SECURE MULTIMEDIA COMMUNICATION
IN SMART DEVICES
REINFORCED BY USING ONE-TIME KEYS

by

ÖMER MERT CANDAN

Submitted to the Graduate School of Engineering and Natural Sciences
in partial fulfillment of
the requirements for the degree of
Master of Science

Sabancı University
July 2017

SECURE MULTIMEDIA COMMUNICATION

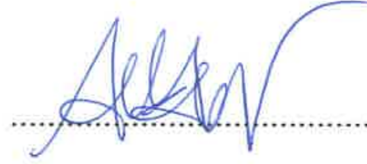
IN SMART DEVICES

REINFORCED BY USING ONE-TIME KEYS

APPROVED BY:

Prof. Albert Levi ....

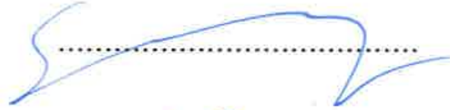
(Thesis Supervisor)



Asst. Prof. Cengiz Toğay ....

(Thesis Co-supervisor)

Assoc. Prof. Selim Balcısoy ....



Asst. Prof. Kamer Kaya ....

Prof. Erkay Savaş ....

DATE OF APPROVAL: 27/07/2017

© Ömer Mert Candan 2017

All Rights Reserved

ABSTRACT

SECURE MULTIMEDIA COMMUNICATION

IN SMART DEVICES

REINFORCED BY USING ONE-TIME KEYS

ÖMER MERT CANDAN

M.Sc. Thesis, July 2017

Supervisor: Prof. Albert Levi

Co-supervisor: Asst. Prof. Cengiz Toğay

Keywords: one-time key, multimedia communication, smart card, hash chain, security

Recently, smart devices have become more and more prevalent in the daily life. The spread of these devices introduced various use cases; however, communication has always been their primary functionality. With the development of WebRTC (Web Real-Time Communication) and the availability of this technology on smart devices, applications offering real-time multimedia communication features will become more pervasive. Though WebRTC presents a promising set of standards and interfaces for the task of carrying data from one end to another, there are security issues that are left in the hands of the application developers. In this thesis, we aim to achieve secure multimedia communication by tackling the key generation and distribution issue of WebRTC platform using a smart card for secure storage and operations. We tested different cryptographic algorithms on smart cards, and resultantly we designed a mechanism based on hash chains.

This mechanism allowed synchronous generation of keys at both sides. The mechanism was implemented and tested on different brands of Java Cards. The results of the tests indicate that it is possible to produce a key under one-second time. In addition, the results were analyzed to optimize generation times of particular keys by adjusting chain length parameter of the mechanism. Consequently, the key generation method was integrated into Media Security Platform of Netaş Telecommunications A.Ş., which is based on WebRTC. The integration was performed under the guidance of a signaling scheme drafted for the message traffic for the key agreement. In conclusion, the successful integration and better results indicate an improvement over a previously used public key system.

ÖZET

Akıllı Cihazlarda Tek Kullanımlık Anahatar

ile Güçlendirilmiş

Güvenli Çoklu Ortam İletişimi

ÖMER MERT CANDAN

Master Tezi, Temmuz 2017

Danışman: Prof. Dr. Albert Levi

Eş-danışman: Asst. Cengiz Toğay

Anahtar Sözcükler: tek kullanımlık anahatar, multimedya iletişim, akıllı kart, özet zinciri, güvenlik

Son yıllarda akıllı cihazlar günlük hayatta önemli bir yer edindi. Bu cihazların yaygınlaşması, onlara birçok yeni işlev kazandırmakla beraber, yine de temel amaçları iletişim olarak kaldı. WebRTC (İnternet Tabanlı Gerçek Zamanlı İletişim) teknolojisinin ortaya çıkması ve akıllı cihazlarda kullanılabilir olması, gerçek zamanlı multimedya iletişimine olanak veren uygulamaların artmasına neden olacaktır. WebRTC'nin amacı uçtan uca bilgi taşınması için standartlar ve programcı arayüzleri belirlemek olsa da, işin güvenlik kısmı uygulama geliştiricilere bırakılmıştır. Bu tezde, akıllı kartların sağladığı güvenli depolama ve işlem özelliklerinin yardımı ile, WebRTC için güvenli anahtar üretilmesi ve dağıtımı sorunları ele alınarak güvenli çoklu ortam iletişimi kurulumu hedeflenmektedir. Değişik kriptografik algoritmalar akıllı kartlar üzerinde denenmiş ve sonuç olarak özet zinciri

üzerine bir yöntem kullanılmasina karar verilmiştir. Tasarlanan mekanizma değişik marka Java kartlar üzerinde calıştırılmış ve testlerin sonucları 1 saniyenin altında bir sürede anahtar üretiminin mümkün olduğunu göstermiştir. Buna ek olarak, özet zinciri uzunluğu değiştirilerek çeşitli analizler yapılmış ve bunun sonucunda hedeflenen bir anahtarın mümkün olan en iyi sürede üretilebilmesi için gerekli olan zincir uzunlukları hesaplanmıştır. Devamında, anahtar üretim mekanizmasının WebRTC teknolojisine dayanan Medya Güvenlik Platformu ile entegrasyonuna yer verilmiştir. Mekanizmanın sisteme uyumu için tasarlanan sinyalleşme trafiği göz önüne alınarak, entegrasyon başarı ile tamamlanmıştır. Sonuçlar, daha önce kullanılan açık anahtarlı sisteme göre daha iyi performans alındığına işaret etmektedir.

dedicated to everyone who've been patient with me,

especially my family

ACKNOWLEDGMENTS

Special thanks to Prof. Albert Levi, my advisor and a major contributor of this thesis.
I am truly grateful for his endless guidance and sincere attention.

I also thank my co-supervisor Asst. Prof. Cengiz Toğay for his contributions and support along the way.

I learned a great deal from the lectures of Assoc. Prof. Selim Balcısoy, Asst. Prof. Kamer Kaya and Prof. Dr. Erkay Savaş, and thank them for participating in my defense jury.

The project under which this thesis has been produced has been supported by T.C. Ministry of Science, Industry and Technology under San-Tez program grant STZ.0805.2014 (later the project has been taken over by TÜBİTAK TEYDEB grant 112D032).

TABLE OF CONTENTS

1 Introduction **1**

2 Background **4**

2.1 WebRTC (Web Real-Time Communication) 4

2.2 Java Card 6

2.3 Cryptographic Hash Functions 8

2.3.1 Preimage Resistance 9

2.3.2 Second Preimage Resistance 9

2.3.3 Collision Resistance 10

2.4 HMAC 10

2.5 One-Time Password 11

2.5.1 HOTP 11

2.5.2 TOTP 12

2.6 Hash Chain 12

3 Proposed Method **15**

3.1 Introduction 15

3.2 Proposed Structure 18

3.3 Call Establishment Protocol 20

3.3.1 Possible Scenarios During Key Establishment 22

3.3.2 Distribution of Seed Values into Smart Cards 24

3.4 Integration with Media Security Platform 24

3.4.1 Pre-integration Preparation Work 25
3.4.2 The Integration 30

4 Performance Evaluation 36

4.1 Key Generation Timings 36
4.2 Calculating the Optimal Chain Length for a Target Key 38
4.3 Post-integration Timings 40
4.4 Memory Usage on Java Card 42

5 Conclusion 43

LIST OF TABLES

3.1 Unit Times on Different Cards (ms) 16

3.2 Error Codes of the Protocol 23

4.1 The Optimal Chain Lengths for Selected Keys 40

4.2 Tests with MSP Application Using SHA-1 in Netaş's intranetwork 41

4.3 Tests with MSP Application Using SHA-256 in Netaş's intranetwork . . . 41

4.4 Tests with MSP Application Using SHA-256 outside Netaş network . . . 41

4.5 Memory Usage of Applets on Java Card 42

LIST OF FIGURES

2.1 Key Agreement in WebRTC 5
2.2 Overview of Public Key Setting in Media Security Platform 6
2.3 Smart Card Chip 7
2.4 Java Card Architecture (taken from [9]) 8
2.5 Preimage Resistance 9
2.6 Second Preimage Resistance 9
2.7 Collision Resistance 10
2.8 Lamport's Password Authentication Scheme 13

3.1 Hash Chain Tests Using SHA-1 17
3.2 Hash Chain Tests Using SHA-256 17
3.3 Hash Chaining MS1 19
3.4 Hash Chaining MS2 19
3.5 Second Phase of the Mechanism 19
3.6 Two Way Hash Chain Mechanism 20
3.7 Signalling Protocol for Media Security Platform. 21
3.8 Test Application on Android 26
3.9 Result of a Test Run on Android 27
3.10 SHA1 Chain Length with Android 28
3.11 SHA256 Chain Length with Android 28
3.12 $JavaCardApi$ Class 29
3.13 $JavaCardKey$ Class 29

3.14 $JavaCardApiIml$ Class 30
3.15 API Calls After Integration with the Media Security Platform 31
3.16 The Caller Initiating in Android Application 32
3.17 Caller Sends the Call Request 33
3.18 The Callee's Screen After Receiving the Call Request 33
3.19 The Call is Accepted 34
3.20 The Call has Started on Caller's Application 34
3.21 The Call has Started on Callee's Application 35

4.1 Tests with Different Chain Lengths (ms) 37
4.2 Generation Time of Different Keys (ms) 39

Chapter 1

Introduction

The technology of mobile devices has shown a great deal of improvement over the past decade. While this fast paced trend continued, the devices became more powerful and smaller in size. The transformation from brick-like phones to pocket sized phones, from room-sized computers to handheld tablets led these devices invade daily life. All the mobile devices, ranging from a smart watch to a portable personal computer have one aspect in common, that is Internet connectivity. The major demand of Internet access provoked the rapid development of the infrastructure of Internet as well as the services that rely on it. One of these services, namely WebRTC (Web Real-Time Communication), aims to allow audio and video communication over the Internet [23]. Since the Internet is publicly accessible and therefore insecure, WebRTC is designed to provide security by end-to-end encryption. What WebRTC does not provide is a standard for generation and distribution of the keys required for the security of the communication.

Media Security Platform (MSP) developed by Netaş Telecommunication A.Ş., tackles the problem of key generation for multimedia communication in WebRTC environment. The platform utilizes a public key cryptography based setting, each participating user having a public/private key pair. The mobile environment is not the ideal place to store private keys [20]; therefore they are stored in smart cards which also provide secure cryptographic functions. In the existing MSP of Netaş, before establishing a secure communication channel, parties generate a session key and encrypt it with the public key of

the other party. The encrypted key is sent after being signed with the sender's private key. Four modular exponentiation operations are needed in this setup, two of them performed in the smart card environment, which slows the initiation process before call. Our main purpose is to devise a mechanism that will shorten the time required by the initial signaling process. The proposed scheme will allow remote users to create a unique and common key in a secure and an efficient way. To eliminate the cost of communication spent for key exchange, we came up with a structure based on the idea of hash chains. The users will create their keys by applying hash functions in a chain-wise manner to some initial data. Since this is a deterministic process, applying hash chain on the same data would yield the same result for different users. In order to generate the same key at both ends, both parties need to share a secret information beforehand. The security of the protocol we design depends solely on the shared secret, namely the "seed". Therefore, the seeds will be stored inside smart cards and any computations on them will also be performed in the card. The whole key generation operation begins and ends inside the card, only the result is visible at the end of the operation. This way, the seeds or the intermediate values generated from the seeds never leave the card. The possible security threats existing in the mobile environment are circumvented with this approach.

