receives first message
rule B_1_receive:
[
St_B_1(B,~id, ltkB, pkA, A)
, In(aenc{A, na}pkB)
]
–[ Recv(B, aencA, napkB)
, Secret(na), Honest(B), Honest(A), Role('B')
]->
[ St_B_2(B,~id, ltkB, pkA, A, na)
]
lemma executable:
exists-trace
"Ex A B m #i #j. Send(A,m)@i & Recv(B,m) @j"
lemma secret_A:
"All n #i. Secret(n)@i & Role('A') @i ==>
(not (Ex #j. K(n)@j)) | (Ex X #j. Reveal(X)@j & Honest(X)
@i)"
lemma secret_B:
"All n #i. Secret(n)@i & Role('B') @i ==>
(not (Ex #j. K(n)@j)) | (Ex X #j. Reveal(X)@j & Honest(X)
@i)"

lemma secrecy_PFS_A:
exists-trace
"not All x #i.
Secret(x) @i & Role('A') @i ==>
not (Ex #j. K(x)@j)
| (Ex B #r. Reveal(B)@r & Honest(B) @i & r < i)"

end
```

4.3.6 Authentication Properties

In this section, we will define a hierarchy of increasingly stronger authentication properties. which means, the non-injective agreement implies both weak agreement and aliveness. In general, two types of claim events are used to analyze most authentication properties as follows: The protocol ends with the producing the Commit action fact by the role A, "Commit (a, b, <'A','B',t>)". The other role; B in his turn produce the corresponding action fact "Running(b, a, < 'A', 'B', t>)". A and B are roles for the agent names a and b respectively, with known term t. The relationship between these two facts will need different requirements that imposed between each of the following properties, except aliveness [29].

4.3.6.1 Aliveness

The aliveness of an agent b is guaranteed by the protocol; to an agent a in role A, if the agent a completes a run of the protocol, apparently with b in role B, then b has previously been running the protocol, as shown in the lemma.

```
lemma aliveness:
 "All a b t #i.
  Commit(a,b,t) @i
   ==> (Ex id #j.  Create(b,id) @ j)
     | (Ex C #r.  Reveal(C)@ r & Honest(C) @ i)"
```

4.3.6.2 Weak agreement

The weak agreement of an agent b is guaranteed by the protocol with an agent a if, whenever the agent a completes a run of the protocol, apparently with b in role B, then b has previously been running the protocol, apparently, with a, as shown in the lemma.

```
lemma weak_agreement:
 "All a b t1 #i.
  Commit(a,b,t1) @i
   ==> (Ex t2 #j.  Running(b,a,t2) @j)
     | (Ex C #r.  Reveal(C)@ r & Honest(C) @ i)"
```

4.3.6.3 Non-injective agreement

The non-injective agreement of an agent b in role B on a message t is guaranteed by the protocol with an agent a if, whenever the agent a completes a run of the protocol, apparently with b in role B, then b has previously been running the protocol, apparently with a, and b was acting in role B in his run, and the two principals agreed on the message t, as shown in the lemma.

```
lemma  noninjective_agreement:
 "All  a  b  t  #i.
   Commit(a,b,t)  @i
     ==>  (Ex #j.   Running(b,a,t)  @j)
       | (Ex  C  #r.   Reveal(C)@  r  &  Honest(C)  @  i)"
```

4.3.6.4 Injective agreement

The injective agreement of an agent b in role B on a message t is guaranteed by the protocol with an agent a if, whenever the agent a completes a run of the protocol, apparently with b in role B, then b has previously been running the protocol, apparently with a, and b was acting in role B in his run, and the two principals agreed on the message t. Moreover, each run of agent a in role A corresponds to a unique run of agent b, for example, each agent Commit corresponds to a unique Running by the partner agent.

Keeping in mind that preventing replay attacks achieved using injective agreement. So, we need to involve a fresh value in each run via term t that must keep the freshness of such value [62].

```
lemma  injectiveagreement:
 "All  A  B  t  #i.
   Commit(A,B,t)  @i
   ==>  (Ex #j.   Running(B,A,t)  @j
     &  j  <  i
     &  not  (Ex  A2  B2  #i2.   Commit(A2,B2,t)   @i2
     &  not  (#i2  =  #i)))
       | (Ex  C  #r.   Reveal(C)@r  &  Honest(C)  @i)".
```

Chapter 5

Modeling and Analyzing of TLS 1.3 Handshake Modes

In this section, we will briefly show the model for the three basic key exchange modes that are supported in TLS 1.3 by showing the models we depends on in [29] then showing our updates on that model:

- (EC)DHE (Diffie-Hellman over either finite fields or elliptic curves)
- PSK only
- PSK with (EC)DHE

The FIGURE 5.1 show the full TLS1.3 handshake protocol.

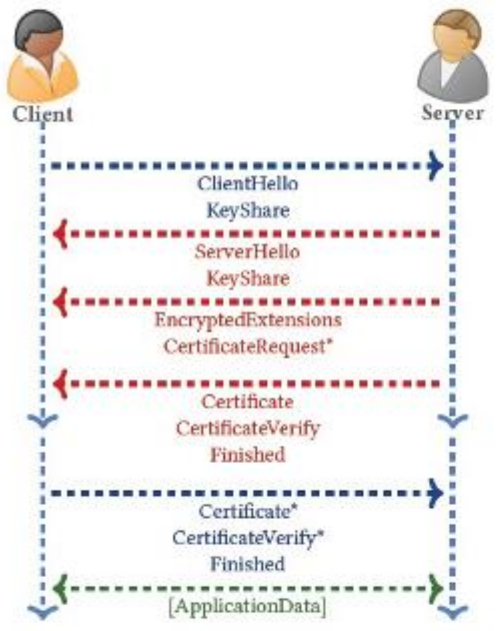


FIGURE 5.1: TLS1.3 Full handshake protocol [65]

The **ClientHello** message contains a random nonce, it also offers the protocol ver- sion, and a list of symmetric hash pairs; key shares that provided by (EC)DHE or pre-shred key that provided by PSK or a combination of both PSK with (EC)DHE,

ClientHello also includes some other extensions and fields for middlebox compatibility.

The **ServerHello** message determines the negotiated cryptographic parameters, and the shared keys with the client, which is a key share extension in Diffie-Hellman (EC)DHE) or pre-shared key extension in a PSK or using both together (EC)DHE and PSK. This thesis will relay on the above handshake basics to start our modeling process in the following sections [66]

5.0.1 DHE Mode Modeling and Analysis

Foremost, we use a pure DHE handshake when no pre-shared key available between the communicating parties. Moreover, DHE establishes a resumption secret to be used by subsequent handshakes to derive a pre-shared key. The server-side authentication is required in this mode.

DHE authenticate peers using the public key cryptography alongside with certificates. However, the certificate side of the client is optional and needs the certificate request from the server.

It is good to mention that, we can derive around four more modes for the DHE itself according to **HelloRetry** request and the **Certificate** request (i.e. DHE with client/server authentication or without client/server authentication).

5.0.1.1 DHE Mode Modeling

We introduce a general modeling of the main roles (client and server) during the handshake process in order to exchange messages between the send rule and its corresponding receive rule.

The initialization for both roles; client and server is the state fact that we have to begin with, to show participating the agent in the protocol. The modeling of client and server initialization using Tamarin tool as shown below:

```
Server Initialization:
rule server_Initialize:
 [!Ltk(S, ltkS)]
   -[ Create(S) ]->
    [St_Dh_S_1(S, ltkS)].

Client Initialization:
rule Client_Initialize:
   [!Ltk(C, ltkC)]
     -[ Create(C) ]->
      [St_Dh_C_1(C, ltkC)]
```

5.0.1.2 Key Exchange

This phase will include the main messages of key exchange protocol, **ClientHello, ServerHello**, and **HelloRetryRequest**. These messages includes a fresh nonce, the Diffie-Hellman key share part. **ClientHello**: The model of sending clientHello rule:

```
rule dh_client_send_ch:
let
 Msg = < ~NC, 'dhe', 'g'^~ec >
in
   [St_Dh_C_1(C, ltkC), Fr(~Nc), Fr(~ec)]
    -[ InEq(C,$S) ]->
      [St_Dh_C_2(C,$S, ltkC, ~ec, CH(Msg)), Out(Msg)]
```

When the server receives the **ClientHello**, the modeling will be the same for the server, but with the following server state fact: [St_Dh_S_2(S, $C, ltkS, Y, CH(Msg))]. (Note, Y = 'g'^ec).

Hello Retry Request As mentioned before; that **HelloRetryRequest** is an option that can be used in need. This increases the possibility of rules; to be two rules for each role (client) that is applicable to the other role (server). The server role will only model the server role; this could be applicable to the client role as well:

```
rule dh_server_send_hrr:
let
 Msg = 'hrr'
in
   [St_Dh_S_2(S, C, ltkS, Y, CH(m))]
    ->
      [St_Dh_S_2a(S, C, ltkS, <CH(m), HRR(Msg)>), Out(Msg)]
```

The above rule shows that we have two different state fact, one of them is applicable for the server that has **ClientHello** in the transcript. This means, only one **HelloRetryRequest** is allowed. The state fact [St_Dh_S_2a(S, C, ltkS, TSH) show that receiving the second **ClientHello** event; produces the state fact [St_Dh_S_2a(S, C, ltkS, Y, TSH), where TSH = <CH(m1), HRR(m2), CH(m3)>, TSH is included in generating every key, it represents the handshake process till the ServerHello, CH represents **ClientHello**, **HRR** represents **HelloRetryRequest**, m1, m2, m3 are messages.

ServerHello: The model of sending the **ServerHello** rule to the client:

```
rule dh_server_send_sh:
let
  Msg = < ~Ns, 'dhe', 'g'^~es > MS = Y^~es
in
   [St_Dh_S_2(S, C, ltkS, Y, TSH), Fr(~Ns), Fr(~es)]
    ->
      [St_Dh_S_3(S, C, ltkS, MS, <TSH, SH(Msg)>), Out(Msg)]
```

The (TSH) in the state fact (St_Dh_S_2) might contain one **ClientHello** message, or it might contain three messages in case the server sends the **HelloRetryRequest** message. In this case; the state fact (St_Dh_S_3) includes MS term, which represented by the Diffie-Hellman key, all keys are derived from MS, since no pre-shared keys are used in this mode to finally sending or receiving the **ServerHello** rule.

5.0.1.3 Server Parameters

To determine the reset of the handshake process after **ServerHello**; we need messages that contains information from the server, which are **EncryptedExtentions** and **CertificateRequest** that are encrypted using encryption key enckey.

Encrypted Extensions. The **EncryptedExtensions (EE)** message is the first message that encrypted using enckey. So, it MUST be sent directly after the **ServerHello** message.

The EE could include extensions that are not associated with individual certificates even if they are protected. Therefore, the client MUST abort the handshake with an alert, after looking for any forbidden extensions in (RFC8446).

Certificate Request requires the client to authenticate its identity if the server already requested it. So, we need to consider both, existence and absence of the optional **CertificateRequest** in our modeling, as shown:

```
rule dh_client_receive_cr:
let
  encKey = HKDF(« MS, 'enc'>, h(TSH) >)  Msg = 'cr'
in
     [St_Dh_C_4(C, S, ltkC, MS, TSH, TST), In(sencMsgencKey)]
     ->
       [St_Dh_C_5(C, S, ltkC, MS, TSH, <TST,CR(Msg)>, 'cauth')]
rule dh_server_skip_cr:
  [St_Dh_S_4(S, C, ltkS, MS, TSH, TST)]
    ->
     [St_Dh_S_5(S, C, ltkS, MS, TSH, TST, 'no_cauth')]
```

Where MS is the source of key derivation, besides the handshake hash stages, TST includes all subsequent messages that derives the traffic key trKey. According to the applied rule; the resulting state fact either is tagged with the term 'cauth' or 'no_cauth' to determines whether the server authenticate itself at first; in order to decide whether to send an authentication message to the client or not while modeling.

Authentication Messages. As we have two roles; client and server, thus we also need to have two messages blocks, server/client blocks. In case of using the Diffie-Hellman, the server will always need to send the full authentication block whereas no pre-shared key is ready yet to alternatively authenticate the server.

Server Authentication Block. The rule for receiving the certificate of the server for the client rule is represented as follows, taking into account the adopted assumption in this model is a perfect public key infrastructure, which means; the certificates are represented as pairs of (name and public key).

