Usage and performance of cryptographically generated addresses

Aymen Boudguiga, Tony Cheneau, Maryline Laurent

Telecom & Management sudParis

Usage and Performance of Cryptographically Generated Addresses

Authors:
Aymen BOUDGUIGA Tony CHENEAU Maryline LAURENT-MAKNAVICIUS

Department:
Software and Networks

Reference:
08015-LOR

September 22, 2008

Abstract

Neighbor Discovery (ND) controls IPv6 nodes and routers interactions through ICMPv6 messages. It provides address resolution and duplicated address detection. Nonetheless, it does not offer security protection to the exchanged messages. In order to improve its security, Secure Neighbor Discovery (SEND) has been specified using the same ND messages and adding new options providing authentication and replay attacks control. SEND uses Cryptographically Generated Addresses (CGAs) as IPv6 addresses and RSA as key and signature generation algorithm.

CGAs are created using RSA public key. In fact, they contain a cryptographic based identifier. Their interface identifier is generated using SHA-1 hash function over a parameters structure which contains the generating node's public key. Our project consists on the study of CGAs different usages in wired, multihoming and mobile networks. In addition, we studied CGA generation and verification algorithms performance and improve it with the use of Elliptic Curve Cryptography (ECC). We generated and conducted a set of tests to assess CGA generation performance in different platforms. Moreover, through a set of experiments, we evaluated the potential brought by ECC in cryptographic address generation.

Maryline LAURENT-MAKNAVICIUS
Professor
TELECOM & Management SudParis Department LOR
9 rue Charles Fourier 91011 Evry cedex
E-mail: Maryline.Maknavicius@int-edu.eu

Tony CHENEAU
PhD student
TELECOM & Management SudParis Department LOR
9 rue Charles Fourier 91011 Evry cedex
E-mail: tony.cheneau@int-edu.eu

Aymen BOUDGUIGA
Internship student
TELECOM & Management SudParis Department LOR
9 rue Charles Fourier 91011 Evry cedex
E-mail: aymen.boudguiga@int-edu.eu

Contents

List of Figures

List of Tables

List of Acronyms

ARP	Address Resolution Protocol
CPS	Certification Path Solicitation
CPA	Certification Path Advertisement
CGA	Cryptographically Generated Address
DAD	Duplicated Address Detection
DoS	Denial of Service
DDoS	Distributed Denial of Service
ECC	Elliptic Curve Cryptography
ECDSA	Elliptic Curve Digital Signature Algorithm
HBA	Hash Based Address
ICMPv6	Internet Control Message Protocol for the Internet Protocol Version 6
IPv6	Internet Protocol Version 6
ISP	Internet Service Provider
MIPv6	Mobile IPv6
NA	Neighbor Advertisement
ND	Neighbor Discovery
NS	Neighbor Solicitation
RA	Router Advertisement
RS	Router Solicitation
RST	Rivest Shamir Tauman ring signature
SEND	Secure Neighbor Discovery
SHIM6	Site Multihoming by IPv6 Intermediation

Introduction

Neighbor discovery(ND) controls IPv6 hosts and routers through ICMPv6 messages. Nodes use it to determine the link layer addresses of neighbors belonging to the same local link and to purge cached values that become invalid. In addition, hosts use it to find neighboring routers that are going to forward their packets and that are offering valid and routable prefixes. This protocol is used when detecting unreachable nodes. In fact, it mainly serves to keep track of neighbors which are reachable and to detect nodes having changed their link layer addresses.

ND is the protocol that controls all IPv6 network nodes exchanges. One problem with this protocol is its security. It has been discovered lately that ND messages are vulnerable to Denial of Services (DoS) and replay attacks [1]. Therefore, to secure them, some specifications have added new ICMPv6 messages and options concerning message authentication to transform the classical ND to a new protocol called Secure Neighbor Discovery (SEND).

SEND [2] aims to make ND useful in all situations and in different kinds of networks. Its goal is offering messages and identities authentication for every node. So in order to make this security concept feasable even in mobile networks which present generally a lack of ressources and power, SEND introduced a new address generation mechanism. It consists in creating a cryptographically generated address (CGA) [3]. The word 'cryptographically'stands for the fact that this kind of IPv6 address is generated using the node public key and a hash function to construct the address interface identifier. Hence, the address identifier binds the node's public key to the address.

CGA offers a strong concept which avoids the use of centralised certification authority in order to bind a public key to its owner. In fact, when used with a signature option, CGA becomes equivalent to a degital certificate: the address contains the public key that will be used to verify the signature sent with it. So the signature verifier has to get the public key from the address and then uses it to verify the signature. This method decentralizes the notion of certification authority and becomes more efficient when used in mobile networks where it is very difficult to get access to a centralized unit due to the mobility.