We have set out to design a new key generation system that produces one-time keys in a type of smart card called Java Card. Java Cards provide secure storage and atomic transactions on a Java-based environment [3]. The product of our design had to meet strict security requirements. One of these requirements was the security of the future and past keys generated by the users of the platform. We had to keep in mind that, in the event of a key compromise, none of the past keys should be revealed. In addition, when a key is captured, our mechanism must keep on producing consequent keys not guessable by any means. The other requirement was that two separate parties must have been able to produce the same key without exchange of information. Another requirement was to produce these keys in a timely manner. To achieve these requirements, we knew that the involving parties should share some information ahead of time. However, this shared information is the most vulnerable part of the whole setting. Therefore, we have made the decision

to choose an external device that will store this information, process this information and display an output when needed. The choice of smart cards, especially Java Cards for our case brought upon its challenges. We have noticed that the smart card platform is indeed very limited in the aspect of memory and computing power. These restrictions made us realize that we are not free in our decisions while we are designing our system. As a result of this, we have moved to perform tests with different cryptographic algorithms. As soon as we received the results of our tests, we observed significant performance disparity between them. Some of the algorithms have performed considerably slow or did not work at all on the smart cards. After a brief analysis of these preliminary results, we determined the main component of our design by the process of elimination. Therefore, we attained to design a scheme that does not only work in theory, but is feasible, implemented and tested on real devices. Our main contribution can be attained to the fact that our mechanism is designed carefully to perform well in the current restricted state of smart card technology. We have performed necessary tests on smart cards and shown that the mechanism does work successfully under one-second threshold, which is acceptable in the context of real-time applications.

The rest of this thesis is organized as follows. In Chapter 2, we provide some background information necessary for understanding the basis of the work. In Chapter 3, we explain our design and how it works in detail. The reasons behind the choices made in the design will be revealed. We will provide steps for generating a single key. Chapter 4 consists of the performance evaluation of the provided scheme. We disclose the tests and their results. In Chapter 5, we provide a conclusion to wrap up.

Chapter 2

Background

2.1 WebRTC (Web Real-Time Communication)

WebRTC (Web Real-Time Communication) is a collection of APIs for modern browsers, such as Google Chrome and Mozilla Firefox, that enable peer-to-peer Real-Time Communication (RTC) without plugins or other requirements [22]. The components of WebRTC technology provide infrastructure for high quality audio, video and other data transfer between browsers or any application that implement the WebRTC API. The aim of WebRTC project is to provide a set of standards that will define the future of web based communication. WebRTC is an open source project, which makes it an important move for web based technology in a sense that relieves developers from relying on proprietary solutions. The media transfer between clients happens in a peer-to-peer fashion; however, this transfer of information is preceded by an initiation process that requires a server in between. With the help of an intermediary server, the clients have to exchange information to establish the channel with the help of Session Description Protocol (SDP) [8]. These initialization events are part of the signaling process which is designed to overcome the difficulties introduced by Network Address Translation (NAT) [19]. Translation of dynamic addresses to private addresses and vice versa puts a hold on the possibility of creating an immediate peer-to-peer connection[1]. Interactive Connectivity Establishment (ICE) framework comes into action at this point; it helps to initialize the connection by trying out different

connection methods, when the channel cannot be established with convenience [18]. The key agreement part of the signaling phase is relevant for the scope of our work.

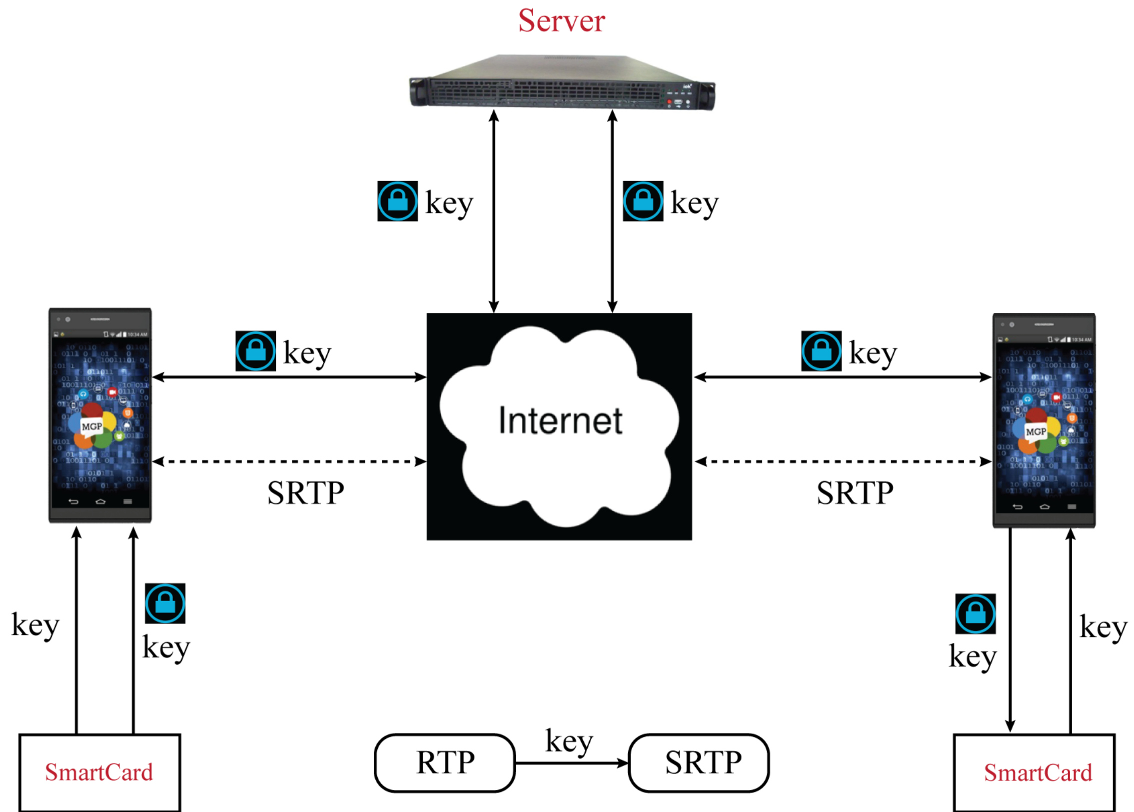


Figure 2.1: Key Agreement in WebRTC

In Netaş's Media Security Platform, key agreement happens at the signaling phase, shown in Figure 2.1. The key generated by the caller's smart card is encrypted with the public key of the other party and transferred to the demanding application. The encrypted key is signed by the application and sent to the intermediary Media Security Server, which then transfers the key to the receiver of the call. The callee verifies the signature and passes the encrypted key to the smart card for decryption. The plain key returned from the smart card is used to establish a secure multimedia communication channel. An overview of this public key based scheme is shown in Figure 2.2.

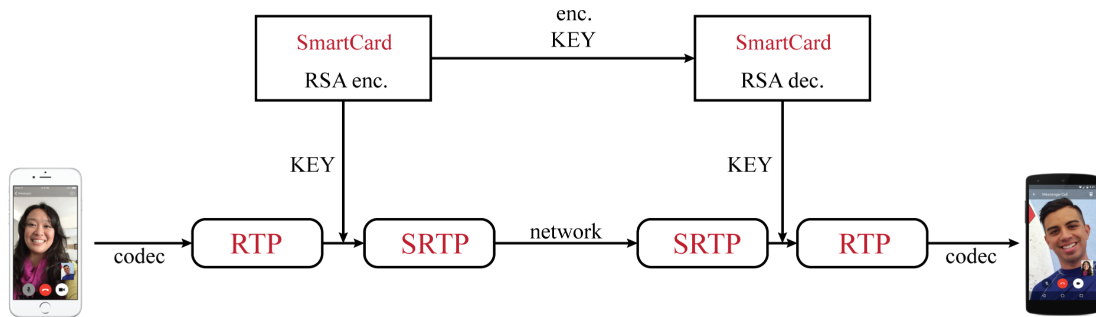


Figure 2.2: Overview of Public Key Setting in Media Security Platform

2.2 Java Card

Smart cards are widely used in areas such as banking, security systems, personal identification and so on. A smart card is a plastic card that houses a chip containing a Central Processing Unit (CPU), Read-Only Memory (ROM), Electrically Erasable Programmable Read-Only Memory (EEPROM), Random Access Memory (RAM) and a unit for input/output operations, simply presented in Figure 2.3. For the time being, smart cards carry approximately 1KB of RAM and 64KBs of flash memory space (EEPROM). The ROM section of the smart card is used to store the operating system of the smart card and is not accessible by developers. The contents of RAM is lost when the smart card is unpowered, while EEPROM provides persistent but very slow storage compared to RAM. Smart cards are operated by a card reader or sometimes called Card Acceptance Device (CAD) that is connected to a computer, a terminal or as in our case a smart device. The card operators provide power and clock signals to the chip embedded in the card.

Smart cards are assumed to securely store its contents, however, this assumption may not always hold true. While current smart cards are marketed with the promise of tamper-resistance, there have been various tampering techniques introduced in the past [10]. In our case, the smart card environment is a far better alternative to implement our key generation mechanism into than the operating system environment of a smart device.

Java Card is a specific type of smart card that utilizes a restricted subset of Java Environment. Java Cards are initialized in the manufacturing process with Java Card Runtime

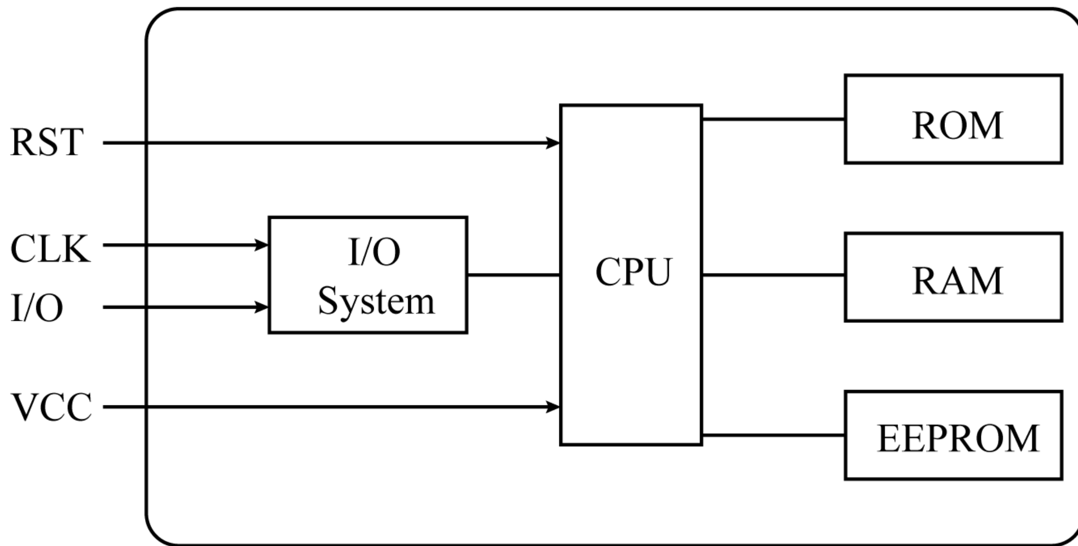


Figure 2.3: Smart Card Chip

Environment (JCRE) written in their ROMs. EEPROM stores the applications on Java Card - or applets - and static data related to those applets, while the RAM is used by the applets as temporary storage during runtime. JCRE holds Java Card Virtual Machine and Java Card API on top of which the applets operate, as illustrated in Figure 2.4. Contrary to the earlier smart card applications, Java Card applets work on any card that runs JCRE, independent of the brand of the chip. A Java Card can hold multiple applets and provides a firewall to restrict access between said applets.

