**DOKUZ EYLÜL UNIVERSITY**
**GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES**

# ENHANCING SECURITY OF LOW-COST RESOURCE STRICKEN IOT DEVICES AND GATEWAYS

**by**
**Halit TÜRKSÖNMEZ**

**June, 2022**
**İZMİR**

# ENHANCING SECURITY OF LOW-COST RESOURCE STRICKEN IOT DEVICES AND GATEWAYS

**A Thesis Submitted to the**
**Graduate School of Natural and Applied Sciences of Dokuz Eylül University**
**In Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy in Computer Engineering**

**by**
**Halit TÜRKSÖNMEZ**

**June, 2022**
**İZMİR**

**Ph.D. THESIS EXAMINATION RESULT FORM**

We have read the thesis entitled "**ENHANCING SECURITY OF LOW-COST RESOURCE STRICKEN IOT DEVICES AND GATEWAYS**" completed by **HALİT TÜRKSÖNMEZ** under the supervision of **ASSOC.PROF.DR. MEHMET HİLAL ÖZCANHAN** and we certify that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Doctor of Philosophy.

Assoc. Prof. Dr. Mehmet Hilal ÖZCANHAN

Supervisor

| | |
|---|---|
| Assoc. Prof. Dr. Gökhan DALKILIÇ | Assoc. Prof. Dr. Mete EMİNAĞAOĞLU |
| Thesis Committee Member | Thesis Committee Member |
| Prof. Dr. Mehmet Süleyman ÜNLÜTÜRK | Prof. Dr. Murat KOMESLİ |
| Examining Committee Member | Examining Committee Member |

Prof. Dr. Okan FISTIKOĞLU

Director

Graduate School of Natural and Applied Sciences

**ACKNOWLEDGEMENTS**

First of all, I would like to express my sincere gratitude to my supervisor, Assoc. Prof. Dr. Mehmet Hilal ÖZCANHAN, for his immense knowledge, continuous support, encouragement, and helpful suggestions along with this study. His guidance helped me all the time in my research and writing of this thesis. It has been an honor to be his Ph.D. student.

Besides my advisor, I would like to thank the rest of my thesis committee, Assoc. Prof. Dr. Gökhan DALKILIÇ, and Assoc. Prof. Dr. Mete EMİNAĞAOĞLU, for their insightful comments and encouragement.

I would like to dedicate this thesis to my wife, Güner TÜRKSÖNMEZ, my daughter, Ela, and my grandson, Ege, without whose support I would not have been able to complete this work.

Halit TÜRKSÖNMEZ

**ENHANCING SECURITY OF LOW-COST RESOURCE STRICKEN IOT DEVICES AND GATEWAYS**

## ABSTRACT

In addition to its benefits, the Internet of Things (IoT) technology has also opened the way to novel privacy and security issues. The basis of IoT security starts with the trust in the IoT hardware and its supply chain. Since the integrated circuits (ICs) are the main building blocks of any computer system, IC and IoT supply chain security should be considered together. In previous studies, IC and IoT supply chains have been studied separately and separate solutions have been proposed. The most important of these IC supply chain-related solutions is the Supply Chain Hardware Integrity for Electronics Defense (SHIELD) scheme. Besides, four RFID (Radio Frequency Identification)-enabled solutions by the same authors have been proposed to secure the entire IoT supply chain. The works propose a new RFID-enabled supply chain tracing procedure. In each of these works, a variant of the same is proposed. However, the same variant authentication protocol coupled with the offline supply chain has such security vulnerabilities that make the whole supply chain unsafe. Besides that, the newly started SHIELD scheme has not matured yet, and the proposed authentication protocol improvements are still vulnerable to known non-invasive, side-channel attacks.

In this thesis; (i) a novel authentication protocol based on strong mutual authentication is proposed for SHIELD, which resists the demonstrated attacks on previous schemes, and (ii) an online supply chain hop-tracking procedure supported by a novel RFID mutual authentication protocol is proposed. The proposed authentication protocols have been verified by two well-accepted formal protocol analyzers. The verification results demonstrate that the proposed protocols overcome the previous works' vulnerabilities. The proposed novel online supply chain tracing procedure removes the previous offline supply chain tracing procedure weaknesses.

**Keywords:** Integrated circuits, Internet of Things, IC security, IoT security, supply chain security

# DÜŞÜK MALİYETLİ VE KISITLI KAYNAKLARA SAHİP NESNELERİN İNTERNETİ (IOT) AYGITLARI VE YÖNLENDİRİCİLERİNİN GÜVENLİĞİNİN ARTIRILMASI

## ÖZ

Sağladığı yararların yanı sıra, popüler Nesnelerin İnterneti (IoT) teknolojisi, yeni güvenlik ve gizlilik sorunlarının da yolunu açmıştır. IoT güvenliğinin temeli, IoT donanımına ve tedarik zincirine duyulan güven ile başlar. Tümleşik devreler (IC'ler) herhangi bir bilgisayar sisteminin ana yapı taşları olduğundan, IC ve IoT tedarik zincirlerinin güvenliği birlikte düşünülmelidir. Daha önceki çalışmalarda IC ve IoT tedarik zincirleri ayrı ayrı incelenmiş ve ayrı çözümler önerilmiştir. Söz konusu IC tedarik zinciri ile ilgili çözümlerin en önemlisi Elektronik Savunma İçin Tedarik Zinciri Donanım Bütünlüğü (Supply Chain Hardware Integrity for Electronics Defense - SHIELD) tasarımıdır. Ayrıca, tüm IoT tedarik zincirini güvence altına almak için aynı yazarlar tarafından dört RFID (Radio Frequency Identification) özellikli çözüm önerilmiştir. Çalışmalar, yeni bir RFID özellikli tedarik zinciri izleme prosedürü önermektedir. Bu çalışmaların her birinde, aynı çözümün bir varyantı önerilmektedir. Bununla birlikte, çevrimdışı tedarik zinciriyle birleştirilen aynı varyant kimlik doğrulama protokolü, tüm tedarik zincirini güvensiz hale getiren güvenlik açıklarına sahiptir. Bunun yanı sıra, yeni başlatılan SHIELD tasarımı henüz olgunlaşmamıştır ve önerilen kimlik doğrulama protokolü iyileştirmeleri, bilinen istilacı olmayan, yan kanal saldırılarına karşı hala savunmasızdır.

Bu tezde; (i) SHIELD için, önceki tasarımlarda gösterilen saldırılara direnen, güçlü karşılıklı kimlik doğrulamaya dayalı yeni bir kimlik doğrulama protokolü önerilmiştir ve (ii) yeni bir RFID karşılıklı kimlik doğrulama protokolü tarafından desteklenen bir çevrimiçi tedarik zinciri aşama izleme prosedürü önerilmiştir. Önerilen kimlik doğrulama protokolleri, iyi kabul görmüş iki resmi protokol analizcisi tarafından doğrulanmıştır. Doğrulama sonuçları, önerilen protokollerin önceki çalışmaların güvenlik açıklarını giderdiğini göstermektedir. Önerilen çevrimiçi tedarik zinciri

izleme prosedürü, önceki çevrimdışı tedarik zinciri izleme prosedürünün zayıf yönlerini ortadan kaldırmaktadır.

**Anahtar kelimeler:** Tümleşik devreler, Nesnelerin İnterneti, tümleşik devre güvenliği, IoT güvenliği, tedarik zinciri güvenliği

**CONTENTS**

**Page**

## LIST OF FIGURES

## LIST OF TABLES

**LIST OF SYMBOLS**

$\oplus$ : XOR operator

$\|$ : Concatenation operator

**ABBREVIATIONS**

| | |
|---|---|
| AES | : Advanced Encryption Standard |
| CDIR_CTR | : Counter-based Combating Die/IC Recycling |
| CDIR_RO | : Ring Oscillator-based CDIR |
| CDTA | : Counterfeit Detection, Traceability, and Authentication |
| CL-AtSe | : Constraint-logic-based Attack Searcher |
| CNTR-SHIELD | : Counter SHIELD |
| C-SCRM | : Cybersecurity Supply Chain Risk Management |
| DARPA | : Defense Advanced Research Projects Agency |
| DB | : Database |
| DoS | : Denial of Service |
| ECID | : Electronic Chip ID |
| EEPROM | : Electronically Erasable Programmable Read-Only Memory |
| HLPSL | : High-level Protocol Specification Language |
| HMAC | : Hash Message Authentication Code |
| $I^2C$ | : Inter-Integrated Circuit |
| IC | : Integrated Circuits |
| IF | : Intermediate Format |
| IoT | : Internet of Things |
| IoT-SC | : IoT-based Supply Chain |
| I-SCRAM | : Supply Chain Risks in IoT Iystems |
| NFC | : Near Field Communication |
| OFMC | : On-the-Fly Model Checker |
| PCB | : Printed Circuit Board |
| PUF | : Phisically Unclonable Functions |
| ReSC | : RFID-Enabled Supply Chain |
| RF | : Radio Frequency |
| RFID | : Radio-Frequency Identification |
| SATMC | : Boolean Satisfiability (SAT)-based Model Mhecker |
| SBMABS | : Strongly Bonded and Mutually Authenticated Protocol for SHIELD |

| | |
|---|---|
| SHIELD | : Supply Chain Hardware Integrity for Electronics Defense |
| SPDL | : Security Protocol Description Language |
| SPI | : Serial Peripheral Interface |
| SRAM | : Static Random Access Memory |
| STRAP | : Strong RFID Authentication Protocol |
| TA4SP | : Tree Automata-based Protocol Analyzer |
| TRNG | : True Random Number Generator |
| UART | : Universal Asynchronous Receiver-Transmitter |
| WiFi | : Wireless Fidelity |

# CHAPTER 1
# INTRODUCTION

## 1.1 General

The Internet of Things (IoT) can be described as a network of diverse computation-capable entities, such as the present-day ubiquitous systems that have sensors and actuators, sitting on top of the current Internet infrastructure. Running on the global Internet, IoT is changing the habits of modern society, by facilitating human life on a scale that was not imagined to come so soon. Adapting well to individual contexts, IoT is changing the way we live, work and conduct our business. The total number of connected IoT devices on earth is estimated to nearly triple, from 11.57 billion predicted in 2022 to more than 25.44 billion in 2030 (See, 2022). One other reason for the speed of IoT solutions increase is the fast progress in the areas of cloud computing and wireless sensor networks (Perrig, Stankovic, & Wagner, 2004). One of the benefits of IoT is the enhancement of the information and service exchange in the worldwide supply chains of commercial goods. IoT is helping in the monitoring of supply chain activities which is very difficult, due to the increasing global good traffic. It has been shown that the journey of critical and valuable goods can be tracked from anywhere and anytime, by using IoT technology. The good-transfer system malfunctions can be determined and localized easily and quickly, by the monitoring of the goods movement at their supply chain hops. Not only the commercial goods but also the IoT supply chain itself is one of the most critical chains requiring protection.

However, IoT does not only offer advantages and benefits but also brings new privacy and security issues (Kanuparthi, Karri, & Addepalli, 2013). While the traditional Internet was a network of uniform computer devices, IoT exhibits heterogeneity. Usually, IoT devices are resource-stricken in computing power, wireless operation distance, and storage capabilities. Consequently, IoT privacy and security are much harder to provide than conventional systems. Therefore, the protection of both the IoT tracked supply chains and the IoT devices supply chains

themselves is not an easy task. A general structure of the IoT supply chain and its vulnerabilities are shown in Figure 1.1.

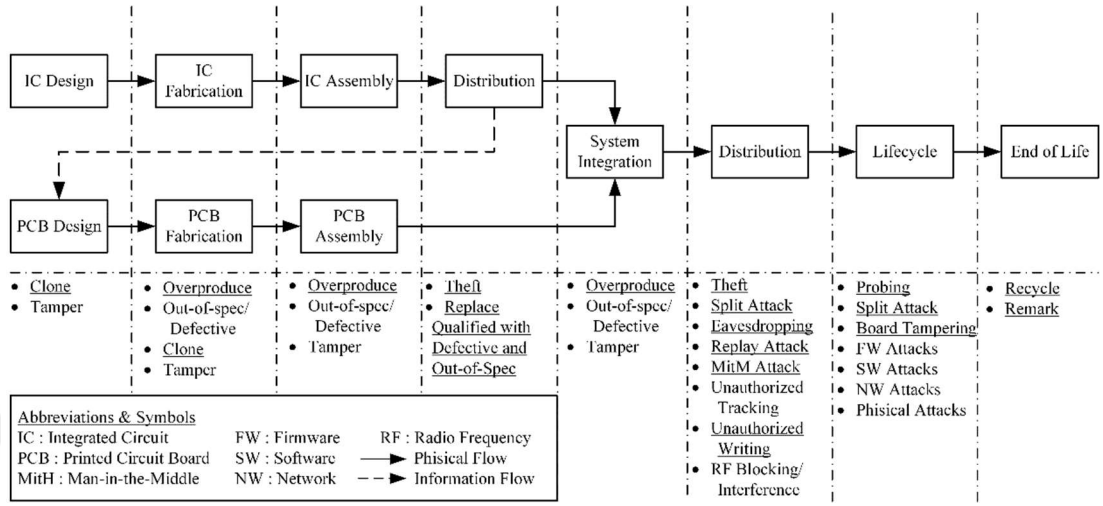



Figure 1.1 General structure of the IoT supply chain and vulnerabilities faced at different stages (Yang, Forte, & Tehranipoor, 2017)

At the Integrated Circuit (IC) design level, there is intellectual property theft, which is a fact but, a different scope than our field of study. At the printed circuit board (PCB) design-fabrication-assembly and integration levels, the produced IC or PCB can be physically tampered with, overproduced, cloned i.e. counterfeited, or tampered with (Chakraborty, Narasimhan, & Bhunia, 2009; Cobb, Garcia, Temple, Baldwin, & Kim, 2010; Guin, DiMase, & Tehranipoor, 2014; Sathyanarayana, Robinson, & Beyah, 2013; Subramanyan et al., 2013; Türksönmez & Özcanhan, 2021). For example, hardware Trojans can be planted in IoT devices. In the distribution, Genuine IoT devices may be mixed with fakes or clones by untrusted supply chain partners, or their high-cost components may be stolen or replaced. IoT devices may be recycled or reintroduced into the market, or the supply chain after their lifetime. Thereafter, the stolen, cloned or recycled IoT devices would eventually be installed at different places other than their originally registered locations.

As a result, the supply chains of IoT devices and semiconductors are also in need of protection from rogue element cyber-attacks. Yang, Forte, & Tehranipoor (2015a) have proposed a full solution, after comprehensively summarizing the privacy and

security problems of IoT devices and their supply chain. The confidentiality of exchanged messages is provided by using symmetric encryption such as Advanced Encryption Standard (AES), in the work. To verify data integrity, the keyed Hash Message Authentication Code (HMAC) is used. Most of the known typical cyber-attacks can be stopped by the used methods but, the authentication protocols can have weaknesses that cause vulnerabilities that blow away the security of the whole proposal.

## 1.2 Purpose

Challenges are progressive in IoT technology and have to be reciprocated by equaled counter research. The weaknesses in previously proposed solutions motivated us for providing better security techniques; not only for the solutions proposed in the works (Jin & van Dijk, 2019; Yang, Forte, & Tehranipoor, 2015a, 2015b, 2017, 2018) but also for other consumer supply chains as well. Therefore, our main purpose is to provide stronger security for both RFID-enabled devices and RFID-enabled supply chains. However, the lack of support for innovative academic solutions to develop attack detection and prevention on supply chains is aggravating the challenge (Guin, Huang, DiMase, Tehranipoor, & Makris, 2014). The absence of a formal announcement about a secure supply chain proves that the security challenges have not yet been met satisfactorily.

## 1.3 Contributions of this Thesis

The contributions of this thesis to RFID-enabled IC and IoT supply chain authentication protocols are explained in the following sub-sections.

### *1.3.1 Contributions of SBMAPS to IC Supply Chain*

The contributions of the novel Strongly Bonded and Mutually Authenticated Protocol for SHIELD (SBMAPS) protocol (Özcanhan, & Türksönmez, 2020) to the studied previous proposals are:

i. Elimination of exaggerated assumptions and the need for the trusted smartphones in the supply chain,

ii. Elimination of the threat by rogue employees and the need to keep sensor status bits on the dielet,

iii. Strong mutual authentication of the dielet and the server at every hop in the supply chain,

iv. Shared secret keys enumerated and controlled by the server,

v. New (changing) session key for every new round,

vi. Optional ID update, to prevent malicious tracking.

### *1.3.2 Contributions of STRAP to IoT Supply Chain*

The contributions of the novel Strong RFID Authentication Protocol (STRAP) protocol to meeting the security challenges faced in previous proposals are:

i. Elimination of exaggerated assumptions and the need for trusted RFID readers in the supply chains,

ii. Elimination of the threat by rogue employees,

iii. Strong mutual authentication of the embedded RFID tags and the server at every hop in the supply chain,

iv. Shared secret keys enumerated and controlled by the server,

v. New (changing) session keys for all tags, for every new round.

## 1.4 Organization of the Thesis

This thesis consists of six chapters. In this chapter, general information about IoT, IoT supply chain, and its vulnerabilities are mentioned. Then the purpose of this thesis, the motivation behind it, and the contribution to the literature are explained. The rest of the thesis is organized as follows.

Chapter 2 provides general information about related studies and a literature review on IoT security and IoT supply chain security.

In Chapter 3, background information is given about the solutions aimed at enhancing the security of RFID enabled IC and IoT supply chain. The security vulnerabilities and weaknesses of the solutions previously proposed and the attacks exploiting these vulnerabilities and weaknesses are also explained.

In Chapter 4, AVISPA and Scyther Tool, which are used for the formal verification and validation of the security protocols are presented, and two novel authentication protocols are proposed to overcome the security vulnerabilities and weaknesses of the solutions previously proposed.

In Chapter 5, the formal and informal security and performance analyses on the novel authentication protocols proposed in Chapter 4 are explained.

Finally, in Chapter 6, conclusions, and future works are presented.

# CHAPTER 2
# LITERATURE REVIEW AND RELATED WORK

A mutual authentication scheme based on the adoption of Physically Unclonable Functions (PUFs) has been proposed by the authors of work (Barbareschi, De Benedictis, La Montagna, Mazzeo, & Mazzocca, 2019). Furthermore, they discussed the security of the protocol by presenting different attack scenarios and showed that it is strong against tampering and man-in-the-middle attacks. But, the authors of work (Adeli & Bagheri, 2019) has shown that the proposed scheme is vulnerable to traceability, desynchronization, and impersonation attacks.

The survey (Sey, 2018) has been focused on lightweight mutual authentication methods for IoT. The survey has showed that only a few of the various authentication methods used in the IT industry are suitable for IoT. The importance of mutual authentication for IoT has been shown in the survey. Because of the constraint on the IoT devices' resources, lightweight authentication has been stated as very practical by the author, considering the areas such as low bandwidth.

The authors of the work (Arasteh et al., 2016) studied the relationship between the IoT and the smart city. Furthermore, they also explained challenges and crucial experiences in this area. They specified that the system might be open to some attacks like side-channel attacks in conditions where all data is gathered and analyzed on a common IoT platform.

An identity-based authentication protocol for IoT has been proposed by the authors of the work (Alzahrani & Mahmood, 2021). They analyzed the security of their authentication protocol informally without using formal verification tools.

The authors of work (Ziegeldorf, Morchon & Wehrle, 2014) examined the privacy and confidentiality problems in IoT particularly. They studied the progressing features in IoT and classified the privacy threats.

An authentication protocol for IoT devices has been proposed by Ding, Wang, Xie & Li (2021), based on signcryption and elliptic curve techniques. They exposed that the protocol reduced the computation and communication costs of the client devices while ensuring the security requirements when compared to other protocols.

A previously proposed mutual authentication protocol for IoT devices has been analyzed by Özcanhan & Utku (2018). Their recommendation provided stronger security by resolving the weaknesses of the protocol studied.

The authors of the work (Bae & Kwak, 2017) proposed a smartcard-based secure authentication protocol for multi-gateway IoT environments. By using the AVISPA tool, they formally validated and verified the proposed protocol. They showed that the proposed protocol is secure against session key leakage attacks, user impersonations, and some other attacks.

Smartcard-based two-factor authentication protocol for multi-gateway IoT proposed by the authors of the work (Bae & Kwak, 2017) has been reviewed in the work (Lee, Yu, K. Park, Y. Park & Y. Park, 2019). The authors showed that the protocol analyzed was vulnerable to gateway spoofing attacks, user impersonation attacks, and session key disclosure, and did not provide mutual authentication. They proposed a three-factor mutual authentication protocol to handle the security vulnerabilities of the multi-gateway IoT environments.

A multi-key-based mutual authentication mechanism between server and IoT device has been proposed by the authors of work (Shah & Venkatesan, 2018). They made the power and security analysis of the proposed algorithm. They defined that the algorithm they proposed is secure against side-channel attacks.

Authors of work (Farooq & Zhu, 2019) analyzed supply chain risks of the IoT systems and their individual characteristics, examined research areas in supply chain security, and showed future research directions. They concluded that the first direction is the accurate mapping of the threat view in point of the supply chain. A full risk

assessment and impact analysis are then required. At last, mitigation plans are required to serve as a guide for building best practices.

In the work (Boyens et al., 2022), the authors provide direction to the organizations on recognizing, evaluating, and reducing cybersecurity risks all over the supply chain at all levels of their organizations. (Boyens et al., 2022) combines cybersecurity supply chain risk management (C-SCRM) with risk management activities by applying a multilevel, C-SCRM-specific approach, comprising direction on the development of C-SCRM strategy excution plans, C-SCRM plans, C-SCRM policies, and risk evaluations for services and products.

In the work (Omitola & Wills, 2018), the authors discussed the security problems covering an IoT supply chain. They first started by explaining what an IoT endpoint is, its different types, and the associated security challenges. Then they described the endpoint as an IoT supply chain's output. They used the list of security challenges to define what IoT attack surfaces are, along with the threats and vulnerabilities faced by a typical IoT supply chain. They concluded by discussing mitigation measures against possible security vulnerabilities in such a supply chain.

Abdel-Basset, Manogaran, & Mohamed (2018) proposed a framework for IoT-based supply chain. The proposed system automates the products' identification process, tracks and traces products worldwide, obtains transparency, decreases cost and time, and then will obtain customer satisfaction.

Hiromoto, Haney, & Vakanski (2017) advised the development of an IoT supply chain risk management architecture that is cyber-secure. The proposed system was designed to mitigate vulnerabilities of hostile supply chain risks by implementing distributed system coordination, cryptographic hardware monitoring, and machine learning technics.

The authors of the work (Hassija, Chamola, Gupta, Jain, and Guizani, 2020) examined the application areas of the supply chain that is security-critical and provided

a comprehensive review of security problems in the recent supply chain architecture. Several emerging technologies such as physically unclonable functions (PUF), machine learning, and blockchain as solutions to vulnerabilities in the current infrastructure of the supply chain were also explained. The latest research examined in this study showed an increased sensitivity in the industry to new and emerging technologies such as machine learning, blockchain, and IoT. Blockchain has not been widely adopted while many organizations have adopted IoT applications and artificial intelligence systems in their businesses. They also found that authentication systems based on PUF have gained much ground over the past decade. They concluded that a correct reference model for their implementation in complicated supply chains is still not present.

The authors of the work (Kieras, Farooq, & Zhu, 2021) proposed a framework to analyze supply chain risks in IoT systems and to support risk-mitigating decisions (I-SCRAM). After defining an expanded system model that consists of interconnected components and a hierarchy of component vendors, they developed and proposed metrics to quantify systemic risks. Finally, they presented a decision framework that helps in the selection of vendors to mitigate supply chain risk. As a result, they demonstrated that I-SCRAM successfully minimizes system risk as higher budget and more reliable component resources become available while providing flexibility in prioritizing risk sources.

In the work (Shahzad, Zhang, & Gherbi, 2020), the authors designed and modeled a scaled IoT-based supply chain (IoT-SC) system, containing different participants and operations, and deploys mechanisms to increase the security and a digital signature scheme to increase potentially secured non-repudiation security service for the worst-case scenario and to increase privacy to keep users' sensitive location and personal information that is protected against adversarial entities to the IoT-SC system.

Zhang & Sakurai (2020) introduced the general history and background of the supply chain and related technologies, which have been applied or are to be applied to the supply chain to lower cost and increase security. They provided a detailed

evaluation of recent research studies and industrial cases from various popular companies. They also illustrated the IoT enablement and security problems of the current digital supply chain system and the existing blockchain's role in this kind of digital system. Finally, they concluded several potential or existing security problems and challenges that supply chain management is facing.

In the work (Kieras, Farooq, & Zhu, 2020), the authors developed a security risk assessment framework (RIoTS) inspired by system reliability theory to combine the supply chain. They also analyzed the impact of grouping within suppliers that may pose hidden risks to the systems from malicious supply chain actors. The results showed that the proposed analysis can reveal hidden threats intended to the IoT ecosystem from potential supplier collusion.

## 2.1 Related Studies on IC Supply Chain Security

The IC supply chain and its vulnerabilities are studied in detail from design to end of life stage, in the literature (Jin & van Dijk, 2019; Yang, Forte, & Tehranipoor, 2017). After identifying vulnerabilities and threats, developing countermeasures gathered momentum among researchers. The threat models that aim at gaining unfair advantages and benefits in the IC supply chain are summarized below (Guin, Huang, DiMase, Tehranipoor, & Makris, 2014; Jin & van Dijk, 2019; Rostami, Farinaz, & Karri, 2014):

i. *Hardware Trojans:* Adding malicious circuitry or modifying original ICs.

ii. *IP Piracy:* Illegally pirating a design without the consent of the designer.

iii. *IC Overproduction:* Building more than the ordered number to sell the excess in the gray market.

iv. *Reverse Engineering-Tampering:* Tampering with ICs to manufacture counterfeits.

v. *Side-channel Analysis:* Extracting secret information by exploiting physical phenomena (electro-magnetic emission, timing, power consumption) involving the ICs.

vi. *Counterfeiting/Cloning:* Illegally forging or imitating the original ICs.

vii. *Recycling:* Reselling defective, end of service ICs.

The same works also detail the countermeasures. The most comprehensive countermeasure is identified as the Supply Chain Hardware Integrity for Electronics Defense (SHIELD) program, by Defense Advanced Research Projects Agency (DARPA)'s Microsystems Technology Office (DARPA, 2022). Hardware named "dielet" is at the core of the initiative. The hardware is required to be no more than a 100x100 micron circuitry, inserted in the host IC package.

The circuitry has to include an encryption engine, passive sensors, near-field power, and communications technologies to verify the identity and authenticity of the package, at any stage after the insertion process. By capturing tampering attempts through sensors and reporting to a secure server via radio frequency (RF) communication, the integrity of the IC package is tracked. The SHIELD's dielet is shown in Figure 2.1 and it is expected to have the following features (DARPA, 2022):

i. A circuitry that self-destructs upon any attempt to physically detach it from the host,

ii. A root-of-trust for identification (ID) and cryptographic key (that never leaves the dielet) storage that is prohibitively expensive and time-consuming to reverse engineer,

iii. An on-board, complete, and compact encryption engine that encrypts the communication using the recorded cryptographic key,

iv. Passive sensors to detect any tampering attempt on the package,

v. Inductive powering and RF communication for contactless operation,

vi. Resiliency against component probing,

vii. A very cheap price per unit.

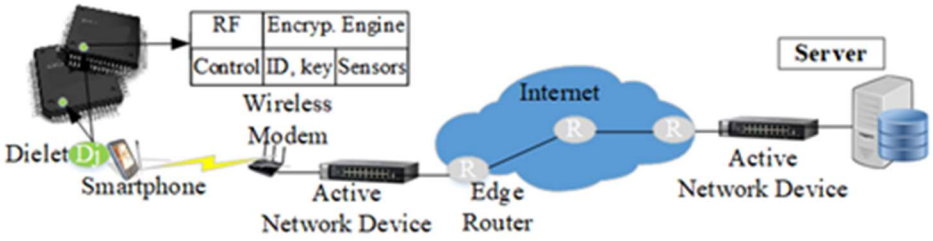



Figure 2.1 The 100 x 100 µm "dielet" on a host IC package. Status information of passive sensors is sent in encrypted form to a server, via the RF component

A smartphone (Figure 2.1) with a suitable probe is used, as the mediator between the dielet and the server (Jin & van Dijk, 2019). The remote server contains each dielet's identification number $D_j\_ID$, cryptographic key, and the passive sensor status bits. The dielet-smartphone communication channel is a near field wireless network; while the smartphone-server connection is over the Internet. There are many proposals about the hardware architecture of the dielet. But, the effectiveness of the SHIELD initiative is yet to be seen, as DARPA partners with companies for IC production (Koushanfar & Karri, 2014).

Observing the disadvantages and weaknesses of the SHIELD protocol, researchers offered rectifications by proposing additions and corrections. Physically unclonable functions (PUFs) are extensively offered as security primitive for generating the dielet $D_j\_ID$ (Guin, Huang, DiMase, Tehranipoor, & Makris, 2014; Jin & van Dijk, 2019; Rostami, Majzoobi, Koushanfar, Wallach, & Devadas, 2014; Rührmair, Devadas, & Koushanfar, 2012).

Although PUFs can serve as a private fingerprint for each IC, they are susceptible to side-channel attacks (Rostami, Farinaz, & Karri, 2014). True Random Number Generators (TRNG) found inherently in the ICs are considered natural alternatives to PUFs (Jin & van Dijk, 2019).

## 2.2 Related Studies on IoT Supply Chain Security

In work (Yang, Forte, & Tehranipoor, 2015b), Yang et al. proposed an RFID-enabled supply chain solution for network devices that address the entire supply chain's security and management problems. In the rest of this thesis, the proposed solution in work (Yang et al., 2015b) will be mentioned as RFID-Enabled Supply Chain -1 (ReSC-1). ReSC-1 is proposed as a solution resistant to illegal network service access, product theft, and counterfeit injection by joining the techniques of (i) unique tag trace which records tag history information, and (ii) one-to-one mapping between tag identity and control chip identity.

Yang et al. modified the original ReSC-1 solution they proposed in work (Yang et al., 2015b) and presented a new RFID-enabled solution in work (Yang, Forte, & Tehranipoor, 2015a) that aims at protecting endpoint devices in the IoT supply chain. In the rest of this thesis, the proposed solution in work (Yang et al., 2015a) will be mentioned as RFID-Enabled Supply Chain – 2 (ReSC-2). ReSC-2 has the following properties and benefits when compared with the original ReSC-1 approach: (i) ReSC-2 is specific to the supply chain of IoT devices, (ii) ReSC-2 provides mutual authentication between RFID readers and tags, and (iii) ReSC-2 does not need RFID readers to upload backups of tag traces to the central database (Yang et al., 2015a).

Furthermore, Yang et al. improved their ReSC-1 and ReSC-2 solutions and presented one more RFID-enabled solution in work (Yang, Forte, & Tehranipoor, 2018) for defending the IoT supply chain. In the rest of this article, the proposed solution in work (Yang et al., 2018) will be mentioned as RFID-Enabled Supply Chain – 3 (ReSC-3). Similar to ReSC-1, ReSC-2 and ReSC-3 use the same general supply chain framework aimed at solving different problems in the supply chain of IoT devices. ReSC-3 is proposed as a solution resistant to split attacks, product theft throughout the entire supply chain, counterfeit injection, illegal network service access, and device recycling by incorporating the techniques of (i) one-to-one mapping between RFID tag identity and control chip identity, (ii) unique tag trace which records tag provenance and history information, and (iii) neighborhood attestation of IoT devices.

Besides the ReSC solutions stated in previous paragraphs, Yang et al. presented another RFID-enabled solution in work (Yang, Forte, & Tehranipoor, 2017) aimed at the detection of counterfeit ICs and tampered boards, enabling traceability and authentication in the IoT supply chain. Yang et al. called this solution as Counterfeit Detection, Traceability, and Authentication (CDTA). As in the ReSC solutions previously stated, CDTA also uses the same supply chain framework. Unlike the ReSC solutions, CDTA aims at the detection of all counterfeit chips used in IoT devices, not just a counterfeit control chip or an RFID chip.

# CHAPTER 3
# BACKGROUND INFORMATION

## 3.1 SHIELD Authentication Protocol

The recommended SHIELD authentication protocol between the dielet and the server is shown in Figure 3.1. The five-step authentication is in fact three steps; since step 1 is merely a request and step 5 is a notification of the smartphone. Authentication protocols consisting of message exchanges of less than 4 steps are not considered mutual authentications and have known weaknesses (Dalkilic, Özcanhan, & Çakir, 2013). The dielet never authenticates the smartphone or the server and message blocking is not considered. Although the channel between the smartphone and the server is assumed secure, nothing is stopping an adversary from sending a wireless fabricated "OK" result to the smartphone, in the final step. Other works based on known attacks against computers point to more weaknesses and vulnerabilities.