```
rule dh_client_receive_cts:
let
  encKey = HKDF(« MS, 'enc'>, h(TSH) >), Msg = < S, pkltkS >
in
 [St_Dh_C_5(C, S, ltkC, MS, TSH, TST, cr),
 In(sencMsgencKey), !Pk(S, pkltkS)]
 ->
 [St_Dh_C_6(C, S, ltkC, pkltkS, MS, TSH, <TST,CTS(Msg)>, cr)]
```

This rule contains the public key infrastructure, which modeled by the fact !Pk(S, pkltkS). In this case; the server sends **CertificateVerify** after sending it's certificate, as shown:

```
rule dh_server_send_cvs:
let
  encKey = HKDF(« MS, 'enc'>, h(TSH) >)
  Msg = sign(h(<TSH,TST>), ltkS)
in
   [St_Dh_S_6(S, C, ltkS, MS, TSH, TST, cr)]
    ->
      [St_Dh_S_7(S, C, MS, TSH,<TST, CVS(Msg)>, cr),
      Out(sencMsgencKey)]
```

Finally, the **Server Finished** message, which is the last message of the Server authentication block is modeled almost the same. The message Msg = HMAC(h(<TSH, TST>), mackey) has been encrypted by the server and sent out. The hash in this message "Msg = HMAC(**h(<TSH, TST>)**, mackey)" is differ from the hash of the Server Certificate "Msg = sign(**h(<TSH,TST>)**,ltkS)". Moreover, the TST term contains the whole signature. The state fact that represent the agent will be St_Dh_S_8(S, C, MS, TSH, TST, cr) and St_Dh_C_8(C, S, ltkC, MS, TSH, TST, cr). In this case the trKey becomes known, which means that the TST term is complete. Also, the TSR term that used to compute the resumption secrete will be constituted by the remaining messages.

Client Authentication Block. The client Certificate and client CertificateVerify messages will be skipped if the client didn't receive the CertificateRequest. The rules for both roles are applied with no client authentication (**no_cauth**) term.

```
rule dh_client_skip_cauth:
  [St_Dh_C_8(C, S, ltkC, MS, TSH, TST, 'no_cauth')]
   ->
     [St_Dh_C_10(C, S, MS, TSH, TST, 'no_cauth')]
```

In the case of sending the **CertificateRequest**, then we apply both rules dh_client_send_ctc and dh_server_receive_ctc using the client authentication term, as shown below.

```
rule dh_client_send_ctc:
let encKey = HKDF(« MS, 'enc'>, h(TSH) >)
     Msg = < C, pk(ltkC) > in
[St_Dh_C_8(C, S, ltkC, MS, TSH, TST, 'cauth')]
->
[St_Dh_C_9(C,S,ltkC,MS,TSH,TST,CTC(Msg)),
   Out(sencMsgencKey)]

rule dh_server_receive_ctc:
let  encKey = HKDF(« MS, 'enc'>, h(TSH) >)
      Msg = < C, pkltkC > in
 [St_Dh_S_8(S, C, MS, TSH, TST, 'cauth'),
   In(sencMsgencKey), !Pk(C, pkltkC)]
->
[St_Dh_S_9(S, C, pkltkC, MS, TSH, TST, CTC(Msg))]
```

Finally, the **client Finished** message received by the server are modeled as shown below.

```
rule dh_server_receive_fic:
let
macKey = HKDF(« MS, 'mac'>, h(TSH) >)
encKey = HKDF(« MS, 'enc'>, h(TSH) >)
Msg = HMAC(h(«TSH,TST>,TSR>), macKey)
trKey = HKDF(« MS, 'tr'>, h(<TSH,TST>) >)
in
[St_Dh_S_10(S, C, MS, TSH, TST, TSR),
In(sencMsgencKey)]
-[ Dh_Server_Ts(TSH, <TSR,FIC(Msg)>),
Dh_Server_Claim_Secret(S, 'encKey', encKey),
Dh_Server_Claim_Secret(S, 'macKey', macKey),
Dh_Server_Claim_Secret(S, 'trKey', trKey),
Commit(S,C, <'S','C','DH',
 < macKey, encKey, trKey > >),
Honest(S),
Honest(C)
 ]->
[St_Dh_S_11(S, C, MS, TSH, TST, <TSR,FIC(Msg)>)]
```

New Session Ticket. The last message sent during the post handshake phase is the NewSessionTicket (NST) with the value (lable), which can be used as psk_id in the future handshake. This is called handshake resumption and can be used in the future to have the ability of sending data in the first flight of the protocol. In this part, we need to have an outlook about the security properties of resSec. This post handshake message is crucial for the agreement properties satisfied by resSec. This post handshake message is encrypted using the key trKey. The rule of sending the **NewSesstionTicket** message, as shown below.

```
rule dh_server_send_nst:
let
trKey = HKDF(« MS, 'tr'>, h(<TSH,TST>) >)
resSec = HKDF(« MS, 'res'>, h(«TSH,TST>,TSR>) >)
Msg = label
in
[St_Dh_S_11(S,C,MS,TSH,TST,TSR), Fr( label)]
-[ Dh_Server_Ts(TSH, TSR),
Dh_Server_Claim_Secret(S, 'resSec', resSec),
Running(S,C,<'C','S','DH', resSec >),
Commit(S,C, <'S','C','DH', resSec >),
Honest(S), Honest(C)
 ]->
[Out(sencMsgtrKey)]
```

Executions and Lemmas. We need to verify our models if it behaves as expected. To this end; the executable lemmas are the best techniques to do this task, which is checking our security properties model behaviour since executable lemmas are responsible for proving or disproving these security properties. We have four sub modes of handshake protocols (Hello Retry Request , No Hello Retry Request , Client Authentication and No Client Authentication), each sub mode needs a lemma, so we need to specify four different lemmas to describe if the property hold (exist trace), these lemmas are defined by a model that contains a full handshake considering the disability of compromising any agent by the adversary. Two of these lemmas are shown below, the first one for a handshake with client side authentication with no **HelloRetryRequest (HRR)** message. The other lemma is without client authentication, but with HRR:

lemma 1:

```
lemma dh_nhrr_cauth_executable:
 exists-trace
 " Ex TSH m1 m2 TSR #i. Dh_Server_Ts(TSH, TSR)@i &
 TSH = <CH(m1),SH(m2)> &
 not (Ex tail. TSR = <'no_cauth', tail >) &
 not (Ex C #j. Rev(C)@j) "
```

lemma 2:

```
 lemma dh_hrr_ncauth_executable:
 exists-trace
 " Ex TSH TSR tail #i. Dh_Server_Ts(TSH, TSR) @i &
 not (Ex m1 m2. TSH = <CH(m1),SH(m2)>) &
 TSR = <'no_cauth', tail > &
 not (Ex C #j. Rev(C)@j) "
 Dh_Server_Ts(TSH, TSR)
```

These action facts have been used to restrict traces to a specific mode/sub-mode.
If the term TSH = <CH(m1),SH(m2)>) does not contain HRR, then the handshake do not involve HRR message. If the tag 'no cauth' is not included with the TSH term, then no client side authentication is required in the handshake. The automatic verification shows the success of lemmas, and if the security property has accomplished.

5.0.1.4 Diffie-Hellman Mode Analysis

The following analysis has been verified, tested and updated for the following original analysis, which aims to verify the satisfaction of the property for different ways of executing the handshake (with HRR, no cauth ...etc.). Secrecy and authentication are the security properties that we are interested in. Each sub-mode need to be tested for the satisfaction of each property. For efficiency, and according to specifications of the properties in Lowe's hierarchy of authentication specifications [67], proving the "lemmas" for the whole model means that we can conclude that every "mode" satisfies it. But, if we can't prove the lemma for the whole, then we need to be more specific in defining lemmas, for instance, defining lemmas for the handshake with/without client-side authentication.

The security properties have the form of implication, so the original researcher have added the restrictions to the premise to be able to constrain the lemma to a specific sub-mode. To do so, they have added the constraint Dh_Role_Ts(TSH, TSR)@i to the premise. Where, i is a temporal variables. The set of traces will be partitioned into two sub modes; with client side authentication and without client side authentication. The last partition that concerning the **HRR** will not be included here, since it doesn't affect the validity of the analysis in this stage. Moreover, the precision, in this case, is decreased especially with the existence of an attack, e.g., the attack would be applied to the handshake with **HRR** during the mode without client side authentication. For this reason, they consider the additional partition induced by the optional **HRR**, especially, when the attack exploits the structure of this exceptional flow. They have used the restrictions to define sub mode specific lemmas and adding the following restrictions, if they constrain the lemma to consider only the handshakes with client side authentication: not(Ex tail. TSR = <'no_cauth', tail>) as a conjunction to its premise. In contrast, the handshakes without the client side authentication will include the restriction TSR=<'no_cauth', tail> as a conjunction to its premise, binding the tail variable to the all messages. To prove the PFS for a server running Diffie-Hellman mode with client side authentication, the used lemma would be specified as shown.