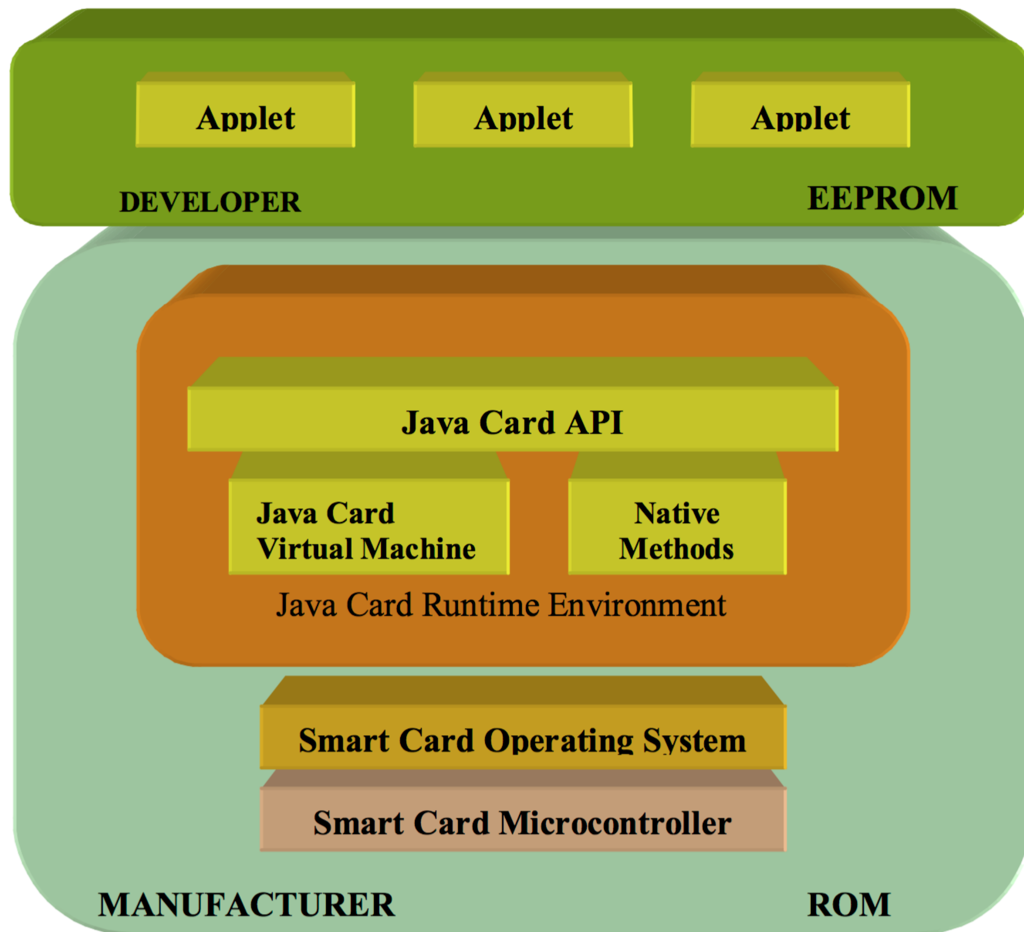


Figure 2.4: Java Card Architecture (taken from [9])

2.3 Cryptographic Hash Functions

A hash function produces a fixed size of output, also called hash value or digest, from inputs of any length. Hash functions are very fast and utilized for various purposes in security. An example usage might be to produce the hash of a long document and securely store this hash for future verification that the document is not modified. If there is a modification in the document, then applying the hash function on the modified version will create a digest that does not match with the original hash. Since the input space is often larger than the output space, the outputs of the hash functions are not in one-to-one correspondence with the inputs. This means that the hash function might map different inputs to the same hash value. The case where two different inputs give the same hash

result is called a collision.

Cryptographic hash functions are a sub-type of hash functions that satisfy some security properties. The security of hash functions depends on preimage resistance, second preimage resistance and collision resistance [17].

2.3.1 Preimage Resistance

From the output of the hash function, it must be difficult to find the input that produces the hash value. The hash value should not reveal any information about the input.

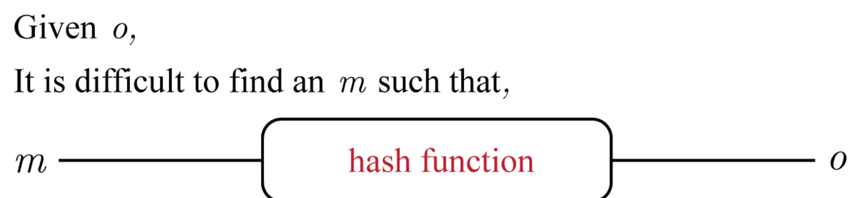


Figure 2.5: Preimage Resistance

2.3.2 Second Preimage Resistance

For an input and its hash, it must be difficult to find a different input that produces the same hash value.

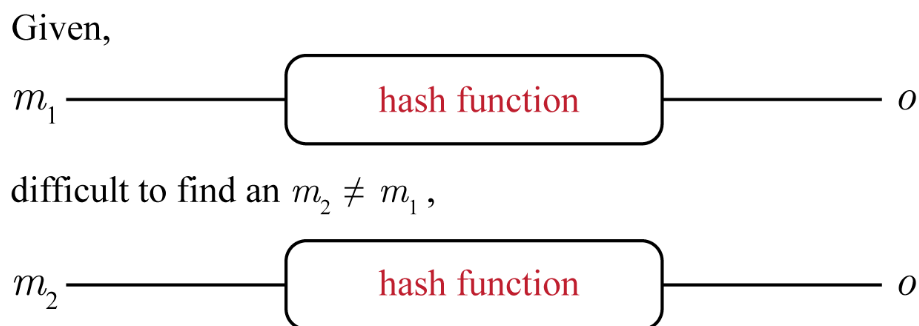


Figure 2.6: Second Preimage Resistance

2.3.3 Collision Resistance

It must be difficult to find any input pairs that produce the same hash output.

It is difficult to find any pair of (m_1, m_2) such that $m_1 \neq m_2$ and,

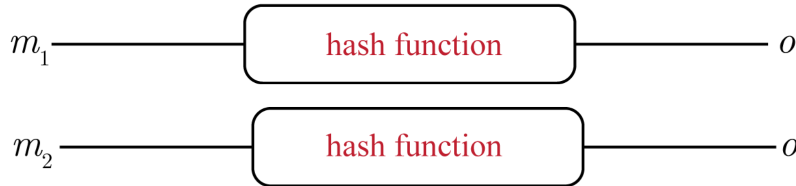


Figure 2.7: Collision Resistance

2.4 HMAC

HMAC (Hash-based Message Authentication Code) is a cryptographic mechanism providing integrity check by validating messages using a secret key [11]. It involves a cryptographic hash function and a secret key. The hash function should be an iterative function that operates on blocks, such as a hash function from SHA family [7][6]. The name of the HMAC scheme is correlated with the underlying hash function, for example if SHA-256 is being used the scheme is called HMAC-SHA-256. In addition, the security of HMAC depends heavily on the quality of the hash function. With this in mind, cryptographically secure hash functions such as SHA-256 and beyond are often the choice for HMAC operations. Calculation of HMAC of message M using key K and hash function h is as follows.

$$HMAC(K, M) = h(K \oplus opad \| h(K \oplus ipad \| M)) \quad (2.1)$$

HMAC consists of two hash operations involving the message, the key and constant values called $ipad$ and $opad$. Let us assume that the hash function operates on blocks of size b and produces a hash output of size n. The $ipad$ and $opad$ constants are 64 bytes long byte strings and are repetition of the bytes 0x36 and 0x5C, respectively. As the first

step of HMAC process, if needed, the key is padded with zeroes until it is of length b. Padded key is XOR'ed with the $ipad$, then concatenated with the message which forms the inner part. The first hash operation is applied on this inner part. As the next step, the padded key is XOR'ed with the $opad$, then concatenated with the digest coming from the previous step to form the outer part. The second hash operation is applied on the outer part to produce the result of the HMAC operation as seen in Equation 2.1.

2.5 One-Time Password

As its name suggests, *one-time password* algorithms are designed to produce passwords that are supposed to be used only once. The main advantage of this scheme is to prevent attack scenarios where the adversary captures a previously used password. Implementing one-time password systems usually are not as straightforward as implementing a system that depends on a single (master) password. Therefore, other devices come to help in the one-time password settings in the creation of one-time passwords. This is known as *two factor authentication* and its security depends on not only what the user knows but also on what the user possesses. There are different algorithms to produce single use passwords. We will discuss two of them, one is based on HMAC and the other one uses the time as a source of input.

2.5.1 HOTP

HMAC-based One-Time Password Algorithm (HOTP) [13], uses a key, a counter and HMAC-SHA-1. Obviously the key must be kept secret and should be of adequate length which is at least 160 bits suggested in the RFC document [13]. After a run of HOTP, the counter value is incremented. This scheme guarantees that for every iteration of the algorithm, the outcome will be different than the previous one. By sharing a secret key, K, and a synchronized counter, C, two remote parties can successfully generate a one-time password as shown below.

$$HOTP(K, C) = Truncate(\textit{HMAC-SHA-1}(K, C)) \tag{2.2}$$

Let us assume that we want to have a result with d digits after the execution of HOTP algorithm. HOTP algorithm applies HMAC-SHA-1 on counter with the key, then selects specific bits of the result and shortens it to d digits as in Equation 2.2. The truncation process looks at the low-order 4 bits of the last byte of the HMAC output. This becomes the $index$ to the bytes that will be selected again from the HMAC result. The bytes that are in the range of $[index, index + 3]$ will make up a 32 bit number. As a final step, modulo operation is applied with modulus being 10^d, to get the result in the expected range.

2.5.2 TOTP

TOTP (Time-based One-Time Password Algorithm) is very similar to HOTP [14]. The only difference is that TOTP uses Unix time as its counter value.

$$TOTP = HOTP(K, T) \tag{2.3}$$

Here in equation 2.3, T is the number that represents how many time steps have been taken from a determined initial time. Assuming that the initial time is determined as T_0, and a time step is defined as X, we calculate the number of time steps in Equation 2.4 below.

$$T = \left\lfloor \frac{CurrentTime - T_0}{X} \right\rfloor \tag{2.4}$$

2.6 Hash Chain

A hash chain is produced by applying hash operation on a given data successively. When hash functions are chained, the result of one hash becomes the input of the upcoming hash function. If we represent the hash function with h and the length of the hash

chain with L, then the hash chain F is:

$$h^L(m) = \underbrace{h...h(h(h(m))...)}_{\text{L times.}} \tag{2.5}$$

The idea of chaining hash functions first appears in the seminal work of Lamport [12]. In the context of remote authentication, a password is used for identification. Instead of choosing a single password and sending this to the server for every access, a scheme based on chaining hash functions is proposed. The user chooses an initial value x and applies hash chain operation on this value. Let us assume that, the length of the hash chain is 1000, which indicates that the hash is taken 1000 consecutive times. The user shares the result of the hash chain with the server, so the server has $F^{1000}(x)$. When the user wants to identify to the server, the previous hash in the chain is sent. Thus, the user calculates $F^{999}(x)$ and sends it to the server. For verification, the server takes the hash of the incoming message from the user and compares it with $F^{1000}(x)$. If the authentication is successful, the server keeps F^{999} and expects to receive F^{998} the next time the user wants to access.

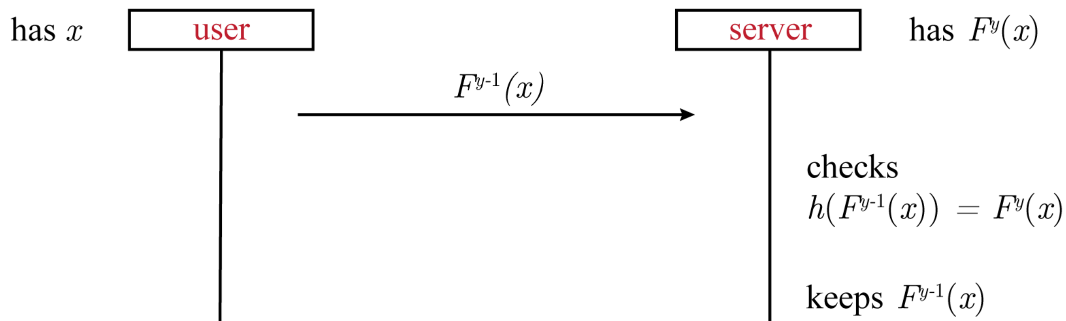


Figure 2.8: Lamport's Password Authentication Scheme

The security of the hash chain method depends on the one-way property of the underlying hash function. This guarantees that, it is very difficult to find F^{y-1} from F^y. Therefore, capturing previously used passwords will not allow to produce future passwords to be used later on. However, obtaining an intermediate hash from a hash chain structure does give away the forward part of the chain since moving forward is simply ap-

plying a hash function. In our proposed scheme, this problem is taken into consideration and dealt with.