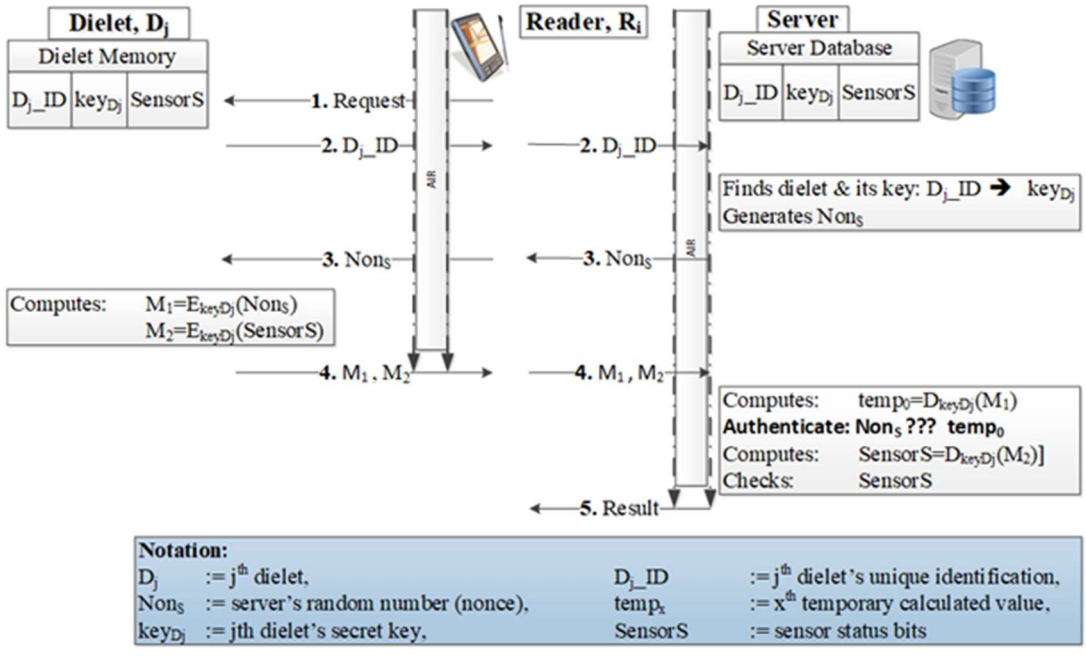



Figure 3.1 The SHIELD protocol, the nonce is passed both in plaintext and ciphertext form; $M_2$ is constant and "*Result*" is vulnerable

For example, the sensor status bits are passed as a repeated, encrypted constant, since sensor "OK" bits are the same. The server's message freshness indicator, random number (nonce) $Nons$ appears both in plaintext and ciphertext form; allowing the formation of a look-up table. The symmetric $key_{Dj}$ of dielet $D_j$ is constant, instead of being variable, allowing the accumulation of side-channel information (Rostami, Farinaz, & Karri, 2014).

## 3.2 CNTR-SHIELD Protocol

Jin & van Dijk (2019) offered countermeasures to overcome the weaknesses of the SHIELD authentication protocol. Claiming that a Denial of Service (DoS) attack on a single dielet is possible in SHIELD, the following corrections are proposed in work (Jin & van Dijk, 2019):

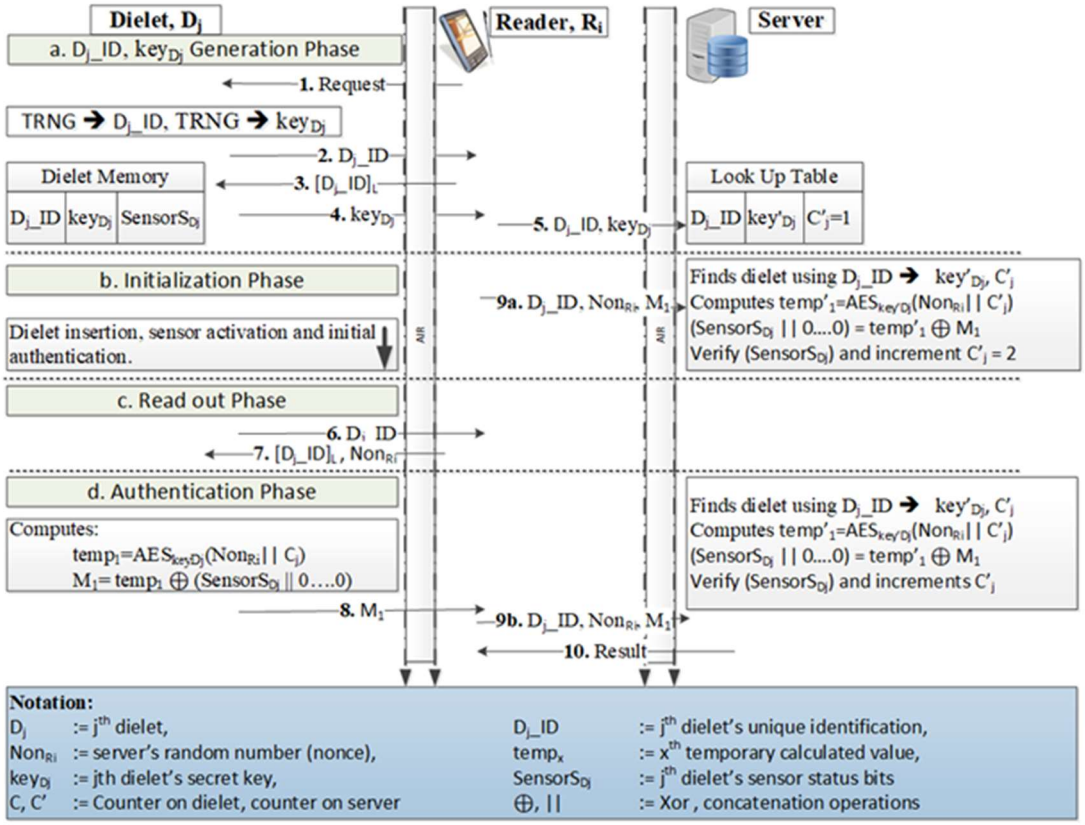



Figure 3.2 The CNTR-SHIELD protocol (Jin, & van Dijk, 2019)

i. Adding a read-out mode before the authentication,

ii. Counter mode encryption to prevent linking ciphertexts to corresponding plaintexts over time,

iii. Four distinct modes: ID generation, initialization, readout, and authentication modes.

Since work (Jin & van Dijk, 2019) is counter-based, it is named CNTR-SHIELD in the present work and given in Figure 3.2. The number of message exchanges has been increased and the encryption has been fortified, leading the authors to claim that CNTR-SHIELD is both secure and efficient. But, the work has two surreal assumptions: The generation and initialization processes cannot eavesdrop; the intermediary smartphone can be trusted.

However, the generation phase involves wireless transmission, where malicious monitoring of electromagnetic emissions is a well-accepted threat, in security literature. Nevertheless, the encryption $key_{DJ}$ is transmitted in plaintext to the smartphone in step 4 (Figure 3.2). This is an over-simplistic design, putting the protocol's security in danger. Since message 4 can travel in the air, beyond the smartphone; an adversary may have a chance to record the keys of the inserted dielets, with a high gain antenna. The proposal is further weakened when the dielet is recommended for "seamless integration" with Electronic Product Code tags. Tags use ultra-high radio frequency identification (RFID) technology. Numerous well-known attacks on UHF tags exist in the literature, based on malicious listening. As an excuse for the weakness, the authors declare smartphone attacks outside their scope and recommend alternative security measures. However, authentication protocols are normally designed, taking the existence of rogue (dishonest, malicious, unauthorized) readers and smartphones into consideration (Özcanhan, 2014; Özcanhan, Baytar, Utku, & Dalkılıç, 2015; Özcanhan, Dalkılıç, & Utku, 2014; Peris-Lopez, Orfila, Castro, & Lubbe, 2011; Peris-Lopez, Orfila, Mitrokotsa, & Van der Lubbe, 2011; Yen, Lo, & Wu, 2012).

The exaggerated assumptions in (Jin & van Dijk, 2019) are as follows:

i. Smartphones used for tracking are secure ─ although rogue workers exist,
ii. Manufacturers and integrators are fully trusted ─ contrasting the gray market of overproduced ICs,
iii. The chain of distribution is known a priori ─ contradicting unplanned/uncontrollable stops,
iv. Only transitions in the supply chain are insecure ─ contradicting most sophisticated, repeated attacks launched at the start or end of the supply chain.

### *3.2.1 Attacks on CNTR-SHIELD Protocol*

Due to the assumptions pointed out above, serious weaknesses emerge in CNTR-SHIELD. The reasons for the weaknesses and their consequences are presented below:

#### *3.2.1.1 DoS Attack*

Message blocking or communication loss is ignored in CNTR-SHIELD, which opens the way for desynchronization of the counter value in the dielet ($C_j$) and the server ($C'_j$). The authors downplay the DoS attack, by claiming it affects a single IC package and does not require an expensive recovery process. However, this is not the case. At an intermediary location, if an adversary or a rogue employee sends the same nonce $Non_{Ri}$ to all of the dielets in range (or one by one); the dielets increment their counters, at the authentication phase. By blocking the communication to the server, the counters at the server-side are all left un-incremented, effectively desynchronizing the dielets and the server.

Furthermore, knowing that the counter value is 2 at the exit of the manufacturer; repeating the above attack many times until the counter reaches its maximum value, puts the dielets completely out of service. Therefore, a single and bunch of DoS attacks are possible on CNTR-SHIELD. The same undesired result will happen, if an

inexperienced employee tries to read the dielets many times, with a broken link to the server.

#### *3.2.1.2 Full-disclosure Attack*

In the generation phase, sending the dielet ID ($D_j\_ID$) and the secret $key_{Dj}$ in plaintext through the air is an open breach of security principles. The manufacturer can be trusted, but malicious eavesdropping is a fact and the reason for developing security measures. Therefore, we will not try to prove the possibility of capturing plaintext messages maliciously, using a high gain antenna. The weakness makes cloning, counterfeiting, and unauthorized recycling possible. For example, an adversary who captures data from the generation phase can also record the constant, encrypted sensor "OK" values, in the initialization phase of the same session. Now, the adversary has $D_j\_ID$, $key_{Dj}$; knows the last hop's counter value and the sensor "OK" value. Counterfeits with foul finite state machines can be prepared, with the information in hand. At the delivery phase to the end-user (last hop), the counterfeit obtains $temp_1$ from the smartphone and prepares $M_1$. A server receiving the correct verification $M_1$ in step 9b (Figure 3.2) can never differentiate a counterfeit from an authentic IC.

#### *3.2.1.3 Lookup Table Attack*

The simple XOR operation ruins the security of the CNTR-SHIELD protocol while forming the verification message $M_1$. The ($SensorS_{Dj}$ || *0....0*) value is constant for "OK" status. The work does not clearly define "OK" but we know that it is equivalent to "1" bits; otherwise the whole ($SensorS_{Dj}$ || *0....0*) value will be zero and XORing $temp_1$ with zero would yield $temp_1$ itself. Knowing the number of sensor status bits suffices to capture $temp_1$. Simply, XORing $M_1$ as in Equation (1) gives $temp_1$:

$$(temp_1) = M_1 \oplus (1\ldots.1 \,||\, 0\ldots.0) \tag{3.1}$$

Exposing $temp_1$ means obtaining $AESkey_{Dj}(Non_{Ri} \,||\, C_j)$. This information puts the authors of CNTR-SHIELD face to face with their own criticism: “Ciphertexts corresponding to the same plaintext are linked over time”. The time merely depends on the smartphone’s different random number generation capacity. The studies investigating the generation of similar pseudo-random numbers by smartphones will not be discussed in the present work.

#### *3.2.1.4 Replay Attack*

Reception of the clipped $D_j\_ID$ from the insecure smartphone is accepted as an authentication of the server. This is a serious weakness, which allows an attacker to perform the following actions. A rogue employee at the end of the supply chain energizes several legitimate dielets and performs an authentication round recording the exchanged messages, without sending them to the server. By doing so, a legal set of step 9b messages are obtained, but the dielet counters are incremented by one. Next, the rogue employee replaces the equal number of legal pieces with counterfeits and replays the recorded data to the server. The counterfeits are now in service while an equal number of legal ICs have been hijacked, and the counters of the hijacked pieces are now synchronized. The counterfeits are rejected in the next control. The attacker puts the hijacked, legal dielets back into service, after illegally benefiting from the cost of the replacement.

### 3.3 ReSC-1 Supply Chain Framework

In work (Yang, Forte, & Tehranipoor, 2015b), Yang *et al.* proposed an RFID-enabled supply chain (named ReSC-1) solution for the management and security of the entire supply chain and hop-transfers. Figure 3.3 shows the proposed ReSC-1 set-up, which has three stages and three transitions between stages. ReSC-1 consists of RFID readers, network devices with embedded RFID tags, locations of each reader, and a central database (DB) that records the information (e.g., control chip identities,

tag identities, tag traces, etc.) and authenticates the network devices. Each stage is described below:

i. Integration Stage: This is basically the production stage of the network device, where the ICs are produced and put together into a final product. The control chip identities (*CC_ID*) are generated using the physically unclonable function property of the embedded SRAM memory inside the chips. The embedded RFID tags are assigned tag identification numbers (*T_ID*) and recorded in the central database online together with the *CC_ID*. The recorded information is used to monitor and verify the device's journey through the insecure supply chain. Initializations are finalized and functionality tests are carried out to ensure proper hardware and software functioning.

ii. Distribution Stage: The temporary storage and transfer (transportation) of network devices are carried out by supply houses, distributors, retailers, and installers. Transportation activities are followed offline, in the ReSC-1 model. The devices are identified by authenticating the embedded tags using the readers at the hop points.

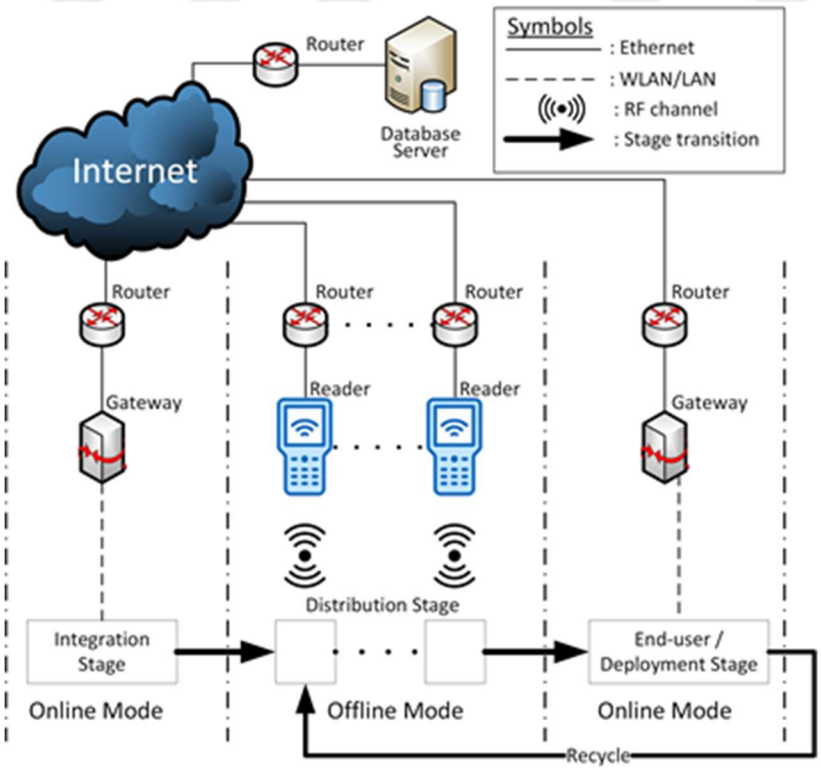



Figure 3.3 Supply chain stages and transitions of works (Yang et al., 2015a, 2015b, 2018)

Each hop reader adds its signature into the tag's memory, to contribute to the unique tag trace. The tag traces are recorded in the database when the readers get online. It is assumed that the database can authenticate readers based on their secret identities as it holds their ordered list.

iii. Deployment Stage: In the final stage, network devices are deployed at the end-user premises. The devices are powered on and connected to the central database over a secure link. Each device is authenticated online and the jointly created unique tag trace is validated. If the tag trace in the IoT device memory is not validated then that device cannot be put to service and it has to be sent for control and re-distribution.

### *3.3.1 Hardware Architecture of ReSC-1*

Figure 3.4 illustrates the hardware architecture of ReSC-1 and the other ReSC solutions that will be explained later, including the entities related and their connections. One central database records reader information (i.e., identity, location, index, public key, etc.), tag information (i.e., tag trace, identity, etc.), and device information (i.e., one-to-one mapping between tag and device, control chip identity, etc.). The database can authenticate readers based on their secret identities (which would not be transmitted in plaintext). The database can also authenticate the devices. It is assumed that the database could inform the readers of their previous readers for each tag in the supply chain. RFID readers are spreaded at distinct stages of the supply chain. Readers can together create a unique tag trace and record that trace in the tag memory in a secure fashion. The readers at distinct locations will write their own signatures to the tag memory to create a unique tag trace. If necessary, the RFID reader could communicate with the database via a secure Ethernet channel to record or retrieve the backup of the tag trace. Every RFID tag has a tag memory storing tag identity and tag trace. The RFID tag communicates with the RFID reader via an RF channel.

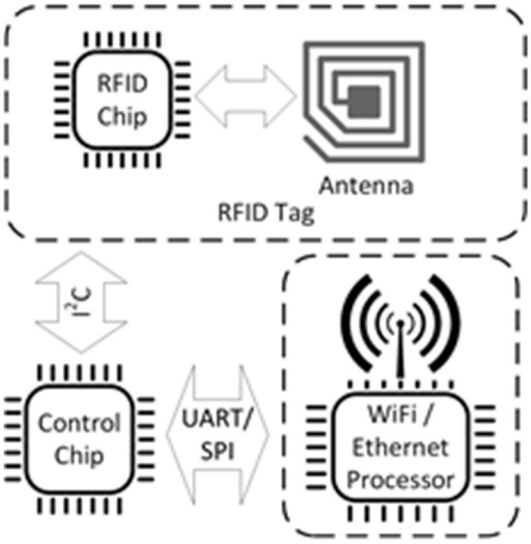



Figure 3.4 General hardware architecture of ReSC solutions

### *3.3.2 ReSC-1 Authentication Protocol*

The authentication procedure of ReSC-1 can be devided into two stages: (i) confirmation of a match between the control chip identity (*CC_ID*) and the tag identity (*T_ID*); and (ii) confirmation of the integrity of the tag trace to ensure that the network device has passed through the valid supply chain before reaching the end user.

#### *3.3.2.1 Tag-Device Matching*

*CC_ID* will be produced from the start-up signature (SRAM PUF) of embedded SRAM inside the control chip at the system integration stage. The tuple (*CC_ID, T_ID*) will be produced and recorded in the central database in online mode for device authentication in the future. The tag memory contains two parts: one unique, read-only tag identity and one unique tag trace produced from the signatures of readers on the distribution path of that tag. A tag trace can be described as valid when it contains all the required signatures of authorized readers on its distribution path.

### *3.3.2.2 Tag Trace Validation*

When the network device is deployed in the supply chain, RFID readers scattered in different locations will combine to create a unique tag trace and save this trace in the tag memory. The contents of tag memory include a static read-only tag identity and a unique tag trace created from the signatures of readers on the distribution path of the tag. Figure 3.5 illustrates the proposed lightweight RFID protocol in ReSC-1. The entire communication flow between RFID reader and RFID tag can be divided into three steps as shown in Figure 4.

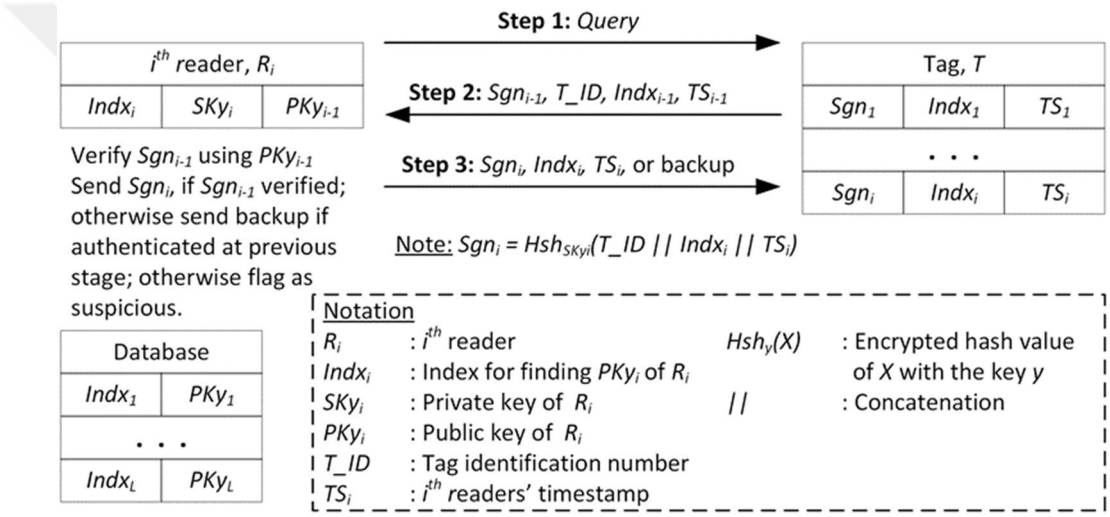



Figure 3.5 Lightweight RFID protocol used in ReSC-1

*Step 1:* At first, the reader $R_i$ interrogates tag $T$ at a hop.

*Step 2*: When the tag $T$ receives the *Query*, the tag $T$ will answer with $T\_ID$ indicating the tags' identity, $Indx_{i-1}$ indicating the previous stage number, $TS_{i-1}$ indicating the timestamp of the reader at the previous stage, and the signature $Sgn_{i-1}$ of the reader at the previous stage

$$Sgn_{i-1} = Hsh_{SKyi-1}(T\_ID \,||\, Indx_{i-1} \,||\, TS_{i-1}) \tag{3.2}$$

where $Hsh_{SKyi-1(X)}$ indicates the encrypted hash value of $X$ with the key $SKy_{i-1}$ of the reader $R_{i-1}$ and || indicates the concatenation operator.

*Step 3*: The reader $R_i$ decrypts $Sgn_{i-1}$ using the public key $PKy_{i-1}$ of the reader $R_{i-1,}$ and gets the hash value of $T\_ID$ || $Indx_{i-1}$ || $TS_{i-1}$. At the same time, the reader $R_i$ calculates the same hash value locally. The reader $R_i$ authenticates tag $T$ by comparing these two hash values. If the authentication is successful, the reader $R_i$ encrypts the hash value of $T\_ID$ || $Indx_i$ || $TS_i$ with its' private key $SKy_i$ to form its own signature $Sgn_i$ and sends $Sgn_i$, $Indx_i$, and $TS_i$ to the tag $T$ to update the tag trace.

### *3.3.3 Authentication of the Network Device at the End-User Stage*

The authentication procedure at the end-user stage includes authentication of the control chip by itself to the central database and validation of the tag trace kept in the memory of the tag attached to the network device. The service offered will be available for the network device after all the authentication procedures are passed successfully.

### *3.3.4 Assumptions and Vulnerabilities of ReSC-1*

The channel between the reader and the server is commonly assumed to be secure, in RFID community. In contrast, the channel between the reader and the tag is accepted to be insecure. The readers are known to be resourceful hardware that supports sophisticated cryptographic tools for encryption and decryption. However, rogue readers are assumed to be present in open environments (Peris-Lopez, Orfila, Castro, & Lubbe, 2011; Peris-Lopez, Orfila, Mitrokotsa, & Van der Lubbe, 2011). Because of these two reasons, attacks in RFID mostly target the readers and message exchanges between the readers and the tags. Briefly, in the formal Dolev-Yao model, adversaries are assumed to have the capability of listening, blocking, changing, and injecting messages into an exchange (Dolev & Yao, 1983). But, ReSC-1 protocol contradicts the formal model and common assumptions, by assuming a trustful integrator, reader operator, and secure readers (Yang, Forte, & Tehranipoor, 2015b).

As a result, although ReSC-1 is designed for providing the management and security of the vulnerable IoT device supply chain, it has serious security weaknesses that jeopardize the safety of the whole project.

The first weakness is the upload of pre-calculated session keys in the tag memories by the system integrator, and the database has no security precautions in case the initialization information is captured.

A second weakness is the lack of authentication or bonding of the reader or its operator with the server, at the hop point. Thus, the opportunity is opened to the use of a rogue reader or the use of a legal reader as an accessory by a malicious user. In either case, attacks exploiting these weaknesses are shown below. The results of the launched attacks are devastating to the security of the whole work.

A third vulnerability is in Step 2 of the protocol, in Figure 3.5. The signature $Sgn_{i-1}$, the tag $T\_ID$, the index $Indx_{i-1}$, and the timestamp $TS_{i-1}$ of the previous supply chain hop are responded to the reader in plaintext, in the present hop. Even identifying the hash function is not necessary, because the hash result is accompanied by the inputs. An attacker team can obtain all $Sgn_{i-1}$ and corresponding inputs of all tags and organize an attack with the help of the reader operators.

The last, simple but devastating vulnerability is at Step 3; where a rogue reader can send a false tag trace backup; or simply block the last message before the batch is sent to the next hop.

### *3.3.5 Launching Attacks on ReSC-1*

With an attacker model weaker than Dolev-Yao, the attacker can be assumed to have access to a common RFID reader or a legal ReSC-1 reader. Naturally, the attacker can be a malicious user too, who can have the reader's memory tampered with, to obtain the secrets inside. Furthermore, the hop numbers, locations, and the transportation path of the supply chains are public, in present-day trading. Therefore,

forming a malicious attacker organization by placing a rogue reader/operator at the hop locations can be assumed possible.

#### *3.3.5.1 Disrupting the Supply-Chain: Tag-Trace Contamination Attack on ReSC-1*

The simplest attack can be launched by blocking the last message of ReSC-1, thus disrupting the supply chain. When the message to the tags in Step 3 is blocked and the batch continues to the next hop, the trace records of the tags are missing a hop entry that nobody is aware of. Tampered trace-records result in the rejection of the delivered batch of devices. A slightly more sophisticated attack can take place when a rogue reader sends a false tag trace backup, in Step 3. The false $Sgn_i$ or backup to the tags contaminates the tag trace records and again results in the disruption of the supply chain.

#### *3.3.5.2 Tag Cloning by Hop Table Attack on ReSC-1*

Attackers placed at the hop locations can query all the RFID tags associated with the network devices with the hop's legal reader. The messages at the second and third steps can be recorded at every hop. The tag trace shown in Figure 3.5 is completed at the last hop. Thus, the attacker ring has all the entries in the tag memory of all genuine RFID tags. Thus, the valid signature chain of the devices can be copied to a set of fake RFID tag memories. The fake tags only need a legal *CC_ ID* to become a clone of the legal tag. But, the *CC_IDs* of the network devices are produced from the start-up signature of the embedded static random access memory (SRAM) inside the control chips, at the integration stage. SRAM signatures are extensively offered as physically unclonable function (PUFs) security primitives for generating IDs (Guin, Huang, DiMase, Tehranipoor, & Makris, 2014; Peris-Lopez, Orfila, Mitrokotsa, & Van der Lubbe, 2011; Rostami, Majzoobi, Koushanfar, Wallach, & Devadas, 2014; Rührmair, Devadas, & Koushanfar, 2012). Although PUFs can serve as a private fingerprint for each IC, they are susceptible to side-channel attacks (Rostami, Farinaz, & Karri, 2014).

ReSC-1 authors assume that the integration stage occurs in a trusted environment. This exaggerated assumption can be exploited by the above malicious ring and a rogue employee can obtain the *CC_ IDs* of the network devices while they are at the integration stage. Hence, a pirate network device with a cloned RFID tag that has a valid tag trace and *CC_ ID* can pass the authentication process at the End-User Stage of Figure 3.3. The legal devices are now cloned for malicious trading.

## 3.4 ReSC-2 and ReSC-3 Supply Chain Framework

The ReSC-1 proposal was upgraded in two later RFID-enabled solutions by the authors, which will be named ReSC-2 and ReSC-3, respectively (Yang, Forte, & Tehranipoor, 2015a, 2018). The works aimed at protecting the IoT devices and their supply chain, using the same supply chain framework shown in Figure 3.3.

### *3.4.1 Hardware Architecture of ReSC-2 and ReSC-3*

Even though the ReSC-3 hardware architecture specified in the work (Yang, Forte, & Tehranipoor, 2018) seems to have an additional wireless network processor, the hardware architectures of ReSC-1 and ReSC-2 specified in the work (Yang, Forte, & Tehranipoor, 2015a) and work (Yang, Forte, & Tehranipoor, 2015b) have also wired or wireless network processors. Because it is obvious that network devices or IoT devices have wired or wireless network processors communicating with the control chip. Thus, all the ReSC solutions mentioned in this work use the same general hardware architecture shown in Figure 3.4.

### *3.4.2 ReSC-2 and ReSC-3 Authentication Protocols*

The authentication schemes of ReSC-2 and ReSC-3 consist of the same number of steps and phases as in ReSC-1. In addition, ReSC-3 has another procedure named Neighborhood Attestation, which aims to detect theft at the end-user or device recycling locations, by verifying the status of its neighbors. The functionality of this

procedure is out of the scope of this work because there are more serious vulnerabilities even before it reaches the end-user. The authentication protocol between the device tags and the readers is shown in Figure 3.6. Before entering the supply chain, the system integrator loads a set of session keys ($SnKy_1$, $SnKy_2$, ..., $SnKy_n$) generated by encrypting the tag identity, using the distribution path RFID reader keys $Ky_{Ri}$, shared between the central database and $n$ readers. Each session key ($SnKy_i$) is used to encrypt the communication between the tag and one specific reader ($R_i$). *SE* designates symmetric-key encryption algorithm, AES in ReSC-2.

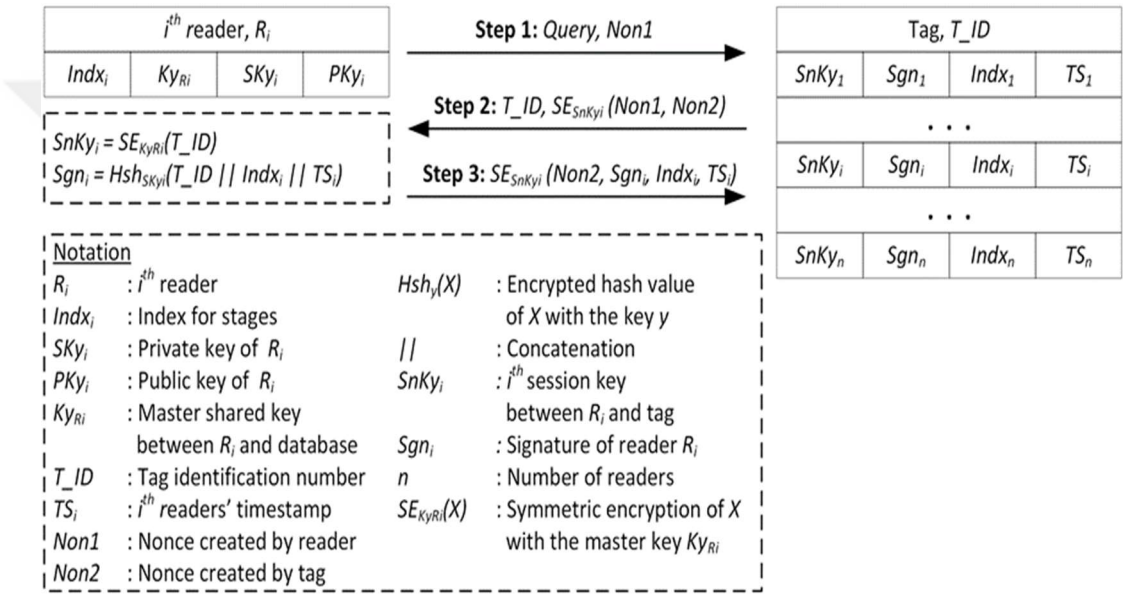



Figure 3.6 Lightweight RFID protocol used in ReSC-2 and ReSC-3

Briefly, the goal is a tag trace in the tag memory validated by the signatures of the authorized readers, on the distribution path. The entire communication flow between the RFID reader and tag can be divided into the following three steps:

*Step 1*: When the IoT device arrives at an intermediate stage, the hop reader $R_i$ first issues a *Query* command to the tag, together with a random number *Non1*.

*Step 2*: After receiving the *Query* command and the random number *Non1*, the tag encrypts the tuple (*Non1*, *Non2*) using the session key $SnKy_i$ and replies with its

identity (*T_ID*) and the ciphertext $SE_{SnKyi}(Non1, Non2)$, where *Non2* is a new random number generated by the tag.

*Step 3*: The reader $R_i$ will generate the session key $SnKy_i$ locally by encrypting the tag identity (*T_ID*) using its master key $Ky_{Ri}$. The reader $R_i$ authenticates the tag by decrypting $SE_{SnKyi}(Non1, Non2)$ and validating *Non1*. If the tag passes the authentication, the reader $R_i$ generates its signature as shown in Figure 3.6,

where $Indx_i$ is the index associated with the $i^{th}$ reader, $TS_i$ denotes the specific time when reader $R_i$ updates the tag, || indicates the concatenation operation, and $Hsh_{SKyi}(X)$ is the hash value of the input *X* using $SKy_i$, the private key of reader $R_i$. Next, the reader $R_i$ encrypts the quad as $SE_{SnKyi}(Non2, Sgn_i, Indx_i, TS_i)$, using the session key $SnKy_i$, and sends it to update the tag trace. The tag authenticates the reader $R_i$ by decrypting $SE_{SnKyi}(Non2, Sgn_i, Indx_i, TS_i)$ and validating *Non2*. If the reader $R_i$ is authenticated, the tag records the reader update ($Sgn_i, Indx_i, TS_i$) in the tag memory.

### *3.4.3 Assumptions and Vulnerabilities of ReSC-2 and ReSC-3*

The same assumptions of ReSC-1 apply to ReSC-2 and ReSC-3. Secure reader and reader operator are still the most notable assumptions. The ReSC-2 and ReSC-3 are improved versions of ReSC-1, but still, security vulnerabilities exist. The basic vulnerability is again the upload of pre-calculated session keys in the tag memories, by the assumedly trusted system integrator, during the integration stage (Yang, Forte, & Tehranipoor, 2015a, 2018). The second is again the lack of authentication and bonding between a legal reader, its operator, and the central database. Thus, the malicious operators and rogue readers have an avenue for access to the secrets inside the readers. A third vulnerability is plaintext *T_ID* and *Non1* in Steps 1 and 2, which are not needed in our attacks but can be used in others. The fourth vulnerability is the unacknowledged message in Step 3, as in ReSC-1.

### *3.4.4 Launching Attacks on ReSC-2 and ReSC-3*

With the same attacker model in ReSC-1, the attacker is assumed to have access to a common or a legal ReSC-n RFID reader. The attacker is assumed to be able to have access to the secrets in the reader's memory. Furthermore, the details of the supply chain hops are public. Therefore, forming a malicious organization by placing a rogue reader/operator at each hop is assumed as possible.