CGA generation algorithm uses a RSA public key and a parameters structure in order to generate the interface identifier using SHA-1 hashing function. This project consisted

in the study of the CGA from two angles: (a) first, we studied its possible uses in different kinds of situations and networks; (b) second, after implementing the CGA generation and verification algorithms, we studied its performance in term of time; (c) finally we proposed an improvement to the CGA generation algorithm and SEND protocol by introducing the use of elliptic curve cryptography (ECC) instead of RSA.

This report starts with an introductive chapter presenting ND messages, cases of use and threats and how SEND makes it more secure.

The second chapter discusses the CGA generation algorithm and its verification process with some discussions concerning the proposal of using a DHCP server in the address generation process.

The third chapter presents CGA use cases in multihoming and mobile networks.

The fourth chapter introduces a study to SEND threats and CGA security consideration.

The fifth and last chapter addresses the CGA generation algorithm implementation and its performance studies. It ends with a comparison between CGA classical generation algorithm and the new solution that we proposed based on ECC.

Chapter 1

Securing Neighbor Discovery

Introduction

Neighbor Discovery (ND) is a protocol developed for IPv6 [4], and defined in [5]. It enables hosts over a link to determine neighboring routers, to make address resolution, to detect unreachable neighbors and duplicated addresses. It is based on the use of ICMPv6 [6] messages and options to define the interaction between the different hosts.

With the evolution and the spread of IPv6, it has been discovered that ND has many security problems and that it was vulnerable to attacks such as Denial of Service attack (DoS) and replay attack. So, to prevent from those attacks, some specifications have been added to this protocol and a new version has appeared and was named Secure Neighbor Discovery (SEND), which is defined in [2]. SEND is based on Cryptographically Generated Addresses [3] which are IPv6 addresses[1] including public key information of the owner, in the interface identifier bits.

In this introductive chapter, we first review the different types of messages and options defined in ND. Then we present usages of this protocol, and possible attacks targeting ND messages. In the second part, we describe SEND messages that were introduced to secure ND and remaining possible attacks.

1.1 Neighbor discovery (ND)

ND defines five ICMPv6 messages to make the dialog between the different nodes belonging to the same local link easier and to replace some other IPv4 [7] protocols functionalities such as ARP and RARP [8]. In the following, we describe these messages is done along with some of their usages.

[1] IPv6 addresses are presented in appendix A

1.1.1 ND messages

The differents ND ICMPv6 messages are: router solicitation, router advertisement, neighbor solicitation, neighbor advertisement and redirect. They are described in this section.

1.1.1.1 Router solicitation (RS)

RS is sent by a node to ask a router to send a Router Advertisement (RA) immediately. The node needs a RA when booting or to get information related to the router. It is sent to the router multicast address ff02::2.

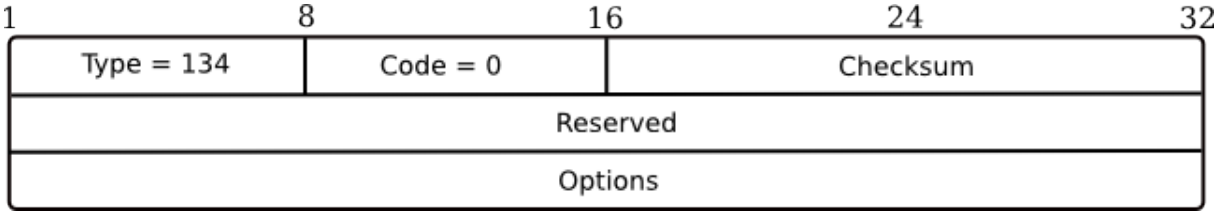

Figure 1.1: Router solicitation message.

This message given in Figure 1.1 is composed of the following fields:

- *Checksum* contains the checksum of the message.
- *Reserved* is reserved for future usage and all its bits are set to zero and must be ignored by the receiver.
- *Options* contain the ICMPv6 options that could be added to this message.

In all the ICMPv6 messages defined hereinafter, the fields checksum, reserved and options have the same aforementioned goal.

1.1.1.2 Router advertissement (RA)

RA is sent periodically or in response to RS. It has the following structure (Figure 1.2):

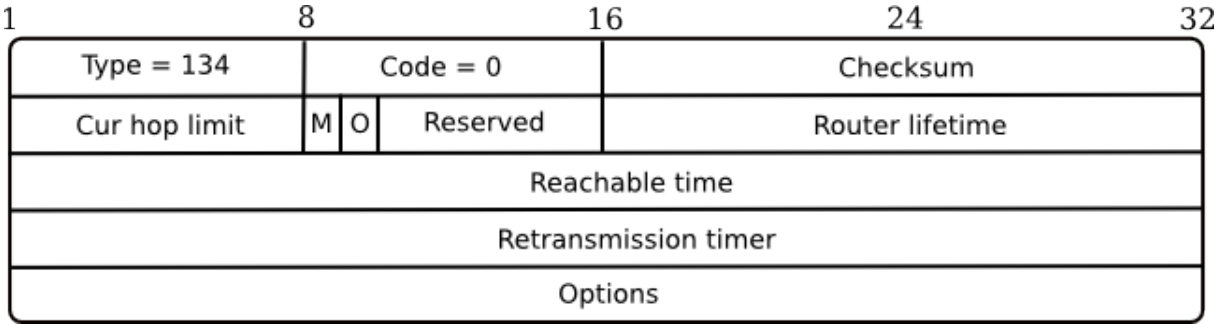

Figure 1.2: Router advertissement message.