Chapter 3

Proposed Method

3.1 Introduction

In this section, we explain our new key generation structure in detail. We will introduce the smart card environment and demonstrate the preliminary test results. The results of test implementations on the cards will reveal the resolution process behind the choices we have made. Then, we will present our two-way hash mechanism step by step.

Our aim, as we have pointed out before, is to generate one-time encryption keys on remote parties for voice/video communication. However, this does not come naturally. Being able to reach the same key requires some pre-shared knowledge. This shared secret, however, is the most essential element concerning the security of the key generation process. Therefore, it is of utmost importance that the shared values must never be disclosed to the outside world. The applications on smart phones are able to store these mentioned secrets, but they do not provide a secure storage. However, the smart cards do make the promise of secure data storage.

We have chosen to work with Java Cards, a type of smart card that runs Java applets. Like other Java applications, Java Card applets run on a virtual machine called Java Card Virtual Machine [3]. This provides separation between the applet and the underlying structure of the card and allows for an applet to be run on different Java Card brands without an issue. There may be more than one applet on the same card. With the applet

Table 3.1: Unit Times on Different Cards (ms)

Algorithm	Variant	Feitian A22	Feitian A40	NXP JCOP	G&D Java Card
HMAC	SHA1	35.7	29.5	45.3	36.8
HMAC	SHA256	55.3	48.4	55.5	39.8
HOTP	HMAC-SHA-1	42.6	36.3	50.2	44.0
RSA 2048	Encrypt	59.0	49.0	205.0	289.7
RSA 2048	Decrypt	677.0	607.0	742.0	767.1

firewall in Java Cards, the applets have no access to each other's data, if not explicitly allowed. The data in an applet is stored in persistent memory (ROM). There is a small non-persistent memory (RAM) to store temporary results of operations. Java Cards are inserted into card reader devices and they are powered through them. When the card is unpowered, the RAM is reset and the data residing in RAM is lost. Overall memory in Java Cards are very limited and usually do not exceed 64 KB. There are different versions Java Card Platform Specifications, and the cards are designed according to one of these specifications. Although the cards are supposed to perform the specified operations in the documents, the cards usually have documented and undocumented missing functionalities. In addition, the performance of the cards are highly varying. While one operation on a card is performed fast, the same operation takes noticeably longer on another brand of Java Card. Since the environment have such restrictions and quirks, initial tests were performed to measure the performance of basic operations. Table 3.1 shows the timing results of some operations tested on different cards.

We have also tested hash chain operations with different chain lengths. Figures 3.1 and 3.2 show how timings vary with different chain lengths and hashing algorithms.

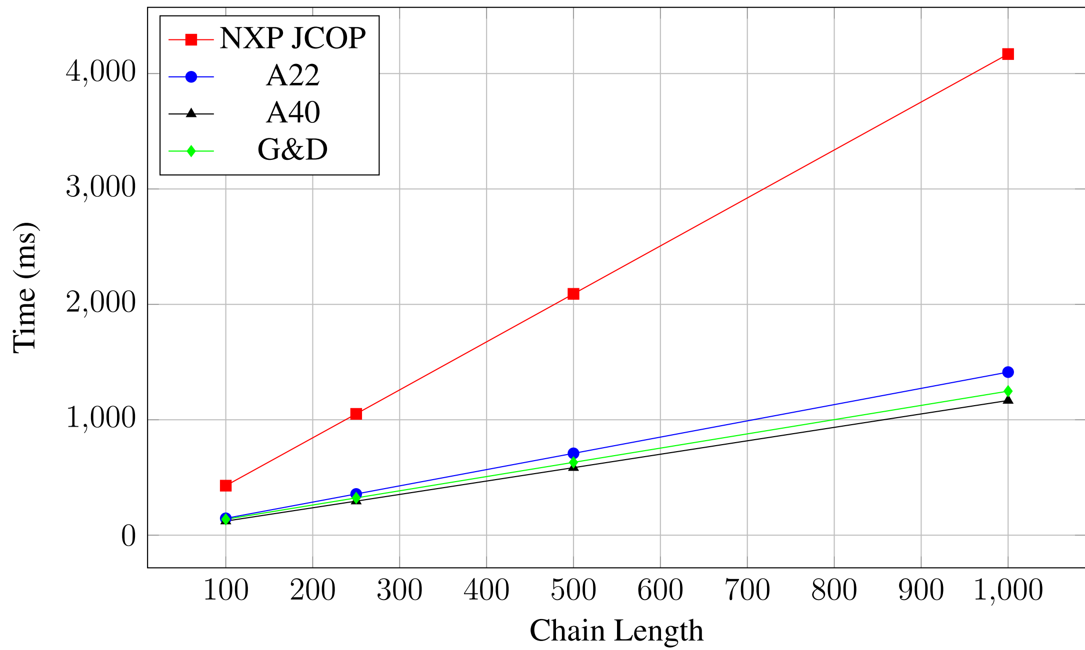


Figure 3.1: Hash Chain Tests Using SHA-1

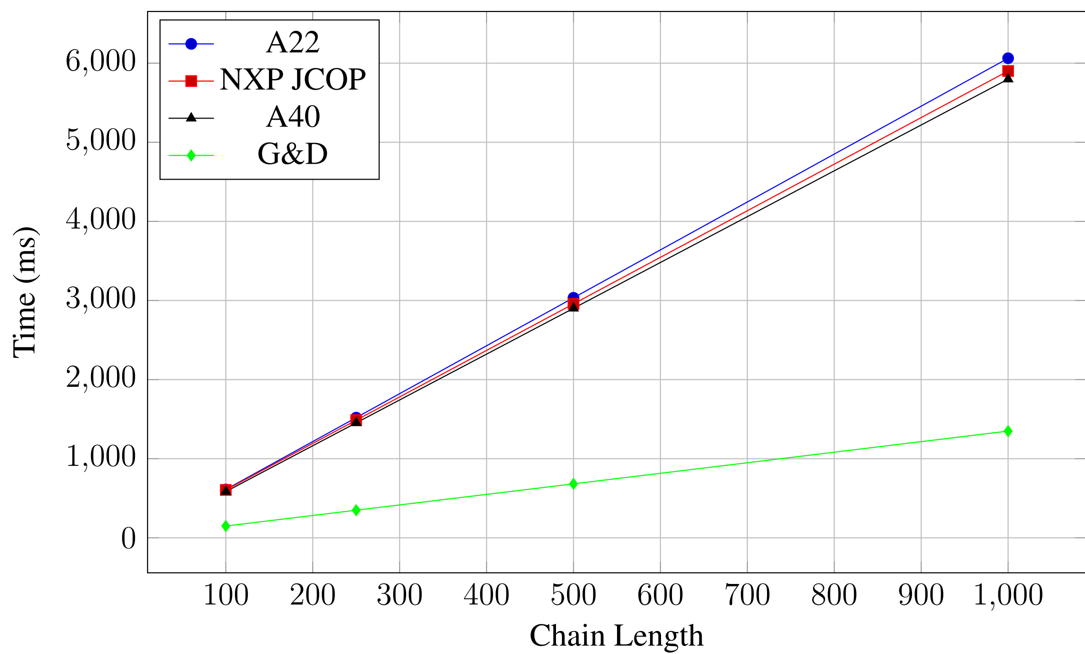


Figure 3.2: Hash Chain Tests Using SHA-256

3.2 Proposed Structure

After a brief analysis of the preliminary tests, we have selected hash chain to be the main component of our key generation method. One-time password scheme proposed by Lamport which is shown in Figure 2.8, utilizes the hash chain in a straight forward manner. The one-way property of the hash function used in the chain guarantees that even if one of the keys are compromised, none of the future keys can be generated. However, previously generated keys until the captured key are easily produced. For our case, the secrecy of the communication depends on the cryptographic key used in that session. Therefore, if any of the past keys are discovered, then the encrypted communication can be deciphered. With this in mind, we set out to draft a scheme that will not only provide security of future conversations, but the past conversations as well. In an event of a key compromise, our scheme should withstand attacks on both past and future keys. The property of backward secrecy requires that if a key is compromised, this must not allow discovery of the keys used in the past. For the other way around, if a system satisfies forward secrecy, then previously captured keys must not lead to prediction of any future keys. As these definitions imply, an important requirement of our system is to ensure both backward and forward secrecy [16]. As a result of this requirement, we have decided to use two separate hash chains. This way, we would be generating keys that would be combination of the results of two hash chain operations, preventing the discovery of past keys.

The purpose of our mechanism is to generate the same keys on two different ends. This is only possible using some secret information that is being known by the involving parties. This secret information, or *master seed*, will be the only factor that the security of our mechanism depends on. The seeds will be written in secured area of smart cards and in no circumstances the seeds will leave that secured area.

Our decision to use two way hash chains requires two separate seeds and we have named these seeds as $MasterSeed1$ and $MasterSeed2$. In addition, we have decided to follow a two-dimensional approach. Therefore, our mechanism needs to keep two different counters for each dimension. These counters are both set to one when the cards

are initialized. We have named these counters simply i and j for the first dimension and the second dimension respectively. The second dimension constitutes the two way hash chain part of our method. We have limited the number of keys generated for each value of the counter (i) in the first dimension. We call this the chain length of our mechanism and represent it with L shortly. Then, the first operation of our mechanism is performing hash chain operation of length i on these seeds as shown in Figures 3.3 and 3.4.

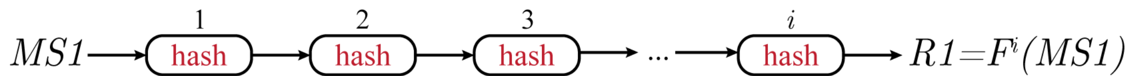


Figure 3.3: Hash Chaining MS1

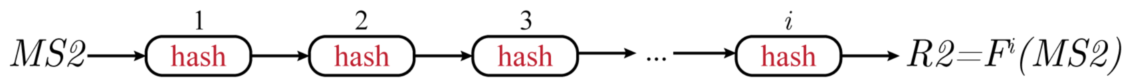


Figure 3.4: Hash Chaining MS2

The intermediate results of chain hashing $MasterSeed1$ ($MS1$) and $MasterSeed2$ ($MS2$) is represented by $R1$ and $R2$ respectively. This concludes the first step of the mechanism. The second phase of the mechanism involves applying hash chain operations on $R1$ and $R2$. $R1$ is chained j times whereas $R2$ is chained $L - j$ times, thus, in total L hash operations are performed. Then, the results obtained from the hash chain operations in the second dimension are XOR'ed together to generate a key, shown in Figure 3.5.