#### *3.4.4.1 Disrupting the Supply-Chain: Tag-Trace Contamination Attack on ReSC-2 and ReSC-3*

The same attack can be launched on ReSC-2 and ReSC-3, without any changes in the methods. Therefore, the supply chain of all ReSC solutions is disruptable. The formalized step by step procedure of the launched attack is below:

i. Fabricate a constant *Non1*,
ii. Query all tags with any reader using *Non1*,
iii. After receiving the replies in Step 2 (Figure 3.6), play a fabricated reply to all tags,
iv. Repeat the above three steps infinite times with an automated piece of software.

At the end of the attack, any tag that has a *Non2* match, will update its memory with fake information and increment its address pointer. Now, the tag has a corrupted tag trace record at the end of its journey that nobody is aware of. The above attack can be launched before or after the tags are queried by a legal reader; or even while in transit. Repeating the attack at many hops increases the success rate of the tag-trace contamination.

#### *3.4.4.2 Tag Cloning by Full Disclosure Attack on ReSC-2 and ReSC-3*

The exaggerated assumption of a secure system integrator facilities fails when a rogue employee can obtain the *CC_ID*, session keys $SnKy_i$, and *T_ID* of the corresponding IoT devices. Thus, the rogue employee may be a part of the attacker ring or may sell the information to a ring. In fact, there are two ways of obtaining the critical tag session keys. All secrets except the *CC_ID* can also be captured by tampering with the reader's memories, as well. After capturing the session keys of the corresponding tags, the attackers can find out all the secret information of the IoT device tag traces. Now launching an attack is trivial. Since the attacker has either the reader key $Ky_{Ri}$ or the session key $SnKy_i$, the *Non2* sent in Step 2 is captured. The message in Step 3 is formed. At the last hop, the procedure to capture the full tag trace is complete. Thus the attacker ring has all the entries in the tag memory of all genuine RFID tags. Now, the attacker ring can introduce a fake network device with a cloned RFID tag and a valid *CC_ ID*. The fake network device can pass the authentication process at the End-User Stage. Hence, pirate network devices with cloned RFID tags with a valid tag trace record and *CC_ ID* can be exchanged with legal devices for malicious trading.

## 3.5 CDTA Supply Chain Framework

Yang et al. presented yet another RFID-enabled solution named CDTA in work (Yang, Forte, & Tehranipoor, 2017), which uses the same supply chain framework in previous ReSC works (Figure 3.3). CDTA aims to detect all counterfeit chips used in IoT devices with a new hardware architecture. At least one sensor is embedded inside the chips to send data periodically to the RFID chip, to be recorded in the EEPROM of the device. Because of its common creators and the same supply chain framework, it would be appropriate to put CDTA in the ReSC family.

### *3.5.1 Hardware Architecture of CDTA*

Yang *et al.* propose a new generic hardware infrastructure for IoT devices to combat counterfeiting and enhance supply chain management in the CDTA solution. The general hardware architecture proposed in the CDTA solution is shown in Figure 3.7. As seen in Figure 3.7, it is proposed that there will be at least one sensor/primitive (S1, S2, …, Sn/Sm/Sp) embedded inside the chips (including RFID chip) building the IoT device. These sensors send their data to the RFID chip periodically to be recorded in the EEPROM (Electronically Erasable Programmable Read-Only Memory) of the chip when the device is powered on.

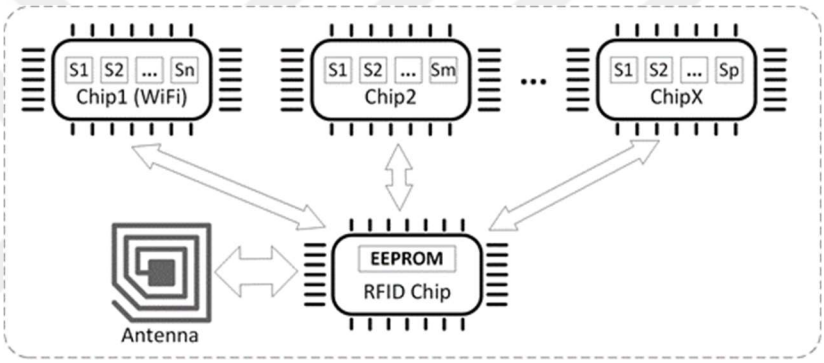



Figure 3.7 General hardware architecture of CDTA

Four types of sensors are frequently referred to in the literature (Suh & Devadas, 2007; Zhang & Tehranipoor, 2014; Zhang, Tuzzio, & Tehranipoor, 2012; Guin, Zhang, Forte, & Tehranipoor, 2014; Guin, Forte, & Tehranipoor, 2016) to handle specific types of counterfeits, and a novel board ID generator are used in the CDTA solution. The type and functionality of each sensor are listed in Table 1.

The board ID in the CDTA solution is generated by combining each individual chip's PUF with a special function. Therefore, each individual chip should have at least an embedded PUF sensor/primitive. Detailed functionality of sensors and Board ID generation process is out of the scope of this article and will not be mentioned here.

Table 3.1 Types and functionality of CDTA sensors

| Type of Sensor | Functionality |
|---|---|
| CDIR_CTR (Counter-based Combating Die/IC Recycling) | Tracking of chip usage time (fine granularity) and detection of recycled ICs |
| CDIR_RO (Ring Oscillator-based CDIR) | Tracking of chip usage time (coarse granularity) and detection of recycled ICs |
| PUF (Physical unclonable function) | ● Unique identification of silicon chips<br>● Detection of overproduced/cloned ICs |
| ECID (Electronic Chip ID) | Protection of the chip against over grading/remarking |
| Board ID | ● Identification and tracking of boards in the supply chain<br>● Detection of board cloning and overproduction<br>● Detection of unauthorized substitution of chips on the board |

### *3.5.2 CDTA Authentication Protocol*

Figure 3.8 shows the lightweight authentication protocol of CDTA. The tag memory has a static, read-only *T_ID*, a set of session keys ($SnKy_1, SnKy_2, \ , SnKy_n$) where *n* is the number of readers on the supply chain, and the collected sensor data. Before entering the supply chain, the system integrator assigns the session keys to the tag memory of the IoT device. Each session key ($SnKy_i$) is used to encrypt the communication between the tag and one specific reader ($R_i$). The session key ($SnKy_i$) is just the encryption of the plaintext *T_ID*, as in $SnKy_i = AES_{KyRi}(T\_ID)$, where $Ky_{Ri}$ is the master key shared between $R_i$ and the database. The communication between the RFID reader and tag is as follows:

*Step 1*: When the IoT device arrives at a transfer hop, that hop's reader $R_i$ issues a *Query* to the tag along with a random number *Non1*.

*Step 2*: After receiving the *Query* and *Non1*, the tag replies with its identity (*T_ID*) and its own random number *Non2*.

*Step 3*: After receiving the 2-tuple (*T_ID*, *Non2*), the reader $R_i$ will first generate the session key $SnKy_i$ locally by encrypting the tag identity (*T_ID*) using its master key $Ky_{Ri}$. Afterward, the reader $R_i$ will encrypt *Non2* using the session key $SnKy_i$ and send the ciphertext $AES_{SnKyi}(Non2)$ to the tag to authenticate itself.

*Step 4*: The tag authenticates the reader $R_i$ by decrypting $AES_{SnKyi}(Non2)$ and validating *Non2*. If the reader $R_i$ passes the authentication, the tag will encrypt *Non1* || *SDTA* using the session key $SnKy_i$ and send the ciphertext $AES_{SnKyi}(Non1 \,||\, SDTA)$ to the reader $R_i$, where *SDTA* stands for CDTA sensor data and || indicates the concatenation operation.

*Step 5:* Similarly, the reader $R_i$ authenticates the tag by decrypting $AES_{SnKyi}(Non1 \,||\, SDTA)$ and validating *Non1*. The reader $R_i$ will trust CDTA sensor data if the tag passes the authentication.

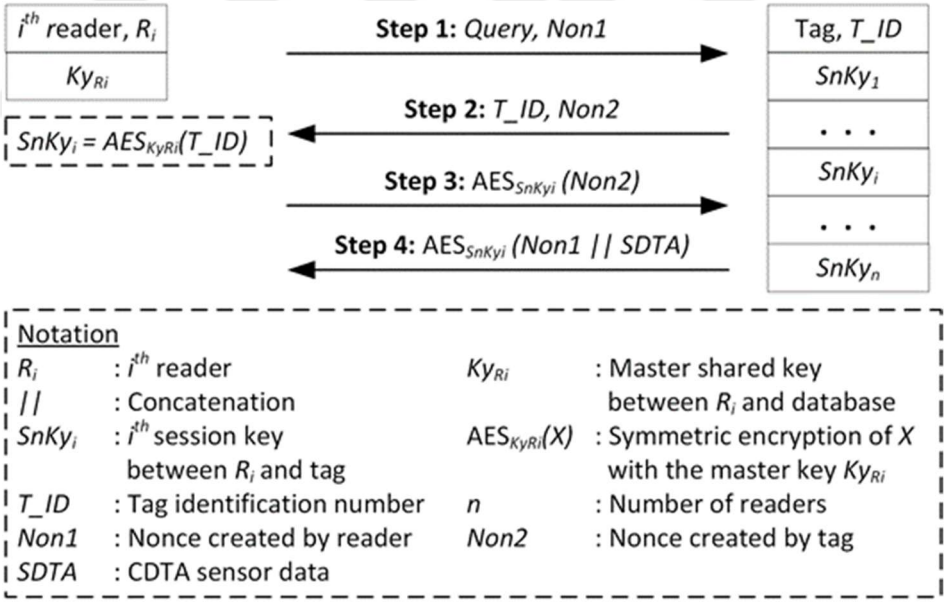



Figure 3.8 Lightweight RFID protocol used in CDTA

### *3.5.3 Assumptions and Vulnerabilities of CDTA*

The same assumptions of the ReSc-n family apply to CDTA, too (Yang, Forte, & Tehranipoor, 2017). The serious weakness of no authentication or bonding between a

legal reader and its legal user is carried to the CDTA, as well. Thus, the same vulnerabilities exist. The devastating secure system integrator assumption results in the same additional vulnerabilities because the existence of malicious users in system integrator facilities is considered to be possible.

Another vulnerability is handed to attackers at Steps 1 and 2 of the protocol when both *Non1* and *Non2* of the peers are transmitted in plaintext, which allows plaintext ─ ciphertext matching.

The last vulnerability is the data recorded offline on the reader, which can be captured through memory tampering. However, disrupting the CDTA supply chain with the previous ReSC-n methods is not possible, because the tag has a second reply as a fourth step.

### *3.5.4 Tag Cloning by Full Disclosure Attack on CDTA*

Due to the vulnerabilities explained above, the session keys $SnKy_i$ and *T_ID* of the devices are vulnerable to theft. Thus, capturing the critical information makes the monitoring of the supply chain easy, as in the attack scenario on ReSC-2 and ReSC-3. Hence, the attacker knows the $SnKy_i$ with the corresponding *T_ID* and index number $Indx_i$. Now an attack can be carried out with a rogue reader. The rogue reader queries the target RFID tag with any *Non1*. The tag responses with *T_ID* and *Non2* tuple in plaintext. The rogue reader encrypts valid *Non2* with the compromised session key $SnKy_i$ and transmits the encrypted reply to the tag, in Step 3. After decrypting the received message, the RFID tag encrypts *Non1* and the sensor data *SDTA* with the session key $SnKy_i$ and transmits the encrypted message to the rogue reader. After decrypting the message, the rogue reader learns the critical sensor data. Thus the attacker ring has all the information necessary to introduce fake network devices with cloned RFID tags and valid *CC_ IDs*. The fake network devices can pass the authentication process at the End-User Stage because the valid SDTA has been captured.

# CHAPTER 4
# MATERIALS AND METHODS

## 4.1 Formal Verification Tools for Security Protocols

Security protocol verification by using validation tools has become popular, in recent years. AVISPA and Scyther are the most widely used formal analysis tools for security protocols (Armando et al., 2005; Clarke et al., 2018; Cremers, 2008). In the next sub-sections, AVISPA and Scyther Tools will be explained.

### *4.1.1 AVISPA Formal Analysis Tool*

Automated Validation of Internet Security Protocols and Applications (AVISPA) is a popular authentication protocols verification tool with its security protocol animator (SPAN) (Armando et al., 2005; Clarke et al., 2018; Cremers, 2008). The structure of the AVISPA Tool is shown in Figure 4.1

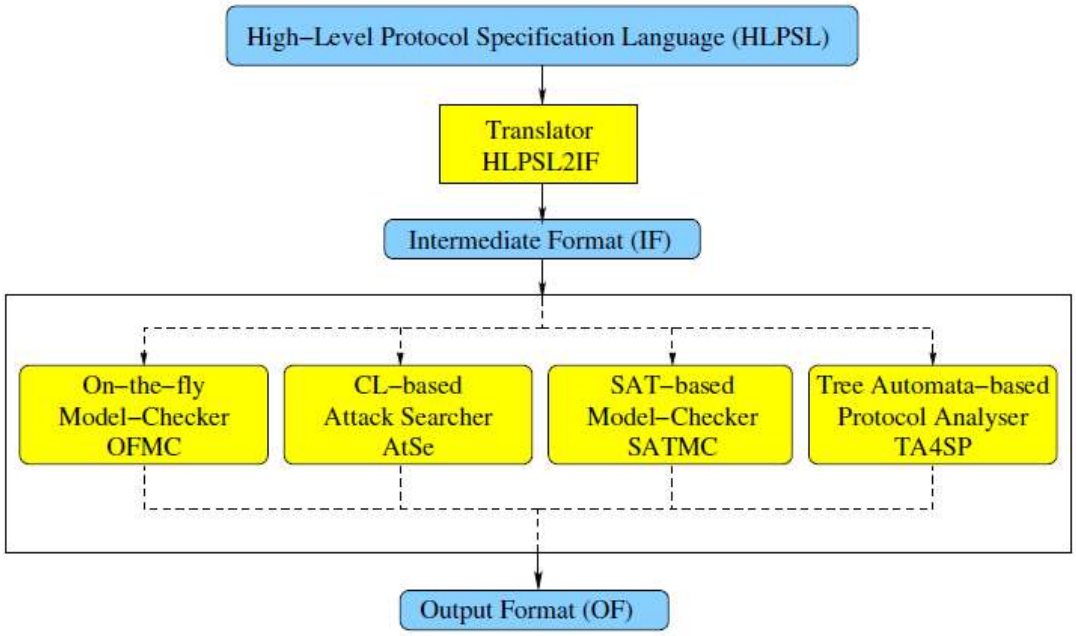



Figure 4.1 Architecture of the AVISPA Tool

SPAN provides a graphical interface for testing authentication protocols encoded using high-level protocol specification language (HLPSL) to specify protocols and their properties. Protocols are defined in HLPSL with regard to their roles, using control flow patterns, data structures, and alternative adversary models, together with different cryptographic primitives and their algebraic properties. HLPSL specifications have declarative semantics based on Lamport's Temporal Logic of Actions (Lamport, 1994) and an operational semantics defined together with a rewrite-based formalism called the intermediate format (IF). Different model checking back-ends interpret the IF and can be used for verification or falsification (Clarke et al., 2018).

SPAN Tool (AVISPA) has four back-ends:

i. On-the-fly model checker (OFMC),
ii. Constraint-logic-based attack searcher (CL-AtSe),
iii. Boolean satisfiability (SAT)-based model checker (SATMC),
iv. Tree automata-based protocol analyzer (TA4SP).

OFMC and CL-AtSe checkers are more popular than SATMC and TA4SP. OFMC checks the submitted protocol and warns when it finds an attack against the protocol. Various studies are using AVISPA (Sadhukhan, Ray, Biswas, Khan & Dasgupta, 2021; Thakur et al., 2021; Mohanta, Jena & Patnaik, 2021; Zargar, Shahidinejad & Ghobaei-Arani, 2021) for the verification of authentication protocols.

### *4.1.2 Scyther Formal Analysis Tool*

Scyther is a powerful popular tool for the verification of security protocols against known vulnerabilities (Cremers, 2008). The tool has been used in many works in the formal analysis of proposed authentication protocols (Adeli, Bagheri, & Meimani, 2021; Azrour, Mabrouki, Guezzaz, & Farhaoui, 2021; Safkhani, Camara, Lopez, & Bagheri, 2021). The tool automatically analyzes submitted schemes, carefully observing the proposed scheme's reaction to the known attacks.

## 4.2 SBMAPS: The Proposed Authentication Protocol for IC Supply Chain

The proposed method in the novel protocol (Özcanhan, & Türksönmez, 2020) involves three phases:

i. Bonding a legal user with a legal smartphone,

ii. The mutual authentication of the dielet and the server, with the creation of a secure session key,

iii. Dielet status tracking, by checking sensor status.

The novel protocol is Strongly Bonded and Mutually Authenticated Protocol for SHIELD, or briefly SBMAPS. The secret keys of each dielet are enumerated and transferred to the manufacturer through a secure channel. Similarly, the IDs of the produced dielets are transferred to the IP owner, in an ordered list. The manufacturer must insert the keys into the dielets, in the ID list's order given to the IP owner. The tuples (ID, keys) are loaded into the server memory. The generation of dielet identification numbers by either using PUFs or TRNGs is preserved in our work. Therefore, dielet ID generation is omitted in our work. The assumptions of SBMAPS are more realistic than CNTR-SHIELD's and require stronger countermeasures. Our assumptions are:

i. The air channels between the dielet and smartphone; smartphone and server are both insecure,

ii. Neither the smartphone nor its operator can be trusted,

iii. Physically tampering with the dielet and capturing the memory contents without disturbing the sensors is not possible (indispensable criterion of SHIELD),

iv. Valid *usr_ID*, *usr_key* pairs for every hop are previously recorded in the server database,

v. Valid smartphone *cpu_IDs* for every hop in the supply chain are previously enrolled to the server database.

### *4.2.1 Phases of SBMAPS*

#### *4.2.1.1 Phase 1 (User and Smartphone Registration and Bonding)*

The protocol starts by bonding an authentic user with a valid smartphone, as shown in Figure 4.2. The dielet has no role in Phase 1. The $k^{th}$ user has to enter his/her username $usr\_ID_k$ and password in the custom-made application, on a legal smartphone. The password is padded and encrypted with software like Linux's "crypt" command to form the $usr\_key_k$. Using the key, the application encrypts the $cpu\_ID_{Ri}$ of the smartphone together with the system time $t_{Ri}$ to form message $M_1$ and sends message $M_1$ and $usr\_ID_k$ to the server, through an insecure air medium. We will not go into the concatenation and padding before encryption, as they are encryption-specific operations. Using the login name $usr\_ID_k$ the server finds the user's key $usr\_key_k$ in its database and decrypts message $M_1$. The server searches its database to see whether there is an entry that matches the decrypted $cpu\_ID_{Ri}$. After verifying that the smartphone is a valid device, the specific user is paired with it, which we define as user-smartphone bonding (Özcanhan, & Türksönmez, 2020).

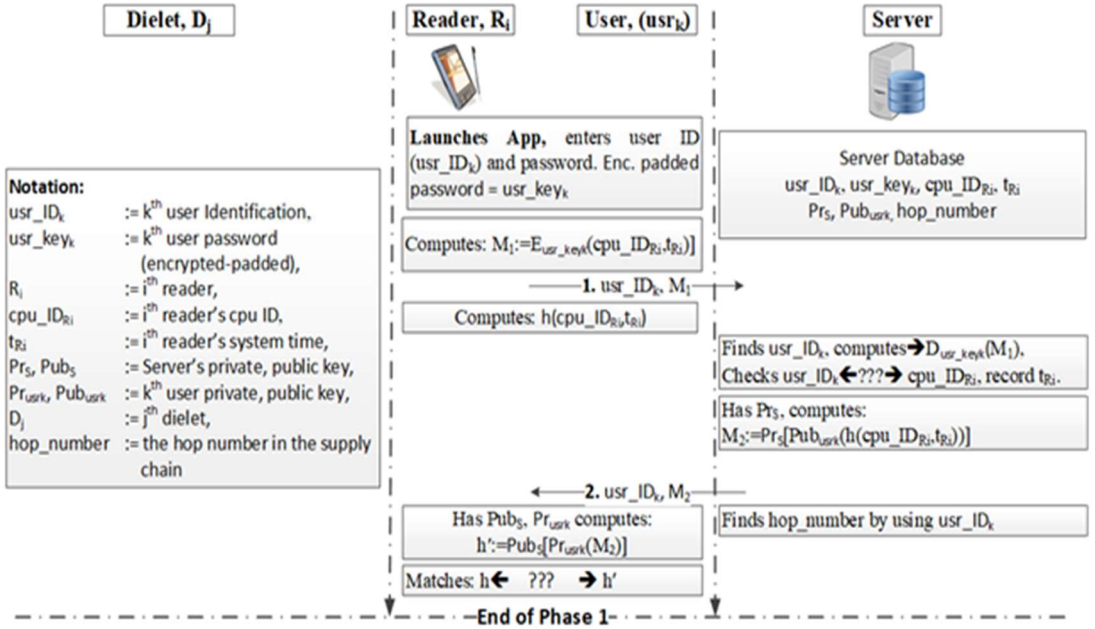



Figure 4.2 Phase 1 of SBMAPS "The bonding of a legal smartphone with an authenticated user"

The smartphone's system time $t_{Ri}$ is recorded. Next, the server calculates the hash of the tuple ($cpu\_ID_{Ri}$, $t_{Ri}$), which proves that it has the user key to decrypt message $M_1$. After encrypting the hash value with the user's public key and then signing the encrypted value with its private key (message $M_2$ is formed), the server sends message $M_2$ to the smartphone (Özcanhan, & Türksönmez, 2020).

While waiting, the smartphone calculates the same hash. After receiving $M_2$, the smartphone can decrypt $M_2$ and obtain the server's version of the hash, if it is an authorized device; i.e. if it has the server's public and the user's private key. If the calculated and unraveled hash values match, the server has authenticated the user and bonded the user name with the smartphone, in hand. Hence, the mutual authentication of the user and server, and the bonding of the user with the smartphone are finished. Meanwhile, using the user name, the server finds the hop number and decides which key number (*Key_No*) should be used, in the next phases of SBMAPS (Özcanhan, & Türksönmez, 2020).

#### *4.2.1.2 Phase 2 (the Mutual Authentication of the Dielet and the Server, Creation of Session Key)*

The bonded smartphone requests the identification of the targeted dielet, as shown in Figure 4.3. From then on, the smartphone is merely a repeater. The dielet sends its identification number *Dj_ID* to the server. The server finds the dielet in its database and determines the *Key_No* and the key $key\mathbf{x}_{Dj}$, by using the *hop_number* decided in Phase 1. After generating the random number $Non_S$, it is encrypted with the chosen key$\mathbf{x}_{Dj}$ key and sent to the dielet, with a prefix *Key_No* indicating the key used. The dielet finds the key pointed by the server and uses it to decrypt message $M_3$. The value $temp_0$ now contains the server's nonce. Next, the dielet prepares message $M_4$ to show the server that it can decrypt server messages. Then, the dielet generates its nonce $Non_{Dj}$ and hides it in message $M_5$. After receiving $M_4$ and $M_5$ from the dielet, the server decrypts $M_4$ and XORs it with $M_3$. The obtained value is matched with the generated $Non_S$. A match authenticates the dielet. Next, $M_5$ is decrypted and XORed with $M_4$. The obtained $temp_3$ value is the nonce of the dielet. The server computes $M_6$ to prove

to the dielet that it can decrypt messages sent by the dielet. After sending $M_6$, the server calculates the session key *key**S*** by XORing the generated and received nonces. The dielet decrypts $M_6$ and XORs it with the previously received $M_5$. If the obtained value matches its generated nonce $Non_{Dj}$, then the server is authenticated. As the last step, the dielet calculates its own version of the session *key**S***. If Phase 2 is stopped at any step, the mutual authentication has to be started from the beginning.

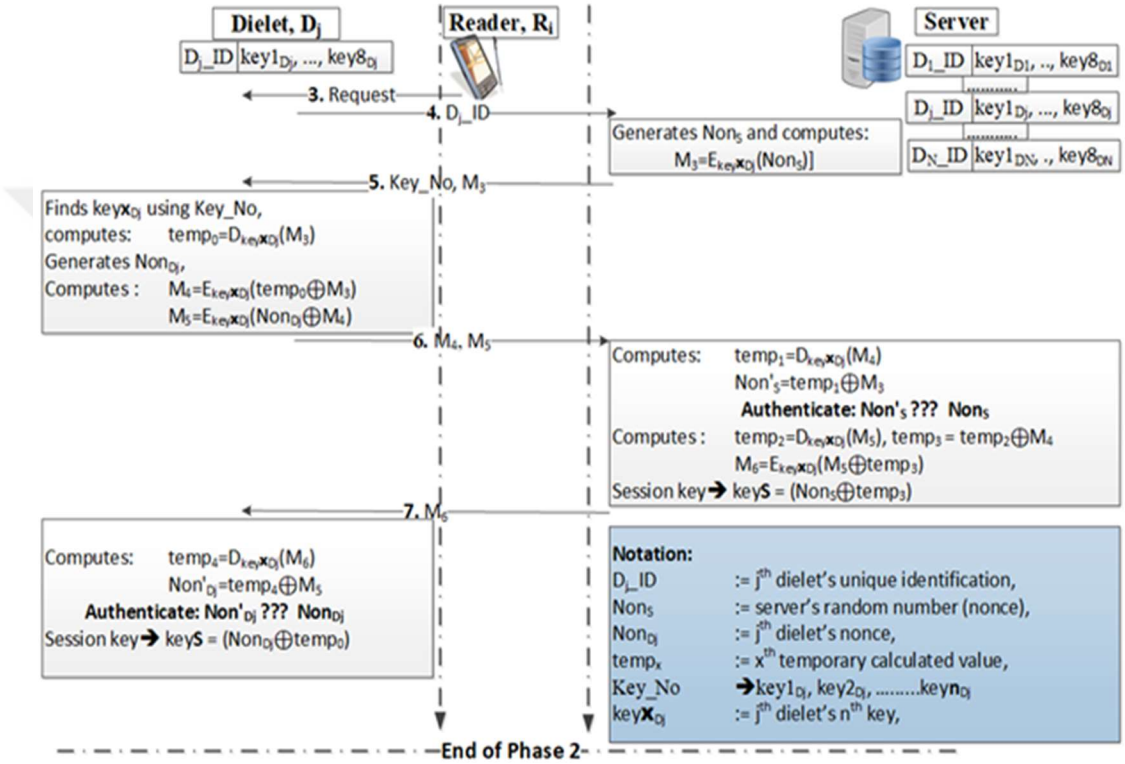



Figure 4.3 Phase 2 of SBMAPS "Dielet-Server mutual authentication and session key generation"

#### *4.2.1.3 Phase 3 (Dielet Tracking by Sensor Status Checking)*

In the last phase, the server encrypts its system time with the session key and sends it to the dielet (Figure 4.4). The dielet decrypts the received message $M_7$, and extracts and checks $Non_{Di}$. Then obtains and saves $t_S$ to a temporary location, i.e. another register. Then, $t_S$ is XORed with the old-time *old_t*$_S$ found in the IC's nonvolatile memory to form $M_8$. Before the dielet's first registration, initially, *old_t*$_S$ can be assumed to be or ANDed with zero. Next, the dielet sensors are read and their status bits are XORed with $t_S$ and then encrypted with the session key to form $M_9$. Before

sending messages $M_8$ and $M_9$, the new time $t_S$ is over-written on $old\_t_S$ in IC's secure memory, dedicated to saving dielet's time of access. After receiving the messages, the server decrypts $M_8$ to check if the dielet knows the session key and has the correct $old\_t_S$ value. After the verification, the server decrypts $M_9$ to obtain the sensor status bits by XORing $M_8$ with $temp_8$. Then, the sensor status bits are checked. If the sensors are not reporting foul play, the time from the start of Phase 1 ($t_{Ri}$) to the end of Phase 3 ($t_{Scheck}$) is checked to verify that the protocol has ended within an acceptable period of time. Next, the server increments the *hop_number* and saves the tuple (*hop_number, $t_S$*). The server uses the accumulated tuples to track the dielet's movement, in the supply chain. The server finally prepares $M_{10}$ and sends it to the smartphone, as a notification of the result of the overall protocol. $M_{10}$ can only be opened up by the smartphone and the result is found by matching the calculated hashes and the received hash. At every session, the result is a variable, obscured, and double encrypted value. Optionally, the server and the dielet can update (change) the $D_j\_ID$, with the generated session key.

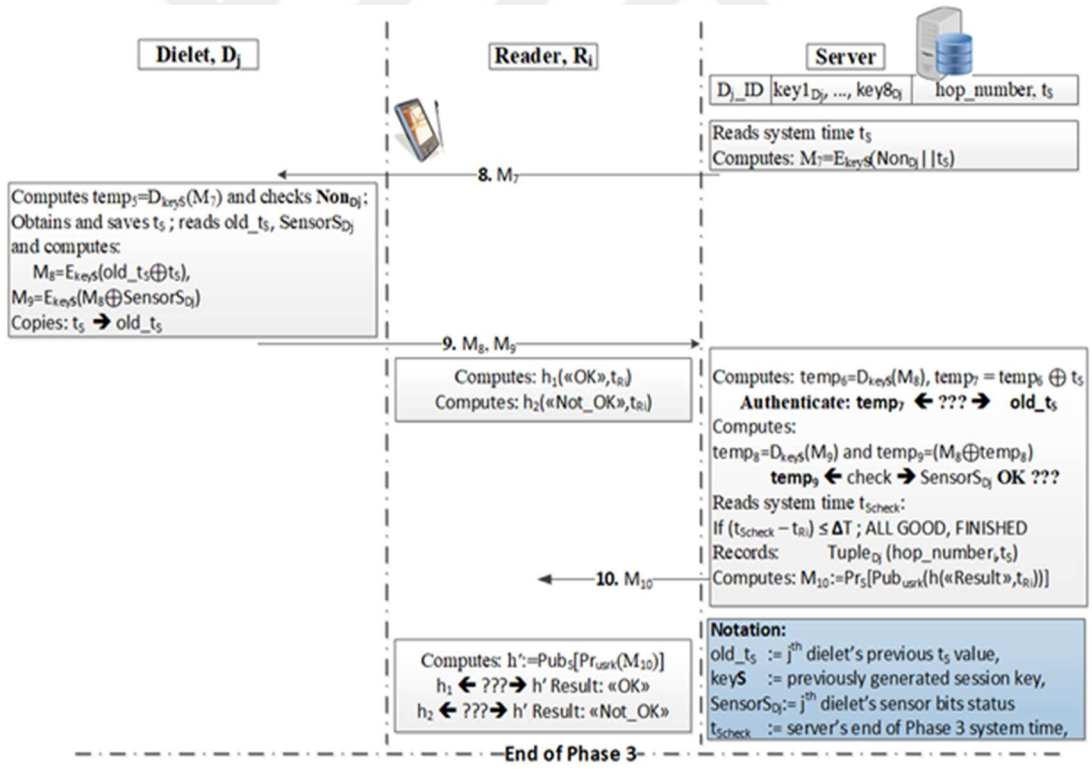



Figure 4.4 Phase 3 of SBMAPS "Dielet tracking by checking sensor status bits and counting hops, in the supply chain"

We will not go further into the equations needed for ID update, as it will merely prolong the explanations. The dielet can update its $D_j\_ID$, before sending messages $M_8$, $M_9$, and the server before sending $M_{10}$. After sending $M_{10}$ another authentication attempt at the same hop -for the same dielet- will be dropped by the server; because a successful authentication has been completed and reported to the reader (Figure 4.4).

## 4.3 STRAP: The Proposed Authentication Protocol for IoT Supply Chain

In this novel solution, a smartphone equipped with a near field communication (NFC) reader is used as the mediator between the NFC tag of the IoT device and the server. The remote server contains all cryptographic keys and the tag *T_ID* of each IoT or network device. The tag-smartphone communication channel is a near field wireless network; while the smartphone-server connection is over the Internet. Thus, all the hops on the supply chain are online mode.

The proposed method in the novel protocol involves three phases:

i. Bonding a legal user with a legal smartphone,

ii. The mutual authentication of the tag and the server, with the creation of a secure session key,

iii. IoT device status tracking, by checking signature chain and *SDTA* status (only in CDTA).

The novel protocol is **ST**rong **R**FID **A**uthentication **P**rotocol for IoT supply chain, or briefly STRAP. The secret keys of each tag are enumerated and transferred to the manufacturer through a secure channel. Similarly, the *T_ID* of the produced tags is transferred to the intellectual property (IP) owner, in an ordered list. The manufacturer must insert the keys into the tags, in the *T_ID* list's order given to the IP owner. The tuples (*T_ID*, key) are loaded into the server memory. The generation of keys and tag identification numbers are preserved in our work and therefore omitted. The assumptions of STRAP are more realistic than ReSC family:

i. The air channels between the RFID or NFC tags and smartphones are insecure,

ii. Neither the smartphone nor its operator can be trusted,

iii. Valid user identification and user key (*UID*, *UKey*) pairs are previously generated and recorded in the server database,

iv. Valid smartphone *CPUIDs* are previously enrolled in the server database.

### *4.3.1 Phases of STRAP*

#### *4.3.1.1 Phase 1 (User and Smartphone Registration and Bonding)*

The protocol starts by bonding an authentic user with a valid smartphone, as shown in Figure 4.5. The RFID tag has no role in Phase 1. The $k^{th}$ user has to enter his/her username $UID_k$ and password in the custom-made application, on a legal smartphone. The password is padded and encrypted with software like Linux's "crypt" command to form the $UKey_k$. Using the key, the application encrypts the $CPUID_{Spi}$ of the smartphone together with the system time $t_{Spi}$ to form and send the message $Msg_1$ and $UID_k$ to the server, through the insecure air medium. We will not go into the encryption-specific operations of concatenation and padding.