This message given in Figure 1.2 is composed of the following fields:

- *Cur hop limit* contains the value of the field hop number of the IPv6 header.
- *M* indicates the way the address must be conFigured. When set to one, the host must use stateful address configuration.
- *O* indicates that there are other stateful information to use while configuring the address.
- *Router lifetime* gives the lifetime of the default router.
- *Reachability time* is the period of validity of information existing in the a nodes cache: it is the time when a node assumes that a neighbor is still reachable after receiving a reachability confirmation.
- *Retransmission timer* represents the period between two RS transmissions.

1.1.1.3 Neighbor solicitation (NS)

NS is a message used when attempting to collect information about neighbors in the same link. It can be multicasted or unicasted. Its structure is presented in Figure 1.3 and includes:

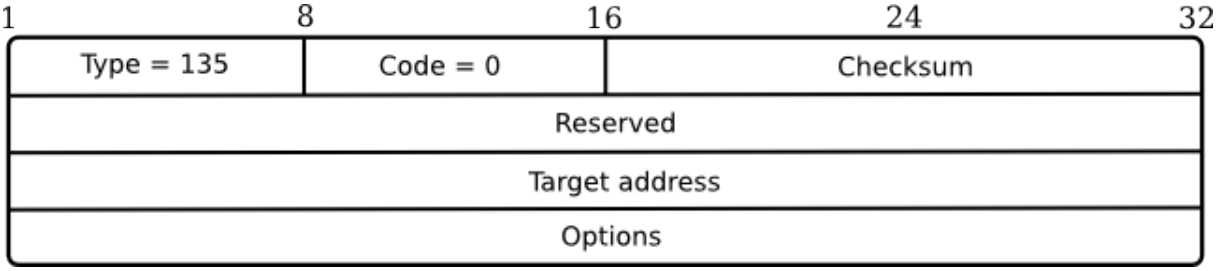

Figure 1.3: Neighbor solicitation message.

- *Target address* contains the address of the target of the solicitation.

1.1.1.4 Neighbor advertisement (NA)

It is a message used to respond to a NS message or to propagate new information about the interface. It is presented in Figure 1.4.

- *R* represents the router flag. If it is equal to one, the sender is a router.
- *S* is the solicitation flag. A flag set to one means that the advertisement is sent in response to a NS.
- *O* is the override flag. The advertisement overrides an existing cache entry and uploads the cached link layer address when this bit is set to one.

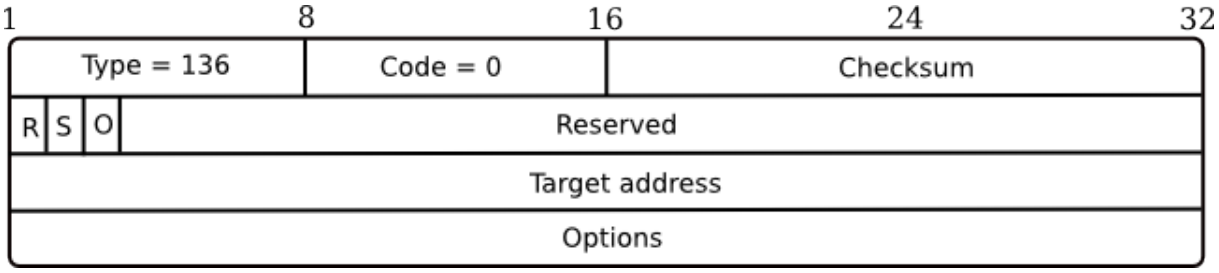

Figure 1.4: Neighbor advertissement message.

- *Target address* is equal to the target address of the NS message if the bit S is set to one, else it will be equal to the local-link address of the sender.

1.1.1.5 Redirect

Router sends redirect message (Figure 1.5) to inform a host of a better first hop node.

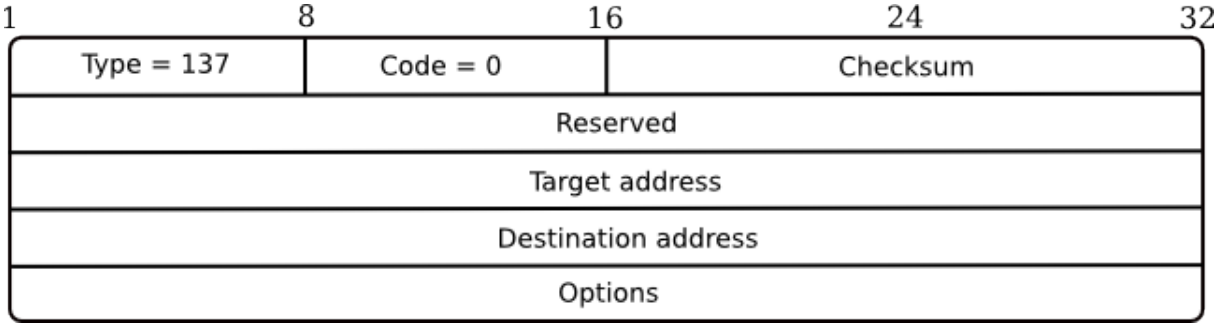

Figure 1.5: Redirect message.