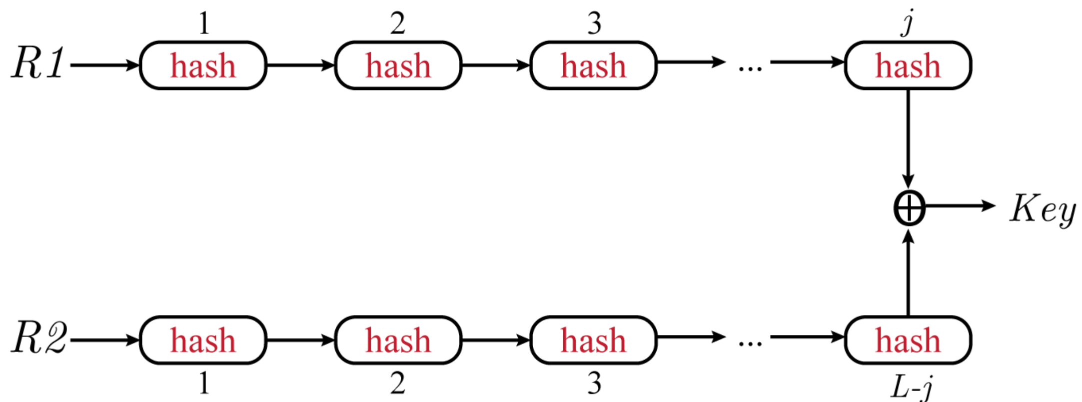


Figure 3.5: Second Phase of the Mechanism

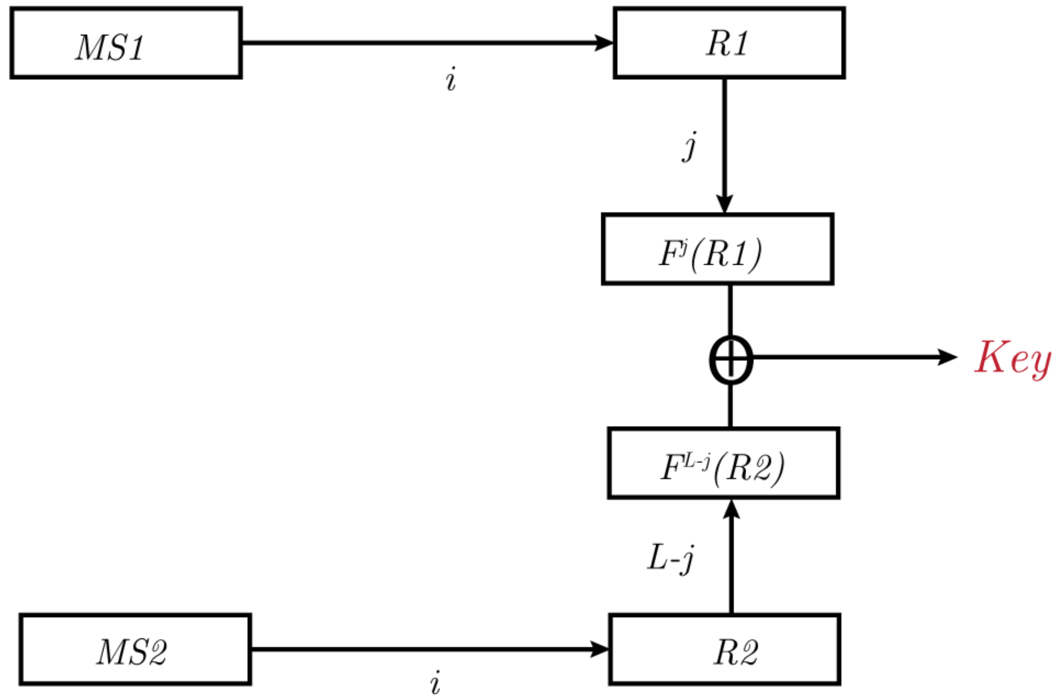


Figure 3.6: Two Way Hash Chain Mechanism

The complete picture of the mechanism is shown in Figure 3.6. Whenever a key is generated, the counter j is incremented. If j reaches the chain length value L, then it is set back to one and the counter i is incremented. This means, for each i, we produce L keys in total, before moving on to the next value of i.

3.3 Call Establishment Protocol

The hash chain mechanism provides a secure way to generate one-time keys. This mechanism can be set up on different ends with the same parameters to generate synchronized keys. In our case, we plan to provide keys to Media Security Platform which is a real world application developed by Netaş Telecommunications A.Ş. Since it is quite possible that the synchronization might be lost in a real life scenario, we devised a protocol to remedy that.

Let us define a limit to the number of tries when generating keys to match a given

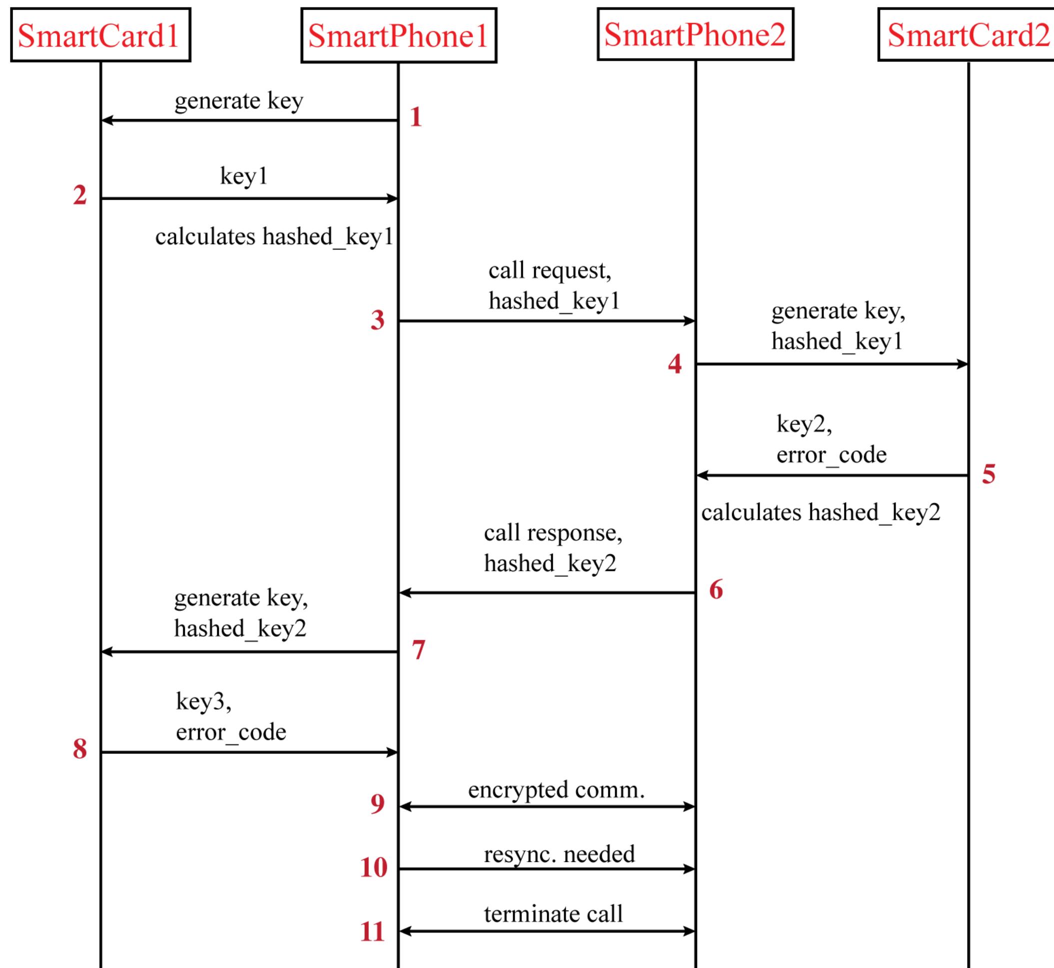


Figure 3.7: Signalling Protocol for Media Security Platform.

key. From here on, we represent this number with ϵ. We set ϵ at the initialization process where we load and install our mechanism to the smart cards.

Assuming that SmartPhone1 initiates the call, the protocol follows these steps:

1. SmartPhone1 requests a key from SmartCard1.

2. SmartCard1 produces the next key and returns it back to SmartPhone1, SmartPhone1 generates the hash of the received key, by performing $hashed_key1 = h(key1)$.

3. SmartPhone1 generates a call request using the WebRTC library. The call request

and the $hashed_key1$ is sent to SmartPhone2.

4. SmartPhone2 transfers the $hashed_key1$ to SmartCard2, requesting a key to be generated.
5. At this point, SmartCard2 keeps producing keys until ϵ number of keys have been produced or until it produces a key with the hash that matches with $hashed_key1$. If the same key on both ends are generated, SmartCard2 returns this key and produces an error code of 0. If the keys do not match after ϵ keys, SmartCard2 returns the first key it generated along with an error code of -1.
6. SmartPhone2 hashes the key returned from SmartCard2, by $hashed_key2 = h(key2)$ operation and transfers call response and $hashed_key2$ to SmartPhone1.
7. If the received $hashed_key2$ is a match with $hashed_key1$, SmartCard1 moves to step 9. If not, $hashed_key2$ is transferred to SmartCard1 and another key is requested.
8. SmartCard1 keeps producing keys until ϵ keys have been produced or until it produces a key with the hash that matches with $hashed_key2$. If a matching key is found, this key along with an error code of 0 is returned back to SmartPhone1. If not, SmartCard returns an error code of -2, then moves to step 10.
9. Both parties possesses the same key and the call begins.
10. The parties have not established a key, the call is dropped. Resynchronization is needed.
11. The call is terminated by one of the parties.

The error codes produced by the cards are defined in table 3.2

3.3.1 Possible Scenarios During Key Establishment

We will shortly refer to the client initiating the call as client A, and the client at the receiving end as client B. The most common case is both clients' cards stay synchronized,

Table 3.2: Error Codes of the Protocol

Error Code	Description
0	The key is generated successfully.
-1	The key generation failed. The first available key is returned.
-2	The key generation failed. Resynchronization required.

if the protocol and the mechanism produces keys without any mishap. In this situation, client A's smart card is ready to produce the next key; likewise the smart card of client B is going to produce the exact same key. When client A begins the call, $hashed_key1$ will be received by client B and subsequently sent to the smart card of B. At this point, B's card will produce a key identical to the one on the caller's side. Client B sends $hashed_key2$ across and upon receiving the hash value client A verifies that a common key is created. Client A will initiate necessary operations to begin the call.

One alternative scenario to the one mentioned above is the clients' cards might be out of synchronization within the specified limit ϵ. Assuming client A (caller) has produced more keys than client B, therefore is forward in the mechanism compared to client B (callee). When client A initiates the call, $hashed_key1$ sent to B will not match the first key generated by smart card of B. However, since the synchronization is not broken out of the boundary of ϵ, B's card will eventually find the matching key and return this key along with 00 error code. The rest of the call setup will progress in the same manner as the synchronized scenario.

A similar scenario is the reverse of this situation where client B (callee) is in forward position in the mechanism. When this happens, $hashed_key1$ received from client A cannot be matched by client B's smart card, because the mechanism only moves in the forward direction. Upon not being able to find a matching key after trying ϵ keys in the sequence, smart card of B returns the first key it tried. Then, this key is hashed ($hashed_key2$) and transferred back to client A. When $hashed_key2$ is received by client A's smart phone, it will discover that a common key has not been established. Next,

another key generation command is sent to A's smart card, this time requesting a key to be matched with $hashed_key2$. Since the synchronization is still intact within ϵ, the smart card of A will catch up to the key produced on client B's card. Consequently, the common key will be established and the call will begin afterwards.

The last possible scenario is the cards being more than ϵ away from each other. When this is the case, all the events mentioned in the paragraph above will take place. The only difference is that, A's smart card will not be able to find a matching key, therefore returning error code of -2. This error code is an indicator to the fact that the synchronization is broken beyond (ϵ) the usual flow of the protocol and needs to be established externally. In this condition, the call is aborted and the intermediary server is notified.

3.3.2 Distribution of Seed Values into Smart Cards

Although it is not in the scope of project and thesis, we offer methods for the generation and distribution of seed values. If the clients are in physically separated environments, then the seeds can be generated and shared in a Diffie-Hellman setting [5] using the application interface. A second suggestion for the remote client scenario is using an alternative secure channel to share the seed values This secure channel is only needed during the exchange of seeds. After the seeds are exchanged, they are stored in the secure area of the smart cards and never leave the card afterwards.

If the clients are in the same environment, then the seeds can be shared by the Bluetooth interface of the devices. Another possible method is relying on QR codes to display the seeds created by one party and then detect them by the other party. It might also be possible to transfer seed values between clients by simply inputting them as hexadecimal characters in the application.

3.4 Integration with Media Security Platform

In this section, we provide details on the integration of our key generation method with the ongoing Android application side of the Media Security Platform. The process of

integration consists of two steps. The first being the preparation work before introducing our code into the platform and the second step is the actual merging of the mechanism into the call establishment phase.

3.4.1 Pre-integration Preparation Work

The motivation behind the design of the key generation mechanism was providing keys to an application running on smart devices, especially those that run Android Operating System. Therefore, we have developed an application on the Android platform using a library that provides support for communication between the application and the smart card reader [2]. With this application, we are able to send commands to the smart card and receive the responses produced by the card, illustrated in Figures 3.8 and 3.9. We have performed our preliminary tests using the Android application, and discovered no discrepancy between these tests and the previous test run on a laptop computer. In Figues 3.10 and 3.11, we present tests on different chain lengths, which are the most relevant ones in the context of our mechanism.