Using the login name $UID_k$ the server finds the user's key $UKey_k$ in its database and decrypts the message $Msg_1$. The server searches its database to see whether there is a $CPUID_{Spi}$. After verifying that the smartphone is a valid device, the specific user is paired with it, which we define as user-smartphone bonding. The smartphone's system time $t_{Spi}$ is recorded. Next, the server calculates the hash of the tuple ($CPUID_{Spi}$, $t_{Spi}$), which proves that it has the user key to decrypt message $Msg_1$. After encrypting the hash value with the user's public key and then signing the encrypted value with its private key (message $Msg_2$ is formed), the server sends message $Msg_2$ to the smartphone. While waiting, the smartphone calculates the same hash. After receiving $Msg_2$, the smartphone can decrypt $Msg_2$ and obtain the server's version of the hash, if it is an authorized device; i.e. if it has the server's public and the user's private key.

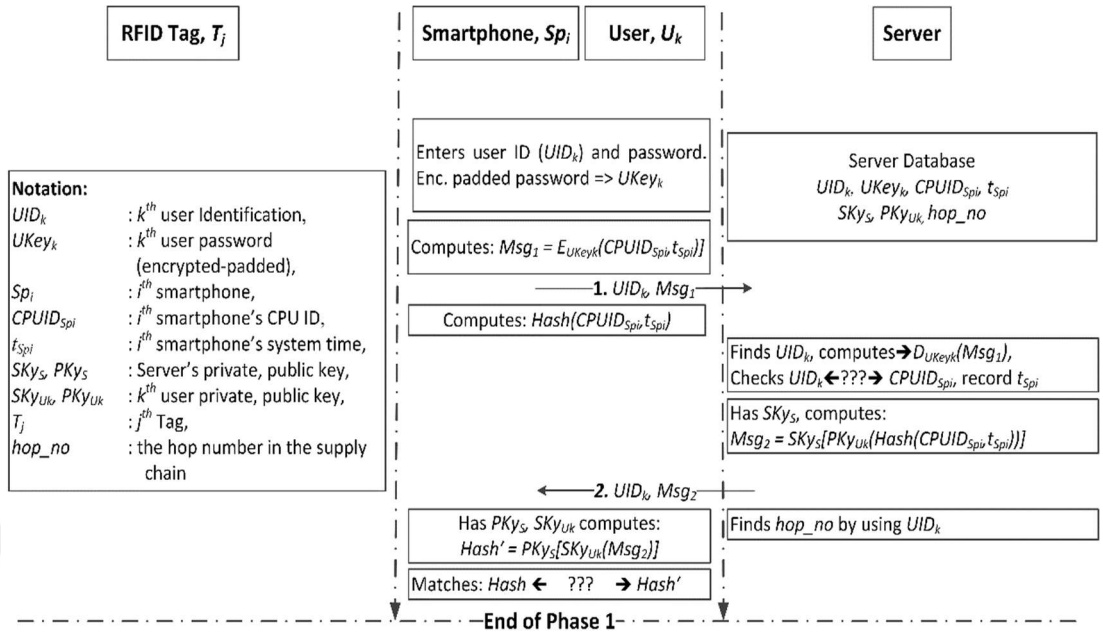



Figure 4.5 Phase 1 of STRAP, "The bonding of a legal smartphone with an authenticated user"

If the calculated and unraveled hash values match, the server has authenticated the user and bonded the user name with the smartphone, in hand. Hence, the mutual authentication of the user and server, and the bonding of the user with the smartphone are finished. Meanwhile, using the user name, the server finds the hop number and decides which key number *(KyNo*) should be used, in the next phases of STRAP.

#### *4.3.1.2 Phase 2 (Mutual Authentication of the RFID Tag and the Server, Creation of Session Key)*

The bonded smartphone requests the identification of the targeted RFID tag, as shown in Figure 4.6. From then on, the smartphone is merely a repeater. The RFID tag sends its identification number $T_j\_ID$ to the server. The server finds the tag in its database and determines the *KyNo* and $KyX_{Tj}$, by using the *hop_no* decided in Phase 1. After generating the random number $Nonce_S$, the server encrypts it using the chosen $KyX_{Tj}$ and sends it to the tag with the prefix *KyNo*, indicating only the key number used. The tag finds the key pointed by the server and uses it to decrypt message $Msg_3$. The value *tmp0* now contains the server's nonce. Next, the tag prepares message $Msg_4$

to show the server that it can decrypt server messages. Then, the tag generates its nonce $Nonce_{Tj}$ and hides it in message $Msg_5$, before sending it. After receiving $Msg_4$ and $Msg_5$ from the tag, the server decrypts $Msg_4$ and XORs it with $Msg_3$. The obtained value is matched with the generated $Nonces_S$. A match authenticates the tag. Next, $Msg_5$ is decrypted and XORed with $Msg_4$. The obtained *tmp3* value is the nonce of the tag. The server computes $Msg_6$ to prove that it can decrypt messages sent by the tag. After sending $Msg_6$, the server calculates the session key *SesKey* by XORing the generated and received nonces. The tag decrypts $Msg_6$ and XORs it with the previously received $Msg_5$. If the obtained value matches its generated nonce $Nonce_{Tj}$, then the server is authenticated. As the last step, the tag calculates its own version of the session *SesKey*. Hence, mutual authentication and new session key generation steps are complete. If the protocol is stopped at any step of Phase 2, then mutual authentication has to be started from the beginning.

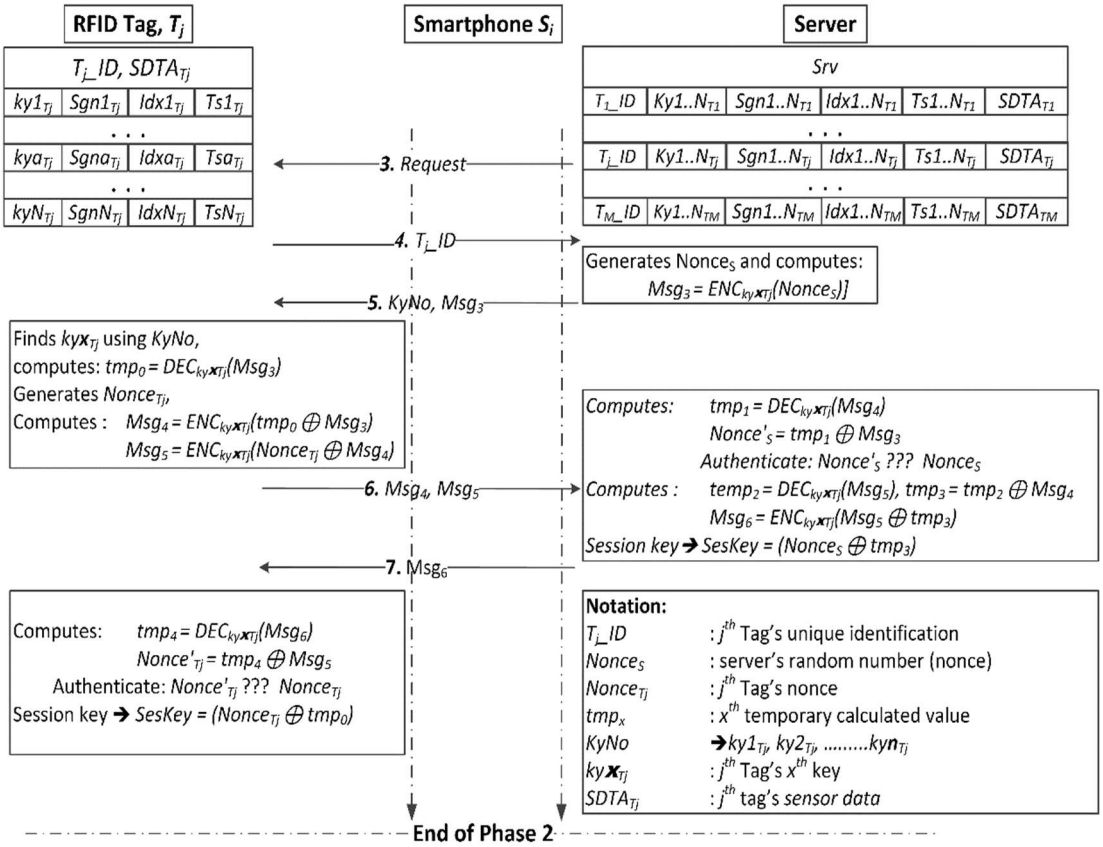



Figure 4.6 Phase 2 of STRAP, "Tag-Server mutual authentication and session key generation"

#### 4.3.1.3 Phase 3 (Tag Signing, and Tag Tracking by SDTA Checking)

In the last phase (Figure 4.7), the server calculates the signature ($Sngi_{Tj}$) for the current stage as $Sngi_{Tj} = Hash_{KyiTj}(T_j\_ID \,||\, Indi_{Tj} \,||\, Ts)$, where $Hash_{KyiTj}(X)$ indicates the encrypted hash value of $X$ with key $Kyi_{Tj}$ of the tag $Tj$ and || indicates concatenation operator. The server encrypts its system time $Ts$, nonce of the tag, and calculated signature generated in phase 2, and sends it to the tag. The tag decrypts message $Msg_7$, extracts, and checks $\text{Nonce}_{Ti}$. Then, the tag saves the signature in its memory and saves $Ts$ to a temporary register. Then, $Ts$ is XORed with the old-time $oldTs$ found in the IC's nonvolatile memory to form $Msg_8$. Initially, the $oldTs$ can be assumed to be zero. Next, $SDTA$ is formed and XORed with $Ts$ and then encrypted with $SesKey$ to form $Msg_9$. Before sending messages $Msg_8$ and $Msg_9$, the new time $oldTs$ are over-written by $Ts$, in the dedicated secure memory, registering a time of tag access. After receiving the messages, the server decrypts $Msg_8$ to check if the tag knows the session key and has the correct $oldTs$ value. After the verification, the server decrypts $Msg_9$ to obtain the sensor status bits by XORing $Msg_8$ with $temp8$.

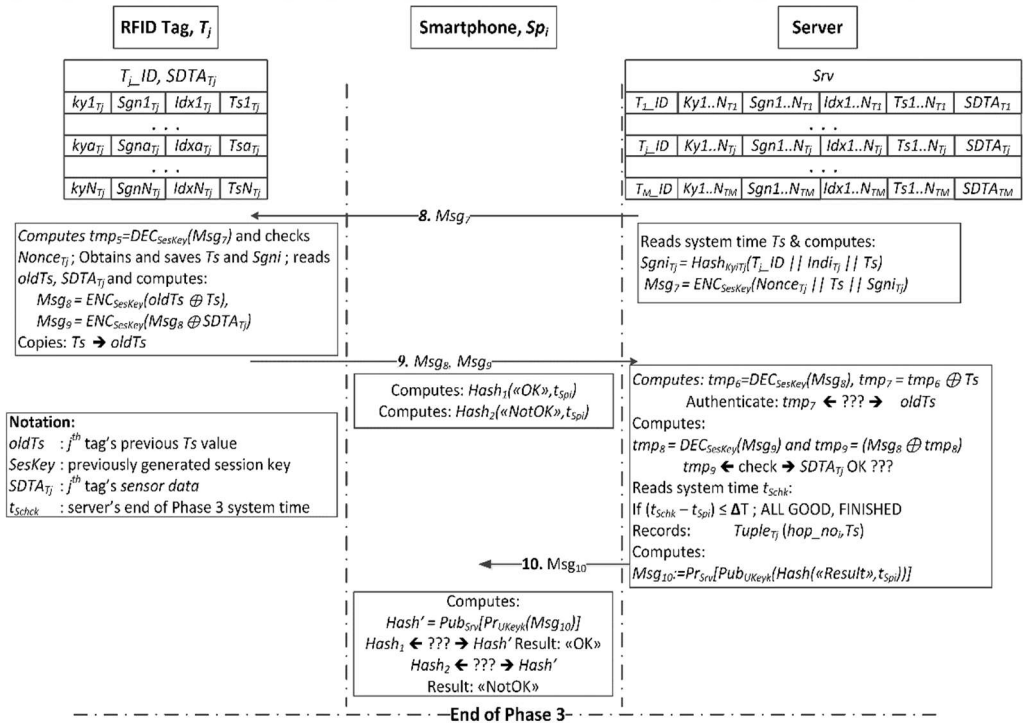



Figure 4.7 Phase 3 of STRAP, "Device tracking by checking SDTA and counting hops in the supply chain"

Then, the *SDTA* is checked. If the sensors are not reporting foul play, the time from the start of Phase 1 ($t_{Spi}$) to the end of Phase 3 ($t_{Schk}$) is checked to verify that the protocol has ended within an acceptable period of time. Next, the server increments the *hop_no* and saves the current record of the tag. The server uses the accumulated records to track the tag's movement, in the supply chain. The server finally prepares $Msg_{10}$ and sends it to the smartphone, as a notification of the result of the overall protocol. $Msg_{10}$ can only be opened up by the smartphone and the result is found by matching the calculated hashes and the received hash. At every session, the result is a variable, obscured, and double encrypted value. After sending $Msg_{10}$ another authentication attempt at the same hop -for the same tag- will be dropped by the server; because a successful authentication has been completed and reported to the smartphone (Figure 4.7).

# CHAPTER 5
# EXPERIMENTAL STUDIES AND RESULTS

## 5.1 Overview

This section contains the security and performance analysis of the proposed SBMAPS and STRAP protocols, and the comparison results of the analysis. Informal and formal security analysis of the proposed protocols was carried out. Formal analysis was conducted with well-accepted protocol validation tools Scyther and AVISPA (Armando et al., 2005; Clarke, Henzinger, Veith, & Bloem, 2018; Cremers, 2008). Our attacker or adversary model is as strong as in the Dolev-Yao model, where the adversary is assumed to have the capability of listening, blocking, changing, and even injecting messages into an exchange (Dolev & Yao, 1983).

## 5.2 Security Analysis of SBMAPS

### *5.2.1 Informal Security Analysis of SBMAPS*

Although the authors of CNTR-SHIELD present mathematical proof of their protocol's security, four attacks have been demonstrated in paragraph 3.2.1. The security analysis of SBMAPS for the same attack types is below (Özcanhan, & Türksönmez, 2020).

#### *5.2.1.1 DoS Attack*

Apparently, counter desynchronization of the dielet and the server is not possible in SBMAPS, since there are no separate, synchronized counters. Message blocking, altering, or loss is also not a threat, because:

a. If phases 1 or 2 are blocked, authentication fails and the protocol stops or times out,
b. If the key number is altered, dielet authentication fails, and the protocol halts,
c. If phases 1 or 2 are disrupted, protocol times out.

#### *5.2.1.2 Full-disclosure Attack*

Full disclosure is out of the question in SBMAPS, as none of the messages are passed in plaintext. The ID is the only information in plaintext, which is required for tracking ICs.

None of the encrypted messages exchanged can be exposed, since the shared keys and private keys are secure. Even if a rogue employee enters its credentials and hands the smartphone to an attacker, the messages cannot be decrypted; simply because the smartphone is a time checked and a message relaying device, unable to see any secrets.

#### *5.2.1.3 Lookup Table Attack*

It is not possible to construct a plaintext-ciphertext pair table in SBMAPS, simply because no plaintext messages are passed. Additionally, the passed messages are always variable, as in the critical sensor status bits and time combination. Furthermore, messages are encrypted with a new session key, every round. The result notification of the authentication to the smartphone at the end of every round is also encrypted and variable.

#### *5.2.1.4 Replay Attack*

None of the messages exchanged are re-playable because the encrypted messages change every round. In addition, the authentication keys and the generated session keys used for encrypting the messages change every round.

### *5.2.2 Formal Security Analysis of SBMAPS*

We have analyzed SBMAPS with both AVISPA and Scyther tools. The analysis results of both tools are presented in the next sub-sections.

#### *5.2.2.1 Verification of SBMAPS using AVISPA*

HLPSL implementation of SBMAPS is based on three roles: *Reader*, *Server*, and *Dielet*. Figure 5.1 shows the Reader role codes of SBMAPS in the HLPSL language.

```
%%-----------------ROLE READER--------------------------------
role role_Reader(
    Reader, Server, Dielet  : agent,
    Usr_ID, CPU_ID          : text,
    Usr_Key, KeyRD          : symmetric_key,
    Hash                    : hash_func,
    ReadReq                 : nat,
    Usr_PK, Srv_PK          : public_key,
    SND, RCV                : channel(dy)
) played_by Reader def=
local
    State           : nat,
    Rdr_t, Result   : text,
    M10             : {{function}_public_key}_inv(public_key)
init State := 0
transition
    %%------------- Phase-1 starts here ----------------------
    0.  State = 0 /\ RCV(start) =|>
        State' := 1 /\ Rdr_t' := new()
        /\ SND(Usr_ID.{Reader.Server.CPU_ID.Rdr_t'}_Usr_Key)
        /\ secret(CPU_ID, sec_cpuid_rs, {Reader, Server})
        /\ secret(Rdr_t', sec_rdrt_rs, {Reader, Server})
    1.  State = 1 /\ RCV({{Hash(Server.Reader.CPU_ID'.Rdr_t)}
        _Usr_PK}_inv(Srv_PK)) =|>

        %%-------------Phase-2 starts here ------------------
        State' := 2 /\ SND({Reader.Dielet.ReadReq}_KeyRD)
        %%Reader checks that it receives the same timestamp
        %%that it sent at step 0
        /\ request(Reader, Server, auth_rdrt_rs, Rdr_t)

    %%-------------Phase-3 starts here ----------------------
    2.  State = 2 /\ RCV(Server.Reader.M10') =|>
        State' := 3
end role
```

Figure 5.1 *Reader* role codes of SBMAPS in HLPSL

As shown in Figure 5.1, a role named *role_Reader* is defined first. The steps of the protocol executed on the reader side are placed in this role. The role definition has parameters of *Reader*, *Server*, *Dielet* in *agent* type, *Usr_ID*, *CPU_ID* in *text* type, *Usr_Key*, *KeyRD* in *symmetric_key* type, *Hash* in *hash_func* type, *ReadReq* in *nat* type, *Usr_PK*, *Srv_PK* in *public_key* type, and *SND*, *RCV* in *channel(dy)* type. The attribute *dy* of the *channel* type represents the intruder model (Dolev-Yao) of the channel. All the variables in the HLPSL language start with a capital letter.

In Figure 5.1, the parameter *Reader* is placed in the *played_by* section, which means that *Reader* represents the name of the agent that plays the role *role_Reader*. In the *local* section, local variables that will be used in the scope of the *role_Reader* role are defined. The *state* variable showing the protocol steps is set to zero in the *init* section of the role *role_Reader*.

The *transition* section contains the set of transitions that shows the received and sent messages. Each transition has a name distinguishing it from other transitions. The /\ operator represents that the specified actions are executed together.

The *start* message is only available in one role and specifies the beginning of the execution of the protocol. In transition 0, if the *State* variable is set to 0 and the *start* message is received through the *RCV* channel then, the *State* variable is set to 1, and the *Rdr_t* variable keeping the time of the *Reader* is set to a new value. The tuple *{CPU_ID*, *Rdr_t}* is encrypted with the symmetric key *Usr_Key* and sent to *Server* through the *SND* channel.

The *secret* property indicates that the secret information is not related to an unauthorized party throughout the message exchange in the protocol. The *secret* property is used to control the secrecy of *CPU_ID* and *Rdr_t'* to be transmitted between the *Reader* agent and the *Server* agent.

In transition 1, if the *State* variable is set to 1 and the hashed value of the tuple *{CPU_ID', Rdr_t}* encrypted with *Usr_PK* public key and signed with *inv(Srv_PK)*

private key is received through the *RCV* channel, then the *State* variable is set to 1 and *ReadReq* variable encrypted with the symmetric key KeyRD is sent to *Dielet* through *SND* channel. The *request* event together with the corresponding *witness* event is used to check strong authentication. With the *request* event here, *Reader* checks that it receives the same timestamp *Rdr_t* that it sent at step 0 (transition 0) and it was not replayed from a previous session.

In transition 2, if the *State* variable is set to 2 and the *M10* variable sent by *Server* is received through the *RCV* channel, then the State variable is set to 3 and the role of the agent *Reader* ends here.

```
%%--------------ROLE SERVER-----------------------------------
role role_Server(
    Reader, Server, Dielet  : agent,
    Usr_Key, KeyDS          : symmetric_key,
    Hash                    : hash_func,
    Usr_PK, Srv_PK          : public_key,
    CurrHop                 : nat,
    SND, RCV: channel(dy)
) played_by Server def=

local
    State               : nat,
    KeySession          : symmetric_key,
    Ts, Usr_ID, CPU_ID, Rdr_t, D_ID, NonS, Temp3, SensorS,
    Result, Old_Ts      : text,
    M1                  : {text.text}_symmetric_key,
    M3, M4, M5, M6      : {text}_symmetric_key,
    M7                  : {text.text}_symmetric_key,
    M8, M9              : {message}_symmetric_key,
    M10                 : {{function}_public_key}_inv(public_key)

init State := 0
transition
    %%-------------Phase-1 starts here --------------------------
    0.  State = 0 /\ RCV(Usr_ID'.{Reader.Server.CPU_ID'.Rdr_t'}_Usr_Key) =|>
        State' := 1
        /\ SND({{Hash(Server.Reader.CPU_ID'.Rdr_t')}_Usr_PK}_inv(Srv_PK))
        /\ secret(CPU_ID', sec_cpuid_sr, {Reader, Server})
        /\ secret(Rdr_t', sec_rdrt_sr, {Reader, Server})
        %%Server hopes that Rdr_t will permit to authenticate him
        /\ witness(Server, Reader, auth_rdrt_rs, Rdr_t')
```

Figure 5.2 Server role codes of SBMAPS in HLPSL (Part-1)

The first part of the *Server* role codes of SBMAPS in HLPSL is shown in Figure 5.2. The steps of the protocol executed on the server-side are placed in this role. The role definition has parameters of *Reader*, *Server*, *Dielet* in agent type, *Usr_Key*,

*KeyRD* in symmetric_key type, *Hash* in hash_func type, *Usr_PK*, *Srv_PK* in public_key type, *CurrHop* in nat type, and *SND*, *RCV* in channel(dy) type.

The parameter *Server* is placed in the *played_by* section, which means that *Server* represents the name of the agent that plays the role *role_Server*. In the *local* section, local variables that will be used in the scope of the *role_Server* role are defined. The *state* variable showing the protocol steps is set to zero in the *init* section of the role *role_Server*.

```
    %%-------------Phase-2 starts here -------------------------
    1. State = 1 /\ RCV({Dielet.Server.D_ID'}_KeyDS) =|>
       State' := 2 /\ NonS' := new() /\ M3' := {NonS'}_KeyDS
       /\ SND(Server.Dielet.CurrHop.M3')
       /\ secret(NonS', sec_nons_sd, {Server, Dielet})

    2. State = 2 /\ RCV(Dielet.Server.M4'.{xor(Temp3', M4')}_KeyDS) =|>
       State' := 3 /\ M5' := {xor(Temp3', M4')}_KeyDS
       /\ M6' := {xor(M5', Temp3')}_KeyDS
       /\ KeySession' := xor(NonS, Temp3') /\ Ts' := new()
       /\ M7' := {Temp3'.Ts'}_KeySession'
       /\ SND(Server.Dielet.M6'.M7')
       /\ secret(Temp3', sec_temp3_sd, {Server, Dielet})
       /\ secret(NonS, sec_nonsl_sd, {Server, Dielet})
       /\ secret(Ts', sec_ts_sd, {Server, Dielet})
       %%Server checks that it receives the same nonce that it sent at step 1
       /\ request(Server, Dielet, auth_nons_sd, NonS)
       %%Server hopes that NonD(Temp3) will permit to authenticate him
       /\ witness(Server, Dielet, auth_nond_ds, Temp3')

    %%-------------Phase-3 starts here ----------------------
    3. State = 3 /\ RCV(Dielet.Server.M8'.{xor({xor(Old_Ts', Ts)}_KeySession',
           SensorS')}_KeySession') =|>
       State' := 4 /\ Result' := new()
       /\ M10' := {{Hash(Result'.Rdr_t')}_Usr_PK}_inv(Srv_PK)
       /\ SND(Server.Reader.M10')
       /\ secret(Rdr_t', sec_rdrt_sr, {Server, Reader})
       /\ secret(Result', sec_result_sr, {Server, Reader})
       %%Server checks that it receives the same timestamp
       %%that it sent at step 2
       /\ request(Server, Dielet, auth_ts_sd, Old_Ts')
end role
```

Figure 5.3 Server role codes of SBMAPS in HLPSL (Part-2)

In transition 0 in Figure 5.2, if the *State* variable is set to 0, and the *Usr_ID'* variable and the tuple *{CPU_ID', Rdr_t'}* encrypted with the *Usr_Key* symmetric key is received through the *RCV* channel then, the *State* variable is set to 1 and the hashed value of the tuple *{CPU_ID', Rdr_t'}* encrypted with *Usr_PK* public key and signed with *inv(Srv_PK)* private key is sent to *Reader* through the *SND* channel. The *secret*

property is used to control the secrecy of *CPU_ID'* and *Rdr_t'* to be transmitted between the *Reader* agent and the *Server* agent. With the *witness* event here, *Server* hopes that *Rdr_t'* will permit to authenticate it.

In transition 1 in Figure 5.3, if the *State* variable is set to 1 and the *D_ID'* variable encrypted with the *KeyDS* symmetric key by the *Dielet* agent is received through the *RCV* channel then, the *State* variable is set to 2, the *NonS'* nonce is created. *NonS'* encrypted with the *KeyDS* symmetric key is assigned to the *M3* variable, and the *CurrHop* variable with the *M3* variable is sent to the *Dielet* agent through the *SND* channel. The *secret* property is used to control the secrecy of *NonS'* to be transmitted between the *Server* agent and the *Dielet* agent.

In transition 2, if the *State* variable is set to 2, and XOR'ed *{Tem3', M4'}* tuple encrypted with the *Key_DS* symmetric key, and the *M4'* variable are received through the *RCV* channel between the Reader and the Dielet agents then, the *State* variable is set to 3 and XOR'ed *{Temp3', M4'}* tuple encrypted with the *Key_DS* is assigned to the *M5'* variable. XOR'ed *{M5', Temp3'}* tuple encrypted with the *Key_DS* is assigned to the *M6'* variable. XOR'ed *{NonS, Temp3'}* tuple is assigned to the *KeySession'* variable and a new timestamp is assigned to the *Ts'* variable. *{Temp3', Ts'}* tuple encrypted with the *KeySession'* session key is assigned to the *M7'* variable, and *{M6', M7'}* tuple is sent to the *Dielet* agent through the *SND* channel.

The *secret* property is used to control the secrecy of *Temp3', NonS,* and *Ts'* to be transmitted between the *Server* agent and the *Dielet* agent. With the *request* event, *Server* checks that it receives the same nonce *NonS* that it sent at step 1 (transition 1) and it was not replayed from a previous session. With the *witness* event here, *Server* hopes that *Temp3'* will permit it to authenticate it.

In transition 3, if the *State* variable is set to 3, and the M8' variable and *{xor({xor(Old_Ts', Ts)}_KeySession', SensorS')}_KeySession'* expression (*M9'*) sent by the *Dielet* agent are received through the *RCV* channel then, the *State* variable is set to 4 and a new value is assigned to the *Result'* variable. The hashed tuple *{Result',*

*Rdr_t'}* encrypted with *Usr_PK* public key and signed with *inv(Srv_PK)* private key is assigned to the *M10'* variable. The *M10'* variable is sent to the *Reader* agent through the *SND* channel.

The *secret* property is used to control the secrecy of *Rdr_t'*, and *Result'* to be transmitted between the *Server* agent and the *Reader* agent. With the *request* event, *Server* checks that it receives the same timestamp *Old_Ts'* that it sent at step 2 (transition 2) and it was not replayed from a previous session.

```
%%------------------ROLE DIELET----------------------------
role role_Dielet(
    Reader, Server, Dielet  : agent,
    ReadReq                 : nat,
    KeyRD, KeyDS            : symmetric_key,
    D_ID, Old_Ts            : text,
    SND, RCV                : channel(dy)
) played_by Dielet def=

local
    State, ReadReq1, CurrHop    : nat,
    KeySession                  : symmetric_key,
    M3, M4, M5, M6              : {text}_symmetric_key,
    M7                          : {text.text}_symmetric_key,
    Temp0, NonD, Tss, SensorS   : text,
    M8, M9                      : {message}_symmetric_key

init State := 0
transition
    %%-------------Phase-2 starts here -------------------------
    0.  State = 0 /\ RCV({Reader.Dielet.ReadReq1'}_KeyRD)
        /\ ReadReq = ReadReq1' =|>
        State' := 1 /\ SND({Dielet.Server.D_ID}_KeyDS)
        /\ secret(D_ID, sec_did_ds, {Dielet, Server})

    1.  State = 1 /\ RCV(Server.Dielet.CurrHop'.{Temp0'}_KeyDS) =|>
    %%---Dielet.Temp0 = Server.NonS----
        State' := 2 /\ M3' := {Temp0'}_KeyDS
        /\ M4' := {xor(Temp0', M3')}_KeyDS
        /\ NonD' := new() /\ M5' := {xor(NonD', M4')}_KeyDS
        /\ KeySession' := xor(NonD', Temp0')
        /\ SND(Dielet.Server.M4'.M5')
        /\ secret(Temp0', sec_temp0_ds, {Dielet, Server})
        /\ secret(NonD', sec_nond_ds, {Dielet, Server})
        %%Dielet hopes that NonS(Temp0) will permit to authenticate him
        /\ witness(Dielet, Server, auth_nons_sd, Temp0')
```

Figure 5.4 Dielet role codes of SBMAPS in HLPSL (Part-1)

The first part of the *Dielet* role codes of SBMAPS in HLPSL is shown in Figure 5.4. The steps of the protocol executed on the dielet side are placed in this role. The role definition has parameters of *Reader*, *Server*, *Dielet* in agent type, *ReadReq* in *nat*

type, *KeyRD* and *KeyDS* in symmetric_key type, *D_ID* and *Old_Ts* in text type, and *SND*, *RCV* in *channel(dy)* type.

The parameter *Dielet* is placed in the *played_by* section, which means that *Dielet* represents the name of the agent that plays the role *role_Dielet*. In the *local* section, local variables that will be used in the scope of the *role_Dielet* role are defined. The *state* variable showing the protocol steps is set to zero in the *init* section of the role *role_Dielet*.

In transition 0 on Figure 5.4, if the *State* variable is set to 0, and the *ReadReq1'* variable encrypted with the *KeyRD* symmetric key sent by the *Reader* agent is received through the *RCV* channel, and the *ReadReq* variable is equal to the *ReadReq1'* variable then, the *State* variable is set to 1 and the *D_ID* variable encrypted with the *KeyRD* symmetric key is sent to the *Server* agent through the *SND* channel. The *secret* property is used to control the secrecy of *D_ID* to be transmitted between the *Server* agent and the *Dielet* agent.

In transition 1, if the *State* variable is set to 1, and the *CurrHop'* variable and the *Temp0'* variable encrypted with the *KeyDS* symmetric key by the *Server* agent is received through the *RCV* channel then, the *State* variable is set to 2, and *Temp0'* variable encrypted with the *KeyDS* symmetric key is assigned to *M4'* variable. A new nonce is assigned to the *NonD'* variable. The XOR'ed *{NonD', M4'}* tuple encrypted with the *KeyDS* symmetric key is assigned to the *M5'* variable, and the XOR'ed *{NonD', Temp0'}* tuple is assigned to the *KeySession'* variable. The *{M4', M5'}* tuple is sent to the *Server* agent through the SND channel.

The *secret* property is used to control the secrecy of *Temp0'*, and *NonD'* to be transmitted between the *Server* agent and the *Dielet* agent. With the *witness* event here, *Dilelet* hopes that *Temp0'* (*NonS*) will permit it to authenticate it.

```
    2.  State = 2 /\ RCV(Server.Dielet.M6'.{NonD.Tss'}_KeySession') =|>
    %%-------------Phase-3 starts here ----------------------
    State' := 3
        /\ SensorS' := new()
        /\ M8':= {xor(Old_Ts, Tss')}_KeySession'
        /\ M9' := {xor(M8', SensorS')}_KeySession'
        /\ SND(Dielet.Server.M8'.M9')
        /\ secret(Old_Ts, sec_oldts_ds, {Dielet, Server})
        /\ secret(Tss', sec_tss_ds, {Dielet, Server})
        /\ secret(SensorS', sec_sensors_ds, {Dielet, Server})
        %%Dielet checks that it receives the same nonce that it sent at step 1
        /\ request(Dielet, Server, auth_nond_ds, NonD)
        %%Dielet hopes that Old_Ts will permit to authenticate him
        /\ witness(Dielet, Server, auth_ts_sd, Old_Ts)
end role
```

Figure 5.5 Dielet role codes of SBMAPS in HLPSL (Part-2)

In transition 2 on Figure 5.5, if the *State* variable is set to 2 the *M6'* variable and the *{NonD, Ts'}* tuple encrypted with the *KeySession'* symmetric key by the *Server* agent is received through the *RCV* channel then, the *State* variable is set to 3 and a new value is assigned to the *SensorS'* variable. The XOR'ed *{Old_Ts, Tss'}* tuple encrypted with the *KeySession'* symmetric key is assigned to the *M8'* variable. The XOR'ed *{ M8', SensorS'}* tuple encrypted with the *KeySession'* symmetric key is assigned to the *M9'* variable. The *{M8', M9'}* tuple is sent to the *Server* agent through the *SND* channel.

The *secret* property is used to control the secrecy of *Old_Ts*, Tss', and *SensorS'* to be transmitted between the *Server* agent and the *Dielet* agent. With the *request* event, *Dielet* checks that it receives the same nonce *NonD* that it sent at step 1 (transition 1) and it was not replayed from a previous session. With the *witness* event here, *Dilelet* hopes that *Old_Ts* will permit it to authenticate it.