- Target address is the link local address of the best first hop.
- Destination address is the address of the destination to which the packet is redirected.

1.1.2 ND options

Options might complement previous messages. They all have the same TLV format (TLV for Type-Length-Value) and are defined in [5].

1.1.2.1 Source/Target link layer address

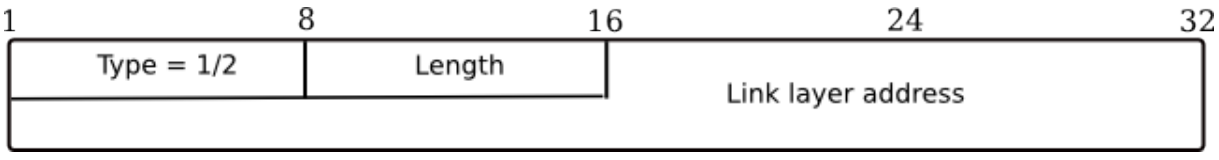

Figure 1.6: Source/Target link-layer option.

This option given in Figure 1.6 is composed of the following fields:

- *Type* field is equal to one when the link-layer address used is the source one.
- *Type* field is equal to two when the link-layer address used is the destination one.
- *Length* field represents the length of the option in words of 64 bits.

1.1.2.2 Information about the prefix

This option contains the information needed during the configuration of the equipment.

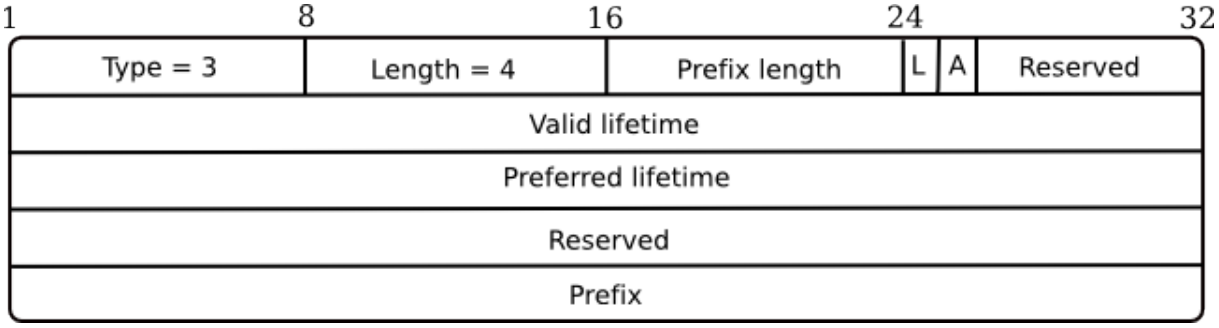

Figure 1.7: Prefix option.

This option given in Figure 1.7 is composed of the following fields:

- *Prefix length* field contains the number of leading bits that are valid.
- *L* is the on-link flag. When set to one, it means that this prefix can be used for on-link determination.
- *A* is the autonomous address configuration flag. When set to one, it indicates that it could be used for stateless address autoconfiguration.
- *Valid lifetime* contains the length of time in seconds that the prefix is valid for the purpose of on-link determination.
- *Preferred lifetime* represents the period of time when the addresses generated from the prefix via stateless address autoconfiguration remain preferred.
- *Prefix* contains the prefix itself.

1.1.2.3 Redirected header

This option is used with the redirection message and contains all or parts of the packet that is being redirected. It is defined as presented in Figure 1.8.

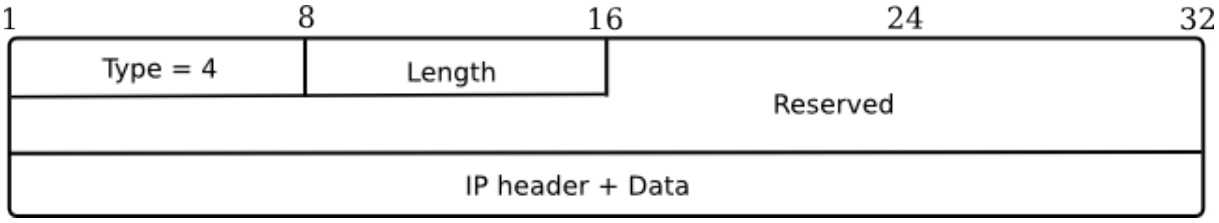

Figure 1.8: Redirected header option.