We have designed an Application Programming Interface (API) for the interaction between Java Card and the Android application. Encapsulating the details of command execution in Java Card, the API offers simple functions to ease the integration process. In addition, when the smart card side of the project requires modification, the application does not need to be modified as well. This avoid the necessity of repeating the whole integration phase in the likely event of a minor change in our mechanism. We have defined three separate classes for the API which are overviewed in the following part of this section.

JavaCardApi class

In this class, as represented in Figure 3.12, we define all of the data members and method necessary to communicate with the Java Card. The card is connected to the smart device via a card reader attached to the one of the USB ports. The first step of reaching the card is to be able to control the USB interface through an UsbManager object defined. Then, the card reader device is stored in a UsbDevice object, after the manager connects

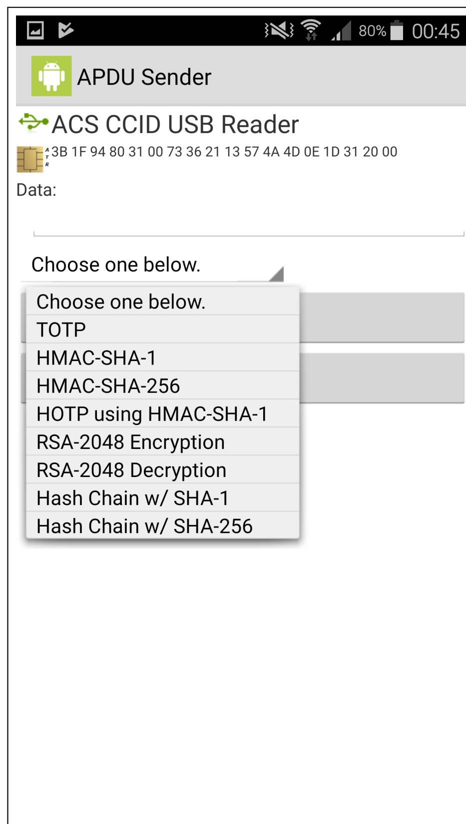


Figure 3.8: Test Application on Android

to the reader. Following that, communication with the actual reader interface requires a definition of a Reader variable provided by the external library. In addition, the generated key is stored in a byte array along with another byte array for the error code produced.

The constructor of this class needs two parameters, one UsbManager and one UsbDevice. When an object of this class is created, the parameters are provided by the application and set into the corresponding member variables. The $initialize$ method defines the $Reader$ object and starts the reader that is linked to the $UsbDevice$. The presence of a card in the reader is being queried with the $checkCard$ method, which returns $true$ when the card is inserted and $false$ otherwise. Actually connecting to the card and providing power is possible by $connectCard$ class method.

When the caller is in need of a key, $generateKey$ method is called. This method, firstly, selects the applet on the smart card that is responsible for key generation. If the applet is selected and the PIN verification process is successfully completed, a command

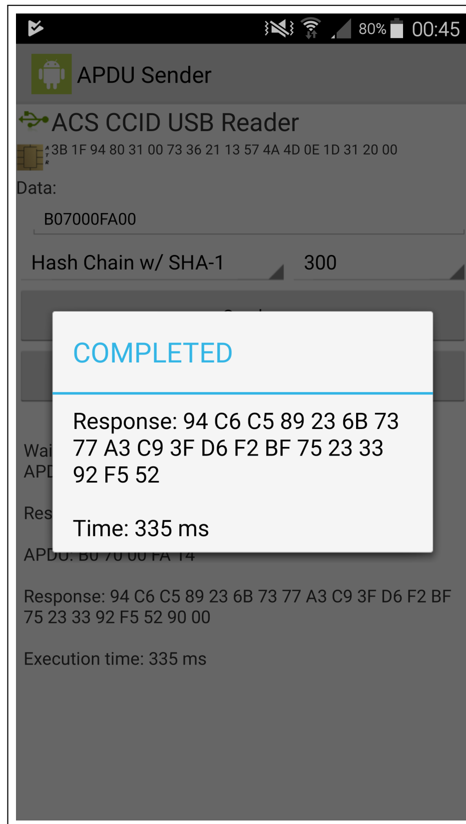


Figure 3.9: Result of a Test Run on Android

for key generation is sent to the card. The card is expected to return a key and an error code to this method, and this method itself returns an object ($JavaCardKey$) to its caller.

In our signaling scheme, the parties hash their keys before transmitting them across the channel. When the hashed key is received, the recipient produces a key of its own. For this, $generateKeyWithLimit$ method is called with the received *hashed_key* supplied as a parameter. The smart card then produces a series of keys until a match is found or the preset number of tries has been reached. If a matching key is found, the key is returned alongside with an error code of 0x00. If not, the first tried key is returned with an error code of either 0x01 or 0x02, depending on the side performing the operation. If the callee does not successfully create a matching key, then the error code is 1, in the caller's case the error code is output as 0x02. The result of this method is collected in a $JavaCardKey$ object and returned, as it happens in the $generateKey$ method.

Media Security Platform requires 96 bytes long keys in order to establish a call be-

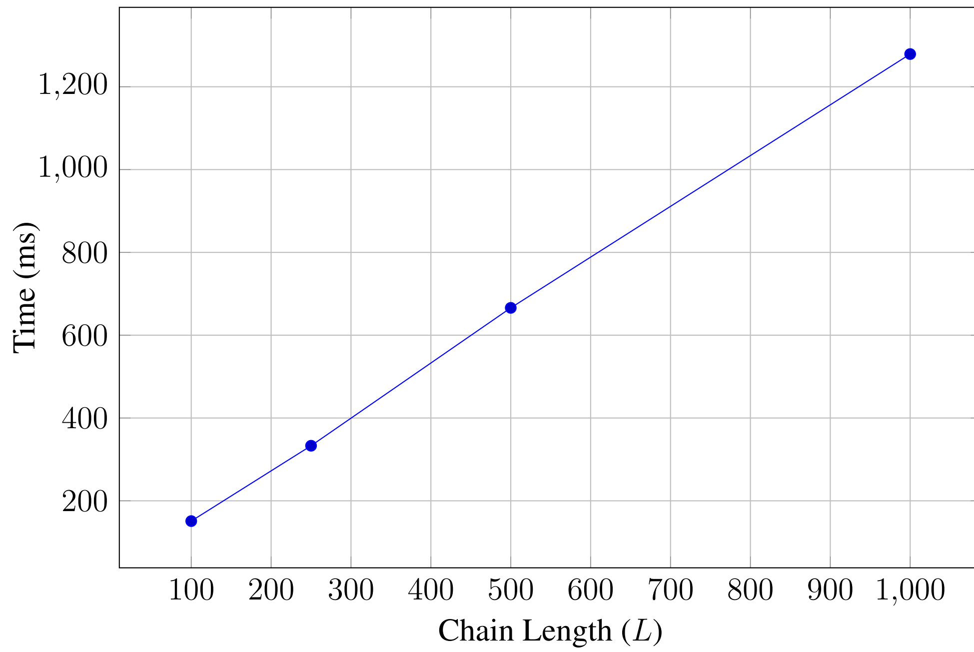


Figure 3.10: SHA1 Chain Length with Android

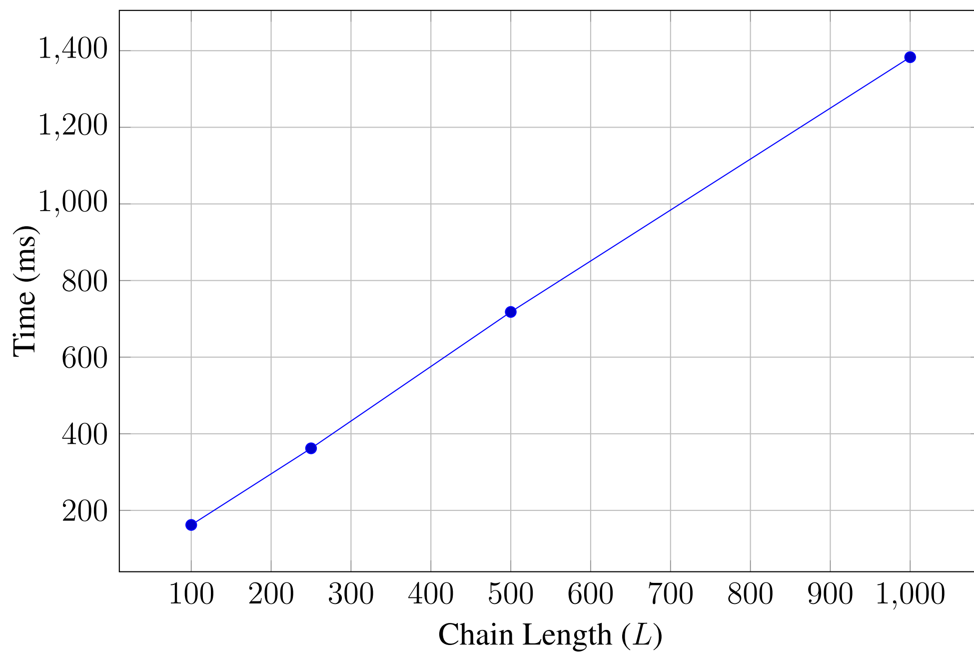


Figure 3.11: SHA256 Chain Length with Android

JavaCardApi

reader:Reader
usbManager:UsbManager
usbDevice:UsbDevice
boolean:connected

initialize()
checkCard()
connectCard()
generateKey()
generateKeyWithLimit()
createSrtpKey()

Figure 3.12: $JavaCardApi$ Class

tween its clients. We performed our tests by using hash chains with SHA-1 algorithm. The output of SHA-1 hash is only 20 bytes long, leaving us short of 76 bytes when compared to the actual key length in need. We lengthen our key by using a known data expansion method called P_hash [4]. The final method in this class, $createSrtpKey$ gets 20 a bytes key and returns a 96 bytes long key created using P_SHA1 method.

JavaCardKey

boolean:success
String:code
String:message
String:key
String:keyHash

isSuccess()
setSuccess()
getCode()
setCode()
getKey()
setKey()
getKeyHash()
setKeyHash()

Figure 3.13: $JavaCardKey$ Class

JavaCardKey class

This class is a simple container for a key and related information about the key as seen in Figure 3.13. The first member variable, $success$ is set to $false$ if an error occurred during the generation of the key. This variable is an indicates that whether the key is available or not. The second variable holds the error code returned from the card. The $error$ variable may take following values: $0x00$, $0x01$ or $0x02$. The key variable stores the key generated and returned by the smart card. The last member variable, $hashedKey$ is used to store the hashed version of the key.

Figure 3.14: $JavaCardApiIml$ Class

JavaCardApiIml class

This class encapsulates the previously defined $JavaCardKey$ and $JavaCardApi$ classes, and provides singular methods. The purpose of the design of this class is to further ease the integration by gathering multi-line commands under one operation. A UML representation of this class is presented in Figure 3.14.

3.4.2 The Integration

As the first step, we merged our Java Card interface with the Media Security Platform's code base. From the Android application, appropriate API functions are called in the signaling phase. In Figure 3.15, which functions are supposed to be called at what point are shown.