The SBMAPS protocol is analyzed using the OFMC backend of the AVISPA SPAN Tool. The analysis result of the proposed protocol is shown in Figure 5.6. Line 4 of the OFMC result shows that the proposed SBMAPS protocol is 'SAFE' against attacks tested by AVISPA. Details of the analysis are summarized after the 'SAFE' declaration of the tool. The tool confirms that the protocol had a bounded number of checker sessions. The location of the protocol file is also given in the results to ensure the user about the HLPSL code analyzed. 'as_specified' declaration of the tool confirms that the security goals described in the HLPSL code have been reached.

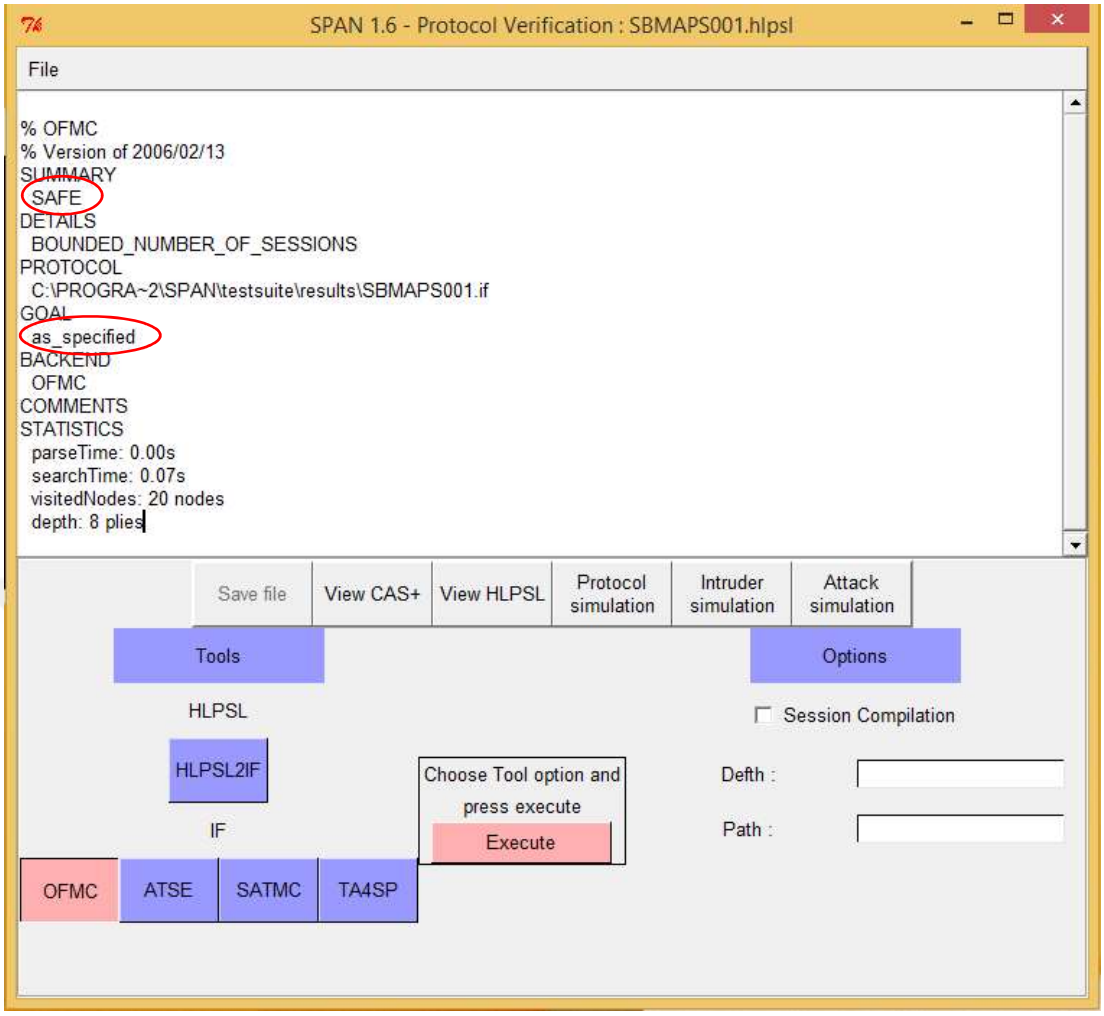



Figure 5.6 Security evaluation result of SBMAPS with AVISPA Span Tool

### 5.2.2.2 *Verification of SBMAPS using Scyther Tool*

SBMAPS was coded using the Security Protocol Description Language (SPDL). During the analysis, the smartphone equipped with an RFID reader was given the *Reader* role, the server was given *Server* role, and the dielet on IC was given *Dielet* role. The SPDL code of the Reader role is shown in Figure 5.7. Local variables and constants are declared after the role definition. The *NotOK* and *OK* variables are declared in *Nonce* type and assigned fresh values. The *TR* variable is declared in *TimeStamp* user-defined type and assigned a fresh value. The *CpuID* variable in *String* user-defined type, the *Tr* variable in *TimeStamp* type, and the *Result* variable in *Nonce* type are declared. The *USRID*, *CPUID*, and *REQ* constants are declared in String type.

In phase-1 of role Reader, the *{USRID, CPUID, TR}* tuple is encrypted with the *k* symmetric key shared between Reader and Server and assigned to the *M1* macro. The *M1* macro is sent to *Server* from *Reader* by the *send_1* command. After sending *M1*, the hash value of the *{CPUID, TR}* tuple is calculated and assigned to the *H1* macro. The hashed value of the *{CpuID, Tr}* tuple encrypted with *Reader's* public key and signed with *Server's* private key is received by the *recv_2* command. Then the hashed value of the *{CpuID, Tr}* tuple is assigned to the *H2* macro and the *H1* and *H2* macros are compared for a match.

In phase-2 of role Reader, the *{Reader, REQ}* tuple encrypted with the *k* symmetric key shared between *Reader* and *Dielet* is sent from *Reader* to *Dielet* by the *send_3* command.

In phase-3 of role Reader, the hashed value of the *{OK, TR}* tuple is assigned to the *h1* macro, the hashed value of the *{NotOK, TR}* tuple is assigned to the *h2* macro, and the hashed value of the *{Result, Tr}* tuple is assigned to the *ResHash* macro. The *ResHash* macro encrypted with the *Reader's* public key and signed with the *Server's* private key is received from *Server* by the *recv_10* command. Then the *h1* and *h2* macros are compared with the *ResHash* macro for a match.

```
//----------------ROLE READER----------------------------------------------------------------
    role Reader
    {
        fresh NotOK, OK: Nonce;
        fresh TR: TimeStamp;
        var CpuID: String;
        const USRID, CPUID, REQ: String;
        var Tr: TimeStamp;
        var Result: Nonce;

//-------------------------Phase-1 starts here-----------------------------------------
        macro M1 = {USRID, CPUID, TR}k(Reader, Server);
        send_1(Reader, Server, M1);
        macro H1 = Hash(CPUID, TR);
        recv_2(Server, Reader, {{Hash(CpuID, Tr)}pk(Reader)}sk(Server));
        macro H2 = Hash(CpuID, Tr);
        match(H1, H2);

//-------------------------Phase-2 starts here-----------------------------------------
        send_3(Reader, Dielet, {Reader, REQ}k(Reader, Dielet));

//-------------------------Phase-3 starts here-----------------------------------------
        macro h1 =  Hash(OK, TR);
        macro h2 =  Hash(NotOK, TR);
        macro ResHash = Hash(Result, Tr);
        recv_10(Server, Reader,{{ResHash}pk(Reader)}sk(Server));
        match(h1, ResHash);
        match(h2, ResHash);

//-------------------------Reader Claim Descriptions-----------------------------------

        claim_r1(Reader, Secret, CPUID);
        claim_r2(Reader, Secret, TR);

        claim_r3(Reader, Niagree);
        claim_r4(Reader, Nisynch);
        claim_r5(Reader, Alive);
        claim_r6(Reader, Weakagree);
    }
```

Figure 5.7 Reader role codes of SBMAPS in SPDL

The SPDL code of the Server role is shown in Figure 5.8. Local variables and constants are declared after the role definition. The *NonS*, *TS*, *Result*, *CurrHop* variables are declared in *Nonce* type and assigned fresh values. The *Tr* variable is declared in *TimeStamp* user-defined type. The *CpuID, UsrID,* and *Did* variables in *String* user-defined type, the *Tr* variable in *TimeStamp* type, and the *NonSS*, *NonDD*, *OldTs*, *Ts*, and *SensorSS* variables in *Nonce* type are declared. The *CPUID* and *DID* constants are declared in String type.

```
//------------ROLE SERVER-----------------------------------------------------------------
    role Server
    {
        var CpuID, UsrID, Did: String;
        fresh NonS, TS, Result, CurrHop: Nonce;
        var NonSS, NonDD, OldTs, Ts, SensorSS: Nonce;
        var temp3, temp6, temp7, temp8, temp9, M4, M7, M8, M9, ResHash;
        const CPUID, DID: String;
        var Tr: TimeStamp;

//-------------------------Phase-1 starts here-----------------------------------------
        recv_1(Reader, Server, {UsrID, CpuID, Tr}k(Reader, Server));
        match(CPUID, CpuID);
        macro M2 = {{Hash(CpuID, Tr)}pk(Reader)}sk(Server);
        send_2(Server, Reader, M2);

//-------------------------Phase-2 starts here-----------------------------------------
        recv_4(Dielet, Server, {Did}k(Dielet, Server));
        match(DID, Did);
        macro M3 = {NonS}k(Dielet, Server);
        send_5(Server, Dielet, CurrHop, M3);
        recv_6(Dielet, Server, M4, {fxor(NonDD, M4)}k(Dielet, Server));
        macro M5 = {fxor(NonDD, M4)}k(Dielet, Server);
        macro temp1 = fxor(NonS, M3);
        match(NonSS, temp1);
        macro temp2 = fxor(NonDD, M4);  // temp2 = NonD x M4
        match(temp3, fxor(temp2, M4)); // temp3 = NonD
        macro KeyS =  fxor(NonS, NonDD);
        macro M6 =  {fxor(M5, temp3)}k(Dielet, Server);
        send_7(Server, Dielet, M6);

//-------------------------Phase-3 starts here-----------------------------------------
        macro M7 = {NonDD, TS}KeyS;
        send_8(Server, Dielet, M7);
        macro M8 = {fxor(OldTs, Ts)}KeyS;
        macro M9 = {fxor(M8, SensorSS)}KeyS;
        recv_9(Dielet, Server,M8, M9);
        macro temp6 = fxor(OldTs, Ts);
        macro temp7 = fxor(temp6, Ts);
        match(temp7, OldTs); // temp7 =? OldTs
        macro temp8 = fxor(M8, SensorSS);
        macro temp9 = fxor(M8, temp8);
        match(temp9, SensorSS); // temp9 =? SensorSS
        macro ResHash = Hash(Result, Tr);
        send_10(Server, Reader,{{ ResHash}pk(Reader)}sk(Server));

//---------------------Server Claim Descriptions-----------------------------------

        claim_s1(Server, Secret, CpuID);
        claim_s2(Server, Secret, Tr);

        claim_s3(Server, Niagree);
        claim_s4(Server, Nisynch);
        claim_s5(Server, Alive);
        claim_s6(Server, Weakagree);
    }
```

Figure 5.8 Server role codes of SBMAPS in SPDL

In phase-1 of role Server, the *{UsrID, CpuID, Tr}* encrypted with the *k* symmetric key shared between *Reader* and *Server* is received from *Reader* by the *recv_1* command. Then *CPUID* and *CpuID* are compared for a match. The hashed value of the *{CpuID, Tr}* tuple is assigned to the *M2* macro and the *M2* macro is sent to Reader by the *send_2* command.

In phase-2 of role Server, *Did* that is encrypted with the *k* symmetric key shared between *Dielet* and *Server* is received from *Dielet* by the *recv_4* command. Then DID and Did are compared for a match. The *NonS* nonce encrypted with the *k* symmetric key shared between *Dielet* and *Server* is assigned to the *M3* macro and the *M3* macro with *CurrHop* is sent to *Dielet* by the *send_5* command. The XOR'ed value of the *{NonDD, M4}* tuple that is encrypted with the *k* symmetric key shared between *Server* and *Dielet*, and M4 itself is received from *Dielet* by the *recv_6* command. The XOR'ed value of the *{NonDD, M4}* tuple encrypted with the *k* symmetric key shared between *Server* and *Dielet* is assigned to *M5* macro. The XOR'ed value of *NonS* and *M3* is assigned to the *temp1* macro. Then *NonSS* and *temp1* are compared for a match. The XOR'ed value of *NonDD* and *M4* is assigned to the *temp2* macro. The XOR'ed value of *temp2* and *M4* is compared with *temp3* for a match. Then the XOR'ed value of *NonS* and *NonDD* is assigned to the *KeyS* macro that will be used as a session key later. The XOR'ed value of the *{M5, temp3}* tuple encrypted with the *k* symmetric key shared between *Server* and *Dielet* is assigned to the *M6* macro and the *M6* macro is sent to *Dielet* by the *send_7* command.

In phase-3 of role Server, the *{NonDD, TS}* tuple encrypted with the *KeyS* session key is assigned to the *M7* macro and the *M7* macro is sent to *Dielet* by the *send_8* command. The XOR'ed value of *OldTs* and *Ts* encrypted with *KeyS* is assigned to the *M8* macro and the XOR'ed value of *M8* and *SensorSS* encrypted with *KeyS* is assigned to the *M9* macro. The *M8* and *M9* macros are received from *Dielet* by the *recv_9* command. The XOR'ed value of *OldTs* and *Ts* is assigned to the *temp6* macro and the XOR'ed value of *temp6* and *Ts* is assigned to the *temp7* macro. Then *temp7* and *OldTs* are compered for a match. The XOR'ed value of *M8* and *temp8* is assigned to the *temp9* macro. The *temp9* macro and *SensorSS* are compared for a match. The hashed value

of the *{Result, Tr}* tuple is assigned to the *ResHash* macro and the *ResHash* macro encrypted with *Reader's* public key and signed with the *Server's* private key is sent to *Reader* by the *send_10* command.

```
//-------------------------ROLE DIELET-----------------------------------------------------
    role Dielet
    {
        const REQ, DID: String;
        var Req: String;
        var NonS, NonDD, Ts, CurrHop: Nonce;
        fresh NonD, OldTs, SensorS: Nonce;
        var M6, M7;

//-------------------------Phase-2 starts here---------------------------------------------
        recv_3(Reader, Dielet, {Reader, Req}k(Reader, Dielet));
        match(REQ, Req);
        send_4(Dielet, Server, {DID}k(Dielet, Server));
        recv_5(Server, Dielet, CurrHop, {NonS}k(Dielet, Server));

        macro M3 = {NonS}k(Dielet, Server);
        macro M4 = {fxor(NonS, M3)}k(Dielet, Server);
        macro M5 = {fxor(NonD, M4)}k(Dielet, Server);
        send_6(Dielet, Server, M4, M5);
        recv_7(Server, Dielet, {fxor(M5, NonDD)}k(Dielet, Server));
        macro KeyS = fxor(NonS, NonDD);

//-------------------------Phase-3 starts here---------------------------------------------
        recv_8(Server, Dielet, {NonDD, Ts}KeyS);
        macro M8 = {fxor(OldTs, Ts)}KeyS;
        macro M9 = {fxor(M8, SensorS)}KeyS;
        send_9(Dielet, Server, M8, M9);


//-------------------Dielet  Claim Descriptions--------------------------------------------

        claim_d1(Dielet, Secret, OldTs);
        claim_d2(Dielet, Secret, NonD);
        claim_d3(Dielet, Secret, SensorS);
        claim_d4(Dielet, Secret, DID);
        claim_d5(Dielet, Secret, NonS);
        claim_d6(Dielet, Secret, Ts);

        claim_d7(Dielet, Niagree);
        claim_d8(Dielet, Nisynch);
        claim_d9(Dielet, Alive);
        claim_d10(Dielet, Weakagree);
    }
```

Figure 5.9 Dielet role codes of SBMAPS in SPDL

The SPDL code of the Dielet role is shown in Figure 5.9. Local variables and constants are declared after the role definition. The *NonD*, Old*Ts*, *SensorS* variables

are declared in *Nonce* type and assigned fresh values. The *Req* variable is declared in *String* user-defined type, the *NonS, NonDD, Ts,* and *CurrHop* variables in *Nonce* type, and *M6* and *M7* variables with no type are declared. The *REQ* and *DID* constants are declared in String type.

Since there is no phase-1 of Dielet role, it starts from phase-2. In phase-2 of role Dielet, the {Reader, Req} tuple encrypted with the *k* symmetric key shared between *Reader* and *Dielet* is received from *Reader* by the *recv_3* command, and *REQ* and *Req* are compared for a match. DID that is encrypted with the *k* symmetric key shared between *Server* and *Dielet* is sent to *Server* by the *send_4* command and *CurrHop* with *NonS* encrypted with the *k* symmetric key shared between *Server* and *Dielet* is received from *Server* by the *recv_5* command. *NonS* encrypted with the *k* symmetric key shared between *Server* and *Dielet* is assigned to the *M3* macro, the XOR'ed value of *NonS* and *M3* encrypted with the *k* symmetric key shared between *Server* and *Dielet* is assigned to the *M4* macro, and the XOR'ed value of *NonD* and *M4* encrypted with the *k* symmetric key shared between *Server* and *Dielet* is assigned to the *M5* macro. Then the *M4* and *M5* macros are sent to *Server* by *send_6* command. The XOR'ed value of *M5* and *NonDD* encrypted with the *k* symmetric key shared between *Server* and *Dielet* is received from *Server* by the *recv_7* command. The XOR'ed value of *NonS* and *NonDD* is assigned to the *KeyS* macro. Then the *{NonDD, Ts}* tuple encrypted with the *KeyS* session key is received from *Server* by *recv_8* command. The XOR'ed value of *OldTs* and *Ts* encrypted with the *KeyS* session key is assigned to the *M8* macro and the XOR'ed value of *M8* and *SensorS* encrypted with the *KeyS* session key is assigned to the *M9* macro. Then the *M8* and *M9* macros are sent to *Server* by the *send_9* command.

The Scyther Tool automatically analyzes submitted schemes, carefully observing the proposed scheme's reaction to the known attacks. SBMAPS was coded using the Security Protocol Description Language (SPDL). During the analysis, the smartphone equipped with an RFID reader was given the *Reader* role, the server was given *Server* role, and the IC equipped with a dielet was given *Dielet* role. Figure 5.10 shows the analysis output of our SBMAPS protocol.

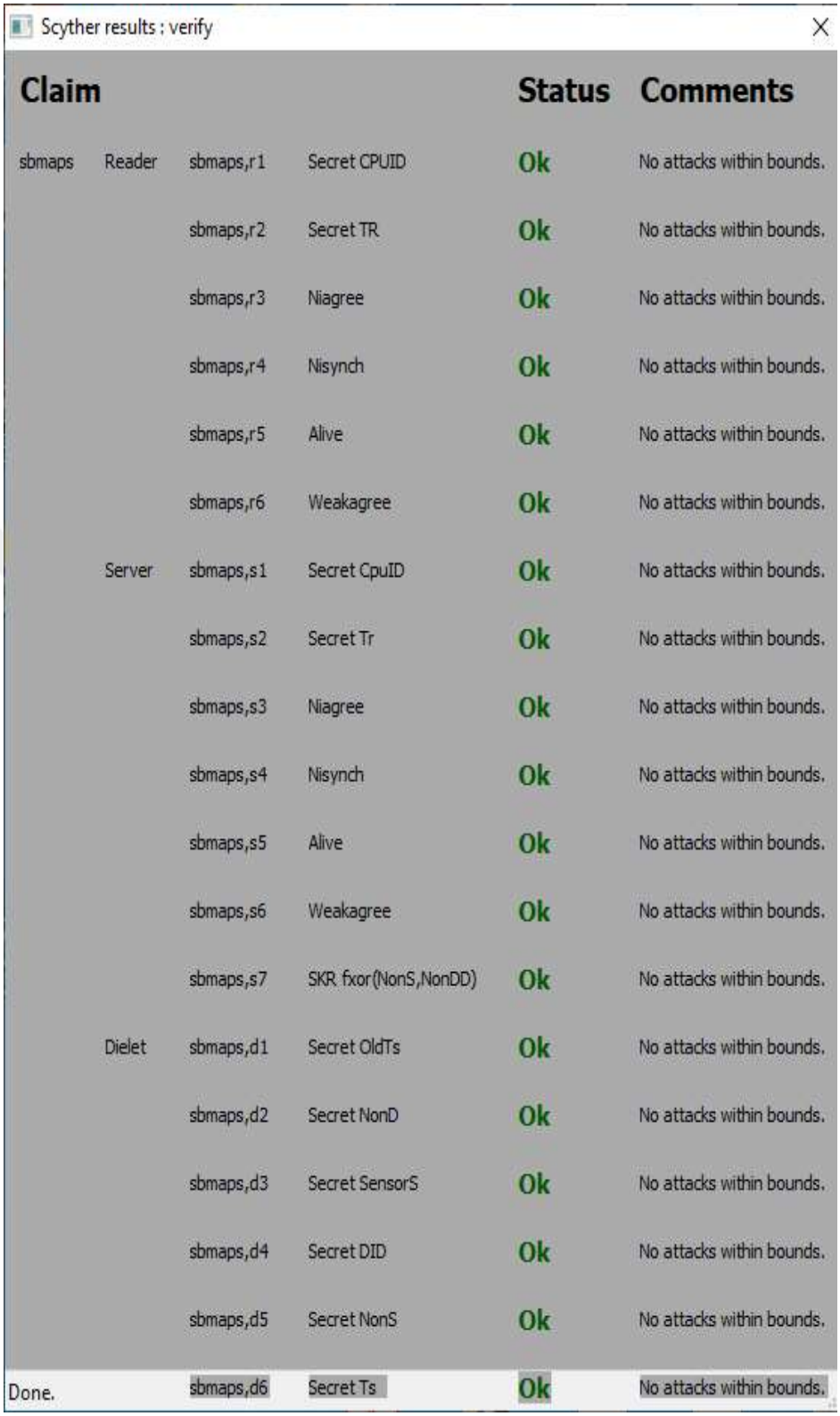



Figure 5.10 Analysis result of SBMAPS via the Scyther Tool

The ten message exchanges of SBMAPS were coded as "send" and "receive" parameters. The "claim" events are used to control the security of the message transferred. The "claim" requires parameters such as 'Secret', 'Alive', 'Weakagree', session-key reveal ('SKR'), 'Niagree' and 'Nisynch'. The ‘Secret’ parameter specifies that the coded SBMAPS parameters’ secrecy (*CPUID, TR, Tr, CpuID, OldTs, NonD, SensorS, DID,* and *NonS*) is expected to be verified by a “No attacks within bounds” comments, as in Figure 5.10. The Sychter response to 'Alive' parameter specifies that communicating partners are alive and the availability of the authentication peers is ensured. Verification of the 'weakagree' parameter ensures that the protocol is immune to impersonation attacks (Nikooghadam & Amintoosi, 2020). Verification of the secrecy of the session key KeyS is verified by 'SKR' parameter. The use of 'Niagree' on all three roles ensures that non-injective agreement is achieved, i.e. no messages can be injected into the message exchange between the peers. Hence, the message exchange between the peers is uncorrupted. The synchronization of the ten steps (non-injective synchronization) is verified by the 'nisynch' parameter, ensuring that all messages sent by reciprocal have been received by the recipient. The 'OK' declarations in Figure 5.10 demonstrate that the protocol was completed successfully, and no attacks were detected against the submitted protocol. Hence, the submitted SBMAPS was verified.

## 5.3 Performance and Security Comparison of SBMAPS

### *5.3.1 Security Comparison of SBMAPS*

Table 5.1 compares the security characteristics of the three studied protocols. CNTR-SHIELD tries to improve the SHIELD protocol by generating the ID, using the dielet’s inherent TRNG characteristic. There is no difference between protocols, in this property. But the dielet’s ID and its secret key are sent in plaintext. In contrast, all messages are encrypted in SBMAPS. Thus, plaintext-ciphertext pairing is possible in both SHIELD and CNTR-SHIELD, the same is not true for SBMAPS. Therefore, our protocol is secure compared to the two previous schemes.

Table 5.1 Security comparison of the studied three SHIELD protocols

| Authentication Protocol | TRNG | Plain Text | SPhone Use | Key Update | DoS Attack | Full Disclosure Attack | Lookup Table Attack | Replay Attack |
|---|---|---|---|---|---|---|---|---|
| SHIELD | No | Yes | No | No | Yes | No | Yes | Yes |
| CNTR-SHIELD | Yes | Yes | No | No | Yes | Yes | Yes | Yes |
| SBMAPS | Yes | No | Yes | Yes | No | No | No | No |

The security of the intermediary smartphone and its user has critical importance. Only SBMAPS guarantees the elimination of attacks by rogue smartphones, or their users. Another important security primitive is the key update. Neither of the previous protocols accommodates key updates, but SBMAPS provides a new session key, every round. Hence, transmitted SBMAPS messages are changed every round, by changing the encryption keys. Due to weaknesses, the original SHIELD and CNTR-SHIELD are vulnerable to four types of attacks. On the other hand, SBMAPS does not show the same vulnerabilities and thus complies with the requirements of SHIELD. A further improvement in SBMAPS is the requirement for a login name-password tuple for each chain hop. Although many, there are a finite number of hops. Therefore, a small size password file on the server is sufficient to provide extended security.

As a last security comparison, the four adversarial models given in CNTR-SHIELD will be discussed.

#### *5.3.1.1 The attacker obtains a list of valid IDs and tries to produce a fake dielet*

As shown in paragraph 3.2.1 this is possible in CNTR-SHIELD but not in SBMAPS, because the secret keys are controlled by the IP owner, and all messages passed in the initialization and authentication phases are encrypted.

*5.3.1.2 The attacker has a "black box" (i.e. rogue smartphone) to access legitimate dielets and collects exchanged messages to produce a fake dielet*

As shown in paragraph 3.2.1 this is possible in CNTR-SHIELD. But since all messages are variable, encrypted, and controlled by the secure server, this adversarial model cannot succeed in SBMAPS.

*5.3.1.3 The attacker has the capability to separate a dielet from its legitimate host IC and reuse the dielet in a fake IC*

Separating the dielet without setting off the passive sensors is against the requirements of SHIELD. Nevertheless, this type of attack is resisted in SBMAPS by hiding the time in non-volatile, secure IC memory. Hence, a dissected dielet cannot provide the time trace to produce a fake.

*5.3.1.4 The attacker extracts secret keys from dielets by physical attacks to produce a fake dielet*

SBMAPS is not strong against capturing the secrets by tampering with the dielet, without setting off the sensors. Physical attacks are in a different adversarial model category and have to be countered by hardware methods. Our work's scope covers the non-invasive side-channel attacks.

### *5.3.2 Performance Comparison of SBMAPS*

The present study is not challenging the hardware characteristics or ID generation methods proposed, in previous works. In the present study, the security of the IC and supply chain is improved by keeping hardware equal but designing a stronger dielet-server mutual authentication. Therefore, the parameters studied in the CNTR-SHIELD authentication protocol will be compared. The exchange between the smartphone and

the server is not evaluated in either work, as it is wireless communication over a high bandwidth communication through the Internet.

Table 5.2 Performance comparison of the studied three SHIELD protocols

| Protocol Compared | Number of steps | Number of bits | Number of enc. | Memory (Bits) | Extra Counter | Latency (n times) |
|---|---|---|---|---|---|---|
| SHIELD | 5 | 448 | 2 | 178 | 0 | 2 |
| CNTR-SHIELD | 10 | 258 | 1 | 178 | 1 | 1 |
| SBMAPS | 10 | 934 | 7 | 1074 | 0 | 3 |

Table 5.2 shows the compared properties of the three protocols. The first is the number of steps taken by the protocols. SHIELD finishes verifying the dielet in 5 steps, without a mutual authentication. CNTR-SHIELD finishes verifying the dielet in 4 phases and 10 steps, also without a mutual authentication. SBMAPS also finishes in 10 steps, but with a strong 3-phase mutual authentication. Therefore, SBMAPS and CNTR-SHIELD have equal step sizes. By unrealistically declaring the smartphone as secure, CNTR-SHIELD exchanges 258 bits. SBMAPS exchanges use 934 bits, almost doubling the number used by the original SHIELD. As observed, SBMAPS passes more information in an equal number of steps, compared to CNTR-SHIELD. Obviously, information exchange is not efficient in CNTR-SHIELD. Similarly, our SBMAPS dielet executes a total of 7 encryption-decryption operations to complete one round. SHIELD and CNTR-SHIELD dielets carry out much fewer encryption operations, at the expense of passing plaintext data. The above three parameters impact the energy consumed by the protocols. The amount of energy consumed in message exchange is obviously higher in SBMAPS. But, dielet's energy efficiency is not a disadvantage in the present context, because it is energized by the near field of the smartphone and not by a battery source. The only disadvantage of SBMAPS is the extended time taken by the dielet operations, which is a few extra microseconds for a processor with a megahertz clock. While CNTR-SHIELD introduces an additional counter to overcome known ciphertext attacks of constant key protocols; the SBMAPS dielet instead uses 1074 bits of additional memory. Our strategy is increasing keyspace; hence increasing security, at the expense of extra memory. But, the extra

amount of memory needed does not critically burden the die area, according to calculations made in CNTR-SHIELD, based on work (Jeong et al., 2012). Apart from the above, SBMAPS requires no additional hardware; therefore, has no critical hardware disadvantages. While SHIELD contacts the server a couple of times, CNTR-SHIELD contacts once and loses security for that matter. SBMAPS eliminates the smartphone involvement by communicating with the server 3 times more than SHIELD, to mutually authenticate the dielet and the server. In other words, our protocol has roughly 3 times more latency than SHIELD, in verifying a dielet. This is a typical performance and security trade-off in computer science. This disadvantage can be diminished by increasing the communication speed between the smartphone and the server. It is obvious that our proposed SBMAPS takes a longer time than previous protocols, due to strong mutual authentication. But we trust that while making decisions security officers consider the performance/security ratio of a protocol. As such, SBMAPS is the only strong mutual authentication protocol with no weaknesses to known attacks, complying with the primary goal of SHIELD.

## 5.4 Security Analysis of STRAP

### *5.4.1 Informal Security Analysis of STRAP*

The informal security analysis of STRAP for four attack types is given below. As informal analyses are subjective, they are kept short.

#### *5.4.1.1 DoS Attack*

Message blocking, altering, or loss is not a threat for STRAP, because:

i. If phases 1 or 2 are blocked, authentication fails and the protocol stops or times out,

ii. If the key number is altered, RFID authentication fails, and the protocol halts,

iii. If phases 1 or 2 are disrupted, protocol times out.

#### *5.4.1.2 Full Disclosure Attack*

Full disclosure cannot succeed against STRAP, as none of the messages are passed in plaintext. The $T_j$_ID is the only information in plaintext, which is required for tracking tags. None of the encrypted messages exchanged can be exposed, since the shared keys and private keys are secret. Even if a rogue employee enters his/her credentials and hands the smartphone to an attacker, the messages cannot be decrypted. Because, the smartphone is a time-checked message relaying device, unable to see any secrets.

#### *5.4.1.3 Lookup Table Attack*

It is not possible to construct a plaintext-ciphertext pair table in STRAP, simply because no plaintext messages are passed. Additionally, the passed messages are always variable, as in the critical sensor status bits and time combination. Furthermore, messages are encrypted with a new session key, every round. The result notification of the authentication to the smartphone at the end of every round is also encrypted and variable.

#### *5.4.1.4 Replay Attack*

None of the messages exchanged are re-playable because the encrypted messages change every round. In addition, the authentication keys and the generated session keys used for encrypting the messages change every round.

### *5.4.2 Formal Security Analysis of STRAP*

We have analyzed STRAP with both Scyther and AVISPA tools. The analysis results of both tools are presented in the next sub-sections.

#### 5.4.2.1 Verification of STRAP using AVISPA

HLPSL implementation of STRAP is based on three roles: *SPhone*, *Server*, and *Tag*. Figure 5.11 shows the SPhone role codes of STRAP in the HLPSL language.

```
%%-----------------ROLE SMARTPHONE----------------------------
role role_SPhone(
    SPhone, Server, Tag : agent,
    Usr_ID, CPU_ID          : text,
    Usr_Key, KeySpT         : symmetric_key,
    Hash                    : hash_func,
    ReadReq                 : nat,
    Usr_PK, Srv_PK          : public_key,
    SND, RCV                : channel(dy)
) played_by SPhone def=
local
    State           : nat,
    Sp_t, Result    : text,
    M10             : {{function}_public_key}_inv(public_key)
init State := 0
transition
    %%------------- Phase-1 starts here ----------------------
    0.  State = 0 /\ RCV(start) =|>
        State' := 1 /\ Sp_t' := new()
        /\ SND(Usr_ID.{SPhone.Server.CPU_ID.Sp_t'}_Usr_Key)
        /\ secret(CPU_ID, sec_cpuid_sps, {SPhone, Server})
        /\ secret(Sp_t', sec_spt_sps, {SPhone, Server})
    1.  State = 1 /\ RCV({{Hash(Server.SPhone.CPU_ID'.Sp_t)}
        _Usr_PK}_inv(Srv_PK)) =|>

        %%-------------Phase-2 starts here ------------------
        State' := 2 /\ SND({SPhone.Tag.ReadReq}_KeySpT)
        %%SPhone checks that it receives the same timestamp
        %%that it sent at step 0
        /\ request(SPhone, Server, auth_spt_sps, Sp_t)

    %%-------------Phase-3 starts here ---------------------
    2.  State = 2 /\ RCV(Server.SPhone.M10') =|>
        State' := 3
end role
```

Figure 5.11 SPhone role codes of STRAP in HLPSL

As shown in Figure 5.11, a role named *role_SPhone* is defined first. The steps of the protocol executed on the smartphone side are placed in this role. The role definition has parameters of *SPhone*, *Server*, *Tag* in *agent* type, *Usr_ID*, *CPU_ID* in *text* type, *Usr_Key*, *KeySpT* in *symmetric_key* type, *Hash* in *hash_func* type, *ReadReq* in *nat* type, *Usr_PK*, *Srv_PK* in *public_key* type, and *SND*, *RCV* in *channel(dy)* type. The

attribute *dy* of the *channel* type represents the intruder model (Dolev-Yao) of the channel. All the variables in the HLPSL language start with a capital letter.