- *IP header and data* contain the original packet truncated to ensure that the size of the redirected message does not exceed the minimum MTU required to support IPv6 over the link.

1.1.2.4 MTU

The MTU option (Figure 1.9) is used in RA message to ensure that all nodes on a link use the same MTU value in case when the link MTU is not known.

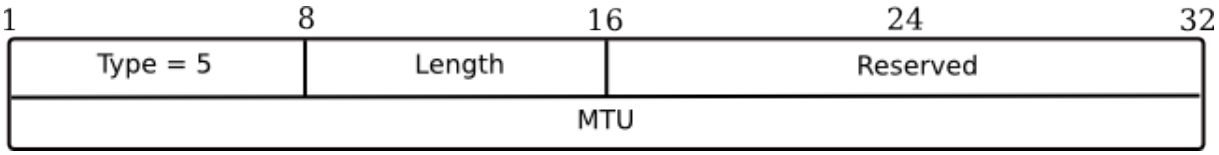

Figure 1.9: MTU option.

1.1.3 ND applications

ND is a protocol used to organize the communication over a link in an IPv6 environment. The main functions of ND [5] are reviewed below.

1.1.3.1 Address resolution

This mechanism consists of sending a NS message whose destination is given by the solicited-node multicast address. The packet's target address field contains the addresses of the neighbors to be solicited. The host responds with a NA, indicating its IPv6 address in the target address field and its physical address in the target link-layer address option. This process is done in IPv4 by the protocol ARP.

1.1.3.2 Host autoconfiguration

An interface can be conFigured manually by the network administrator, or configuration can be performed automatically. This latter possibility has become extremely important in IPv6 because of address length and the need to renumber sites more frequently. In IPv6, each address is associated with a period of validity so that network topology can be

changed automatically. To reduce the impact of renumbering on applications, two timers are used for each address. Initially, an address is classified as preferred, meaning that it can be used without restrictions as either a source or destination address. Once the preferred lifetime expires, the address goes to the deprecated state, i.e., the address is still valid but its use is discouraged for new communications. When the second timer, called the valid lifetime, expires, the address becomes invalid and can no longer be used. In this way, it is highly likely that the applications which use an address that has gone to the deprecated state will end before the address becomes invalid. There are two ways to address autoconfiguration:

- The stateless autoconfiguration which is based on the creation of an interface identifier by the host and then its verification by constructing the link-layer address related to this suffix and verifying it by using the duplication address detection which is explained after.
- The stateful autoconfiguration uses a DHCP server to assign addresses to the interfaces.

1.1.3.3 Duplicated Address Detection (DAD)

After a station gets a unicast address, it must check that it is unique before assigning it to the interface. To do so, the station sends a NS message in which the IPv6 header source address field is set to the unspecified address and the destination address field is set to the solicited-node multicast address. If the same unicast address has already been assigned to another node, the latter responds with a NA. When the node that initiated the DAD procedure receives this message, it disables the use of that address.

1.1.3.4 Neighbor Unreachability Detection (NUD)

This process gives a host the ability to test if a neighbor is still reachable or not. In fact, the host sends a NS to one of its neighbor and waits for NA. If no reply comes, it sends some other NS and in case of no response, it discards the entry related to that neighbor from its cache.

1.1.3.5 Router discovery

When a new node joins the link, it seeks for a new default router by sending a RS to the multicast address specific to routers and waits for an RA. If there is no response, the neighbor assumes that there is no router on the link.

1.1.4 ND attacks

In this section we study some of the attacks that have been discovered and have been identified as possible against ND. Those attacks are detailed in [1].

1.1.4.1 NS/NA spoofing

NS or NA spoofing basically involves the emission of faked NS or NA messages in order to modify the association between the L2 and L3 addresses of a victim's peer in its cache.

An attacker sends a NA or a NS with the target IP address as source address and a fake (or its own) link layer address, in the source link layer option, to a target peer which represents the victim. The latter updates its neighbor cache with the new information he gets. So that the attacker succeeds in spoofing the victim identity.

1.1.4.2 NUD failure

NUD is used to monitor the reachability of local destinations. In case node B left the network, and node A is sending some NUD NS message, an attacker can make a node A thinking B is still connected by sending to node A a NA using B address. So it spoofs the identity of B. In the case where B tries to join the network again, it will not be able to use its previous address and will be denied of service if it can not autoconFigure a new interface identifier.

1.1.4.3 DAD DoS attack

A new node (or a node that has changed its address) in a stateless address autoconfiguration performs DAD testing, and an attacker responds to all its NS messages. As such the host will never be able to get an address.

1.1.4.4 RS/RA DoS attacks

Many scenarios could be developed using RS/RA. Some of them are next described:

- An attacker tries to be selected by other nodes over the link as the default router. In order to be chosen as last hop router, the attacker spoofs the legitimate default router and sends periodic RA with lifetime set to zero. Once accepted, the attacker sends Redirect message to hosts and disappears. That will cause a DoS attack.
- An attacker makes all nodes on the link think that they are all local by killing the default router using a DoS attack or sending a spoofed RA with a lifetime equal to zero.