The first task of the integration was to create a listener for USB permission in the Android application of the platform. This listener is required, because without an external

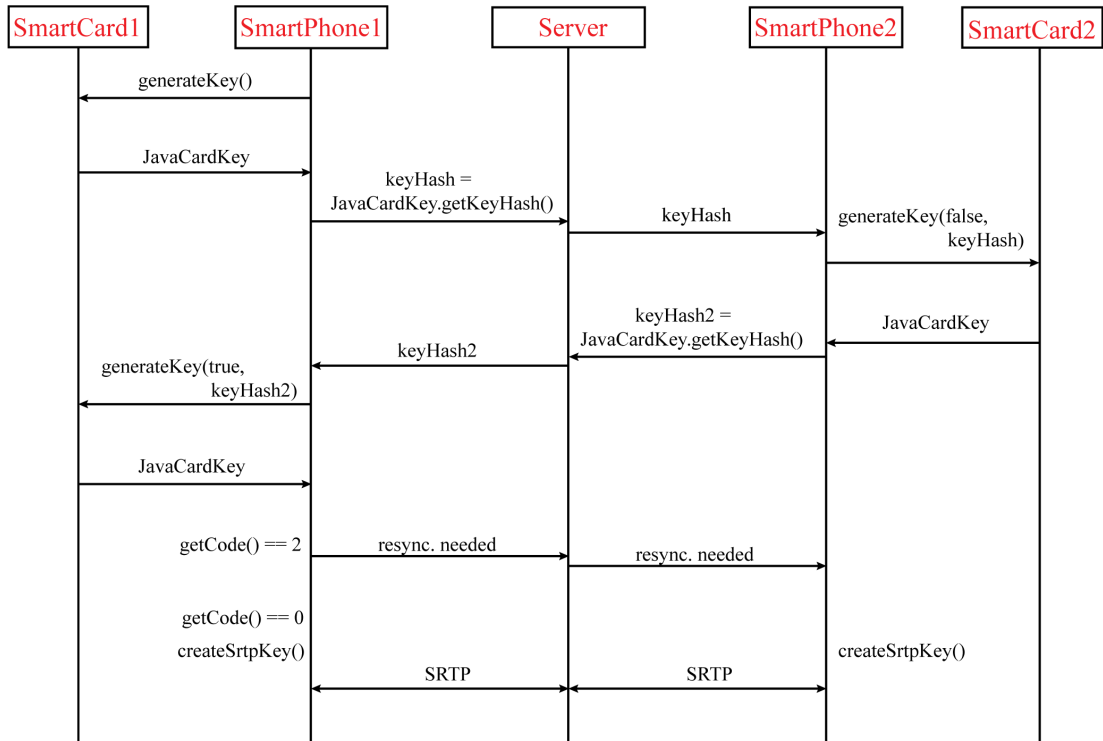


Figure 3.15: API Calls After Integration with the Media Security Platform

USB device permission, the card reader does not work at all. When the card reader is plugged into the USB port of the smart device, the listener catches this event and creates a prompt displaying a permission request to use the USB device. After the permission is granted, the card reader is powered up, and subsequently the application connects to the smart card.

The following events take place when a client (the caller) decides to call another client (the callee). The application on the smart device of the caller, inside its $RtcManager$ class, prompts the first key to be generated. The key returned from the smart card is stored inside a $srtpKey$ variable, and at the same time a hashed version of the key signed and placed inside a call request. The call request is sent to the intermediary server which then transfers it to the application of the callee. On the callee's side, the call request is captured in a method named $onCallOfferCatched$ inside $WebSocketManagerHelper$ class. The hashed version of the key contained in the call request is extracted and provided

to the callee's smart card. As expected the smart card generates a key and an error code. If the callee chooses to answer the call, the key returned from the smart card is hashed and sent back to the caller in the positive response. Otherwise, the the call is cancelled and the caller's notified of the termination.

When the callee accepts the call, the caller receives a hashed key produced and sent by the callee. At this point, the caller has two hashed keys, one of which produced by the other party. The caller compares these hashes, if the hashes are found out to be the same the call is started with the established key. If the hashes turn out to be different, then the caller sends another key generation command to its smart card with the received hashed key. This second key generation is placed inside the method $onCallAnswerNotify$ inside $CallFragment$ class. The smart card tries to generate a matching key, returning 0x00 if a matching key is found and 0x02 if the matching key cannot be found. The call starts after the application verifies the error code to be 0x00. Receiving an error code of 0x02 indicates that the call cannot be started, furthermore the synchronization between the clients are broken exceeding the limits defined by the mechanism.

Here, we present a flow of a call establishment process in the Android application of MSP. In Figure 3.16, the caller is displaying the key generated in the smart card.

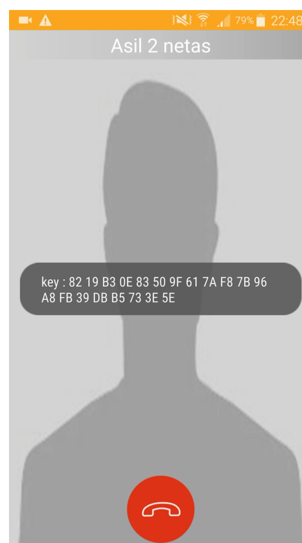


Figure 3.16: The Caller Initiating in Android Application

The call request has been sent to the receiver of the call, as indicated in Figure 3.17.

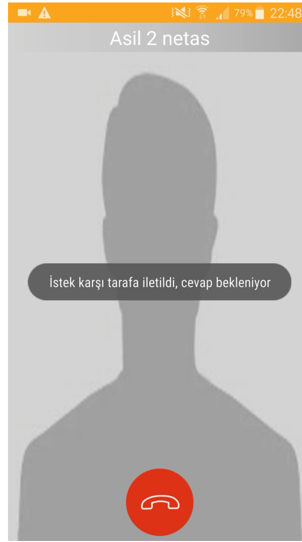


Figure 3.17: Caller Sends the Call Request

Then, the callee's application prompts its smart card to generate a key and displays that key. The callee can either accept or reject the call at this time, as shown in Figure 3.18.

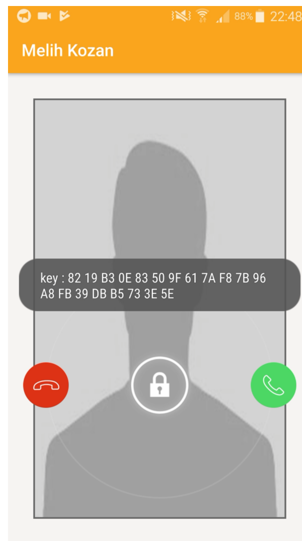


Figure 3.18: The Callee's Screen After Receiving the Call Request

The callee accepts the call, shown in Figure 3.19

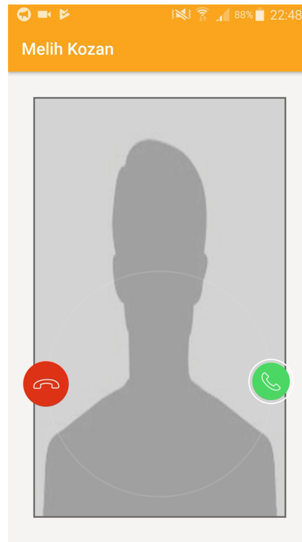


Figure 3.19: The Call is Accepted

The call establishment is completed and the secure communication begins. Figures 3.20 and 3.21 display the screens of both caller and callee, respectively.

Figure 3.20: The Call has Started on Caller's Application

Figure 3.21: The Call has Started on Callee's Application

Chapter 4

Performance Evaluation

4.1 Key Generation Timings

While the production of the keys, our mechanism performs certain number of hash operations. As we produce more and more keys, the time it takes to produce future keys increases. Though, this increase can be controlled by adjusting the chain length of our mechanism. However, the adjustment cannot be done dynamically, in other words, the chain length of our mechanism does not change after initialization. Therefore, we need to determine the chain length at the beginning while considering the key generation performance it entails. Measuring key generation times for different chain length setups, we have produced a graph shown in Figure 4.1. We have tested our mechanism by setting its chain length to 100, 200, 300, 400 and 500.

Our mechanism works in two phases. In the first phase, there are two separate hash chains, each of length i. Thus, the number of hash chain operations in the first part sums up to $2 \times i$. In the second phase, there are again two hash chains, one of length j and one of length $L-j$. The number of hash operations done in the second phase always add up to L no matter the value of the counter j. Then, we can conclude that for a key, the required number of hash operations depends on the value of the counter i and the value of L at the time of production. As a matter of fact, the number of hash operations is equal to $2 \times i + L$. As we produce keys, the value of the counter i will naturally increase and this will result

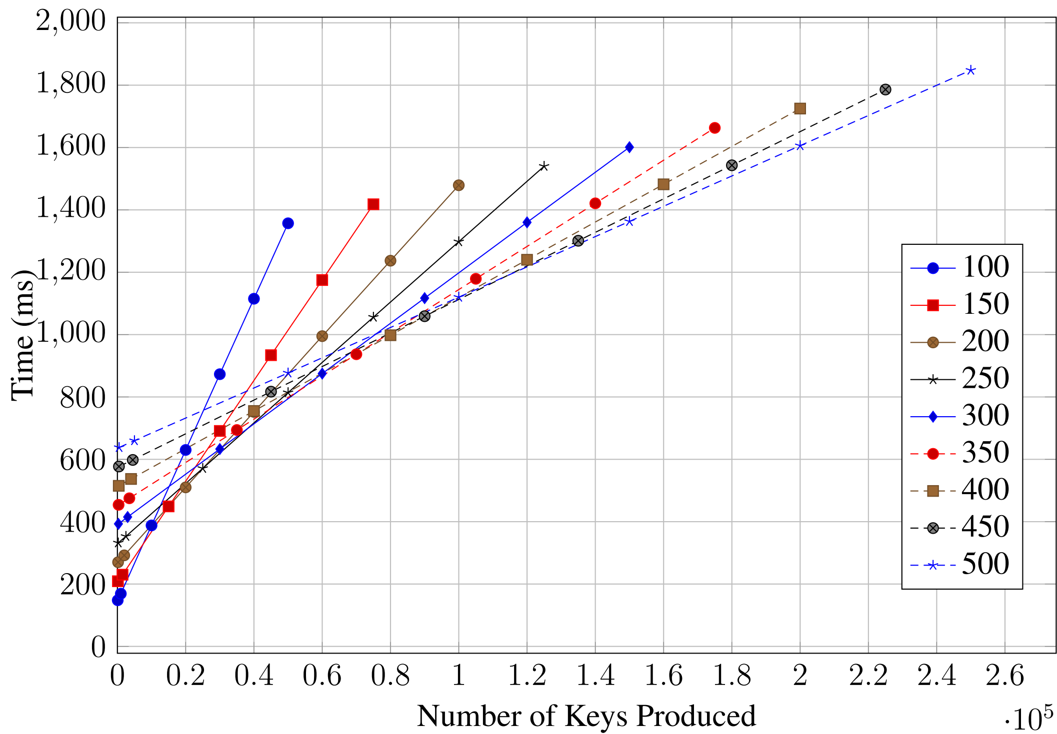


Figure 4.1: Tests with Different Chain Lengths (ms)

in longer times to produce new keys. If we set a relatively small chain length L for our mechanism, then we ensure that the initial keys will be produced faster, because L will be relatively low. Similarly, in a longer chain length setting, the initial keys will require more hash operations, and more time because of the large L. This situation is reversed when the number of keys generated increase. For later keys, the small chain length setups do require more time than the longer chain length ones. This is due to the fact that for each value of counter i, the number of keys produced is exactly L. Therefore, the i counter increases rapidly when the value of L is relatively smaller, which then causes late keys to be generated at a slower rate.

4.2 Calculating the Optimal Chain Length for a Target Key

As mentioned earlier, the completion time of a hash chain operation is directly proportional to the number of hash operations it entails. For a key, our mechanism performs four different hash chains along with an XOR operation. The XOR operation takes negligible amount of time, when compared with the rest of the key generation process. This allows us to make an estimate about the performance of our mechanism by calculating the total number of hash operations. As we have pointed out in Section 4.1, each key requires $2 \times i + L$ hash operations. Now, we want to adjust the chain length L for the kth key, so that it is generated in the shortest time possible. The kth key is produced after $k - 1$ keys have already been produced, so the values of the counters i and j for the kth key are shown below.

$$i = \left\lceil \frac{k}{L} \right\rceil \tag{4.1}$$

$$j = k \mod L \tag{4.2}$$

We have shown that the total number of hash operations n is,

$$n = 2i + L \tag{4.3}$$

If we replace i with $\frac{k}{L}$, then we get the total number of hash operations in terms of k and L.

$$n = \frac{2k}{L} + L = \frac{L^2 + 2k}{L} \tag{4.4}$$

Since we want to optimize the number of hash operations, we take the first derivative of the Equation 4.4 and then set it to zero. This gives us an optimal chain length L for the kth key in the production as shown in Equations 4.5, 4.6 and 4.7.

$$\frac{dn}{dL} = 1 - \frac{2k}{L^2} \tag{4.5}$$

$$1 - \frac{2k}{L_{opt}^2} = 0 \tag{4.6}$$

$$L_{opt} = \sqrt{2k} \tag{4.7}$$

We have experimented with different chain lengths and with different keys and recorded the production times. For an iteration of our experiment, we have set the chain length value to an integer in the range $[50, 700]$. Then, we produced the 10000th, 25000th, 50000th and 75000th key for each chain length. As expected, each key is associated with a chain length that minimizes its production time, as can be seen in Figure 4.2.