In Figure 5.11, the parameter *SPhone* is placed in the *played_by* section, which means that *SPhone* represents the name of the agent that plays the role *role_SPhone*. In the *local* section, local variables that will be used in the scope of the *role_SPhone* role are defined. The *state* variable showing the protocol steps is set to zero in the *init* section of the role *role_SPhone*.

The *transition* section contains the set of transitions that shows the received and sent messages. Each transition has a name distinguishing it from other transitions. The /\ operator represents that the specified actions are executed together.

The *start* message is only available in one role and specifies the beginning of the execution of the protocol. In transition 0, if the *State* variable is set to 0 and the *start* message is received through the *RCV* channel then, the *State* variable is set to 1, and the *Sp_t* variable keeping the time of the *SPhone* is set to a new value. The tuple *{CPU_ID*, *Sp_t}* is encrypted with the symmetric key *Usr_Key* and sent to *Server* through the *SND* channel.

The *secret* property indicates that the secret information is not related to an unauthorized party throughout the message exchange in the protocol. The *secret* property is used to control the secrecy of *CPU_ID* and *Sp_t'* to be transmitted between the *Reader* agent and the *Server* agent.

In transition 1, if the *State* variable is set to 1 and the hashed value of the tuple *{CPU_ID', Sp_t}* encrypted with *Usr_PK* public key and signed with *inv(Srv_PK)* private key is received through the *RCV* channel, then the *State* variable is set to 1 and *ReadReq* variable encrypted with the symmetric key KeySpT is sent to *Tag* through *SND* channel. The *request* event together with the corresponding *witness* event is used to check strong authentication. With the *request* event here, *SPhone* checks that it

receives the same timestamp *Sp_t* that it sent at step 0 (transition 0) and it was not replayed from a previous session.

In transition 2, if the *State* variable is set to 2 and the *M10* variable sent by *Server* is received through the *RCV* channel, then the State variable is set to 3 and the role of the agent *SPhone* ends here.

```
%%-------------ROLE SERVER-----------------------------------
role role_Server(
    SPhone, Server, Tag : agent,
    Usr_Key, KeyTS          : symmetric_key,
    Hash                    : hash_func,
    Usr_PK, Srv_PK          : public_key,
    CurrHop                 : nat,
    SND, RCV: channel(dy)
) played_by Server def=

local
    State               : nat,
    KeySession          : symmetric_key,
    Ts, Usr_ID, CPU_ID, Sp_t, T_ID, NonS, Temp3, SDTA,
    Result, Old_Ts      : text,
    M1                  : {text.text}_symmetric_key,
    M3, M4, M5, M6      : {text}_symmetric_key,
    M7                  : {text.text.hash_func}_symmetric_key,
    M8, M9              : {message}_symmetric_key,
    M10                 : {{function}_public_key}_inv(public_key),
    Sgn                 : message

init State := 0
transition
    %%-------------Phase-1 starts here -------------------------
    0.  State = 0 /\ RCV(Usr_ID'.{SPhone.Server.CPU_ID'.Sp_t'}_Usr_Key) =|>
        State' := 1
        /\ SND({{Hash(Server.SPhone.CPU_ID'.Sp_t')}_Usr_PK}_inv(Srv_PK))
        /\ secret(CPU_ID', sec_cpuid_ssp, {SPhone, Server})
        /\ secret(Sp_t', sec_spt_ssp, {SPhone, Server})
        %%Server hopes that Sp_t will permit to authenticate him
        /\ witness(Server, SPhone, auth_spt_sps, Sp_t')
```

Figure 5.12 Server role codes of STRAP in HLPSL (Part-1)

The first part of the *Server* role codes of STRAP in HLPSL is shown in Figure 5.12. The steps of the protocol executed on the server-side are placed in this role. The role definition has parameters of *SPhone*, *Server*, *Tag* in agent type, *Usr_Key*, *KeyTS* in symmetric_key type, *Hash* in hash_func type, *Usr_PK*, *Srv_PK* in public_key type, *CurrHop* in nat type, *Sgn* in message type and *SND*, *RCV* in channel(dy) type.

The parameter *Server* is placed in the *played_by* section, which means that *Server* represents the name of the agent that plays the role *role_Server*. In the *local* section, local variables that will be used in the scope of the *role_Server* role are defined. The *state* variable showing the protocol steps is set to zero in the *init* section of the role *role_Server*.

In transition 0 on Figure 5.12, if the *State* variable is set to 0, and the *Usr_ID'* variable and the tuple *{CPU_ID', Sp_t'}* encrypted with the *Usr_Key* symmetric key is received through the *RCV* channel then, the *State* variable is set to 1 and the hashed value of the tuple *{CPU_ID', Sp_t'}* encrypted with *Usr_PK* public key and signed with *inv(Srv_PK)* private key is sent to *SPhone* through the *SND* channel. The *secret* property is used to control the secrecy of *CPU_ID'* and *Sp_t'* to be transmitted between the *SPhone* agent and the *Server* agent. With the *witness* event here, *Server* hopes that *Sp_t'* will permit to authenticate it.

In transition 1 on Figure 5.13, if the *State* variable is set to 1 and the *T_ID'* variable encrypted with the *KeyTS* symmetric key by the *Tag* agent is received through the *RCV* channel then, the *State* variable is set to 2, the *NonS'* nonce is created. *NonS'* encrypted with the *KeyTS* symmetric key is assigned to the *M3* variable, and the *CurrHop* variable with the *M3* variable is sent to the *Tag* agent through the *SND* channel. The *secret* property is used to control the secrecy of *NonS'* to be transmitted between the *Server* agent and the *Dielet* agent.

In transition 2, if the *State* variable is set to 2, and XOR'ed *{Tem3', M4'}* tuple encrypted with the *Key_TS* symmetric key, and the *M4'* variable are received through the *RCV* channel between the Server and the Tag agents then, the *State* variable is set to 3 and XOR'ed *{Temp3', M4'}* tuple encrypted with the *Key_TS* is assigned to the *M5'* variable. XOR'ed *{M5', Temp3'}* tuple encrypted with the *Key_TS* is assigned to the *M6'* variable. XOR'ed *{NonS, Temp3'}* tuple is assigned to the *KeySession'* variable and a new timestamp is assigned to the *Ts'* variable. The hashed value of the *{T_ID, CurrHop, Ts'}* tuple is assigned to the *Sgn'* variable. The *{Temp3', Ts', Sgn'}* tuple

encrypted with the *KeySession'* session key is assigned to the *M7'* variable, and *{M6', M7'}* tuple is sent to the *Dielet* agent through the *SND* channel.

```
    %%-------------Phase-2 starts here -------------------------
    1. State = 1 /\ RCV({Tag.Server.T_ID'}_KeyTS) =|>
       State' := 2 /\ NonS' := new() /\ M3' := {NonS'}_KeyTS
       /\ SND(Server.Tag.CurrHop.M3')
       /\ secret(NonS', sec_nons_st, {Server, Tag})

    2. State = 2 /\ RCV(Tag.Server.M4'.{xor(Temp3', M4')}_KeyTS) =|>
       State' := 3 /\ M5' := {xor(Temp3', M4')}_KeyTS
       /\ M6' := {xor(M5', Temp3')}_KeyTS
       /\ KeySession' := xor(NonS, Temp3') /\ Ts' := new()
       /\ Sgn' := Hash(T_ID.CurrHop.Ts')
       /\ M7' := {Temp3'.Ts'.Sgn'}_KeySession'
       /\ SND(Server.Tag.M6'.M7')
       /\ secret(Temp3', sec_temp3_st, {Server, Tag})
       /\ secret(NonS, sec_nonsl_st, {Server, Tag})
       /\ secret(Ts', sec_ts_st, {Server, Tag})
       /\ secret(Sgn', sec_sgn_st, {Server, Tag})
       %%Server checks that it receives the same nonce that it sent at step 1
       /\ request(Server, Tag, auth_nons_st, NonS)
       %%Server hopes that NonT(Temp3) will permit to authenticate him
       /\ witness(Server, Tag, auth_nont_ts, Temp3')

    %%-------------Phase-3 starts here ----------------------
    3. State = 3 /\ RCV(Tag.Server.M8'.{xor({xor(Old_Ts', Ts)}_KeySession',
           SDTA')}_KeySession') =|>
       State' := 4 /\ Result' := new()
       /\ M10' := {{Hash(Result'.Sp_t')}_Usr_PK}_inv(Srv_PK)
       /\ SND(Server.SPhone.M10')
       /\ secret(Sp_t', sec_spt_ssp, {Server, SPhone})
       /\ secret(Result', sec_result_ssp, {Server, SPhone})
       %%Server checks that it receives the same timestamp
       %%that it sent at step 2
       /\ request(Server, Tag, auth_ts_st, Old_Ts')
end role
```

Figure 5.13 Server role codes of STRAP in HLPSL (Part-2)

The *secret* property is used to control the secrecy of *Temp3', NonS, Sgn',* and *Ts'* to be transmitted between the *Server* agent and the *Tag* agent. With the *request* event, *Server* checks that it receives the same nonce *NonS* that it sent at step 1 (transition 1) and it was not replayed from a previous session. With the *witness* event here, *Server* hopes that *Temp3'* will permit it to authenticate it.

In transition 3, if the *State* variable is set to 3, and the M8' variable and *{xor({xor(Old_Ts', Ts)}_KeySession', SDTA')}_KeySession'* expression (*M9'*) sent by

the *Tag* agent are received through the *RCV* channel then, the *State* variable is set to 4 and a new value is assigned to the *Result'* variable. The hashed tuple *{Result', Sp_t'}* encrypted with *Usr_PK* public key and signed with *inv(Srv_PK)* private key is assigned to the *M10'* variable. The *M10'* variable is sent to the *SPhone* agent through the *SND* channel.

The *secret* property is used to control the secrecy of *Sp_t'*, and *Result'* to be transmitted between the *Server* agent and the *SPhone* agent. With the *request* event, *Server* checks that it receives the same timestamp *Old_Ts'* that it sent at step 2 (transition 2) and it was not replayed from a previous session.

```
%%------------------ROLE RFID TAG--------------------------------
role role_Tag(
    SPhone, Server, Tag : agent,
    ReadReq                 : nat,
    KeySpT, KeyTS           : symmetric_key,
    T_ID, Old_Ts            : text,
    SND, RCV                : channel(dy)
) played_by Tag def=

local
    State, ReadReq1, CurrHop    : nat,
    KeySession                  : symmetric_key,
    M3, M4, M5, M6              : {text}_symmetric_key,
    M7                          : {text.text.hash_func}_symmetric_key,
    Temp0, NonT, Tss, SDTA  : text,
    M8, M9                      : {message}_symmetric_key,
    Sgn                         : message

init State := 0
transition
    %%-------------Phase-2 starts here -------------------------
    0.  State = 0 /\ RCV({SPhone.Tag.ReadReq1'}_KeySpT)
        /\ ReadReq = ReadReq1' =|>
        State' := 1 /\ SND({Tag.Server.T_ID}_KeyTS)
        /\ secret(T_ID, sec_tid_ts, {Tag, Server})
```

Figure 5.14 Tag role codes of STRAP in HLPSL (Part-1)

The first part of the *Tag* role codes of SBMAPS in HLPSL is shown in Figure 5.14. The steps of the protocol executed on the tag side are placed in this role. The role definition has parameters of *SPhone*, *Server*, *Tag* in agent type, *ReadReq* in *nat* type, *KeySpT* and *KeyTS* in symmetric_key type, *T_ID* and *Old_Ts* in text type, *Sgn* in message type and *SND*, *RCV* in *channel(dy)* type.

The parameter *Tag* is placed in the *played_by* section, which means that *Tag* represents the name of the agent that plays the role *role_Tag*. In the *local* section, local variables that will be used in the scope of the *role_Tag* role are defined. The *state* variable showing the protocol steps is set to zero in the *init* section of the role *role_Tag*.

In transition 0 on Figure 5.14, if the *State* variable is set to 0, the *ReadReq1'* variable encrypted with the *KeySpT* symmetric key sent by the *SPhone* agent is received through the *RCV* channel, and the *ReadReq* variable is equal to the *ReadReq1'* variable then, the *State* variable is set to 1 and the *T_ID* variable encrypted with the *KeySpT* symmetric key is sent to the *Server* agent through the *SND* channel. The *secret* property is used to control the secrecy of *T_ID* to be transmitted between the *Server* agent and the *Tag* agent.

In transition 1 on Figure 5.15, if the *State* variable is set to 1, and the *CurrHop'* variable and the *Temp0'* variable encrypted with the *KeyTS* symmetric key by the *Server* agent is received through the *RCV* channel then, the *State* variable is set to 2, and *Temp0'* variable encrypted with the *KeyTS* symmetric key is assigned to *M4'* variable. A new nonce is assigned to the *NonT'* variable. The XOR'ed *{NonT', M4'}* tuple encrypted with the *KeyTS* symmetric key is assigned to the *M5'* variable, and the XOR'ed *{NonT', Temp0'}* tuple is assigned to the *KeySession'* variable. The *{M4', M5'}* tuple is sent to the *Server* agent through the SND channel.

The *secret* property is used to control the secrecy of *Temp0'*, and *NonT'* to be transmitted between the *Server* agent and the *Tag* agent. With the *witness* event here, *Tag* hopes that *Temp0'* (*NonS*) will permit to authenticate it.

```
    1.  State = 1 /\ RCV(Server.Tag.CurrHop'.{Temp0'}_KeyTS) =|>
    %%---Tag.Temp0 = Server.NonS----
        State' := 2 /\ M3' := {Temp0'}_KeyTS
        /\ M4' := {xor(Temp0', M3')}_KeyTS
        /\ NonT' := new() /\ M5' := {xor(NonT', M4')}_KeyTS
        /\ KeySession' := xor(NonT', Temp0')
        /\ SND(Tag.Server.M4'.M5')
        /\ secret(Temp0', sec_temp0_ts, {Tag, Server})
        /\ secret(NonT', sec_nont_ts, {Tag, Server})
        %%Tag hopes that NonS(Temp0) will permit to authenticate him
        /\ witness(Tag, Server, auth_nons_st, Temp0')

    2.  State = 2 /\ RCV(Server.Tag.M6'.{NonT.Tss'.Sgn'}_KeySession') =|>
    %%-------------Phase-3 starts here -----------------------
    State' := 3
        /\ SDTA' := new()
        /\ M8':= {xor(Old_Ts, Tss')}_KeySession'
        /\ M9' := {xor(M8', SDTA')}_KeySession'
        /\ SND(Tag.Server.M8'.M9')
        /\ secret(Old_Ts, sec_oldts_ts, {Tag, Server})
        /\ secret(Tss', sec_tss_ts, {Tag, Server})
        /\ secret(SDTA', sec_sensors_ts, {Tag, Server})
        /\ secret(Sgn', sec_sgn_ts, {Tag, Server})
        %%Tag checks that it receives the same nonce that it sent at step 1
        /\ request(Tag, Server, auth_nont_ts, NonT)
        %%Tag hopes that Old_Ts will permit to authenticate him
        /\ witness(Tag, Server, auth_ts_st, Old_Ts)
end role
```

Figure 5.15 Tag role codes of STRAP in HLPSL (Part-2)

In transition 2 in Figure 5.15, if the *State* variable is set to 2 the *M6'* variable and the *{NonT, Tss', Sgn'}* tuple encrypted with the *KeySession'* symmetric key by the *Server* agent is received through the *RCV* channel then, the *State* variable is set to 3 and a new value is assigned to the *SDTA'* variable. The XOR'ed *{Old_Ts, Tss'}* tuple encrypted with the *KeySession'* symmetric key is assigned to the *M8'* variable. The XOR'ed *{ M8', SDTA'}* tuple encrypted with the *KeySession'* symmetric key is assigned to the *M9'* variable. The *{M8', M9'}* tuple is sent to the *Server* agent through the *SND* channel.

The *secret* property is used to control the secrecy of *Old_Ts*, *Tss', Sgn',* and *SDTA'* to be transmitted between the *Server* agent and the *Tag* agent. With the *request* event, *Tag* checks that it receives the same nonce *NonT* that it sent at step 1 (transition 1) and it was not replayed from a previous session. With the *witness* event here, *Tag* hopes that *Old_Ts* will permit to authenticate it.

The STRAP protocol is analyzed using the OFMC backend of AVISPA SPAN Tool. The analysis result of the proposed protocol is shown in Figure 5.16. Line 4 of the OFMC result shows that the proposed SBMAPS protocol is 'SAFE' against attacks tested by AVISPA. Details of the analysis are summarized after the 'SAFE' declaration of the tool. The tool confirms that the protocol had a bounded number of checker sessions. The location of the protocol file is also given in the results to ensure the user about the HLPSL code analyzed. 'as_specified' declaration of the tool confirms that the security goals described in the HLPSL code have been reached.

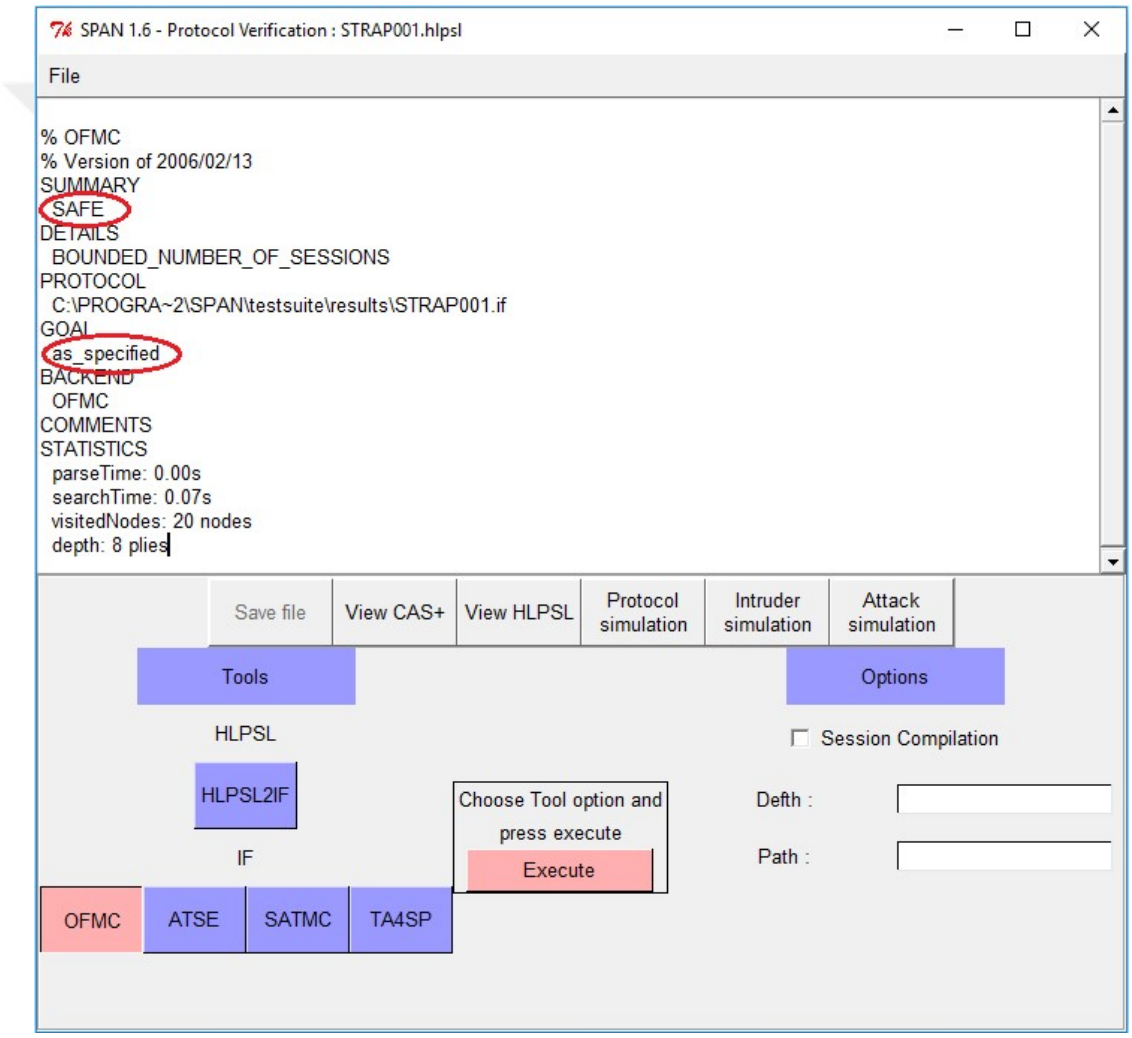



Figure 5.16 Security evaluation result of STRAP with AVISPA Span Tool

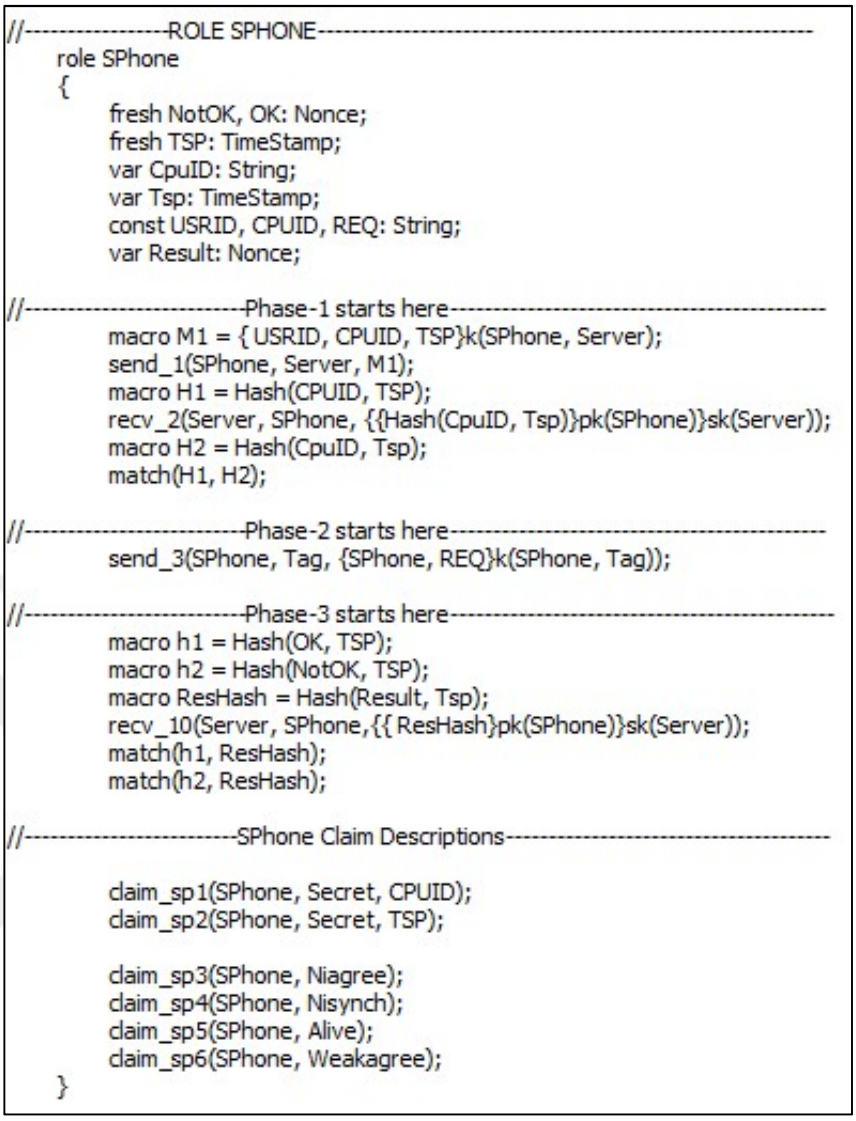
#### *5.4.2.2 Verification of STRAP using Scyther Tool*

STRAP was coded using the Security Protocol Description Language (SPDL). During the analysis, the smartphone equipped with an RFID reader was given the *SPhone* role, the server was given *Server* role, and the IoT device with RFID tag was given the *Tag* role. The SPDL code of the SPhone role is shown in Figure 5.17 SPhone role codes of STRAP in SPDL. Local variables and constants are declared after the role definition. The *NotOK* and *OK* variables are declared in *Nonce* type and assigned fresh values. The *TSP* variable is declared in *TimeStamp* user-defined type and assigned a fresh value. The *CpuID* variable in *String* user-defined type, the *Tsp* variable in *TimeStamp* type, and the *Result* variable in *Nonce* type are declared. The *USRID*, *CPUID*, and *REQ* constants are declared in String type.

In phase-1 of role SPhone, the *{USRID, CPUID, TSP}* tuple is encrypted with the *k* symmetric key shared between SPhone and Server and assigned to the *M1* macro. The *M1* macro is sent to *Server* from *SPhone* by the *send_1* command. After sending *M1*, the hash value of the *{CPUID, TSP}* tuple is calculated and assigned to the *H1* macro. The hashed value of the *{CpuID, Tsp}* tuple encrypted with *SPhone's* public key and signed with *Server's* private key is received by the *recv_2* command. Then the hashed value of the *{CpuID, Tsp}* tuple is assigned to the *H2* macro and the *H1* and *H2* macros are compared for a match.

In phase-2 of role SPhone, the *{SPhone, REQ}* tuple encrypted with the *k* symmetric key shared between *SPhone* and *Tag* is sent from *SPhone* to *Tag* by the *send_3* command.

In phase-3 of role SPhone, the hashed value of the *{OK, TSP}* tuple is assigned to the *h1* macro, the hashed value of the *{NotOK, TSP}* tuple is assigned to the *h2* macro, and the hashed value of the *{Result, Tsp}* tuple is assigned to the *ResHash* macro. The *ResHash* macro encrypted with the *SPhone's* public key and signed with the *Server's* private key is received from *Server* by the *recv_10* command. Then the *h1* and *h2* macros are compared with the *ResHash* macro for a match.



```
//----------------ROLE SPHONE-----------------------------------------------------------
    role SPhone
    {
        fresh NotOK, OK: Nonce;
        fresh TSP: TimeStamp;
        var CpuID: String;
        var Tsp: TimeStamp;
        const USRID, CPUID, REQ: String;
        var Result: Nonce;

//-------------------------Phase-1 starts here-----------------------------------------
        macro M1 = {USRID, CPUID, TSP}k(SPhone, Server);
        send_1(SPhone, Server, M1);
        macro H1 = Hash(CPUID, TSP);
        recv_2(Server, SPhone, {{Hash(CpuID, Tsp)}pk(SPhone)}sk(Server));
        macro H2 = Hash(CpuID, Tsp);
        match(H1, H2);

//-------------------------Phase-2 starts here-----------------------------------------
        send_3(SPhone, Tag, {SPhone, REQ}k(SPhone, Tag));

//-------------------------Phase-3 starts here-----------------------------------------
        macro h1 = Hash(OK, TSP);
        macro h2 = Hash(NotOK, TSP);
        macro ResHash = Hash(Result, Tsp);
        recv_10(Server, SPhone,{{ResHash}pk(SPhone)}sk(Server));
        match(h1, ResHash);
        match(h2, ResHash);

//-------------------------SPhone Claim Descriptions-----------------------------------

        claim_sp1(SPhone, Secret, CPUID);
        claim_sp2(SPhone, Secret, TSP);

        claim_sp3(SPhone, Niagree);
        claim_sp4(SPhone, Nisynch);
        claim_sp5(SPhone, Alive);
        claim_sp6(SPhone, Weakagree);
    }
```

Figure 5.17 SPhone role codes of STRAP in SPDL

The SPDL code of the Server role is shown in Figure 5.18 Server role codes of STRAP in SPDL. Local variables and constants are declared after the role definition. The *NonS*, *TS*, *Result*, and *CurrHop* variables are declared in *Nonce* type and assigned fresh values. The *Tsp* variable is declared in *TimeStamp* user-defined type. The *CpuID, UsrID,* and *Tid* variables in *String* user-defined type, the *Tss* variable in *String* type, and the *NonSS*, *NonTT*, *OldTs*, *Ts*, and *SDTAS* variables in *Nonce* type are declared. The *CPUID* and *TID* constants are declared in String type.

```
//-----------------ROLE SERVER-----------------------------------------------------------------
    role Server
    {
        fresh Tss: String;
        var CpuID, UsrID, Tid: String;
        const CPUID, TID: String;
        fresh NonS,  TS, Result, CurrHop: Nonce;
        var NonSS, NonTT, OldTs, Ts, SDTAS: Nonce;
        var temp3,  M4: Ticket;
        var Tsp: TimeStamp;

//-------------------------Phase-1 starts here-----------------------------------------
        recv_1(SPhone, Server, {UsrID, CpuID, Tsp}k(SPhone, Server));
        match(CPUID, CpuID);
        macro M2 = {{Hash(CpuID, Tsp)}pk(SPhone)}sk(Server);
        send_2(Server, SPhone, M2);

//-------------------------Phase-2 starts here-----------------------------------------
        recv_4(Tag, Server, {Tid}k(Tag, Server));
        match(TID, Tid);
        macro M3 = {NonS}k(Tag, Server);
        send_5(Server, Tag, CurrHop, M3);
        recv_6(Tag, Server, M4, {fxor(NonTT, M4)}k(Tag, Server));
        macro M5 = {fxor(NonTT, M4)}k(Tag, Server);
        macro temp1 = fxor(NonS, M3);
        match(NonSS, temp1);
        macro temp2 = fxor(NonTT, M4); // temp2 = NonT x M4
        match(temp3, fxor(temp2, M4)); // temp3 = NonT
        macro KeyS = fxor(NonS, NonTT);
        macro M6 = {fxor(M5, temp3)}k(Tag, Server);
        send_7(Server, Tag, M6);

//-------------------------Phase-3 starts here-----------------------------------------

        macro Sgn1 = Hash(TID, CurrHop, Tss);
        macro M7 = {NonTT, TS,  Sgn1}KeyS;
        send_8(Server, Tag, M7);
        macro M8 = {fxor(OldTs, Ts)}KeyS;
        macro M9 = {fxor(M8, SDTAS)}KeyS;
        recv_9(Tag, Server,M8, M9);
        macro temp6 = fxor(OldTs, Ts);
        macro temp7 =  fxor(temp6, Ts);// temp7 = OldTs
        macro temp8 =  fxor(M8, SDTAS);
        macro temp9 =  fxor(M8, temp8); // temp9 =? SDTAS
        match(temp9, SDTAS);
        macro ResHash = Hash(Result, Tsp);
        send_10(Server, SPhone,{{ ResHash}pk(SPhone)}sk(Server));

//---------------------Server Claim Descriptions----------------------------------

        claim_s1(Server, Secret, CpuID);
        claim_s2(Server, Secret, Tsp);

        claim_s3(Server, Niagree);
        claim_s4(Server, Nisynch);
        claim_s5(Server, Alive);
        claim_s6(Server, SKR, KeyS);
        claim_s7(Server, Weakagree);
    }
```

Figure 5.18 Server role codes of STRAP in SPDL

In phase-1 of role Server, the *{UsrID, CpuID, Tsp}* encrypted with the *k* symmetric key shared between *SPhone* and *Server* is received from *SPhone* by the *recv_1* command. Then *CPUID* and *CpuID* are compared for a match. The hashed value of the *{CpuID, Tsp}* tuple is assigned to the *M2* macro and the *M2* macro is sent to SPhone by the *send_2* command.

In phase-2 of role Server, *Tid* encrypted with the *k* symmetric key shared between *Tag* and *Server* is received from *Tag* by the *recv_4* command. Then *TID* and *Tid* are compared for a match. The *NonS* nonce encrypted with the *k* symmetric key shared between *Tag* and *Server* is assigned to the *M3* macro and the *M3* macro with *CurrHop* is sent to *Tag* by the *send_5* command. The XOR'ed value of the *{NonTT, M4}* tuple that is encrypted with the *k* symmetric key shared between *Server* and *Tag*, and M4 itself is received from *Tag* by the *recv_6* command. The XOR'ed value of the *{NonTT, M4}* tuple encrypted with the *k* symmetric key shared between *Server* and *Tag* is assigned to *M5* macro. The XOR'ed value of *NonS* and *M3* is assigned to the *temp1* macro. Then *NonSS* and *temp1* are compared for a match. The XOR'ed value of *NonTT* and *M4* is assigned to the *temp2* macro. The XOR'ed value of *temp2* and *M4* is compared with *temp3* for a match. Then the XOR'ed value of *NonS* and *NonTT* is assigned to the *KeyS* macro that will be used as the session key later. The XOR'ed value of the *{M5, temp3}* tuple encrypted with the *k* symmetric key shared between *Server* and *Tag* is assigned to the *M6* macro and the *M6* macro is sent to *Tag* by the *send_7* command.