- A malicious node sends RA advertising bad prefixes. The target uses one of those prefixes to build its own address, and launches successful DAD operation (No response to NS). The target will not be able to communicate over the network.
- An attacker designs parameters concerning the way of the autoconfiguration (stateless or stateful) and sends them to the target which will use them to create its address but in fact it will be denied of service.

1.1.4.5 Replay attacks

All ND messages could be replayed. For example, an attacker could replay RA or Redirect messages of a router that retired from the link to make a DoS attack to all nodes connected. In fact, it can send a RA containing a list of fake prefixes that are no more valid and make the other nodes use them to create their addresses which will not be of a great usage. He could also replay NA to spoof the identity of a node.

1.2 Secure Neighbor Discovery (SEND)

SEND uses ND messages but has added new options and two new messages to secure the exchanges between routers and nodes and between nodes themselves.

We are going now to view the new messages introduced by this protocol and the options that it added, then we will discuss the ways SEND secures ND and at the end we will talk about possible attacks on SEND.
SEND messages and options are defined in [2].

1.2.1 SEND messages

SEND offers new functions such as Authorization Delegation Discovery (ADD) which is a mechanism based on the exchange of two new ICMPv6 messages to authenticate the router by the node before starting information exchange between the two entities.

ADD has been introduced because it was easy to conFigure rogue routers on an unsecured link and it was difficult for a node to distinguish valid source of router information from invalid one. In fact a node needs this information before starting a communication with nodes outside the link. The objective is enabling the node to verify if the local router is true or false.

The two messages that have been introduced are Certification Path Solicitation (CPS) and Advertisement (CPA). In order to authenticate a router, the node sends a CPS (Figure 1.10) asking the router for a certification path to its trust anchor (which is an entity that the node trusts, that helps verifying the routers certificate, and that might be a certification authority). After receiving the CPS, the router generates a CPA (Figure

1.11) containing the certification path from the trust anchor to its own certificate so that certificate verification is made possible by the node itself.

Trust anchor gives also the router the permission to act as a router depending on its certificate.

1.2.1.1 Certification Path Solicitation (CPS)

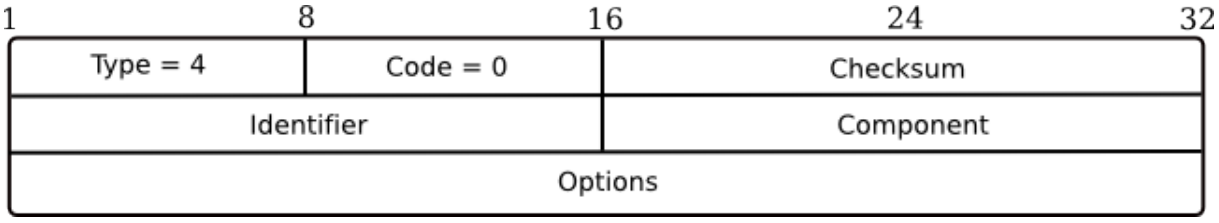

Figure 1.10: CPS message.

- The *identifier* contains an unsigned integer that helps matching advertisements to solicitations.
- The *component* field contains an unsigned integer referring to the component identifier corresponding to the certificate that the receiver wants to retrieve.

1.2.1.2 Certification Path Advertisement (CPA)

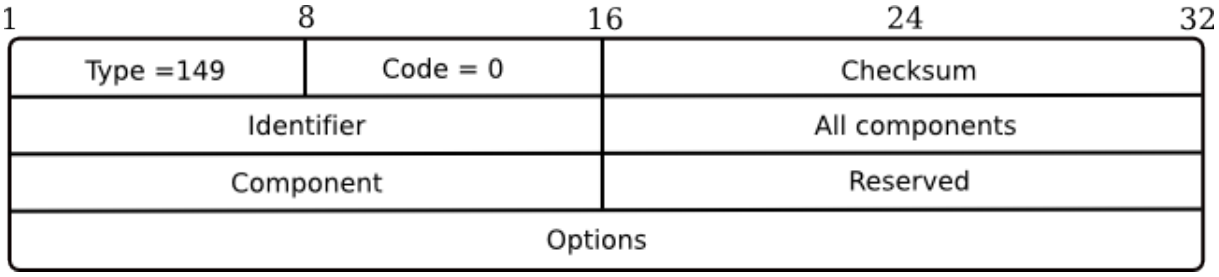

Figure 1.11: CPA message.

- The *identifier* contains the same value of the identifier field than the CPS message.
- *All components* field is used to inform the receiver of the number of certificates in the entire path.
- *Component* field is used to inform the receiver which certificate is being sent. The sending of path components should be ordered so that the certificate after the trust anchor is sent first. Each certificate sent after the first can be verified with the previously sent certificates. The certificate of the sender comes last. The trust anchor certificate should not be sent. A value of zero indicates that there are no more components coming in this advertisement.