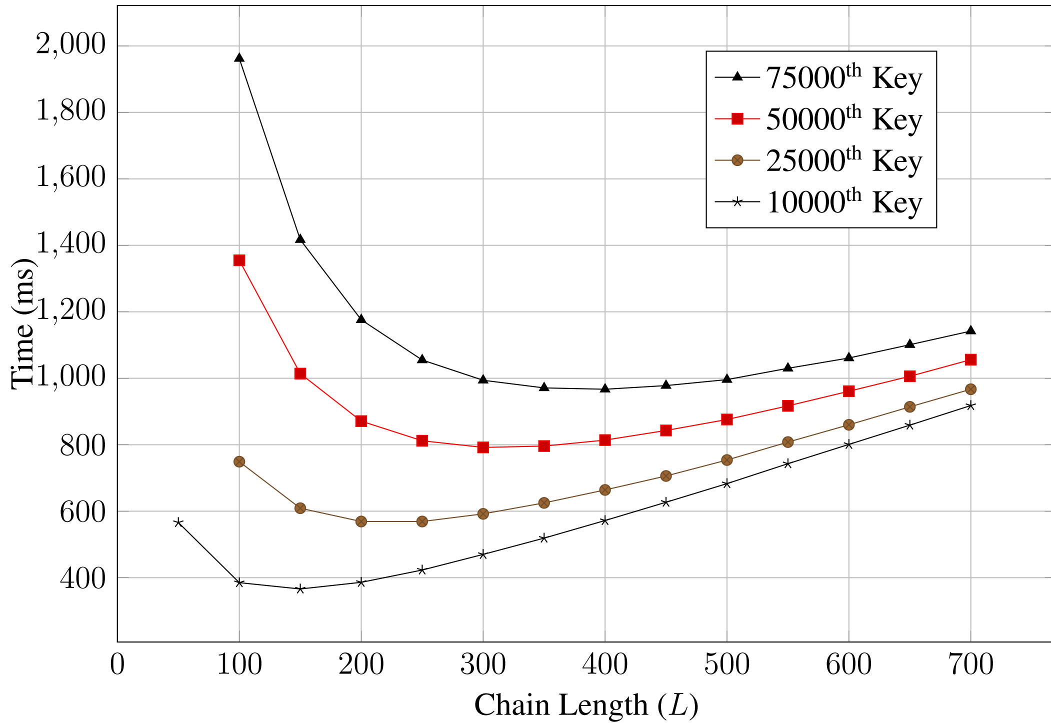


Figure 4.2: Generation Time of Different Keys (ms)

Analytically, the optimal chain length is calculated as $\sqrt{2k}$ (Equation 4.7). We present a comparison between analytical values and the results from our experiments with varying

chain lengths. As shown in Table 4.1, analytically expected optimal chain lengths and the actual optimal chain lengths do have significantly close values.

Table 4.1: The Optimal Chain Lengths for Selected Keys

Production Number of Key	Expected Optimal Chain Length (Analytical)	Actual Optimal Chain Length (Experimental)
10000	141	137
15000	173	177
20000	200	200
25000	224	228
50000	316	311
75000	387	389

4.3 Post-integration Timings

After we successfully integrated our mechanism to the Media Security Platform, we found the opportunity to test the integrated application inside the company's network. The results of these tests with using SHA-1 and SHA-256 algorithms are shown in Tables 4.2 and 4.3, respectively. Each test measures the time it takes for both clients to be ready to establish call. In the testing process, the flow of events happen as introduced in our signaling protocol presented in Figure 3.7. The caller party creates a key, introducing the first delay. Then, this key is sent inside a call request to the recipient party which introduces a network delay while in transfer and processing delays by the applications. Finally, the recipient party generates a key which increases the overall time to establish end-to-end encryption. Therefore, the timing measurements are higher when compared to timings of a single key generation operation, due to delays introduced by the the network and the applications.

The results reveal a performance gap between the hash algorithms used by the mechanism. SHA-1 performs better than SHA-256 as expected. Though, our tests with a specific

Table 4.2: Tests with MSP Application Using SHA-1 in Netaş's intranetwork

Brand of Card	Chain Length (L)	Time (ms)
Feitian A22	100	1580
	200	1890
Feitian A40	100	1630
	200	1780
G&D	100	1570
	200	1790

Table 4.3: Tests with MSP Application Using SHA-256 in Netaş's intranetwork

Brand of Card	Chain Length (L)	Time (ms)
Feitian A22	100	2490
	200	3400
Feitian A40	100	2200
	200	3860
G&D	100	1590
	200	1880

Table 4.4: Tests with MSP Application Using SHA-256 outside Netaş network

Brand of Card	Chain Length (L)	Time (ms)
Feitian A22	100	2540
	200	3820
Feitian A40	100	2640
	200	4010
G&D	100	1850
	200	2150

brand of Java Card, namely G&D, indicate the possibility of using SHA-256 without too much performance degradation. We have also tested the MSP application with SHA-256

on a network outside of Netaş's, and discovered results close to ones found in tests done inside Netaş network. The results, displayed in Table 4.4, show that SHA-256 is feasible when run on an efficient card, even in outside networks where the delay is higher.

4.4 Memory Usage on Java Card

It is not possible to detect memory usage of an applet on Java Card. However, there is an alternative method to estimate the memory usage. The Java Card displays the amount of free memory when probed. We developed the following approach in this context. Firstly, we inquire the free memory from the card, then we load our applet and perform another query to obtain the free memory on the card. By calculating the difference between our inquiries, we are able to find out how much memory our applet consumes on the card. We present the memory consumptions of the tested applets in Table 4.5. Our hash chain mechanism applet consumes around 1000 bytes of memory, which is minimum among other mechanisms.

Table 4.5: Memory Usage of Applets on Java Card

Name of the Applet	Memory Usage (Bytes)
HMAC	1220
HOTP	1048
RSA 2048	2570
Hash Chain Mechanism	1006

Chapter 5

Conclusion

WebRTC is a developing technology that allows development of multimedia communication applications with ease. Still an ongoing project to offer neat interfaces for the future of software, WebRTC does not provide a standard method for key management. In this thesis, we delve on the issue of key generation and key distribution for the Media Security Platform being built by Netaş Telecommunications A.Ş. We founded a method to generate synchronous keys on remote ends with the help of embedded security of smart cards, our scheme depends on fast hash operations. Our key generation mechanism performs better than previously used approach of generating keys relying on a public key setting. We have implemented and tested our solution on different brand of smart cards and then integrated our mechanism to the Android application offered by Media Security Platform. After the integration, we have achieved to establish audio and/or video calls with the keys generated by our key generation algorithm.

The clients in the Media Security Platform communicate with other clients after they mutually agree on a friendship relation. Our solution for key management currently does not address this situation. We provide the actual seeds of friends manually into each of the smart cards while we initialize the applet that runs the mechanism. For future work, a secure method to establish friendships for the clients can be proposed and implemented. For this, the seeds of the clients should be generated privately and transferred into the cards. The same approach can also be used when previously bonded clients lose synchro-

nization beyond the limits set before. When two clients need to be resynchronized, they should be provided with brand new set of seeds on their cards.

Bibliography

[1] *A Study of WebRTC Security*. URL: http://webrtc-security.github.io/ (visited on 07/08/2017).

[2] *ACS Android Library*. URL: https://android.acs.com.hk (visited on 07/11/2017).

[3] Adavalli, S. R. *Smart Card Solution: Highly secured Java Card Technology*. URL: https://www.cs.auckland.ac.nz/courses/compsci725s2c/archive/termpapers/725adavalli.pdf (visited on 07/06/2017).

[4] Dierks, T. and Rescorla, E. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246. 2008. URL: https://tools.ietf.org/html/rfc5246.

[5] Diffie, W. and Hellman, M. E. "New Directions in Cryptography". In: *IEEE Transactions on Information Theory* 22.6 (1976).

[6] Eastlake 3rd, D. and Hansen, T. *US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)*. RFC 6234. 2011. URL: https://tools.ietf.org/html/rfc6234.

[7] Eastlake 3rd, D. and Jones, P. *US secure hash algorithm 1 (SHA1)*. RFC 3174. 2001. URL: http://www.rfc-editor.org/info/rfc3174.

[8] Handley, M., Jacobson, V., and Perkins, C. *SDP: Session Description Protocol*. RFC 4566. 2006. URL: https://tools.ietf.org/html/rfc4566.

[9] Hassler, V., Muller, C., Gordeev, M., and Manninger, M. *Java Card for E-Payment Applications*. Norwood, MA, USA: Artech House, Inc., 2001. ISBN: 1580532918.

[10] Kommerling, O. and Kuhn, M. G. “Design Principles for Tamper-resistant Smartcard Processors”. In: *Proceedings of the USENIX Workshop on Smartcard Technology on USENIX Workshop on Smartcard Technology*. WOST’99. Chicago, Illinois: USENIX Association, 1999, pp. 2–2.

[11] Krawczyk, H., Bellare, M., and Canetti, R. *HMAC: Keyed-Hashing for Message Authentication*. RFC 2104. 1997. URL: https://tools.ietf.org/html/rfc2104.

[12] Lamport, L. “Password authentication with insecure communication”. In: *Commun. ACM* 24.11 (1981), pp. 770–772. DOI: 10.1145/358790.358797.

[13] M’Raihi, D., Bellare, M., Hoornaert, F., Naccache, D., and Ranen, O. *HOTP: An HMAC-Based One-Time Password Algorithm*. RFC 4226. 2005. URL: https://www.ietf.org/rfc/rfc4226.txt.

[14] M’Raihi, D., Machani, S., Pei, M., and Rydell, J. *TOTP: Time-Based One-Time Password Algorithm*. RFC 6238. 2011. URL: https://tools.ietf.org/html/rfc6238.

[15] Page, T. *The Application of Hash Chains and Hash Structures to Cryptography*. Tech. rep. 2009. URL: https://www.ma.rhul.ac.uk/static/techrep/2009/RHUL-MA-2009-18.pdf (visited on 07/15/2017).

[16] Perrig, A. *Cryptographic Properties*. URL: https://users.ece.cmu.edu/~adrian/projects/sec/node6.html (visited on 07/16/2017).

[17] Rogaway, P. and Shrimpton, T. “Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance”. In: *Fast Software Encryption: 11th International Workshop, FSE 2004, Delhi, India, February 5-7, 2004. Revised Papers*. Ed. by Roy, B. and Meier, W. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 371–388. ISBN: 978-3-540-25937-4. DOI: 10.1007/978-3-540-25937-4_24.

[18] Rosenberg, J. *Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols*. RFC 5245. 2010. URL: https://tools.ietf.org/html/rfc5245.

[19] Srisuresh, P. and Egevang, K. *Traditional IP Network Address Translator (Traditional NAT)*. RFC 3022. 2001. URL: https://tools.ietf.org/html/rfc3022.

[20] Srivatsa, S. *Android Security Issues*. URL: https://www.cse.wustl.edu/~jain/cse571-14/ftp/android_security/index.html#sec3.3 (visited on 07/12/2017).

[21] Surendran, D. *Elements of Smart Card Architecture*. URL: http://people.cs.uchicago.edu/~dinoj/smartcard/arch-1.html (visited on 07/14/2017).

[22] *Web Real-Time Communication (WebRTC): Media Transport and Use of RTP*. URL: https://tools.ietf.org/html/draft-ietf-rtcweb-rtp-usage-25 (visited on 07/08/2017).

[23] *WebRTC*. URL: http://www.webrtc.org/ (visited on 07/07/2017).