In phase-3 of role Server, the hashed value of the *{TID, CurrHop, Tss}* tuple is assigned to the *Sgn1* macro. The *{NonTT, TS, Sgn1}* tuple encrypted with the *KeyS* session key is assigned to the *M7* macro and the *M7* macro is sent to *Tag* by the *send_8* command. The XOR'ed value of *OldTs* and *Ts* encrypted with *KeyS* is assigned to the *M8* macro and the XOR'ed value of *M8* and *SDTAS* encrypted with *KeyS* is assigned to the *M9* macro. The *M8* and *M9* macros are received from *Tag* by the *recv_9* command. The XOR'ed value of *OldTs* and *Ts* is assigned to the *temp6* macro and the XOR'ed value of *temp6* and *Ts* is assigned to the *temp7* macro. Then *temp7* and *OldTs* are compered for a match. The XOR'ed value of *M8* and *temp8* is assigned to the *temp9*

macro. The *temp9* macro and *SDTAS* are compared for a match. The hashed value of the *{Result, Tsp}* tuple is assigned to the *ResHash* macro and the *ResHash* macro encrypted with *SPhone's* public key and signed with the *Server's* private key is sent to *SPhone* by the *send_10* command.

```
//-------------------ROLE TAG-------------------------------------------------
    role Tag
    {
        const REQ, TID: String;
        var Req: String;
        var NonS, NonTT, Ts, CurrHop: Nonce;
        fresh NonT, OldTs, SDTA: Nonce;
        var  Sgn: Ticket;

//-------------------------Phase-2 starts here-----------------------------
        recv_3(SPhone, Tag, {SPhone, Req}k(SPhone, Tag));
        match(REQ, Req);
        send_4(Tag, Server, {TID}k(Tag, Server));
        recv_5(Server, Tag, CurrHop, {NonS}k(Tag, Server));
        macro M3 = {NonS}k(Tag, Server);
        macro M4 = {fxor(NonS, M3)}k(Tag, Server);
        macro M5 = {fxor(NonT, M4)}k(Tag, Server);
        send_6(Tag, Server, M4, M5);
        recv_7(Server, Tag, {fxor(M5, NonTT)}k(Tag, Server));
        macro KeyS = fxor(NonS, NonTT);

//-------------------------Phase-3 starts here-----------------------------
        recv_8(Server, Tag, {NonTT, Ts, Sgn}KeyS);
        macro M8 = {fxor(OldTs, Ts)}KeyS;
        macro M9 = {fxor(M8, SDTA)}KeyS;
        send_9(Tag, Server,M8, M9);


//-------------------Tag  Claim Descriptions---------------------------
        claim_t1(Tag, Secret, OldTs);
        claim_t2(Tag, Secret, NonT);
        claim_t3(Tag, Secret, SDTA);
        claim_t4(Tag, Secret, TID);
        claim_t5(Tag, Secret, NonS);
        claim_t6(Tag, Secret, Ts);

        claim_t7(Tag, Niagree);
        claim_t8(Tag, Nisynch);
        claim_t9(Tag, Alive);
        claim_t10(Tag, SKR, KeyS);
        claim_t11(Tag, Weakagree);
    }
```

Figure 5.19 Tag role codes of STRAP in SPDL

The SPDL code of the Tag role is shown in Figure 5.19. Local variables and constants are declared after the role definition. The *NonT*, Old*Ts*, and *SDTA* variables

are declared in *Nonce* type and assigned fresh values. The *Req* variable is declared in *String* user-defined type, the *NonS, NonTT, Ts,* and *CurrHop* variables in *Nonce* type. The *REQ* and *TID* constants are declared in String type.

Since there is no phase-1 of the Tag role, it starts from phase-2. In phase-2 of role Tag, the *{SPhone, Req}* tuple encrypted with the *k* symmetric key shared between *SPhone* and *Tag* is received from *SPhone* by the *recv_3* command, and *REQ* and *Req* are compared for a match. *TID* encrypted with the *k* symmetric key shared between *Server* and *Tag* is sent to *Server* by the *send_4* command and *CurrHop* with *NonS* encrypted with the *k* symmetric key shared between *Server* and *Tag* is received from *Server* by the *recv_5* command. *NonS* encrypted with the *k* symmetric key shared between *Server* and *Tag* is assigned to the *M3* macro, the XOR'ed value of *NonS* and *M3* encrypted with the *k* symmetric key shared between *Server* and *Dielet* is assigned to the *M4* macro, and the XOR'ed value of *NonT* and *M4* encrypted with the *k* symmetric key shared between *Server* and *Tag* is assigned to the *M5* macro. Then the *M4* and *M5* macros are sent to *Server* by the *send_6* command. The XOR'ed value of *M5* and *NonTT* encrypted with the *k* symmetric key shared between *Server* and *Tag* is received from *Server* by the *recv_7* command. The XOR'ed value of *NonS* and *NonDD* is assigned to the *KeyS* macro. Then the *{NonTT, Ts, Sgn}* tuple encrypted with the *KeyS* session key is received from *Server* by the *recv_8* command. The XOR'ed value of *OldTs* and *Ts* encrypted with the *KeyS* session key is assigned to the *M8* macro and the XOR'ed value of *M8* and *SDTA* encrypted with the *KeyS* session key is assigned to the *M9* macro. Then the *M8* and *M9* macros are sent to *Server* by the *send_9* command.

The Scyther Tool automatically analyzes submitted schemes, carefully observing the proposed scheme's reaction to the known attacks. STRAP was coded using the Security Protocol Description Language (SPDL). During the analysis, the smartphone equipped with an RFID reader was given the *SPhone* role, the server was given the *Server* role, and the IoT device equipped with an RFID tag was given the *Tag* role. Figure 5.20 shows the analysis output of our STRAP protocol.


Figure 5.20 Analysis result of STRAP via the Scyther Tool

The ten message exchanges of STRAP were coded as "send" and "receive" parameters. The "claim" events are used to control the security of the message transferred. The "claim" requires parameters such as 'Secret', 'Alive', 'Weakagree', session-key reveal ('SKR'), 'Niagree' and 'Nisynch'. The ‘Secret’ parameter specifies that the coded STRAP parameters’ secrecy (*CPUID, TSP, Tsp, CpuID, OldTs, NonT, SDTA, TID, Ts,* and *NonS*) is expected to be verified by a “No attacks within bounds” comments, as in Figure 5.20. The Sychter response to 'Alive' parameter specifies that communicating partners are alive and the availability of the authentication peers is ensured. Verification of the 'weakagree' parameter ensures that the protocol is immune to impersonation attacks (Nikooghadam & Amintoosi, 2020). Verification of the secrecy of the session key KeyS is verified by 'SKR' parameter. The use of 'Niagree' on all three roles ensures that non-injective agreement is achieved, i.e. no messages can be injected into the message exchange between the peers. Hence, the message exchange between the peers is uncorrupted. The synchronization of the ten steps (non-injective synchronization) is verified by the 'nisynch' parameter, ensuring that all messages sent by reciprocal have been received by the recipient. The 'OK' declarations in Figure 5.20 demonstrate that the protocol was completed successfully, and no attacks were detected against the submitted protocol. Hence, the submitted STRAP was verified.

## 5.5 Performance and Security Comparison of STRAP

Our study is not focusing on the hardware characteristics of the four protocols ReSC family. Nevertheless, STRAP has no hardware disadvantages as it does not require any additional hardware. However, our work demonstrates that the vulnerabilities in the authentication protocols used by the proposed devices are fatal. Our work improves the security of the IoT devices and the supply chain, by proposing a stronger mutual authentication protocol.

### *5.5.1 Performance Comparison of STRAP*

Table 5.3 compares the performances of the five protocols. The first is the number of steps used by the protocols. ReSC-1 finishes verifying the RFID tag in three steps, without a mutual authentication. ReSC-2 and ReSC-3 also verify the RFID tag in three steps, but with mutual authentication. CDTA finishes mutual authentication in four steps, but exchanges nonces in plaintext. Our STRAP finishes authenticating the tag in the 8 steps but with a strong three-way mutual authentication, where two steps are used for the essential authentication of the smartphone and its user in non-power-consuming Phase 1. Authentication protocols consisting of message exchanges of less than four steps are not well-accepted mutual authentications and have known weaknesses (Dalkilic, Özcanhan, & Çakir, 2013). Therefore, STRAP uses extra steps to ensure the complete security of the supply chain. The disadvantage of extra steps is a few extra microseconds for a microcontroller with a megahertz clock. The second performance parameter is the number of encryption/decryption operations carried out by the protocols. In STRAP, RFID tag executes four encryption-decryption operations in one round. The ReSC family, carry out two encryption operations at the expense of passing critical parameters in plaintext, which blows the security of the devices. The third property is the server connection established in the supply chain hops. The only protocol that has a direct online connection between the devices and the server is STRAP. ReSC family records confidential information on the reader, which is only transmitted when an online connection is available. The record and forward method of ReSC family is baseless, as a wireless Internet connection is highly available in multiple types from many smartphone operators. Another performance metric is the footprint (memory consumption) of the protocol proposed. The memory consumption of STRAP is also higher than the ReSC family, to eliminate sharing any information with the insecure smartphone. STRAP consumes more power than the ReSC family, depending on the number of authentication steps and the number of encryptions/decryptions. This is not a problem in devices that are not battery operated, because the RFID tag is powered by the reader.

Table 5.3 Performance comparison of the studied five protocols

| Authentication Protocols | Number of Steps | Number of Encryptions | Online Connection | Memory Consumption (n times) | Power Consumption |
|---|---|---|---|---|---|
| ReSC-1 | 3 | 0 | No | 1 | Low |
| ReSC-2 | 3 | 2 | No | 1 | Low |
| ReSC-3 | 3 | 2 | No | 1 | Low |
| CDTA | 4 | 2 | No | 1 | Low |
| STRAP | 8 | 4 | Yes | 3 | Medium |

### *5.5.2 Security Comparison of STRAP*

Table 5.4 compares the security characteristics of the four protocols ReSC-1, ReSC-2, ReSC-3, and CDTA (ReSC family) against STRAP. ReSC family tries to provide security through hashing and symmetric encryption; however, some parameters are sent in plaintext. Thus, plaintext-ciphertext pairing is possible in ReSC-1, ReSC-2, ReSC-3, and CDTA. As a result, ReSC family is vulnerable to the attacks listed in Table 5.4. In contrast, the messages are encrypted in STRAP; therefore plaintext-ciphertext pairing is impossible in STRAP. Hence, our protocol is more secure compared to the ReSC family. The security of the intermediator smartphone and its user has critical importance. Unfortunately, ReSC family assumes a secure smartphone and user. But STRAP guarantees the elimination of attacks by rogue smartphones, or malicious users. Another important security primitive is the key update. Neither of the ReSC family members provides a key update, but STRAP obscures the messages with a new session encryption key every round. A critical improvement in STRAP is the requirement for a login name-password tuple for each chain hop. There are a finite number of hops, therefore, a small size server password file is sufficient to provide extended security. A very important disadvantage of the ReSC family is the lack of formal verification of the protocols. Meanwhile, STRAP has been verified by not only one but two well-accepted protocol verifiers. As such, STRAP is the only strong mutual authentication protocol with no weaknesses to known attacks, complying with the primary goal of security. It is commonly accepted that security experts always

consider the performance/security ratio when choosing an authentication protocol. STRAP proves to have a considerably superior performance/security than the ReSC family.

Table 5.4 Security comparison of the studied five protocols

| **Authentication Protocols** | **Plain Text** | **Smart Phone Use** | **Key Update** | **DoS Attack** | **Replay & Full Discl. Attacks** | **Lookup Table Attack** | **Formal Verification** |
|---|---|---|---|---|---|---|---|
| ReSC-1 | Yes | No | No | Yes | Yes | Yes | No |
| ReSC-2 | Yes | No | No | Yes | Yes | No | No |
| ReSC-3 | Yes | No | No | Yes | Yes | No | No |
| CDTA | Yes | No | No | No | Yes | No | No |
| STRAP | No | Yes | Yes | No | No | No | Yes |

# CHAPTER 6
# CONCLUSION AND FUTURE WORK

## 6.1 Conclusion

The Internet of Things (IoT) is an important technology that enables many industries to secure their supply chain by monitoring raw materials assets and supplies. However, the supply chain security of IoT itself has often been overlooked for many years. In recent works, RFID-enabled IC and IoT supply chains are studied separately in the context of security and privacy. One of the most significant of these solutions concerning RFID-enabled IC supply chain is the CNTR-SHIELD protocol proposed by (Jin & van Dijk, 2019) to overcome the weaknesses of the DARPA's SHIELD scheme, and the other ones concerning RFID-enabled IoT supply chain are the ReSC family solutions proposed in the works (Yang et al., 2015a, 2015b, 2017, 2018). However, all these solutions have security vulnerabilities and weaknesses.

In this thesis, the security and privacy issues of, DARPA's SHIELD program which is supported by the CNTR-SHIELD protocol, and the ReSC family solutions, are explored. Initially, the security vulnerabilities and attacks of the CNTR-SHIELD protocol, and the lightweight RFID protocols used in ReSC family solutions are analyzed. Then, two novel authentication protocols are proposed, SBMAPS replacing the CNTR-SHIELD protocol and STRAP replacing the lightweight RFID protocols used in ReSC family solutions. Subsequently, an online supply chain hop-tracking procedure supported by a novel RFID mutual authentication protocol called STRAP is presented. The common objective of these proposals is to enhance the security of IoT devices and gateways, which are resource stricken to be low-cost by design, by improving the security of their supply chains.

Firstly, a strongly bonded and mutually authenticated Protocol for SHIELD (SBMAPS) is proposed to overcome the security vulnerabilities and weaknesses of the CNTR-SHIELD protocol. The proposed SBMAPS scheme identifies and bonds a legal

user with a legal smartphone. Even if the credentials of the user are stolen, the proposed protocol degrades the role of the intermediary smartphone into a message repeater, between the dielet and the server. No nonces, secrets, or messages are passed in plaintext. Thus exposing the secrets of the dielets for creating fakes is rejected. The informal security and performance analysis indicates that the energy efficiency of the proposed SBMAPS is poorer and communication latency is higher than in previous works. However, the proposed protocol is secure against four types of attacks; while previous works are vulnerable. The decision reduces to the common dilemma of preference between performance and security. The evaluation is that the essence of the SHIELD initiative dictates preferring security. The only requirement of the proposed protocol is increasing the smartphone-server communication speed. Moreover, the SBMAPS scheme is formally validated via the widely-used Scyther and AVISPA tools. In comparison with SHIELD and CNTR-SHIELD methods, both the security and performance results showed that the proposed SBMAPS is more secure and efficient for IC supply chain.

Secondly, a strong RFID authentication protocol (scheme) is proposed to overcome the security vulnerabilities and weaknesses of the ReSC family protocols. As in the SBMAPS scheme analysis, the informal and formal security and performance analysis are also performed for the STRAP scheme. Compared with the insecure solutions, STRAP has the following merits:

1. STRAP scheme identifies and bonds a legal user with a legal smartphone. Even if the credentials of the user are stolen, the proposed protocol degrades the role of the intermediary smartphone into a message repeater, between the RFID tag and the server. Thus, the supply chain is secured.

2. No nonces, secrets, or messages are passed in plaintext. Thus exposing the secrets of the RFID tags for creating fakes is rejected. Hence, the transported devices are secured.

3. Comparison shows that the power consumption of our proposed STRAP is poorer and communication latency is higher than in previous works. However, STRAP is secure against four types of attacks while previous works are vulnerable. It is well accepted that security comes at a cost. The decision is reduced to the common dilemma of preference between performance and security. The evaluation is in the direction of preferring security. As in the SBMAPS scheme, the only requirement of our proposed protocol is increasing the smartphone-server communication speed.

Thirdly, an online supply chain hop-tracking procedure supported by the novel STRAP scheme is presented. In this procedure, a smartphone equipped with a near field communication (NFC) reader is used as the mediator between the NFC tag of the IoT device and the server. No confidential information is recorded or decrypted on the smartphone. In this way, security vulnerabilities originating from smartphones or RFID readers are eliminated. The only protocol that has a direct online connection between the devices and the server is STRAP. ReSC family records confidential information on the reader, which is only transmitted when an online connection is available. The record and forward method of ReSC family is baseless, as a wireless Internet connection is highly available in multiple types from many smartphone operators.

## 6.2 Future Work

Although the proposed security mechanisms are for the IoT supply chain, they can also be adopted and used for other electronic devices and equipment. In the future work, it is planned to develop a generalized supply chain framework adopted from this work, for the electronic devices and equipment that store, process, and transmit confidential information used in the fields like national security, armed forces, avionics, etc.

## REFERENCES


Abdel-Basset, M., Manogaran, G., & Mohamed, M. (2018). Internet of Things (IoT) and its impact on supply chain: A framework for building smart, secure and efficient systems. *Future Generation Computer Systems, 86*, 614-628.

Adeli, M., Bagheri, N., & Meimani, H. R. (2021). On the designing a secure biometric-based remote patient authentication scheme for mobile healthcare environments. *Journal of Ambient Intelligence and Humanized Computing, 12*, 3075–3089. https://doi.org/10.1007/s12652-020-02465-2

Alzahrani, B. A., & Mahmood, K. (2021). Provable Privacy-Preserving Authentication Solution for Internet of Things Environment. *IEEE Access, 9,* 82857-82865. https://doi.org/10.1109/ACCESS.2021.3086735

Arasteh, H., Hosseinnezhad, V., Loia, V., Tommasetti, A., Troisi, O., Shafie-khah, M., & Siano, P. (2016). IoT-based smart cities: A survey. *2016 IEEE 16th International Conference on Environment and Electrical Engineering (EEEIC)*, 1-6. https://doi.org/10.1109/EEEIC.2016.7555867

Armando A. et al. (2005). The AVISPA Tool for the Automated Validation of Internet Security Protocols and Applications. In Etessami K., Rajamani S.K. (Eds), *Computer-Aided Verification*. *CAV 2005. Lecture Notes in Computer Science*, *3576*. Springer. https://doi.org/10.1007/11513988_27

Azrour, M., Mabrouki, J., Guezzaz, A., & Farhaoui, Y. (2021). New enhanced authentication protocol for Internet of Things. *Big Data Mining and Analytics*, *4*(1), 1-9. https://doi.org/10.26599/BDMA.2020.9020010

Bae, W. & Kwak, J. (2017). Smart card-based secure authentication protocol in multi-server IoT environment. *Multimedia Tools and Applications, 79*, 15793–15811. https://doi.org/10.1007/s11042-017-5548-2

Barbareschi, M., De Benedictis, A., La Montagna, E., Mazzeo, A., & Mazzocca, N. (2019). A PUF-based mutual authentication scheme for Cloud-Edges IoT systems. *Future Generation Computer Systems, 101*, 246–261. https://doi.org/10.1016/j.future.2019.06.012

Boyens, J., Smith, A., Bartol, N., Winkler, K., Holbrook, A., & Fallon, M. (2022). Cybersecurity Supply Chain Risk Management Practices for Systems and Organizations. *National Institute of Standards and Technology*, *NIST SP 800-161r1*, 7-37. https://doi.org/10.6028/NIST.SP.800-161r1

Chakraborty, R. S., Narasimhan, S., & Bhunia, S. (2009). Hardware Trojan: Threats and emerging solutions. *2009 IEEE International High-Level Design Validation and Test Workshop*, 166-171. https://doi.org/10.1109/HLDVT.2009.5340158

Clarke, E. M., Henzinger, T. A., Veith, H., & Bloem, R. (2018). *Handbook of Model Checking*. Springer. https://doi.org/10.1007/978-3-319-10575-8

Cobb, W. E., Garcia, E. W., Temple, M. A., Baldwin, R. O., & Kim, Y. C. (2010). Physical layer identification of embedded devices using RF-DNA fingerprinting. *2010 - MILCOM 2010 Military Communications Conference,* 2168-2173. https://doi.org/10.1109/MILCOM.2010.5680487

Cremers, C. J. F. (2008). The Scyther Tool: Verification, Falsification, and Analysis of Security Protocols. In Gupta A., Malik S. (Eds), *Computer-Aided Verification. CAV 2008. Lecture Notes in Computer Science*, *5123*. Springer. https://doi.org/10.1007/978-3-540-70545-1_38

Dalkilic, G., Özcanhan, M. H., & Çakir, H. Ş. (2013). Increasing keyspace at little extra cost in RFID authentications. *Turkish Journal of Electrical Engineering & Computer Sciences, 22*, 155–165. https://doi:org/10.3906/elk-1201-77

Defense Advanced Research Projects Agency – DARPA. (2022, June 08). Tiny, cheap, foolproof: seeking new components to counter counterfeit electronics. *Microsystems Technology Office (MTO) Broad Agency Announcement.* https://www.darpa.mil/news-events/2014-02-24

Ding, X., Wang, X., Xie, Y., & Li, F. (2021). A Lightweight Anonymous Authentication Protocol for Resource-Constrained Devices in the Internet of Things. *IEEE Internet of Things Journal, 9*(3)*,* 1818-1829. https://doi.org/10.1109/JIOT.2021.3088641

Dolev, D. & Yao, A. C. (1983). On the security of public-key protocols. *IEEE Transactions on Information Theory, 29*, 198-208. https://doi.org/10.1109/TIT.1983.1056650

Farooq, M. J., & Zhu, Q. (2019). IoT supply chain security: overview, challenges, and the road ahead. *arXiv:1908.07828*. https://doi.org/10.48550/arXiv.1908.07828

Guin, U., DiMase, D., & Tehranipoor, M. (2014). Counterfeit Integrated Circuits: Detection, Avoidance, and the Challenges Ahead. *Journal of Electronic Testing*, *30*, 9–23. https://doi.org/10.1007/s10836-013-5430-8

Guin, U., Forte, D., & Tehranipoor, M. (2016). Design of Accurate Low-Cost On-Chip Structures for Protecting Integrated Circuits Against Recycling. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, *24*(4), 1233-1246. https://doi.org/10.1109/TVLSI.2015.2466551

Guin, U., Huang, K., DiMase, D., Tehranipoor, M., & Makris Y. (2014). Counterfeit integrated circuits: A rising threat in the global semiconductor supply chain. *Proceedings of the IEEE, 102*(8), 1207–1228. https://doi.org/10.1109/JPROC.2014.2332291

Guin, U., Zhang, X., Forte, D., & Tehranipoor, M. (2014). Low-cost on-chip structures for combating die and IC recycling. *DAC '14: Proceedings of the 51st Annual Design Automation Conference*, 1-6. https://doi.org/10.1145/2593069.2593157

Hassija, V., Chamola, V., Gupta, V., Jain, S., & Guizani, N. (2020). A survey on supply chain security: Application areas, security threats, and solution architectures. *IEEE Internet of Things Journal, 8*(8), 6222-6246.

Hiromoto, R. E., Haney, M., & Vakanski, A. (2017). A secure architecture for IoT with supply chain risk management. *9th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS), 1*, 431-435.

Jeong, D. S., Thomas, R., Katiyar, R., Scott, J., Kohlstedt, H., Petraru, A., & Hwang, C. S. (2012). Emerging memories: resistive switching mechanisms and current status. *Reports on Progress in Physics, 75*(7). https://doi.org/10.1088/0034-4885/75/7/076502

Jin, C., & van Dijk, M. (2019). Secure and efficient initialization and authentication protocols for SHIELD. *IEEE Transactions on Dependable and Secure Computing, 16*, 156–173. https://doi.org/10.1109/TDSC.2017.2647950

Kanuparthi, A., Karri, R., & Addepalli, S. (2013). Hardware and embedded security in the context of internet of things. *CyCAR '13: Proceedings of the 2013 ACM workshop on Security, privacy & dependability for cyber vehicles,* 61-64. https://doi.org/10.1145/2517968.2517976

Kieras, T., Farooq, J., & Zhu, Q. (2021). I-SCRAM: A framework for IoT supply chain risk analysis and mitigation decisions. *IEEE Access, 9*, 29827-29840. https://doi.org/10.1109/ACCESS.2021.3058338

Kieras, T., Farooq, M. J., & Zhu, Q. (2020). RIoTS: Risk Analysis of IoT Supply Chain Threats. *2020 IEEE 6th World Forum on Internet of Things (WF-IoT)*, 1-6. https://doi.org/10.1109/WF-IoT48130.2020.9221323

Koushanfar, F., & Karri, R. (2014). Can the SHIELD protect our integrated circuits?. *2014 IEEE 57th International Midwest Symposium on Circuits and Systems (MWSCAS)*, 350-353. https://doi.org/10.1109/MWSCAS.2014.6908424

Lamport, L. (1994). The temporal logic of actions. *ACM Transactions on Programming Languages and Systems, 16*(3), 872–923. https://doi.org/10.1145/177492.177726

Lee, J., Yu, S., Park, K., Park, Y., & Park, Y. (2019). Secure three-factor authentication protocol for multi-gateway IoT environments. *Sensors, 19*(10), 2358. https://doi.org/10.3390/s19102358

Leef, S. (2018). Supply Chain Hardware Integrity for Electronics Defense (SHIELD). Software and Supply Chain Assurance Winter Forum 2018. CSRS. https://csrc.nist.gov/CSRC/media/Projects/cyber-supply-chain-risk-management/documents/SSCA/Winter_2018/TuePM2.1-SHIELD.pdf

Mohanta, B. K., Jena, D., & Patnaik, S. (2021). An ECC-based authentication protocol for fog-IoT enabled smart home environment. *International Journal of Intelligent Systems Technologies and Applications, 20*(1), 25-46. https://doi.org/10.1504/ijista.2021.114638

Nikooghadam, M. & Amintoosi, H. (2020). An improved secure authentication and key agreement scheme for healthcare applications. *25th International Computer Conference, Computer Society of Iran (CSICC_)*, 1-7. https://doi.org/10.1109/CSICC_49403.2020.9050069

Omitola, T., & Wills, G. (2018). Towards mapping the security challenges of the Internet of Things (IoT) supply chain. *Procedia Computer Science, 126*, 441-450.

Özcanhan, M. H. (2014). Improvement of a Weak RFID Authentication Protocol Making Drug Administration Insecure. *Life Science Journal, 11*(10), 269–276.

Özcanhan, M. H., & Türksönmez, H. (2020). A Strong Mutual Authentication Protocol for SHIELD. *Advances in Electrical and Computer Engineering, 20*(4), 81-90. https://doi.org/10.4316/AECE.2020.04010

Özcanhan, M. H., & Utku, S. (2018). Attacking A PUF Based Mutual Authentication Protocol Designed For Internet of Things. *International Journal of Computer Science and Software Engineering (IJCSSE), 7*(11), 270-274.

Özcanhan, M. H., Baytar, S., Utku, S., & Dalkılıç, G. (2015). Security Issues of a Recent RFID Multi Tagging Protocol. *International Journal of Advanced Computer Science and Applications (IJACSA), 6*(1), 11–15. https://doi.org/10.14569/IJACSA.2015.060102

Özcanhan, M. H., Dalkılıç, G., & Utku, S. (2014). Cryptographically supported NFC tags in medication for better inpatient safety. *Journal of Medical Systems, 38*(8), 1-15. https://doi.org/10.1007/s10916-014-0061-x

Peris-Lopez, P., Orfila, A., Castro, J. C. H., & Lubbe, J. C. A. (2011). Flaws on RFID grouping-proofs. Guidelines for future sound protocols. *Journal of Network and Computer Applications, 34*(3), 833–845. https://doi.org/10.1016/j.jnca.2010.04.008

Peris-Lopez, P., Orfila, A., Mitrokotsa, A., & Van der Lubbe, J. C. A. (2011). A comprehensive RFID solution to enhance inpatient medication safety. *International Journal of Medical Informatics, 80*(1), 13–24. https://doi.org/10.1016/j.ijmedinf.2010.10.008

Perrig, A., Stankovic, J., & Wagner, D. (2004). Security in wireless sensor networks. *Communications of the ACM*, 53-57. https://doi.org/10.1145/990680.990707

Rostami, M., Farinaz, K., & Karri R. (2014). A primer on hardware security: Models, methods, and metrics. *Proceedings of the IEEE, 102*(8), 1283–1295. https://doi.org/10.1109/JPROC.2014.2335155

Rostami, M., Majzoobi, M., Koushanfar, F., Wallach, D. S., & Devadas, S. (2014). Robust and reverse-engineering resilient PUF authentication and key-exchange by substring matching. *IEEE Transactions on Emerging Topics in Computing, 2*(1), 37–49. https//doi:org/10.1109/TETC.2014.2300635

Rührmair U., Devadas S., & Koushanfar F. (2012). Security Based on Physical Unclonability and Disorder. In Tehranipoor M., Wang C. (Eds), *Introduction to Hardware Security and Trust*. Springer. https://doi.org/10.1007/978-1-4419-8080-9_4

Sadhukhan, D., Ray, S., Biswas, G. P., Khan, M. K., & Dasgupta, M. (2021). A lightweight remote user authentication scheme for IoT communication using elliptic curve cryptography. *The Journal of Supercomputing, 77*(2), 1114-1151. https://doi.org/10.1007/s11227-020-03318-7

Safkhani, M., Camara, C., Lopez, P. P., & Bagheri, N. (2021). RSEAP2: An enhanced version of RSEAP, an RFID based authentication protocol for vehicular cloud computing. *Vehicular Communications*, *28*. https://doi.org/10.1016/j.vehcom.2020.100311

Sathyanarayana, S., Robinson, W. H., & Beyah, R. A. (2013). A network-based approach to counterfeit detection. *2013 IEEE International Conference on Technologies for Homeland Security (HST)*, 473-479. https://doi.org/10.1109/THS.2013.6699050

See, A. V. (2022). *Number of IoT connected devices worldwide 2019-2030*. Statista. https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/

Sey, D. (2018). A survey on authentication methods for the Internet of Things. *PeerJ Preprints*, *6*. https://doi.org/10.7287/peerj.preprints.26474v2

Shah, T., & Venkatesan, S. (2018). Authentication of IoT device and IoT server using secure vaults. *2018 17th IEEE International Conference on Trust, Security and Privacy in Computing and Communications/12th IEEE International Conference on Big Data Science and Engineering (TrustCom/BigDataSE)*, 819-824. https://doi.org/10.1109/TrustCom/BigDataSE.2018.00117

Shahzad, A., Zhang, K., & Gherbi, A. (2020). Intuitive development to examine collaborative IoT supply chain system underlying privacy and security levels and perspective powering through proactive blockchain. *Sensors*, *20*(13), 3760. https://doi.org/10.3390/s20133760

Subramanyan, P., Tsiskaridze, N., Pasricha, K., Reisman, D., Susnea, A., & Malik, S. (2013). Reverse engineering digital circuits using functional analysis. *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE),* 1277-1280. https://doi.org/10.7873/DATE.2013.264

Suh, G. E. & Devadas, S. (2007). Physical Unclonable Functions for Device Authentication and Secret Key Generation. *DAC '07: Proceedings of the 44th annual Design Automation Conference*, 9-14. https://doi.org/10.1145/1278480.1278484

Thakur, V., Indra, G., Gupta, N., Chatterjee, P., Said, O., & Tolba, A. (2021). Cryptographically secure privacy-preserving authenticated key agreement protocol for an IoT network: A step towards critical infrastructure protection. *Peer-to-Peer Networking and Applications*, 1-15. https://doi.org/10.1007/s12083-021-01236-w

Türksönmez, H., & Özcanhan, M. H. (2021). A Survey on Integrated Circuit Trojans. *Computer Engineering and Intelligent Systems, 12*, 1-8. https://doi.org/10.7176/CEIS/12-2-01

Yang, K., Forte, D., & Tehranipoor, M. (2015a). Protecting endpoint devices in IoT supply chain. *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD),* 351–356. https://doi.org/10.1109/ICCAD.2015.7372591

Yang, K., Forte, D., & Tehranipoor, M. (2015b). ReSC: RFID-Enabled Supply Chain Management and Traceability for Network Devices. RFIDsec 2015: *Revised Selected Papers of the 11th International Workshop on Radio Frequency Identification, 9440*, 32–49. https://doi.org/10.1007/978-3-319-24837-0_3

Yang, K., Forte, D., & Tehranipoor, M. (2017). CDTA: A Comprehensive Solution for Counterfeit Detection, Traceability, and Authentication in the IoT Supply Chain. *ACM Transactions on Design Automation of Electronic Systems, 22*(3), 1-31. https://doi.org/10.1145/3005346

Yang, K., Forte, D., & Tehranipoor, M. (2018). ReSC: An RFID-Enabled Solution for Defending IoT Supply Chain. *ACM Transactions on Design Automation of Electronic Systems, 23*(3), 1-27. https://doi.org/10.1145/3174850

Yen, Y. C., Lo, N. W., & Wu, T. C. (2012). Two RFID-based solutions for secure inpatient medication administration. *Journal of Medical Systems, 36*(5), 2769–2778. https://doi: 10.1007/s10916-011-9753-7

Zargar, S., Shahidinejad, A., & Ghobaei-Arani, M. (2021). A lightweight authentication protocol for IoT-based cloud environment. *International Journal of Communication Systems, 34*(11). https://doi.org/10.1002/dac.4849

Zhang, H., Sakurai, K. (2020). Blockchain for IoT-Based Digital Supply Chain: A Survey. In Barolli, L., Okada, Y., Amato, F. (Eds), *Advances in Internet, Data and Web Technologies. EIDWT 2020. Lecture Notes on Data Engineering and Communications Technologies, 47*. Springer. https://doi.org/10.1007/978-3-030-39746-3_57

Zhang, X. & Tehranipoor, M. (2014). Design of On-Chip Lightweight Sensors for Effective Detection of Recycled ICs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 22*(5), 1016-1029. https://doi.org/0.1109/TVLSI.2013.2264063

Zhang, X., Tuzzio, N., & Tehranipoor, M. (2012). Identification of recovered ICs using fingerprints from a light-weight on-chip sensor. *DAC Design Automation Conference 2012*. 703-708

Ziegeldorf, J. H., Morchon, O. G., & Wehrle, K. (2014). Privacy in the internet of things: threats and challenges. *Security and Communication Networks, 7*(12), 2728–2742. https://doi.org/10.1002/sec.795