1.2.2 SEND options

SEND has introduced six new options to secure SEND and ND messages exchanged between the hosts over a link.

1.2.2.1 CGA option

CGA generation process will be studied in chapter 2. Here we only analyse the CGA option format presented in Figure 1.12.

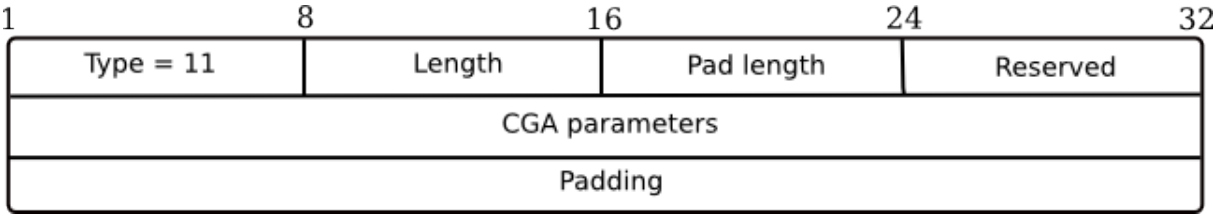

Figure 1.12: CGA option.

- The *length* contains the value of the option length in word of eight octets.
- The *Pad length* contains the number of padding octets.
- *CGA parameters* are defined as in [3] and will be detailed in chapter 2.
- *Padding* field contains bits that are set to zero and that must be ignored by the receiver.
- This option must be present in all SEND messages (we do not consider the case where unspecified address is used; this case is discussed in [2]).

1.2.2.2 RSA signature option

This option carries the RSA signature of every ND message going to be sent. It has the following structure (Figure 1.13):

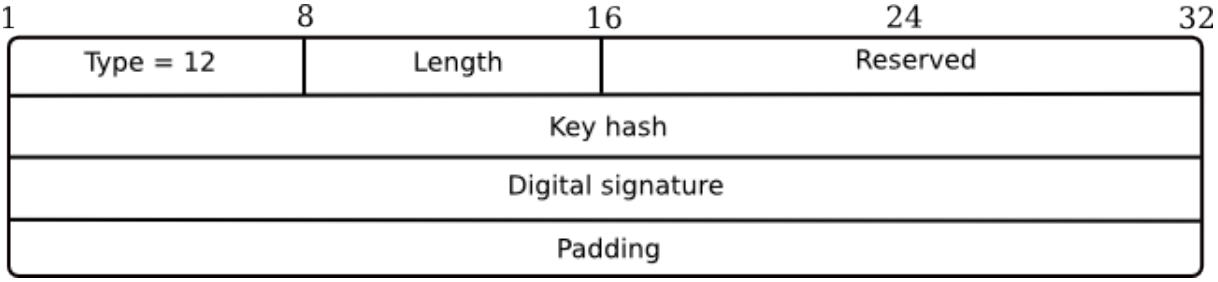

Figure 1.13: RSA signautre option.

- The *key hash* field contains the leftmost 128 bits of a SHA-1 hash of the public key used for constructing the signature. Its purpose is to associate the signature to a particular key known by the receiver (in our case it is the public key of the sender that will be verified using CGA verification described in chapter 2)

- The *digital signature* is realised using RSA signature scheme and covers the following fields: the CGA type tag value for SEND (it will be explained in chapter 2), the source address from the IP header, the destination address of the receiver, the type; code and checksum fields of the ICMPv6 header, the ICMPv6 fields that come after the checksum and before the options, and the ND options preceding the RSA signature option.

This option must be present in all SEND messages (we do not consider the case where unspecified address is used; this case is discussed in [2]).

1.2.2.3 Timestamp option

The purpose of the Timestamp option (Figure 1.14) is to make sure that unsolicited advertisements and redirects have not been replayed. This option must be present in all SEND messages.

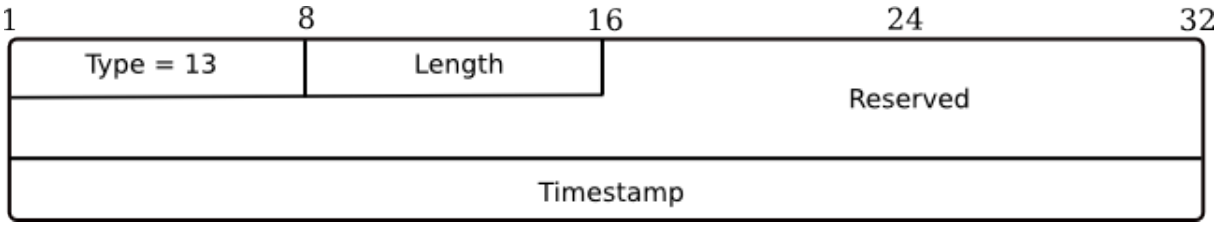

Figure 1.14: Timestamp option.