```
lemma dh_cauth_server_pfs: "All S tag
x TSH TSR #i.(
Dh_Server_Claim_Secret(S, tag, x)@i &
Dh_Server_Ts(TSH, TSR)@i &
not (Ex tail. TSR = <'no_cauth', tail >))
==> (not (Ex #j. K(x)@j)|
(Ex A #j. Rev(A)@j & j<i & Honest(A)@i))"
```

Secrecy. the form of all secrecy claims in this mode will be Dh_Role_Claim_ Secret(agent, termtag, term), where termtag either is encryption key 'encKey', message authentication key 'macKey', traffic key 'trKey',
or resumption secret 'resSec'.
Server role, the rule dh_server_receive_fic includes three of secrecy claims for encKey, macKey, and trKey. The forth secrecy claim for resSec. is added to the rule dh_server_send_nst specifically after the NST message.

Then, we made sure to specify and verify the following lemmas:

dh_server_secrecy, the overall secrecy does not hold, the (×) sign shows an attack.

dh_ncauth_server_secrecy, the secrecy of the anonymous mode also does not hold, the (×) sign shows an attack too.

dh_cauth_server_pfs, the PFS of the authenticated mode have been proven, the ( ) sign shows that the secrecy property has proven to be secure, detailed follows:

The first two properties does not hold since the attacker runs the protocol as an anonymous client to learn the keys. The third property was proven to hold. We conclude both, secrecy and PFS are satisfied using the authentication mode, while the server fails in anonymous handshake mode no client authentication (ncauth).

As a result, the handshake with the client side authentication satisfies secrecy and PFS, regardless sending the **HRR** message or not. However, the handshake without using the client side authentication reveals attacks that violate secrecy and does not satisfy it. Taking into consideration that the secrecy could still satisfy for a specific sub mode using the handshake without client authentication when sending the **HRR**, but this isn't a general situation, so, we do not rely on whether sending **HRR** or not in any future investigations.

Client. The lemma dh_client_pfs has been satisfied and correctly proven by Tamarin prover, which means that client side secrecy is done. As a result, every single sub mode satisfies the secrecy and PFS for this handshake mode.

Agreement. To claim facts in this mode we need to follow a specific structure in Tamarin, which shown below:

Commit(a, b, < 'C','S','DH', t>)
Running(a, b, < 'S','C','DH', t>)

The commit claim and the running claim are made by an agent a running as a client, presumably with agent b running as a server.

Server The defined lemmas for this part is shown in the Table 5.1:

TABLE 5.1: Server agreement

Lemma (security property)	Verified (Satisfied)
dh_server_aliveness	×
dh_ncauth_server_aliveness	×
dh_cauth_server_injectiveagreement	✓

As shown in the Table 5.1, the first two properties do not hold the security specification, which implies an attack; as proven by Tamarin tool. However, if we used the client side authentication, then the injective agreement between the server and the client will be satisfied with every key.

Firstly, for the **Client** we need to specify dh_client_injectiveagreement, in order to analyse the agreement guarantees of the client, with a positive assumption, that it would hold. However, after applying the lemma using Tamarin tool, the attack of disagreement on the client name appears between the client and the server. The cause of this returns to the fact that, for security reasons, the client never sends his name.

The proposed solution by the researcher [29] was to achieve the injective agreement by using a **ClientHello** as a holder to include the agent name of the client, which could be included in the handshake hash. But, this solution needs to change the structure of TLS 1.3 protocol. The problem could be solved if the agent name replaced by an IP address and to be included in the hash function.
The client agreement properties make no much sense in the context of anonymous clients. To show this, we specify client weak agreement (lemma dh), which resulted in disagreement as in dh_client_injectiveagreement. So, the definition of properties of anonymous agreement expressed as follows:

```
lemma dh_client_anonymous_weakagreement:
"All a b m #i.
Commit(a,b,<'C','S','DH', m >)@i
==> (Ex c ts #j.
Running(b,c,ts)@j)
|(Ex X #r. Rev(X)@r & Honest(X)@i)"
```

Note how dh_client_anonymous_weakagreement degenerates to dh_client_aliveness by relaxations on the properties.

```
lemma dh_client_anonymous_injectiveagreement:
"All a b t #i.(
Commit(a,b,<'C','S','DH', t>)@i &
==> (Ex c #j.
continue ...
...continue
Running(b,c,<'C','S','DH', t>)@j &
j < i &
not (Ex a2 b2 mode #i2.
Commit(a2,b2,<'C','S', mode, t>)@i2 &
not (#i2 = #i)))
|(Ex X #r. Rev(X)@r & Honest(X)@i)"
```

The main difference between these two lemmas corresponds to non-anonymous lemma is that they do not require the agent b to have been running apparently with an agent a. Which means, it's OK if agent b does not know the agent name of a. Considering these properties, the verification of satisfaction of these lemmas are show in Table 5.2.

TABLE 5.2: Week agreement vs. injective agreement

Lemma (security property)	Verified (Satisfied)
dh_client_anonymous_weakagreement	✓
dh_client_anonymous_injectiveagreement	✓
dh_cauth_client_injectiveagreement	✓

The Table 5.2 shows the prove of the first two lemmas, which guarantee the anonymous client that the server injectively agrees on the values of encKey, macKey, trKey, and resSec with somebody. The client may be seen as idetified by that specific handshake run, as it's known that we derived the secret values from the secret of Diffie-Hellman and both nonces. The dh_cauth_client_injectiveagreement is proven to be violated if the commit claim (Commit(C,S,<'C','S','DH', < macKey, encKey, trKey > >)) is made in the rule that models the sending of **Client Finished**. In this stage, the client isn't sure if the server has received it's certificate and it's identification. We satisfy the clients claim after receiving an encrypted message with trKey from the server, which is derived partially from the client certificate. Therefore, in the analysis, the rule dh_client_receive_nst defined to include the mentioned claim. With this, dh_cauth_client_injectiveagreement is proven correctly.

5.0.1.5 DHE Mode Results

The summary of the verified Diffie-Hellman results that shown in Table 5.3. Both columns; dh_ncauth and dh_cauth represents no client side authentication and client side authentication respectively. Each column represents security property from the point of view of a specific role.

TABLE 5.3: Diffie-Hellman Model Results

Security property from role point of view	dh_ncauth	dh_cauth
Secrecy for Server	×	✓
PFS for Server	×	✓
Secrecy for Client	✓	✓
PFS for Client	✓	✓
Aliveness for Server	×	✓
Weak Agreement for Server	×	✓
Non-injective Agreement for Server	×	✓
Injective Agreement for Server	×	✓
Aliveness for Client	✓	✓
Weak Agreement for Client	×	✓
Non-injective Agreement for Client	×	✓
Injective Agreement for Client	×	✓
Anonymous Weak Agreement for Client	✓	✓
Anonymous Non-injective Agreement for Client	✓	✓
Anonymous Injective Agreement for Client	✓	✓

5.0.2 Pre-shared keys (PSK) Modes Modeling and Analysis

The FIGURE 5.2 show the TLS1.3 PSK handshake protocol.

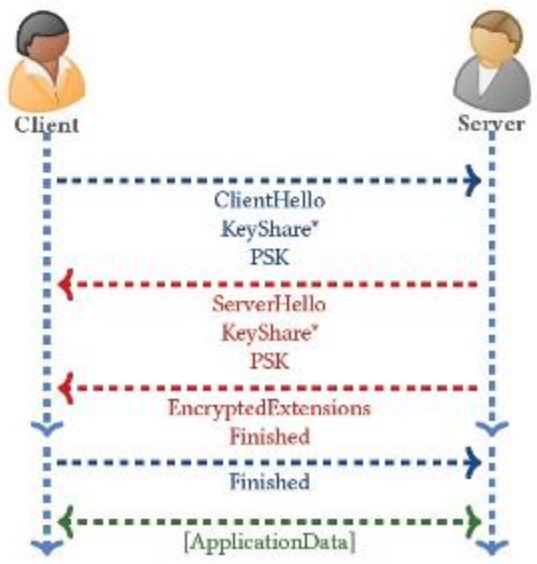


FIGURE 5.2: TLS1.3 PSK handshake protocol [65]

This section includes summary and results of the PSK mode handshakes analysis. The two modes only differ in the key exchange phase. Thus, we will model them with just one mode. Instead of using the pure Diffie-Hellman mode in analyzing the handshakes, we will be modeling the PSKs as fresh nonces that are secret and injectively agreed on. Keeping in mind that the PSK is a result of a previous handshake.

As it is stated in the TLS protocol RFC8446 that no need for the server to authenticate itself in the PSK modes since it is already authenticated by the PSK. While it's a must in the Diffie-Hellman mode. The assumption here is the possibility for the client to authenticate itself in the PSK modes. As we don't use Diffie-Hellman mode in this analysis, in this case, the PSK seems to be generated out of the band. Considering all of the above, it's logical to assume that these PSKs authenticate both parties. The delayed client authentication in PSK modes are included in this model.

We aim here to include this feature in the final model, but while the dependency graphs are not readable, i.e., has a huge size, which prevents us from conducting the sanity checks on that model. So, we need to create smaller models to check the availability of this feature. This allows us to verify that the corresponding rules are defined appropriately, then apply the same technique for the final model.

5.0.2.1 Pre-shared key Modes Modeling

In this section, we describe the model in an abstract level. In addition, we prefer not to present the whole rules for every message rather describe them on a high level. This is because of the messages in the previous handshake (Diffie-Hellman) is almost the same and no new messages will be added to PSK handshake. Finally, we express the modeling process of PSK modes in Tamarin tool.

Key Infrastructure

Although the client side authentication is supported in the PSK modes, the symmetric pre-shared keys are the main source of authenticity. The persistent fact (!) is used to represented PSK, !Psk(S,C,~psk_id, ~psk), the main idea behind this fact is to ensure the freshness and uniqueness of PSK and psk_id terms, in order to prevent the adversary of revealing the PSK directly without the need for producing the reveal action fact.