**APPENDICES**

**APPENDIX A Source Code of SBMAPS in HLPSL**

```
%%-------------------------------------------------------------------
%%        SBMAPS Code in HLPSL
%%-------------------------------------------------------------------
%%-----------------ROLE READER-------------------------------
role role_Reader(
      Reader, Server, Dielet        : agent,
      Usr_ID, CPU_ID                : text,
      Usr_Key, KeyRD                : symmetric_key,
      Hash                          : hash_func,
      ReadReq                       : nat,
      Usr_PK, Srv_PK                : public_key,
      SND, RCV                      : channel(dy)
) played_by Reader def=
local
      State                 : nat,
      Rdr_t, Result         : text,
      M10                   : {{function}_public_key}_inv(public_key)
init State := 0
transition
      %%------------ Phase-1 starts here ----------------------
      0.    State = 0 /\ RCV(start) =|>
            State' := 1 /\ Rdr_t' := new()
            /\ SND(Usr_ID.{Reader.Server.CPU_ID.Rdr_t'}_Usr_Key)
            /\ secret(CPU_ID, sec_cpuid_rs, {Reader, Server})
            /\ secret(Rdr_t', sec_rdrt_rs, {Reader, Server})
      1.    State = 1 /\ RCV({{Hash(Server.Reader.CPU_ID'.Rdr_t)}
            _Usr_PK}_inv(Srv_PK)) =|>

            %%-------------Phase-2 starts here ------------------
```

```
        State' := 2 /\ SND({Reader.Dielet.ReadReq}_KeyRD)
        %%Reader checks that it receives the same timestamp
        %%that it sent at step 0
        /\ request(Reader, Server, auth_rdrt_rs, Rdr_t)

    %%-------------Phase-3 starts here ---------------------
    2.  State = 2 /\ RCV(Server.Reader.M10') =|>
        State' := 3
end role

%%--------------ROLE SERVER-----------------------------------
role role_Server(
    Reader, Server, Dielet      : agent,
    Usr_Key, KeyDS              : symmetric_key,
    Hash                        : hash_func,
    Usr_PK, Srv_PK              : public_key,
    CurrHop                     : nat,
    SND, RCV                    : channel(dy)
) played_by Server def=

local
    State                   : nat,
    KeySession              : symmetric_key,
    Ts, Usr_ID, CPU_ID, Rdr_t, D_ID, NonS, Temp3, SensorS,
    Result, Old_Ts          : text,
    M1                      : {text.text}_symmetric_key,
    M3, M4, M5, M6          : {text}_symmetric_key,
    M7                      : {text.text}_symmetric_key,
    M8, M9                  : {message}_symmetric_key,
    M10                     : {{function}_public_key}_inv(public_key)

init State := 0
```

```
transition
    %%-------------Phase-1 starts here -------------------------
    0.  State = 0 /\ RCV(Usr_ID'.{Reader.Server.CPU_ID'.Rdr_t'}_Usr_Key) =|>
        State' := 1
        /\ SND({{Hash(Server.Reader.CPU_ID'.Rdr_t')}_Usr_PK}_inv(Srv_PK))
        /\ secret(CPU_ID', sec_cpuid_sr, {Reader, Server})
        /\ secret(Rdr_t', sec_rdrt_sr, {Reader, Server})
        %%Server hopes that Rdr_t will permit to authenticate him
        /\ witness(Server, Reader, auth_rdrt_rs, Rdr_t')

    %%-------------Phase-2 starts here -------------------------
    1.  State = 1 /\ RCV({Dielet.Server.D_ID'}_KeyDS) =|>
        State' := 2 /\ NonS' := new() /\ M3' := {NonS'}_KeyDS
        /\ SND(Server.Dielet.CurrHop.M3')
        /\ secret(NonS', sec_nons_sd, {Server, Dielet})

    2.  State = 2 /\ RCV(Dielet.Server.M4'.{xor(Temp3', M4')}_KeyDS) =|>
        State' := 3 /\ M5' := {xor(Temp3', M4')}_KeyDS
        /\ M6' := {xor(M5', Temp3')}_KeyDS
        /\ KeySession' := xor(NonS, Temp3') /\ Ts' := new()
        /\ M7' := {Temp3'.Ts'}_KeySession'
        /\ SND(Server.Dielet.M6'.M7')
        /\ secret(Temp3', sec_temp3_sd, {Server, Dielet})
        /\ secret(NonS, sec_nons1_sd, {Server, Dielet})
        /\ secret(Ts', sec_ts_sd, {Server, Dielet})
    %%Server checks that it receives the same nonce that it sent at step 1
        /\ request(Server, Dielet, auth_nons_sd, NonS)
    %%Server hopes that NonD(Temp3) will permit to authenticate him
        /\ witness(Server, Dielet, auth_nond_ds, Temp3')

    %%-------------Phase-3 starts here ----------------------
    3.  State = 3 /\ RCV(Dielet.Server.M8'.{xor({xor(Old_Ts', Ts)}_KeySession',
```

```
            SensorS')}_KeySession') =|>
        State' := 4 /\ Result' := new()
        /\ M10' := {{Hash(Result'.Rdr_t')}_Usr_PK}_inv(Srv_PK)
        /\ SND(Server.Reader.M10')
        /\ secret(Rdr_t', sec_rdrt_sr, {Server, Reader})
        /\ secret(Result', sec_result_sr, {Server, Reader})
    %%Server checks that it receives the same timestamp
    %%that it sent at step 2
        /\ request(Server, Dielet, auth_ts_sd, Old_Ts')
end role

%%-----------------ROLE DIELET----------------------------
role role_Dielet(
    Reader, Server, Dielet      : agent,
    ReadReq                     : nat,
    KeyRD, KeyDS                : symmetric_key,
    D_ID, Old_Ts                : text,
    SND, RCV                    : channel(dy)
) played_by Dielet def=

local
    State, ReadReq1, CurrHop    : nat,
    KeySession                  : symmetric_key,
    M3, M4, M5, M6              : {text}_symmetric_key,
    M7                          : {text.text}_symmetric_key,
    Temp0, NonD, Tss, SensorS   : text,
    M8, M9                      : {message}_symmetric_key

init State := 0
transition
    %%-------------Phase-2 starts here ------------------------
    0.  State = 0 /\ RCV({Reader.Dielet.ReadReq1'}_KeyRD)
```

```
        /\ ReadReq = ReadReq1' =|>
        State' := 1 /\ SND({Dielet.Server.D_ID}_KeyDS)
        /\ secret(D_ID, sec_did_ds, {Dielet, Server})

    1.  State = 1 /\ RCV(Server.Dielet.CurrHop'.{Temp0'}_KeyDS) =|>
    %%---Dielet.Temp0 = Server.NonS----
        State' := 2      /\ M3' := {Temp0'}_KeyDS
        /\ M4' := {xor(Temp0', M3')}_KeyDS
        /\ NonD' := new() /\ M5' := {xor(NonD', M4')}_KeyDS
        /\ KeySession' := xor(NonD', Temp0')
        /\ SND(Dielet.Server.M4'.M5')
        /\ secret(Temp0', sec_temp0_ds, {Dielet, Server})
        /\ secret(NonD', sec_nond_ds, {Dielet, Server})
        %%Dielet hopes that NonS(Temp0) will permit to authenticate him
        /\ witness(Dielet, Server, auth_nons_sd, Temp0')

    2.  State = 2 /\ RCV(Server.Dielet.M6'.{NonD.Tss'}_KeySession') =|>
    %%-------------Phase-3 starts here ----------------------
    State' := 3
        /\ SensorS' := new()
        /\ M8':= {xor(Old_Ts, Tss')}_KeySession'
        /\ M9' := {xor(M8', SensorS')}_KeySession'
        /\ SND(Dielet.Server.M8'.M9')
        /\ secret(Old_Ts, sec_oldts_ds, {Dielet, Server})
        /\ secret(Tss', sec_tss_ds, {Dielet, Server})
        /\ secret(SensorS', sec_sensors_ds, {Dielet, Server})
        %%Dielet checks that it receives the same nonce that it sent at step 1
        /\ request(Dielet, Server, auth_nond_ds, NonD)
        %%Dielet hopes that Old_Ts will permit to authenticate him
        /\ witness(Dielet, Server, auth_ts_sd, Old_Ts)
end role
```

```
%%----------------ROLE SESSION--------------------------------
role session(
        Reader, Server, Dielet  :       agent,
        Usr_ID                          : text,
        Usr_Key                         : symmetric_key,
        CPU_ID, D_ID                    : text,
        Hash                            : hash_func,
        Usr_PK, Srv_PK                  : public_key,
        Old_Ts                          : text

) def=
local
        SNDR, RCVR, SNDS, RCVS, SNDD, RCVD: channel(dy)
const
        readreq, currhop    : nat,
        keyrd, keyds        : symmetric_key

composition
role_Reader(Reader, Server, Dielet, Usr_ID, CPU_ID, Usr_Key,
                keyrd, Hash, readreq, Usr_PK, Srv_PK, SNDR, RCVR)
/\ role_Server(Reader, Server, Dielet, Usr_Key, keyds, Hash,
                Usr_PK, Srv_PK, currhop, SNDS, RCVS)
/\ role_Dielet(Reader, Server, Dielet, readreq, keyrd,
                keyds, D_ID, Old_Ts, SNDD, RCVD)
end role

%%-------------ROLE ENVIRONMENT----------------------
role environment() def=

const reader, server, dielet            : agent,
        cpu_id1, cpu_id2, cpu_id3       : text,
        usr_id1, usr_id2, usr_id3       : text,
```

```
    d_id1, d_id2, d_id3, old_ts        : text,
    usr_key1, usr_key2, usr_key3       : symmetric_key,
    fhash                              : hash_func,
    sec_cpuid_rs, sec_rdrt_rs, auth_rdrt_rs, sec_cpuid_sr, sec_rdrt_sr,
    sec_did_ds, auth_nons_sd, auth_nond_ds, auth_ts_sd, sec_nons_sd,
    sec_temp3_sd, sec_nons1_sd, sec_ts_sd, sec_result_sr, sec_temp0_ds,
    sec_nond_ds, sec_oldts_ds, sec_tss_ds, sec_sensors_ds: protocol_id,
    usr_pk, srv_pk                     : public_key

intruder_knowledge = {reader, server, dielet, usr_pk, srv_pk}

composition
    session(reader, server, dielet, usr_id1, usr_key1, cpu_id1, d_id1,
            fhash, usr_pk, srv_pk, old_ts)
end role
%%------------------------------------------------------------------
goal
    secrecy_of sec_cpuid_rs, sec_rdrt_rs, sec_cpuid_sr, sec_rdrt_sr, sec_did_ds,
               sec_nons_sd, sec_temp3_sd, sec_nons1_sd, sec_ts_sd, sec_result_sr,
               sec_temp0_ds, sec_nond_ds, sec_oldts_ds, sec_tss_ds, sec_sensors_ds

    authentication_on auth_rdrt_rs, auth_nons_sd, auth_nond_ds, auth_ts_sd
end goal
environment()
```

## APPENDIX B Source Code of SBMAPS in SPDL

```
//-------------------------------------------------------------------------------------------
//                        SBMAPS Protocol Code in SPDL
//-------------------------------------------------------------------------------------------

//The protocol description

usertype String, SessionKey, SymmetricKey, TimeStamp;

hashfunction Hash;
const fxor: Function;

protocol sbmaps (Reader, Server, Dielet)
{

//-----------------ROLE READER-----------------------------------------------------------
      role Reader
      {
            fresh NotOK, OK: Nonce;
            fresh TR: TimeStamp;
            var CpuID: String;
            const USRID, CPUID, REQ: String;
            var Tr: TimeStamp;
            var Result: Nonce;

//--------------------------Phase-1 starts here-----------------------------------------
            macro M1 = { USRID, CPUID, TR}k(Reader, Server);
            send_1(Reader, Server, M1);
            macro H1 = Hash(CPUID, TR);
            recv_2(Server, Reader, {{Hash(CpuID, Tr)}pk(Reader)}sk(Server));
            macro H2 = Hash(CpuID, Tr);
```

```
        match(H1, H2);

//-------------------------Phase-2 starts here----------------------------------------
        send_3(Reader, Dielet, {Reader, REQ}k(Reader, Dielet));

//-------------------------Phase-3 starts here----------------------------------------
        macro h1 =  Hash(OK, TR);
        macro h2 =  Hash(NotOK, TR);
        macro ResHash = Hash(Result, Tr);
        recv_10(Server, Reader,{{ ResHash}pk(Reader)}sk(Server));
        match(h1, ResHash);
        match(h2, ResHash);

//-----------------------Reader Claim Descriptions---------------------------------

        claim_r1(Reader, Secret, CPUID);
        claim_r2(Reader, Secret, TR);

        claim_r3(Reader, Niagree);
        claim_r4(Reader, Nisynch);
        claim_r5(Reader, Alive);
        claim_r6(Reader, Weakagree);
    }

//------------ROLE SERVER-----------------------------------------------------------
    role Server
    {
        var CpuID, UsrID, Did: String;
        fresh NonS, TS, Result, CurrHop: Nonce;
        var NonSS, NonDD, OldTs, Ts, SensorSS: Nonce;
        var temp3, temp6, temp7, temp8, temp9, M4, M7, M8, M9, ResHash;
        const CPUID, DID: String;
```

```
var Tr: TimeStamp;

//-------------------------Phase-1 starts here-----------------------------------------
recv_1(Reader, Server, {UsrID, CpuID, Tr}k(Reader, Server));
match(CPUID, CpuID);
macro M2 = {{Hash(CpuID, Tr)}pk(Reader)}sk(Server);
send_2(Server, Reader, M2);

//-------------------------Phase-2 starts here-----------------------------------------
recv_4(Dielet, Server, {Did}k(Dielet, Server));
match(DID, Did);
macro M3 = {NonS}k(Dielet, Server);
send_5(Server, Dielet, CurrHop, M3);
recv_6(Dielet, Server, M4, {fxor(NonDD, M4)}k(Dielet, Server));
macro M5 = {fxor(NonDD, M4)}k(Dielet, Server);
macro temp1 = fxor(NonS, M3);
match(NonSS, temp1);
macro temp2 = fxor(NonDD, M4);  // temp2 = NonD x M4
match(temp3, fxor(temp2, M4)); // temp3 = NonD
macro KeyS =  fxor(NonS, NonDD);
macro M6 =  {fxor(M5, temp3)}k(Dielet, Server);
send_7(Server, Dielet, M6);

//-------------------------Phase-3 starts here-----------------------------------------
macro M7 = {NonDD, TS}KeyS;
send_8(Server, Dielet, M7);
macro M8 = {fxor(OldTs, Ts)}KeyS;
macro M9 = {fxor(M8, SensorSS)}KeyS;
recv_9(Dielet, Server,M8, M9);
macro temp6 = fxor(OldTs, Ts);
macro temp7 = fxor(temp6, Ts);
match(temp7, OldTs); // temp7 =? OldTs
```

```
            macro temp8 = fxor(M8, SensorSS);
            macro temp9 = fxor(M8, temp8);
            match(temp9, SensorSS); // temp9 =? SensorSS
            macro ResHash = Hash(Result, Tr);
            send_10(Server, Reader,{{ ResHash}pk(Reader)}sk(Server));

//----------------------Server Claim Descriptions-----------------------------------

            claim_s1(Server, Secret, CpuID);
            claim_s2(Server, Secret, Tr);

            claim_s3(Server, Niagree);
            claim_s4(Server, Nisynch);
            claim_s5(Server, Alive);
            claim_s6(Server, Weakagree);
            claim_s7(Server, SKR, KeyS);
      }

//-------------------------ROLE DIELET-------------------------------------------------
      role Dielet
      {
            const REQ, DID: String;
            var Req: String;
            var NonS, NonDD, Ts, CurrHop: Nonce;
            fresh NonD, OldTs, SensorS: Nonce;
            var M6, M7;

//--------------------------Phase-2 starts here-----------------------------------------
            recv_3(Reader, Dielet, {Reader, Req}k(Reader, Dielet));
            match(REQ, Req);
            send_4(Dielet, Server, {DID}k(Dielet, Server));
            recv_5(Server, Dielet, CurrHop, {NonS}k(Dielet, Server));
```

```
        macro M3 = {NonS}k(Dielet, Server);
        macro M4 = {fxor(NonS, M3)}k(Dielet, Server);
        macro M5 = {fxor(NonD, M4)}k(Dielet, Server);
        send_6(Dielet, Server, M4, M5);
        recv_7(Server, Dielet, {fxor(M5, NonDD)}k(Dielet, Server));
        macro KeyS = fxor(NonS, NonDD);

//-------------------------Phase-3 starts here----------------------------------------
        recv_8(Server, Dielet, {NonDD, Ts}KeyS);
        macro M8 = {fxor(OldTs, Ts)}KeyS;
        macro M9 = {fxor(M8, SensorS)}KeyS;
        send_9(Dielet, Server, M8, M9);


//-------------------Dielet  Claim Descriptions---------------------------------------

        claim_d1(Dielet, Secret, OldTs);
        claim_d2(Dielet, Secret, NonD);
        claim_d3(Dielet, Secret, SensorS);
        claim_d4(Dielet, Secret, DID);
        claim_d5(Dielet, Secret, NonS);
        claim_d6(Dielet, Secret, Ts);

        claim_d7(Dielet, Niagree);
        claim_d8(Dielet, Nisynch);
        claim_d9(Dielet, Alive);
        claim_d10(Dielet, Weakagree);
        claim_d11(Dielet, SKR, KeyS);
    }
}
```

**APPENDIX C Source Code of STRAP in HLPSL**

```
%%-------------------------------------------------------------
%%          STRAP Code in HLPSL
%%-------------------------------------------------------------
%%-----------------ROLE SMARTPHONE----------------------------
role role_SPhone(
      SPhone, Server, Tag     : agent,
      Usr_ID, CPU_ID          : text,
      Usr_Key, KeySpT         : symmetric_key,
      Hash                    : hash_func,
      ReadReq                 : nat,
      Usr_PK, Srv_PK          : public_key,
      SND, RCV                : channel(dy)
) played_by SPhone def=
local
      State             : nat,
      Sp_t, Result      : text,
      M10               : {{function}_public_key}_inv(public_key)
init State := 0
transition
      %%------------- Phase-1 starts here ----------------------
   0.  State = 0 /\ RCV(start) =|>
       State' := 1 /\ Sp_t' := new()
            /\ SND(Usr_ID.{SPhone.Server.CPU_ID.Sp_t'}_Usr_Key)
            /\ secret(CPU_ID, sec_cpuid_sps, {SPhone, Server})
            /\ secret(Sp_t', sec_spt_sps, {SPhone, Server})
      1.  State = 1 /\ RCV({{Hash(Server.SPhone.CPU_ID'.Sp_t)}
            _Usr_PK}_inv(Srv_PK)) =|>

            %%-------------Phase-2 starts here ------------------
       State' := 2 /\ SND({SPhone.Tag.ReadReq}_KeySpT)
```

```
        %%SPhone checks that it receives the same timestamp
        %%that it sent at step 0
        /\ request(SPhone, Server, auth_spt_sps, Sp_t)

    %%-------------Phase-3 starts here ---------------------
    2.  State = 2 /\ RCV(Server.SPhone.M10') =|>
        State' := 3
end role

%%--------------ROLE SERVER-----------------------------------
role role_Server(
    SPhone, Server, Tag     : agent,
    Usr_Key, KeyTS          : symmetric_key,
    Hash                    : hash_func,
    Usr_PK, Srv_PK          : public_key,
    CurrHop                 : nat,
    SND, RCV: channel(dy)
) played_by Server def=

local
    State                   : nat,
    KeySession              : symmetric_key,
    Ts, Usr_ID, CPU_ID, Sp_t, T_ID, NonS, Temp3, SDTA,
    Result, Old_Ts          : text,
    M1                      : {text.text}_symmetric_key,
    M3, M4, M5, M6          : {text}_symmetric_key,
    M7                      : {text.text.hash_func}_symmetric_key,
    M8, M9                  : {message}_symmetric_key,
    M10                     : {{function}_public_key}_inv(public_key),
    Sgn                     : message

init State := 0
```

```
transition
    %%-------------Phase-1 starts here -------------------------
  0.  State = 0 /\ RCV(Usr_ID'.{SPhone.Server.CPU_ID'.Sp_t'}_Usr_Key) =|>
    State' := 1
        /\ SND({{Hash(Server.SPhone.CPU_ID'.Sp_t')}_Usr_PK}_inv(Srv_PK))
        /\ secret(CPU_ID', sec_cpuid_ssp, {SPhone, Server})
        /\ secret(Sp_t', sec_spt_ssp, {SPhone, Server})
        %%Server hopes that Sp_t will permit to authenticate him
        /\ witness(Server, SPhone, auth_spt_sps, Sp_t')

    %%-------------Phase-2 starts here -------------------------
    1. State = 1 /\ RCV({Tag.Server.T_ID'}_KeyTS) =|>
      State' := 2 /\ NonS' := new() /\ M3' := {NonS'}_KeyTS
      /\ SND(Server.Tag.CurrHop.M3')
      /\ secret(NonS', sec_nons_st, {Server, Tag})

    2. State = 2 /\ RCV(Tag.Server.M4'.{xor(Temp3', M4')}_KeyTS) =|>
      State' := 3 /\ M5' := {xor(Temp3', M4')}_KeyTS
      /\ M6' := {xor(M5', Temp3')}_KeyTS
      /\ KeySession' := xor(NonS, Temp3') /\ Ts' := new()
      /\ Sgn' := Hash(T_ID.CurrHop.Ts')
      /\ M7' := {Temp3'.Ts'.Sgn'}_KeySession'
      /\ SND(Server.Tag.M6'.M7')
      /\ secret(Temp3', sec_temp3_st, {Server, Tag})
      /\ secret(NonS, sec_nons1_st, {Server, Tag})
      /\ secret(Ts', sec_ts_st, {Server, Tag})
      /\ secret(Sgn', sec_sgn_st, {Server, Tag})
      %%Server checks that it receives the same nonce that it sent at step 1
      /\ request(Server, Tag, auth_nons_st, NonS)
      %%Server hopes that NonT(Temp3) will permit to authenticate him
      /\ witness(Server, Tag, auth_nont_ts, Temp3')
```

```
    %%-------------Phase-3 starts here ----------------------
    3. State = 3 /\ RCV(Tag.Server.M8'.{xor({xor(Old_Ts', Ts)}_KeySession',
                SDTA')}_KeySession') =|>
      State' := 4 /\ Result' := new()
      /\ M10' := {{Hash(Result'.Sp_t')}_Usr_PK}_inv(Srv_PK)
      /\ SND(Server.SPhone.M10')
      /\ secret(Sp_t', sec_spt_ssp, {Server, SPhone})
      /\ secret(Result', sec_result_ssp, {Server, SPhone})
      %%Server checks that it receives the same timestamp
      %%that it sent at step 2
      /\ request(Server, Tag, auth_ts_st, Old_Ts')
end role

%%------------------ROLE RFID TAG-------------------------------
role role_Tag(
    SPhone, Server, Tag     : agent,
    ReadReq                 : nat,
    KeySpT, KeyTS           : symmetric_key,
    T_ID, Old_Ts            : text,
    SND, RCV                : channel(dy)
) played_by Tag def=

local
    State, ReadReq1, CurrHop  : nat,
    KeySession                : symmetric_key,
    M3, M4, M5, M6            : {text}_symmetric_key,
    M7                        : {text.text.hash_func}_symmetric_key,
    Temp0, NonT, Tss, SDTA    : text,
    M8, M9                    : {message}_symmetric_key,
    Sgn                       : message

init State := 0
```

```
transition
    %%-------------Phase-2 starts here ------------------------
    0.  State = 0 /\ RCV({SPhone.Tag.ReadReq1'}_KeySpT)
        /\ ReadReq = ReadReq1' =|>
        State' := 1 /\ SND({Tag.Server.T_ID}_KeyTS)
        /\ secret(T_ID, sec_tid_ts, {Tag, Server})

    1.  State = 1 /\ RCV(Server.Tag.CurrHop'.{Temp0'}_KeyTS) =|>
    %%---Tag.Temp0 = Server.NonS----
        State' := 2     /\ M3' := {Temp0'}_KeyTS
        /\ M4' := {xor(Temp0', M3')}_KeyTS
        /\ NonT' := new() /\ M5' := {xor(NonT', M4')}_KeyTS
        /\ KeySession' := xor(NonT', Temp0')
        /\ SND(Tag.Server.M4'.M5')
        /\ secret(Temp0', sec_temp0_ts, {Tag, Server})
        /\ secret(NonT', sec_nont_ts, {Tag, Server})
        %%Tag hopes that NonS(Temp0) will permit to authenticate him
        /\ witness(Tag, Server, auth_nons_st, Temp0')

    2.  State = 2 /\ RCV(Server.Tag.M6'.{NonT.Tss'.Sgn'}_KeySession') =|>
    %%-------------Phase-3 starts here ----------------------
    State' := 3
        /\ SDTA' := new()
        /\ M8':= {xor(Old_Ts, Tss')}_KeySession'
        /\ M9' := {xor(M8', SDTA')}_KeySession'
        /\ SND(Tag.Server.M8'.M9')
        /\ secret(Old_Ts, sec_oldts_ts, {Tag, Server})
        /\ secret(Tss', sec_tss_ts, {Tag, Server})
        /\ secret(SDTA', sec_sensors_ts, {Tag, Server})
        /\ secret(Sgn', sec_sgn_ts, {Tag, Server})
        %%Tag checks that it receives the same nonce that it sent at step 1
        /\ request(Tag, Server, auth_nont_ts, NonT)
```

```
        %%Tag hopes that Old_Ts will permit to authenticate him
        /\ witness(Tag, Server, auth_ts_st, Old_Ts)
end role

%%----------------ROLE SESSION--------------------------------
role session(
    SPhone, Server, Tag     : agent,
    Usr_ID                  : text,
    Usr_Key                 : symmetric_key,
    CPU_ID, T_ID            : text,
    Hash                    : hash_func,
    Usr_PK, Srv_PK          : public_key,
    Old_Ts                  : text

) def=
local
    SNDSP, RCVSP, SNDS, RCVS, SNDT, RCVT: channel(dy)
const
    readreq, currhop    : nat,
    keyspt, keyts       : symmetric_key

composition
  role_SPhone(SPhone, Server, Tag, Usr_ID, CPU_ID, Usr_Key,
                    keyspt, Hash, readreq, Usr_PK, Srv_PK, SNDSP, RCVSP)
  /\ role_Server(SPhone, Server, Tag, Usr_Key, keyts, Hash,
                        Usr_PK, Srv_PK, currhop, SNDS, RCVS)
    /\ role_Tag(SPhone, Server, Tag, readreq, keyspt,
                        keyts, T_ID, Old_Ts, SNDT, RCVT)
end role

%%-------------ROLE ENVIRONMENT----------------------
role environment() def=
```

```
const sphone, server, tag            : agent,
      cpu_id1, cpu_id2, cpu_id3          : text,
        usr_id1, usr_id2, usr_id3        : text,
        t_id1, t_id2, t_id3, old_ts   : text,
      usr_key1, usr_key2, usr_key3       : symmetric_key,
        fhash                                  : hash_func,
      sec_cpuid_sps, sec_spt_sps, auth_spt_sps, sec_cpuid_ssp, sec_spt_ssp,
        sec_tid_ts, auth_nons_st, auth_nont_ts, auth_ts_st, sec_nons_st,
        sec_temp3_st, sec_nons1_st, sec_ts_st, sec_result_ssp, sec_temp0_ts,
        sec_nont_ts, sec_oldts_ts, sec_tss_ts, sec_sensors_ts,sec_sgn_st,
        sec_sgn_ts                       : protocol_id,
        usr_pk, srv_pk                   : public_key

intruder_knowledge = {sphone, server, tag, usr_pk, srv_pk}

composition
   session(sphone, server, tag, usr_id1, usr_key1, cpu_id1, t_id1,
                fhash, usr_pk, srv_pk, old_ts)
end role
%%-----------------------------------------------------------------
goal
   secrecy_of sec_cpuid_sps, sec_spt_sps, sec_cpuid_ssp, sec_spt_ssp, sec_tid_ts,
                sec_nons_st, sec_temp3_st, sec_nons1_st, sec_ts_st, sec_result_ssp,
                sec_temp0_ts, sec_nont_ts, sec_oldts_ts, sec_tss_ts, sec_sensors_ts,
                sec_sgn_st, sec_sgn_ts

     authentication_on auth_spt_sps, auth_nons_st, auth_nont_ts, auth_ts_st
end goal
environment()
```

**APPENDIX D  Source Code of STRAP in SPDL**

```
//-------------------------------------------------------------------------------------------
//                        STRAP Protocol Code in SPDL
//-------------------------------------------------------------------------------------------

//The protocol description

usertype String,  TimeStamp;

hashfunction Hash;
const fxor: Function;

protocol strap (SPhone, Server, Tag)
{

//-----------------ROLE SPHONE-----------------------------------------------------------
      role SPhone
      {
            fresh NotOK, OK: Nonce;
            fresh TSP: TimeStamp;
            var CpuID: String;
            var Tsp: TimeStamp;
            const USRID, CPUID, REQ: String;
            var Result: Nonce;

//-------------------------Phase-1 starts here-----------------------------------------
            macro M1 = { USRID, CPUID, TSP}k(SPhone, Server);
            send_1(SPhone, Server, M1);
            macro H1 = Hash(CPUID, TSP);
            recv_2(Server, SPhone, {{Hash(CpuID, Tsp)}pk(SPhone)}sk(Server));
            macro H2 = Hash(CpuID, Tsp);
```

```
            match(H1, H2);

//-------------------------Phase-2 starts here-----------------------------------------
            send_3(SPhone, Tag, {SPhone, REQ}k(SPhone, Tag));

//-------------------------Phase-3 starts here-----------------------------------------
            macro h1 = Hash(OK, TSP);
            macro h2 = Hash(NotOK, TSP);
            macro ResHash = Hash(Result, Tsp);
            recv_10(Server, SPhone,{{ ResHash}pk(SPhone)}sk(Server));
            match(h1, ResHash);
            match(h2, ResHash);

//-----------------------SPhone Claim Descriptions-----------------------------------

            claim_sp1(SPhone, Secret, CPUID);
            claim_sp2(SPhone, Secret, TSP);

            claim_sp3(SPhone, Niagree);
            claim_sp4(SPhone, Nisynch);
            claim_sp5(SPhone, Alive);
            claim_sp6(SPhone, Weakagree);
      }

//-----------------ROLE SERVER-------------------------------------------------------
      role Server
      {
            fresh Tss: String;
            var CpuID, UsrID, Tid: String;
            const CPUID, TID: String;
            fresh NonS,  TS, Result, CurrHop: Nonce;
            var NonSS, NonTT, OldTs, Ts, SDTAS: Nonce;
```

```
var temp3,  M4: Ticket;
var Tsp: TimeStamp;

//-------------------------Phase-1 starts here----------------------------------------
recv_1(SPhone, Server, {UsrID, CpuID, Tsp}k(SPhone, Server));
match(CPUID, CpuID);
macro M2 = {{Hash(CpuID, Tsp)}pk(SPhone)}sk(Server);
send_2(Server, SPhone, M2);

//-------------------------Phase-2 starts here----------------------------------------
recv_4(Tag, Server, {Tid}k(Tag, Server));
match(TID, Tid);
macro M3 = {NonS}k(Tag, Server);
send_5(Server, Tag, CurrHop, M3);
recv_6(Tag, Server, M4, {fxor(NonTT, M4)}k(Tag, Server));
macro M5 = {fxor(NonTT, M4)}k(Tag, Server);
macro temp1 = fxor(NonS, M3);
match(NonSS, temp1);
macro temp2 = fxor(NonTT, M4); // temp2 = NonT x M4
match(temp3, fxor(temp2, M4)); // temp3 = NonT
macro KeyS = fxor(NonS, NonTT);
macro M6 = {fxor(M5, temp3)}k(Tag, Server);
send_7(Server, Tag, M6);

//-------------------------Phase-3 starts here----------------------------------

macro Sgn1 = Hash(TID, CurrHop, Tss);
macro M7 = {NonTT, TS,  Sgn1}KeyS;
send_8(Server, Tag, M7);
macro M8 = {fxor(OldTs, Ts)}KeyS;
macro M9 = {fxor(M8, SDTAS)}KeyS;
recv_9(Tag, Server,M8, M9);
```

```
            macro temp6 = fxor(OldTs, Ts);
            macro temp7 =  fxor(temp6, Ts);// temp7 = OldTs
            macro temp8 =  fxor(M8, SDTAS);
            macro temp9 =  fxor(M8, temp8); // temp9 =? SDTAS
            match(temp9, SDTAS);
            macro ResHash = Hash(Result, Tsp);
            send_10(Server, SPhone,{{ ResHash}pk(SPhone)}sk(Server));

//----------------------Server Claim Descriptions---------------------------------

            claim_s1(Server, Secret, CpuID);
            claim_s2(Server, Secret, Tsp);

            claim_s3(Server, Niagree);
            claim_s4(Server, Nisynch);
            claim_s5(Server, Alive);
            claim_s6(Server, SKR, KeyS);
            claim_s7(Server, Weakagree);
      }

//--------------------ROLE TAG-------------------------------------------------
      role Tag
      {
            const REQ, TID: String;
            var Req: String;
            var NonS, NonTT, Ts, CurrHop: Nonce;
            fresh NonT, OldTs, SDTA: Nonce;
            var  Sgn: Ticket;

//-------------------------Phase-2 starts here----------------------------
            recv_3(SPhone, Tag, {SPhone, Req}k(SPhone, Tag));
            match(REQ, Req);
```

```
            send_4(Tag, Server, {TID}k(Tag, Server));
            recv_5(Server, Tag, CurrHop, {NonS}k(Tag, Server));
            macro M3 = {NonS}k(Tag, Server);
            macro M4 = {fxor(NonS, M3)}k(Tag, Server);
            macro M5 = {fxor(NonT, M4)}k(Tag, Server);
            send_6(Tag, Server, M4, M5);
            recv_7(Server, Tag, {fxor(M5, NonTT)}k(Tag, Server));
            macro KeyS = fxor(NonS, NonTT);

//-------------------------Phase-3 starts here-------------------------
            recv_8(Server, Tag, {NonTT, Ts, Sgn}KeyS);
            macro M8 = {fxor(OldTs, Ts)}KeyS;
            macro M9 = {fxor(M8, SDTA)}KeyS;
            send_9(Tag, Server,M8, M9);



//-------------------Tag  Claim Descriptions-------------------------
            claim_t1(Tag, Secret, OldTs);
            claim_t2(Tag, Secret, NonT);
            claim_t3(Tag, Secret, SDTA);
            claim_t4(Tag, Secret, TID);
            claim_t5(Tag, Secret, NonS);
            claim_t6(Tag, Secret, Ts);

            claim_t7(Tag, Niagree);
            claim_t8(Tag, Nisynch);
            claim_t9(Tag, Alive);
            claim_t10(Tag, SKR, KeyS);
            claim_t11(Tag, Weakagree);
      }

}
```