```
rule register_psk:
[Fr(~psk), Fr(~psk_id)]
->
[!Psk(S,C, ~psk_id, ~psk)]
reveal_psk:
[!Psk(S, C, psk_id, psk)]
-[Rev(S), Rev(C)]->
[Out(psk)]
```

Key Exchange

We define the first rules of the handshake protocol for **ClientHello** and **ServerHello** for both pure PSK mode, and PSKDH mode. In this phase, each rule is defined separately for each mode. Moreover, the tags that indicate the mode are also included in the exchanged messages. Also, nonces, psk_id term are used to identify the corresponding PSK.

ClientHello = <~Nc, 'pskid', psk_id >
ServerHello = < ~Ns, 'pskid', psk_id >

The messages of the PSKDH mode additionally contains Diffie-Hellman half keys:

ClientHello = < ~Nc, 'pskdhe', psk_id, 'g'^~ec >
ServerHello = < ~Ns, 'pskdhe', psk_id, 'g'^~es >

Each mode and each role have a rule to model the sending and receiving messages in this phase. For example, the rule pskonly_client_send_ch and pskdh_client_send_ch, these messages (the sending and receiving messages), then collected in the TSH term after wrapping them with the corresponding function. Then, after the handshake phase, TSH is fully defined and the keys encKey and macKey are available to the roles.

The sources of the derived keys differ according to the model of the handshake. MS in the pure PSK mode is set to the value of PSK. While we derive MS in the PSK mode that uses Diffie-Hellman key exchange from both input secrets.

The state facts of the form St_Psk_S_3(S, C, MS, TSH, mode) are produced by the rules modeling the **ServerHello** message. The term mode in the state fact is used to indicate the mode of the handshake, either 'pskonly' or 'pskdh', we use it later when defining lemmas. The rules modeling subsequent messages are also defined for either mode; the mode term here kept abstract during the rewriting of multisets.

Server Parameter

As in Diffie-Hellman mode, this phase contains only the messages Encrypted Extensions and CertificateRequest. These messages sent by the server and received by the client after they are encrypted with encKey. They also form the first of the TST term message. Taking into account that the client authentication is optional, we define the following rules, rule psk_server_skip_cr, and rule psk_client_skip_cr to allow for traces where **CertificateRequest** is not being sent. These rules also use the tags 'no_cauth' and 'cauth' to indicate the use of the **CertificateRequest** or not. Both pairs can be applied to the same state facts, which might only differ in the content of the tag, and the handshake transcript.

Authentication Phase and New Session Ticket (NST)

In this phase, the message **Server Finished** is sent by the server, which is defined as:
Msg = HMAC(h(<TSH,TST>), macKey)
The Finish message is the last one included in the term TST. The tag of the state facts

distinguishes between two applicable sets of rules. Which means, the applicable rule depends on the tag, i.e., if the tag indicates client authentication, then the modeled sending and receiving rules of **Client Certificate** and **Client CertificateVerify** becomes applicable. Furthermore, psk_client_skip_cauth and psk_server_skip_cauth are applied to the produced state facts. The skip rule change the first values of the term TSR from CTC and CVS to add ’no_cauth’ as the first value to TSR instead, which is the same as in the DH mode, because of expecting TSR to be part of the state facts by the rules psk_client_send_fic and psk_server_receive_fic. We keep the client authentication tag in the state facts for future use of submode selection in the definition of lemmas, even though it is not used anymore after the split case on the state fact.

Executability

We define and use the fact Psk_Role_Mode(mode, tag) in order to check if the model has the ability to execute valid protocol traces. The term mode represents either ’PSKDH’ or ’PSKONLY’, while the term tag represents either ’cauth’ or ’no_cauth’. Then, each of the above terms needs to have a separate lemma to represent it, which means that we need to specify four executability lemmas. The handshakes with client authentication in pure PSK mode satisfies the following lemma:

```
lemma pskonly_cauth_executable
exists-trace
"Ex #i.
Psk_Client_Mode(’PSKONLY’, ’cauth’)@i &
not (Ex C #j. Rev(C)@j)"
```

In the next section, we show how to use these action facts to reason about the mode and how it is explained.

5.0.2.2 PSK Mode Analysis

In this section, we consider the secrecy properties, the agreement and the naming of lemmas as they defined in the previous sections. As mentioned in executability section above, we have four modes of execution. The action fact Psk_Role_Mode (mode, tag) is for analyzing/deducing a specific mode. For the server role, we use Psk_Server_Mode(mode, tag). These action facts are created whenever a rule produces a claim for secrecy or agreement.

Secrecy

The action facts of the form Psk_Role_Claim_Secret(agent,term-tag,term) are used to define the corresponding rules of the secrecy properties terms encKey, macKey, trKey, and resSec. The term-tag describes the type of the term. The properties for the analysis of client and server secrecy are defined in Table 5.4. The client authentication (cauth) is not considered in the defining lemmas.

TABLE 5.4: Secrecy

Lemma (security property)	Verified (Satisfied)
psk_server_secrecy	✓
pskonly_server_pfs	×
pskdh_server_pfs	✓
psk_client_secrecy	✓
pskonly_client_pfs	×
pskdh_client_pfs	✓

We have verified the secrecy results for both client and server using Tamarin tool.

To illustrate the mode selection, we define the following lemma:

```
lemma pskonly_client_pfs: " All S tag
x cr #i.( Psk_Client_Claim_Secret(S,
tag, x)@i & Psk_Client_Mode('PSKONLY',
cr)@i)
==> (not (Ex #j. K(x)@j)
|(Ex A #j. Rev(A)@j & j<i & Honest(A)@i) ) "
```

As shown in the Table 5.4, Tamarin reveals an attack, which is No PFS achieved) for both pskonly_server_pfs and pskonly_client_pfs. This refers to deriving the keys of the pure PSK mode from the shared value between two parties, and no ephemeral secrets where used.
In this thesis, we propose the use of compound PSKs by updating Stettler [29] model, specifically, the register_psk rule, and the reveal_psk, which shown later in this thesis. The Tamarin result verifies the PFS for both parties as shown in Table 5.5, which means that the result has been changed from (falsified-attack) to (verified-proof).

TABLE 5.5: Achieving PFS for both parties

Lemma (security property)	Verified (Satisfied)
pskonly_server_pfs	✓
pskonly_client_pfs	✓

Return back to the rest of results shown in the Table 5.4, we recognize that all remaining four lemmas have successfully proved, which means that secrecy on the keys is satisfied in every sub-mode. Additionally, the pskdh mode ensures PFS for both roles. All proved properties assume that the value of PSK is secret and injectively agreed on by both parties.

Agreement

As we are interested in the agreement properties on the terms encKey, macKey, trKey, and resSec. We verify this by adding the action facts to the corresponding rules, as shown:

Commit(a,b,<'S','C','PSK', t>)
Running(b,a,<'S','C','PSK', t>)

We define the following lemmas to analyze the action facts rules:

TABLE 5.6: Injective Agreement

Lemma (security property)	Verified (Satisfied)
psk_server_injectiveagreement	✓
psk_client_injectiveagreement	✓

Tamarin has verified the strongest authentication property, which is the injective agreement. If the PSK is secret and provides authentication then PSK modes satisfy mutual injective agreement. Every agreement in this work have been verified and satisfied by the PSK modes [29].

TABLE 5.7: PSK Model Results - Original Model

Security property from role point of view	pskonly original	pskonly updates	pskdh
Secrecy for Server	✓	✓	✓
PFS for Server	×	✓	✓
Secrecy for Client	✓	✓	✓
PFS for Client	×	✓	✓
Aliveness for Server	✓	✓	✓
Weak Agreement for Server	✓	✓	✓
Non-injective Agreement for Server	✓	✓	✓
Injective Agreement for Server	✓	✓	✓
Aliveness for Client	✓	✓	✓
Weak Agreement for Client	✓	✓	✓
Non-injective Agreement for Client	✓	✓	✓
Injective Agreement for Client	✓	✓	✓

5.0.2.3 PSK Mode Results

In this analysis, we suppose that both parties are already authenticated by the PSK. This leads us to temporarily ignore the delayed client side and to cover it back in session resumption section. The column in Table 5.7 **pskonly-original** represents the original PSK mode modeling by Stettler [29], while the pskonly-updates represent our new modeling and updates made on original modeling. Lastly, the column pskdh represents the combination of both modes (PSK and the ephemeral Diffie-Hellman secret) in order to achieve PFS.

5.0.3 0-RTT Modeling and Analysis

This section introduces the 0-RTT, compare it with the QUIC [68]; Google protocol. Showing advantages and disadvantages of each. The way of implementing, modeling and analysis of 0-RTT and finally to show the anlysis results.

We need to use some cryptographic protocols for instance, Key exchange (KE) protocols; to establish a secure key between two parties. QUIC, TLS, and SSH protocols are combined with KE protocols to establish such secure keys between two parties in a network [15].

Although the performance has always been the main concern for cryptographic protocols, the optimizations mainly focused on the cryptographic operations, which controlled the overall cost of executions for a long period.

Not only the computation process becomes much faster nowadays; but also advancing and deployment of elliptic curve cryptography, which enforces the worker in this field to extremely reduced the cost of cryptographic operations over time because of the very fast progress in technology. As a result, the communication complexity is the most important factor that controlling the overall efficiency of key exchange protocols [14]. However, efficiency must be compatible with other factors such as confidentiality and integrity of the transmitted data that are represented by PFS and preventing replay attacks. To this end, Gunther et al. [15] have proposed a solution that approved full forward secrecy for the transmitted payload messages via constructing their own 0-RTT key exchange protocol to achieves PFS using puncturing method, which allows decrypting each ciphertext just once.

The main idea behind Gunther et al. [15] work, is to generate many keys using only one master key (SK) via HIBE [28] and one-time signature [69]. FIGURE 5.3 summaries the whole process as follows:

- Step1: Generate secrete keys that decrypts one ciphertext each; via master key.
- Step2: Using HIBE and pseudo-random generator to generate step1 keys.
- Step3: Shows an example to use the sk(0101) to decrypt ciphertext(0101).
- step4: Determine the path from root to the intended key (leaf).
- step5: Define nodes' siblings.
- Step6: Remove the secret key path, which makes it a one time use.
- Step7: Shows the number of growing secret keys, which equals two times of security level that we want to achieve, for instance we need 256 keys to achieve 128 bit security as shown in the equation $|sk_{\backslash 0101} \approx 2 * \text{ secpar}$.
- Step8: Shows the number of secrete keys for whole process; using puncturing many times, which is unfortunately a very large size of keys (GBytes) needs to be stored on the server.

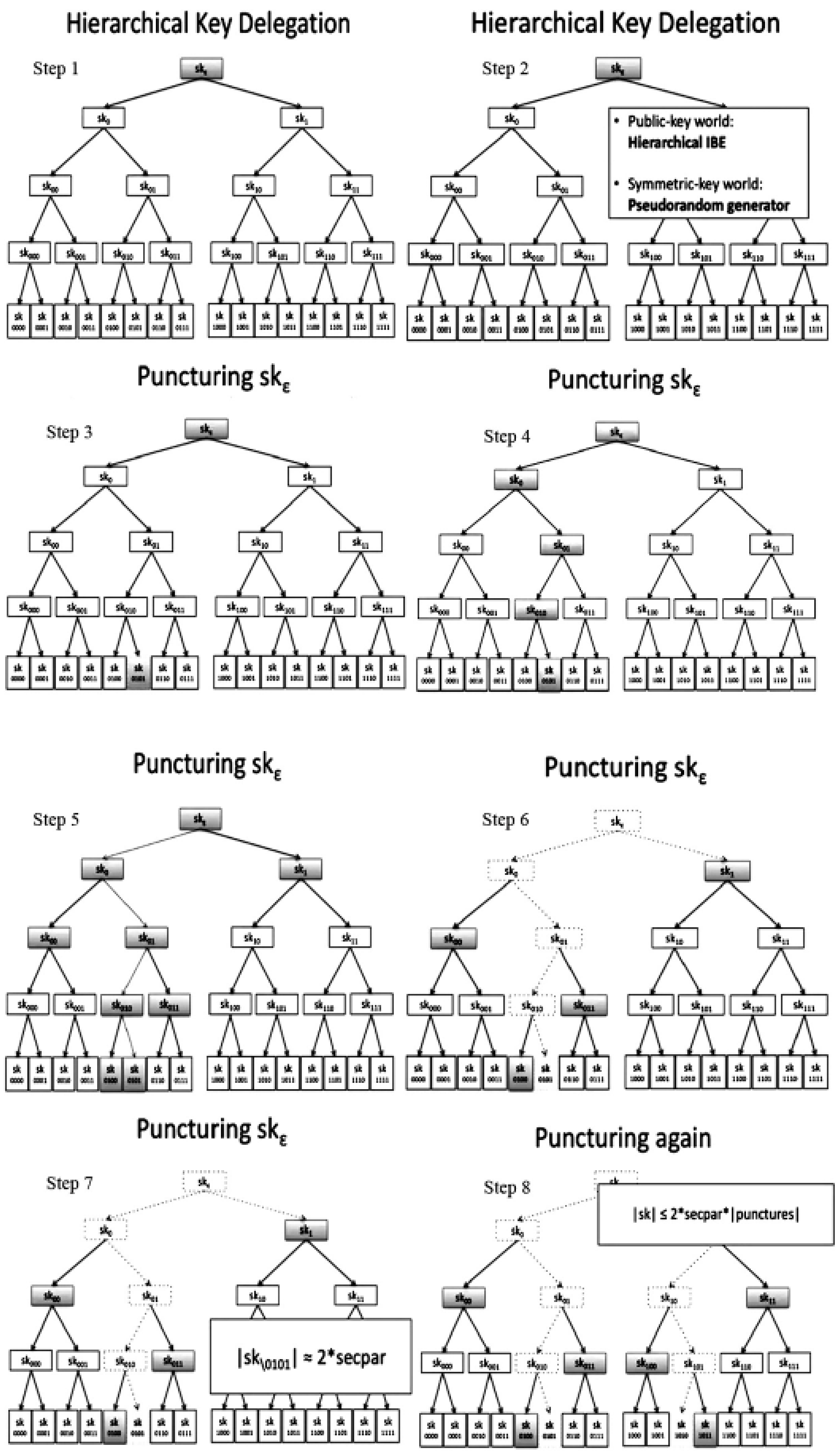

Hierarchical Key Delegation
Step 1
Hierarchical Key Delegation
Step 2
Public-key world: Hierarchical IBE
Symmetric-key world: Pseudorandom generator
Puncturing sk_ε
Step 3
Puncturing sk_ε
Step 4
Puncturing sk_ε
Step 5
Puncturing sk_ε
Step 6
Puncturing sk_ε
Step 7
$|sk_{\setminus 0101}| \approx 2*secpar$
Puncturing again
Step 8
$|sk| \leq 2*secpar*|punctures|$

FIGURE 5.3: Puncturing algorithm to prove PFS [15], [70].

5.0.3.1 0-RTT using Diffie–Hellman exchange

The FIGURE 5.4 show the TLS1.3 0-RTT handshake protocol.
Both QUIC and TLS 1.3 are using Diffie-Hellman key exchange protocol to derive

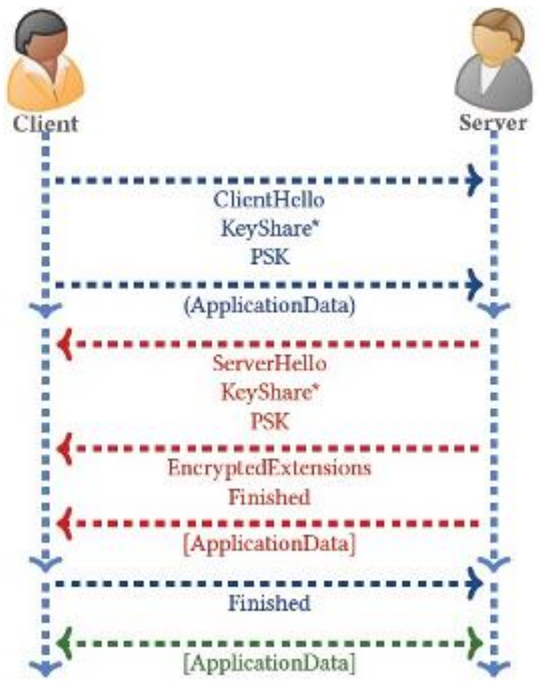


FIGURE 5.4: TLS1.3 0-RTT handshake protocol [65]

a 0-RTT key, which done in the first stage by sending the client key share to a pre-visited server, then upgrade the key to a stronger one by combining the server key share using Diffie-Hellman calculations to achieve forward-secret key. The whole detailed process works as follows. The client already has the server configuration including a semi-static Diffie-Hellman share g^s from the previous connection; the server keeps the exponent (s) for a short period of time (about 2-7 days) to keep it fresh and to use it in the PSK process. The server configuration authentication in QUIC is asymmetrically processed (offline) signed structure that announced by the server, while TLS 1.3 is signed symmetrically (online) during a prior handshake. As the process continue, the client sends an ephemeral Diffie-Hellman share g^x, then the exponential equation $(g^s)^x = g^{xs}$ is used to derive K_1 for the 0-RTT key, this allows the client to send encrypted data directly. The server uses the same process to derive the same key using $(g^x)^s$; this enables 0-RTT data decryption; the server then strengthens the key by responding its own ephemeral key share g^y. Finally, they both agree on the same stronger, forward secrecy key K_2 that is g^{xy}, which remains secure; no matter if the long-term secret has compromised for both parties or even the server configuration key share g^s [15].

5.0.3.2 0-RTT using Pre-shared key exchange

Pre-Shared keys are part of handshake mode in TLS 1.3, which can be established in a previous connection and then can be used to establish a new connection (session resumption). As well, the 0-RTT key K_1 is derived from the previously established secret key. This allows the client to send an immediate early data using K_1, without interacting the full handshake with the server. Then another key K_2 is updated between parties from the PSK for the upcoming messages. To ensure forward secrecy we combine the PSK with a Diffie-Hellman key share.

In this thesis, we are verifying and modeling 0-RTT in the pure PSK mode regarding of showing the secrecy and agreement on the keys used to protect the early data [15].

5.0.3.3 0-RTT Model

We are modeling the set of rules that defining an alternative key exchange phase. The early **EncryptedExtensions** and the **end_of_early_data** are not included here since they are irrelevant to our properties and the latter needs synchronization.

The early keys defined as follows:

```
earlyEncKey = HKDF(<psk,'enc'>, h(CH(..)))
earlyMacKey = HKDF(<psk,'mac'>, h(CH(..)))
earlyTrKey = HKDF(<psk,'tr'>, h(CH(..)))
```

In order to send and receive **ClientHello** messages tagged with pre-shared key identity and early data 'pskided' we need to expand each role of the model with rules that consume the state facts St_Psk_Role_1(..) and produce the state facts St_Pskonly_Ed_Role_1(..). The definition of **ClientHello** as shown:

ClientHello = < ~Nc, 'pskided', psk_id >

There exist other rules to model the **Early Finished**, taking the advantage of using the same state facts as that used in the above **ClientHello** message. They model the sending and receiving of Msg = HMAC(h(CH(..)), earlyMacKey) encrypted using earlyEncKey. The rules modeling the **Early Finished** produces the state facts St_Pskonly_Ed_Role_2(..). They are consumed by rules that model the sending and receiving of the **ServerHello**. Lastly, modeling the reception of the EncryptedExtensions rule. This allows the client to produce the claims for the security properties. The client would have no agreement guarantees if these claims made earlier, i.e., in the rules that receive **ServerHello** [29].

Analysis of 0-RTT

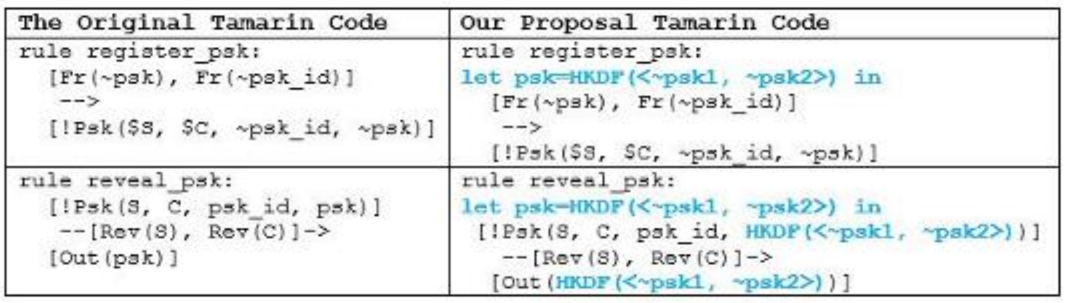
In general 0-RTT messages considered to be part of the PSK modes. However, the previous modeler [29] have decided to took them apart to distinguish the secrecy claims of the form Psk_Ed_Role_Claim_Secret(agent, term-tag, term), from the claims made by roles running in the PSK modes. They have specified the security properties as shown in Table 5.8:

TABLE 5.8: 0-RTT Analysis

Lemma (security property)	Verified (Satisfied)
psked_client_pfs	×
psked_client_secrecy	✓
psked_server_pfs	×
psked_server_secrecy	✓
psked_client_injectiveagreement	✓
psked_server_injectiveagreement	×
psked_server_non_injectiveagreement	✓

Tamarin reveals an attack on both PFS lemmas, in addition to replay attack on the injective agreement between parties. This refers to the derivation of early keys from the static preshared key. If the client is unable to receive messages from the server. This indicates that the server has no freshness guarantee about this data. However, the client has freshness guarantees from the **ClientHello's** nonce. This is why the psked_client_injectiveagreement is proved. The server does a non-injective agreement with the client on the early keys, however, it has no replay protection. This is proven by psked_server_non_injectiveagreement. The early data is unknown to the adversary if both parties are honest, which satisfies both secrecy lemmas [29]. The Table 5.8 depicts the security properties analysis for 0-RTT.

Applying our first approach to resolve the security properties for the above analysis of 0-RTT by combining more than pre-shared key together, which resulted in converting the results of both, the client and the server PFS from an attack (falsified) to secure transition (verified). FIGURE 5.5 shows the Tamarin code for original and our proposed models, followed by the the Tamarin results for both, as shown in FIGURE 5.6.



The Original Tamarin Code

```
rule register_psk:
  [Fr(~psk), Fr(~psk_id)]
   -->
  [!Psk($S, $C, ~psk_id, ~psk)]
```

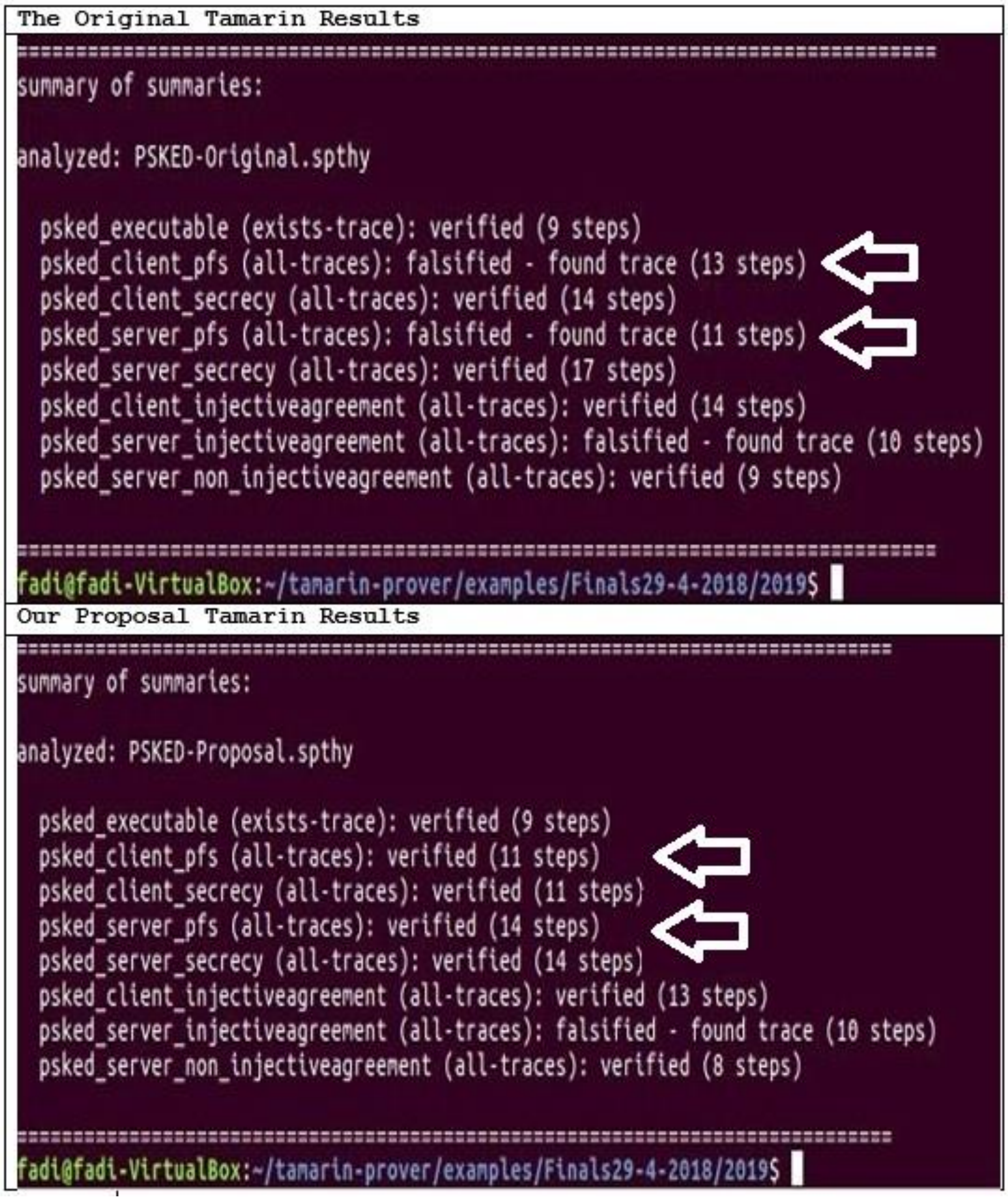
```
rule reveal_psk:
  [!Psk(S, C, psk_id, psk)]
   --[Rev(S), Rev(C)]->
  [Out(psk)]
```

Our Proposal Tamarin Code

```
rule register_psk:
let psk=HKDF(<~psk1, ~psk2>) in
   [Fr(~psk), Fr(~psk_id)]
    -->
   [!Psk($S, $C, ~psk_id, ~psk)]
```

```
rule reveal_psk:
let psk=HKDF(<~psk1, ~psk2>) in
  [!Psk(S, C, psk_id, HKDF(<~psk1, ~psk2>))]
    --[Rev(S), Rev(C)]->
   [Out(HKDF(<~psk1, ~psk2>))]
```

FIGURE 5.5: The original Tamarin code vs. Our Proposal



The Original Tamarin Results

```
==============================================================================
summary of summaries:

analyzed: PSKED-Original.spthy

  psked_executable (exists-trace): verified (9 steps)
  psked_client_pfs (all-traces): falsified - found trace (13 steps)
  psked_client_secrecy (all-traces): verified (14 steps)
  psked_server_pfs (all-traces): falsified - found trace (11 steps)
  psked_server_secrecy (all-traces): verified (17 steps)
  psked_client_injectiveagreement (all-traces): verified (14 steps)
  psked_server_injectiveagreement (all-traces): falsified - found trace (10 steps)
  psked_server_non_injectiveagreement (all-traces): verified (9 steps)

==============================================================================
fadi@fadi-VirtualBox:~/tamarin-prover/examples/Finals29-4-2018/2019$
```

Our Proposal Tamarin Results

```
==============================================================================
summary of summaries:

analyzed: PSKED-Proposal.spthy

  psked_executable (exists-trace): verified (9 steps)
  psked_client_pfs (all-traces): verified (11 steps)
  psked_client_secrecy (all-traces): verified (11 steps)
  psked_server_pfs (all-traces): verified (14 steps)
  psked_server_secrecy (all-traces): verified (14 steps)
  psked_client_injectiveagreement (all-traces): verified (13 steps)
  psked_server_injectiveagreement (all-traces): falsified - found trace (10 steps)
  psked_server_non_injectiveagreement (all-traces): verified (8 steps)

==============================================================================
fadi@fadi-VirtualBox:~/tamarin-prover/examples/Finals29-4-2018/2019$
```

FIGURE 5.6: The original Tamarin Result vs. Our Proposal Result

In order to solve the replay attack problem, we propose that after the server receiving the client's key-shares, it sends its key-shares (a list of keys or a database to be saved by the client) to the client in order to create a session key by combining them (one-by-one) with its own key-share, this allows sending secured data on the first flight. The PFS problem happens when the server key-share is not available for the client at the first step of resumption secret.

This also (more than key-share) leads to prevent the replay attack. For example, The server creates its own (Deffi-Hellma) key-share ($x_1 = g^{a_1}, x_2 = g^{a_2}, x_3 = g^{a_3}, x_4 = g^{a_4}, x_5 = g^{a_5}$) then sends them to the client. The client keeps these key-shares, then start using them one after another. Whenever, the client wants to send

0-RTT data to the server, the client creates a session key using the server key-share ($y_1 = g^{b_1}, y_2 = g^{b_2}, y_3 = g^{b_3}, y_4 = g^{b_4}, y_5 = g^{b_5}$). In this case, both parties are ready to agree on the session keys and to start using each session key once, then drop it. Lastly, when arriving the last session key, the process is repeated to create a new set of shared keys. This will solve the problem, but we need some re-architecture of the TLS 1.3 protocol to be able to do the Diffie-Hellman on the first flight.

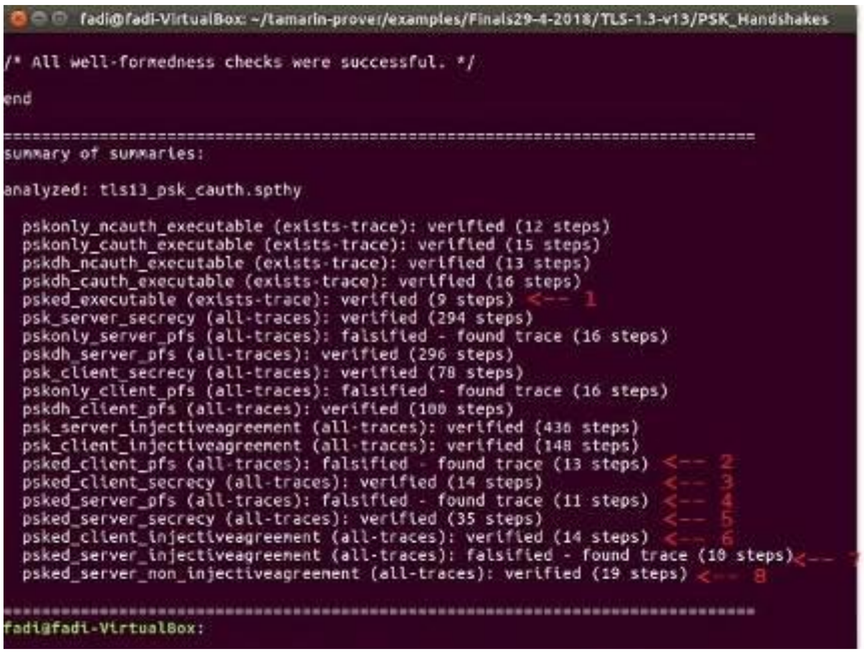
0-RTT Data Analysis Results

The analysis of 0-RTT data properties are shown in the table 5.9. The psked column represents early data of the PSK modes.

TABLE 5.9: 0-RTT Data Analysis Results

Security property from role point of view	psked
Secrecy for Server	✓
PFS for Server	×
Secrecy for Client	✓
PFS for Client	×
Aliveness for Client	✓
Weak Agreement for Client	✓
Non-injective Agreement for Client	✓
Injective Agreement for Client	✓
Aliveness for Server	✓
Weak Agreement for Server	✓
Non-injective Agreement for Server	✓
Injective Agreement for Server	×

The FIGURE 5.7 shows the Tamarin prover screen shot for some of psked results.



```
fadi@fadi-VirtualBox: ~/tamarin-prover/examples/Finals29-4-2018/TLS-1.3-v13/PSK_Handshakes
/* All well-formedness checks were successful. */

end

==============================================================================
summary of summaries:

analyzed: tls13_psk_cauth.spthy

  pskonly_ncauth_executable (exists-trace): verified (12 steps)
  pskonly_cauth_executable (exists-trace): verified (15 steps)
  pskdh_ncauth_executable (exists-trace): verified (13 steps)
  pskdh_cauth_executable (exists-trace): verified (16 steps)
  psked_executable (exists-trace): verified (9 steps) <-- 1
  psk_server_secrecy (all-traces): verified (294 steps)
  pskonly_server_pfs (all-traces): falsified - found trace (16 steps)
  pskdh_server_pfs (all-traces): verified (296 steps)
  psk_client_secrecy (all-traces): verified (78 steps)
  pskonly_client_pfs (all-traces): falsified - found trace (16 steps)
  pskdh_client_pfs (all-traces): verified (100 steps)
  psk_server_injectiveagreement (all-traces): verified (438 steps)
  psk_client_injectiveagreement (all-traces): verified (148 steps)
  psked_client_pfs (all-traces): falsified - found trace (13 steps) <-- 2
  psked_client_secrecy (all-traces): verified (14 steps)            <-- 3
  psked_server_pfs (all-traces): falsified - found trace (11 steps) <-- 4
  psked_server_secrecy (all-traces): verified (35 steps)            <-- 5
  psked_client_injectiveagreement (all-traces): verified (14 steps) <-- 6
  psked_server_injectiveagreement (all-traces): falsified - found trace (10 steps)<-- 7
  psked_server_non_injectiveagreement (all-traces): verified (19 steps) <-- 8

==============================================================================
fadi@fadi-VirtualBox:
```

FIGURE 5.7: The Tamarin psked screen shot

5.0.3.4 Replay Attack Problem

The attacker can replay messages between parties to make them derive the same key twice or to contribute to the derived key by including nonces in the exchanged messages between the client and the server. QUIC protocol strives to solve the problem by reserving a server to store all nonces in "strike register" a size restricted by a server-specific prefix "orbit" including the current time in the nonces and refusing any repeated nonce [15]. This approach has been succeeded to prevent replays on key-derivation on the key exchange level by prohibiting the adversary of making parties derive the same key twice. However, it fails to prevent replays on actual data exchanged (logical replay attack), specifically, working with the clustered and distributed servers. As a result, the replay attack problem is independent of whether the 0-RTT key exchange is based on Diffie-Hellman or pre-shared keys [71]. On the other hand Gillmor et al. [72] have also tried to solve the replay attacks by showing that an attacker can make the encrypted data that sent by the client alongside with 0-RTT key-exchange messages to be delivered twice as shown in FIGURE 5.8. Combining the overall channel protocol that works on delivering the data messages reliably; with any 0-RTT anti-replay mechanism applied at the key exchange level becomes worthless (invalid) and this is because of resending the rejected 0-RTT data by the automatically derived key that aimed to guarantee delivery [15].

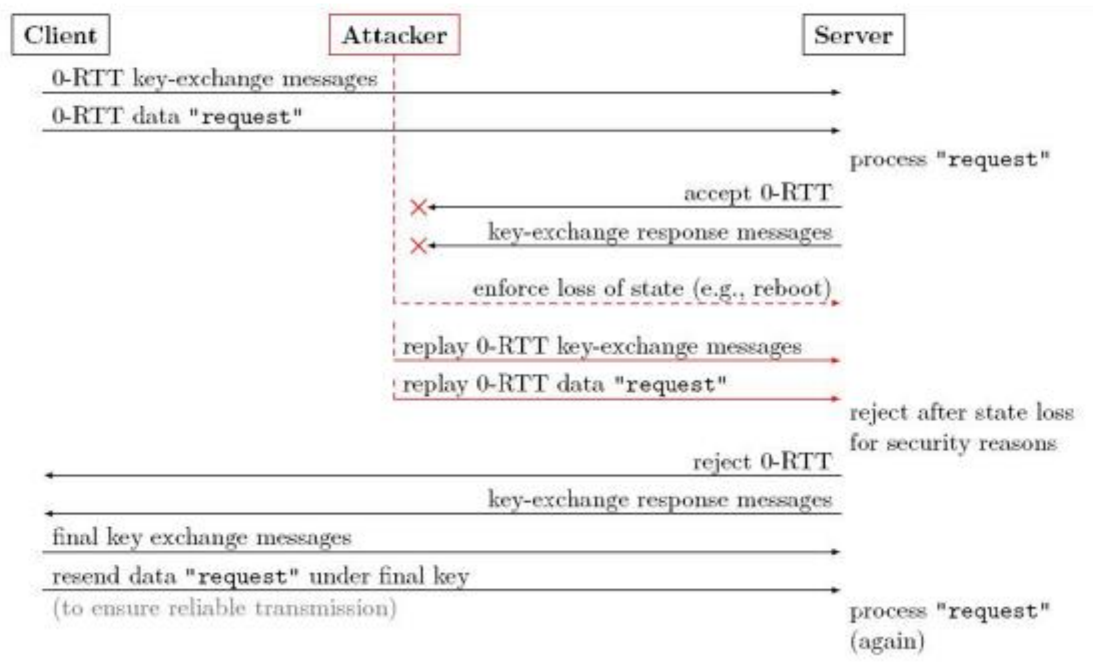

Client
Attacker
Server
0-RTT key-exchange messages
0-RTT data "request"
process "request"
accept 0-RTT
key-exchange response messages
enforce loss of state (e.g., reboot)
replay 0-RTT key-exchange messages
replay 0-RTT data "request"
reject after state loss for security reasons
reject 0-RTT
key-exchange response messages
final key exchange messages
resend data "request" under final key
(to ensure reliable transmission)
process "request" (again)

FIGURE 5.8: Replay attack by Daniel [72].

The attack work as follows:

- The attacker transports the client's 0-RTT message and encrypted data to the server but keeps the server 's key exchange response away from the client.

- Forces the server to reboot in order to lose its state then, re-send the same message to the server.
- After rebooting; the server declines the 0-RTT message of the key exchange to keep secure; instead, the server sends its own key share, which then passed to the client by the attacker.
- The client uses the server key share in deriving the final key then, encrypt the data and re-send to the server using the derived key in order to guarantee reliable delivery.
- The server will re-process the data for the second time, which represents a replay of the contained application data and results in processing a web transaction twice.

The existence of distributed server clusters in real-world; makes it easier for the attacker to forwards the 0-RTT messages to two servers and drops the response of the first server, instead of rebooting the server and keeps the client waiting for a response, as shown in FIGURE 5.8. This attack targets the settings with distributed clusters, which directly affects the cryptographic design of the QUIC protocol. Moreover, achieving full replay attack protection for envisioned 0-RTT seems to be impossible. As mentioned by Langley and Chang [73] that 0-RTT is "designed to die" and the adapted version of TLS 1.3 handshake will replace the 0-RTT protocol. In general, QUIC's strategy [73] resist some of the replay attack kinds, while TLS 1.3 didn't, rather it's accepts replays as inevitable. which must be adapted in the future versions of TLS 1.3.

Chapter 6

Conclusion

In this thesis, we have investigated, verified, modified and re-modeled some of TLS 1.3 security properties; focusing on handshake protocols based on previous models of TLS security protocols using symbolic analysis tool, consideration the updated parts in latest TLS drafts, for instance (draft-28) till releasing RFC8446 few months ago. The Dolev-Yao is the attacker model; using the Tamarin prover tool. Firstly, we have upgraded and combined more than pre-shared key, which resulted in prov- ing the PFS for the client and the server using the mentioned symbolic tool. This drives us one step further our targeted goal, which is reducing the latency overhead to send early payload data in the first flight of resuming the handshake protocol us- ing the PSKs, maintaining critical security guarantees, specifically perfect forward secrecy and preventing replay attacks. This result shows the possibility of solving PFS less expensively than Gunther et al. [15] since we used a simpler and effective way to delete the key rather than puncturing algorithm, where the session key PSK used once then keeps updating different session keys throughout a single session. We also believe that the second part of our proposed solution achieves replay at- tack prevention in theory. Since, exchanging more than Diffie-Hellman key-share between the parties and using them one at a time, then deleting the used key is a logical solution. But, we need to verify the created model in the coming future, be- sides studying the ability of re-structuring the Diffie-Hellman protocol needs more testing to make sure if it considered a valid assumption. We also have verified that the client can be sure of establishing a secure channel with the server under a per- fect public key infrastructure. We have verified that the session resumption of PSK established in Diffie-Hellman satisfies all security properties including PFS.

Bibliography

[1] C. Cremers, M. Horvat, S. Scott, and T. v. d. Merwe, “Automated analysis and verification of tls 1.3: 0-rtt, resumption and delayed authentication”, in 2016 IEEE Symposium on Security and Privacy (SP), 2016, pp. 470–485. DOI: 10.1109/SP.2016.35.

[2] D. Adrian, K. Bhargavan, Z. Durumeric, P. Gaudry, M. Green, J. A. Halderman, N. Heninger, D. Springall, E. Thomé, L. Valenta, et al., “Imperfect forward secrecy: How diffie-hellman fails in practice”, in Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, ACM, 2015, pp. 5–17.

[3] N. J. Al Fardan and K. G. Paterson, “Lucky thirteen: Breaking the tls and dtls record protocols”, in Security and Privacy (SP), 2013 IEEE Symposium on, IEEE, 2013, pp. 526–540.

[4] N. AlFardan, D. J. Bernstein, K. G. Paterson, B. Poettering, and J. C. Schuldt, “On the security of rc4 in tls”, in Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13), 2013, pp. 305–320.

[5] N. J. AlFardan and K. G. Paterson, “Plaintext-recovery attacks against datagram tls”, in Network and Distributed System Security Symposium (NDSS 2012), The Internet Society, 2012.

[6] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and J. K. Zinzindohoue, “A messy state of the union: Taming the composite state machines of tls”, in 2015 IEEE Symposium on Security and Privacy, IEEE, 2015, pp. 535–552.

[7] K. Bhargavan, A. D. Lavaud, C. Fournet, A. Pironti, and P. Y. Strub, “Triple handshakes and cookie cutters: Breaking and fixing authentication over tls”, in 2014 IEEE Symposium on Security and Privacy, IEEE, 2014, pp. 98–113.

[8] K. Bhargavan, C. Fournet, R. Corin, and E. Zălinescu, “Verified cryptographic implementations for tls”, ACM Transactions on Information and System Security (TISSEC), vol. 15, no. 1, p. 3, 2012.

[9] G. V. Bard, “A challenging but feasible blockwise-adaptive chosen-plaintext attack on ssl.”, in SECRYPT, 2006, pp. 99–109.

[10] D. Bleichenbacher, “Chosen ciphertext attacks against protocols based on the rsa encryption standard pkcs# 1”, in Advances in Cryptology, —CRYPTO’98, Springer, 1998, pp. 1–12.

[11] B. Canvel, A. Hiltgen, S. Vaudenay, and M. Vuagnoux, “Password interception in a ssl/tls channel”, in Annual International Cryptology Conference, Springer, 2003, pp. 583–599.

[12] J. Jonsson and B. S. Kaliski Jr, “On the security of rsa encryption in tls”, in Annual International Cryptology Conference, Springer, 2002, pp. 127–142.

[13] V. Klima, O. Pokornỳ, and T. Rosa, “Attacking rsa-based sessions in ssl/tls”, in International Workshop on Cryptographic Hardware and Embedded Systems, Springer, 2003, pp. 426–440.

[14] M. Fischlin and F. Günther, “Replay attacks on zero round-trip time: The case of the tls 1.3 handshake candidates”, in Security and Privacy (EuroS&P), 2017 IEEE European Symposium on, IEEE, 2017, pp. 60–75.

[15] F. Günther, B. Hale, T. Jager, and S. Lauer, “0-rtt key exchange with full forward secrecy”, in Annual International Conference on the Theory and Applications of Cryptographic Techniques, Springer, 2017, pp. 519–548.

[16] J. C. Mitchell, M. Mitchell, and U. Stern, “Automated analysis of cryptographic protocols using mur/spl phi”, in Security and Privacy, 1997. Proceedings., 1997 IEEE Symposium on, IEEE, 1997, pp. 141–151.

[17] L. C. Paulson, “The inductive approach to verifying cryptographic protocols”, Journal of computer security, vol. 6, no. 1-2, pp. 85–128, 1998.

[18] P. F. Syverson and P. C. Van Oorschot, “On unifying some cryptographic protocol logics”, in Research in Security and Privacy, 1994. Proceedings., 1994 IEEE Computer Society Symposium on, IEEE, 1994, pp. 14–28.

[19] A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuéllar, P. H. Drielsma, P.-C. Héam, O. Kouchnarenko, J. Mantovani, et al., “The avispa tool for the automated validation of internet security protocols and applications”, in International Conference on Computer Aided Verification, Springer, 2005, pp. 281–285.

[20] G. Lowe, “Breaking and fixing the needham-schroeder public-key protocol using fdr”, in International Workshop on Tools and Algorithms for the Construction and Analysis of Systems, Springer, 1996, pp. 147–166.

[21] B. Blanchet et al., “An efficient cryptographic protocol verifier based on prolog rules.”, in csfw, vol. 1, 2001, pp. 82–96.

[22] C. J. Cremers, “The scyther tool: Verification, falsification, and analysis of security protocols”, in International Conference on Computer Aided Verification, Springer, 2008, pp. 414–418.

[23] S. Meier and B. Schmidt, The tamarin prover: Source code and case studies, Accessed: 2017-04-27. [Online]. Available: http://hackage.haskell.org/package/tamarin-prover-0.8.2.0.

[24] C. J. Cremers, P. Lafourcade, and P. Nadeau, “Comparing state spaces in automatic security protocol analysis”, in Formal to Practical Security, Springer, 2009, pp. 70–94.

[25] A. T. Luu, J. Sun, Y. Liu, J. S. Dong, X. Li, and T. T. Quan, “Seve: Automatic tool for verification of security protocols”, Frontiers of Computer Science, vol. 6, no. 1, pp. 57–75, 2012.

[26] S. Meier, “Advancing automated security protocol verification”, PhD thesis, ETH Zürich, 2013.

[27] R. Sasse, J. Dreier, C. Cremers, and D. Basin, Security protocol analysis using the tamarin prover-teaching material, 2017. [Online]. Available: https://github.com/tamarin-prover/teaching/blob/master/Tamarin-Tutorial-morning.pdf.

[28] S. Agrawal, D. Boneh, and X. Boyen, “Efficient lattice (h) ibe in the standard model”, in Annual International Conference on the Theory and Applications of Cryptographic Techniques, Springer, 2010, pp. 553–572.

[29] D. Basin, “Formally analyzing the tls 1.3 proposal”, [Online]. Available: https://www.ethz.ch/content/dam/ethz/special-interest/infk/inst-infsec/information-security-group-dam/research/software/TLS-1.3_thesis_vincent_stettler.pdf.

[30] I. Ristic, Bulletproof SSL and TLS: Understanding and Deploying SSL/TLS and PKI to Secure Servers and Web Applications. Feisty Duck, 2013.

[31] H. Krawczyk and H. Wee, “The optls protocol and tls 1.3”, in Security and Privacy (EuroS&P), 2016 IEEE European Symposium on, IEEE, 2016, pp. 81–96.

[32] W. M. Petullo, X. Zhang, J. A. Solworth, D. J. Bernstein, and T. Lange, “Minimalt: Minimal-latency networking through better security”, in Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security, ACM, 2013, pp. 425–438.

[33] J. P. a. Christof Paar, Understanding Cryptography: A Textbook for Students and Practitioners, 1st ed. Springer-Verlag Berlin Heidelberg, 2010.

[34] W. Stallings, Network Security Essentials: Applications and Standards, 4/e. Pearson Education India, 2000.

[35] A. Satapathy and J. L. L. M., “A comprehensive survey on ssl/ tls and their vulnerabilities”, International Journal of Computer Applications, vol. 153, no. 5, pp. 31–38, 2016, ISSN: 0975-8887. DOI: 10.5120/ijca2016912063. [Online]. Available: http://www.ijcaonline.org/archives/volume153/number5/26401-2016912063.

[36] W. Stallings, Cryptography and Network Security: Principles and Practice, International Edition: Principles and Practice. Pearson Higher Ed, 2014.

[37] T. Dierks and E. Rescorla, The transport layer security (tls) protocol version 1.2, 2008. [Online]. Available: https://www.rfc-editor.org/rfc/rfc5246.txt.

[38] R. Holz, Y. Sheffer, and P. Saint-Andre, "Recommendations for secure use of transport layer security (tls) and datagram transport layer security (dtls)", 2015.

[39] K. G. Paterson and T. van der Merwe, "Reactive and proactive standardisation of tls", in International Conference on Research in Security Standardisation, Springer, 2016, pp. 160–186.

[40] M. Marlinspike, "More tricks for defeating ssl in practice", Black Hat USA, 2009.

[41] Y Sheffer, R Holz, and P Saint-Andre, "Summarizing known attacks on transport layer security (tls) and datagram tls (dtls)", Tech. Rep., 2015.

[42] T. Duong and J. Rizzo, "Here come the ninjas", Unpublished manuscript, vol. 320, 2011.

[43] S. Vaudenay, "Security flaws induced by cbc padding—applications to ssl, ipsec, wtls...", in International Conference on the Theory and Applications of Cryptographic Techniques, Springer, 2002, pp. 534–545.

[44] J. Salowey, A. Choudhury, and D. McGrew, "Aes-gcm cipher suites for tls", 2008.

[45] P. Gutmann, "Encrypt-then-mac for transport layer security (tls) and datagram transport layer security (dtls)", 2014.

[46] B. Möller, T. Duong, and K. Kotowicz, "This poodle bites: Exploiting the ssl 3.0 fallback", Security Advisory, 2014.

[47] S. Bruce, "Applied cryptography: Protocols, algorithms, and source code in c", John Wiley & Sons, Inc., New York, 1996.

[48] G. Paul and S. Maitra, "Permutation after rc4 key scheduling reveals the secret key", in International Workshop on Selected Areas in Cryptography, Springer, 2007, pp. 360–377.

[49] I. Mantin and A. Shamir, "A practical attack on broadcast rc4", in International Workshop on Fast Software Encryption, Springer, 2001, pp. 152–164.

[50] S. Fluhrer, I. Mantin, and A. Shamir, "Weaknesses in the key scheduling algorithm of rc4", in International Workshop on Selected Areas in Cryptography, Springer, 2001, pp. 1–24.

[51] N. J. AlFardan, D. J. Bernstein, K. G. Paterson, B. Poettering, and J. C. Schuldt, "On the security of rc4 in tls.", in Usenix security, Washington DC, USA, vol. 2013, 2013.

[52] T. Duong and J. Rizzo, "The crime attack", in Presentation at ekoparty Security Conference, 2012.

[53] T. Be'ery and A. Shulman, "A perfect crime? only time will tell", Black Hat Europe, vol. 2013, 2013.

[54] A. Prado, N. Harris, and Y. Gluck, "Ssl, gone in 30 seconds", Breach attack, 2013.

[55] D. Brumley and D. Boneh, "Remote timing attacks are practical", Computer Networks, vol. 48, no. 5, pp. 701 –716, 2005, Web Security, ISSN: 1389-1286. DOI: https://doi.org/10.1016/j.comnet.2005.01.010. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1389128605000125.

[56] C. Brubaker, S. Jana, B. Ray, S. Khurshid, and V. Shmatikov, "Using frankencerts for automated adversarial testing of certificate validation in ssl/tls implementations", in Security and Privacy (SP), 2014 IEEE Symposium on, IEEE, 2014, pp. 114–129.

[57] S. Farrell and H. Tschofenig, "Pervasive monitoring is an attack", 2014.

[58] N. Mavrogiannopoulos, F. Vercauteren, V. Velichkov, and B. Preneel, "A cross-protocol attack on the tls protocol", in Proceedings of the 2012 ACM conference on Computer and communications security, ACM, 2012, pp. 62–72.

[59] D Gillmor, A Langley, N Modadugu, and B Moeller, "Negotiated finite field diffie-hellman ephemeral parameters for tls", Work in Progress, draft-ietf-tls-negotiated-ff-dhe-10, 2015.

[60] S. Fluhrer and Y. Sheffer, "Additional diffie-hellman tests for the internet key exchange protocol version 2 (ikev2)", 2013.

[61] M. Ray, S. Dispensa, E. Rescorla, et al., "Transport layer security (tls) renegotiation indication extension", Transport, 2010.

[62] TheTamarinTeam, Tamarin prover manual., Accessed: 2018-06-19. [Online]. Available: https://tamarin-prover.github.io/manual/index.html.

[63] B. Schmidt, "Formal analysis of key exchange protocols and physical protocols", PhD thesis, Citeseer, 2012.

[64] L.-S. Huang, S. Adhikarla, D. Boneh, and C. Jackson, "An experimental study of tls forward secrecy deployments", IEEE Internet Computing, vol. 18, no. 6, pp. 43–51, 2014.

[65] C. Cremers, M. Horvat, J. Hoyland, S. Scott, and T. van der Merwe, "A comprehensive symbolic analysis of tls 1.3", in Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, ACM, 2017, pp. 1773–1788.

[66] E. Rescorla, "The transport layer security (TLS) protocol version 1.3", RFC, vol. 8446, pp. 1–160, 2018.

[67] G. Lowe, "A hierarchy of authentication specifications", in Computer security foundations workshop, 1997. Proceedings., 10th, IEEE, 1997, pp. 31–43.

[68] A. M. Kakhki, S. Jero, D. Choffnes, C. Nita-Rotaru, and A. Mislove, “Taking a long look at quic: An approach for rigorous evaluation of rapidly evolving transport protocols”, in Proceedings of the 2017 Internet Measurement Conference, ACM, 2017, pp. 290–303.

[69] V. Lyubashevsky, “Lattice signatures without trapdoors”, in Annual International Conference on the Theory and Applications of Cryptographic Techniques, Springer, 2012, pp. 738–755.

[70] Ruhrsec 2017: "0-rtt key exchange with full forward secrecy", prof. dr. tibor jager. [Online]. Available: https://www.youtube.com/watch?v=QmTlzBFQheU.

[71] M. Fischlin and F. Günther, “Multi-stage key exchange and the case of google’s quic protocol”, in Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, ACM, 2014, pp. 1193–1204.

[72] E. Rescorla, 0-rtt and anti-replay (ietf tls working group mailing list). March 2015. [Online]. Available: https://www.ietf.org/mail-archive/web/tls/current/msg15594.html.

[73] A. Langley and W.-T. Chang, Quic crypto, 2013. [Online]. Available: https://docs.google.com/document/d/1g5nIXAIkN_Y-7XJW5K45IblHd_L2f5LTaDUDwvZ5L6g/edit.