- The *timestamp* field contains the value of the number of seconds since a specific date (January 1, 1970).

1.2.2.4 Nonce option

The purpose of the nonce option is to make sure that an advertisement is fresh and matches a solicitation sent by the node earlier. This option must be present in all solicitation messages. It has the following format (Figure 1.15):

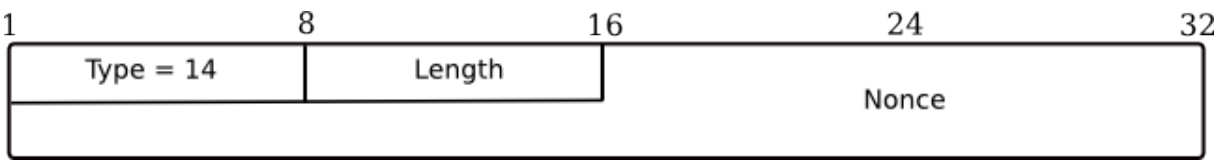

Figure 1.15: Nonce option.

- *Nonce* is a field containing a random number selected by the sender of the solicitation message (at least six bytes).

1.2.2.5 Trust anchor option

This option is used only in CPS and CPA messages. It is presented in Figure 1.16.

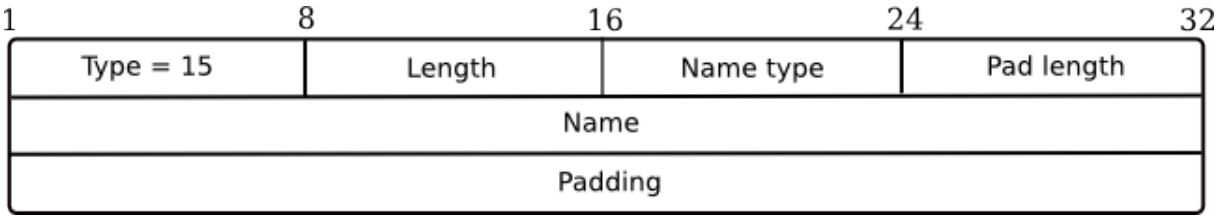

Figure 1.16: Trust anchor option.

- The *name type* contains the type of the name included in the name field. It can be: DER encoded X.501 name or fully qualified domain name (FQDN).
- The *pad length* contains the number of padding octets.
- The *name field* the name identifying the trust anchor.
- The *padding field* bits that are set to zero.

1.2.2.6 Certificate option

This option is used only in CPA messages and contains only one certificate. It is presented in Figure 1.17.

Figure 1.17: Certificate option.

- The *cert type* field contains the type of the certificate included in the certificate field.
- The *certificate* field contains the certificate to be sent to the node.

1.2.3 SEND usage

When a new node joins a network, it starts by sending a RS message to get information about the possible prefixes that it can use to generate its CGA. When it receives a response from a router, it can not be sure about its identity and the validity of its prefixes. So in order to authenticate the router, the node sends him a CPS message (Figure 1.18) including the name of a trust anchor (known and trusted by the sender) using a trust anchor option. So the router responds with a CPA message containing a list of certificates matching its certificate to the trust anchor' s one.

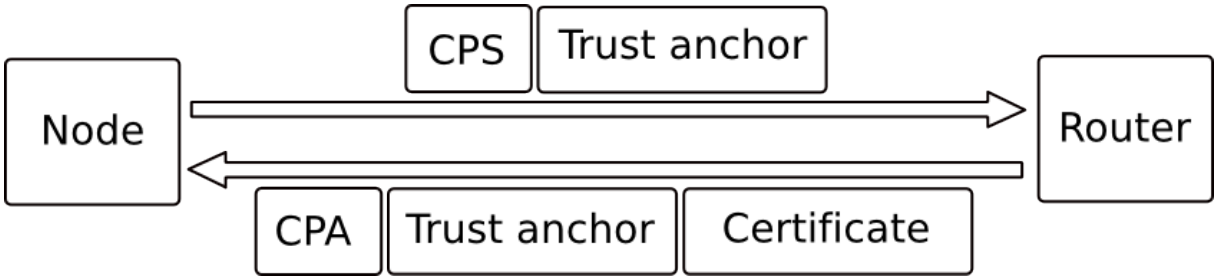

Figure 1.18: CPS/CPA messages.

After authenticating the router via its certificate, the node uses one of its prefix with some parameters to compute its CGA and to form the corresponding CGA parameters structure. The node is now able to communicate with its neighbors using SEND messages.

Two situations are possible:

- When the node sends a solicitation message (NS or RS), it has to include the following options: CGA parameters option, RSA-signature option, timestamp option and nonce option. The use of nonce is mandatory in this case because the message sent is a solicitation. The corresponding response (advertisement) has to contain the same nonce value in order to create a binding with the solicitation (Figure 1.19).

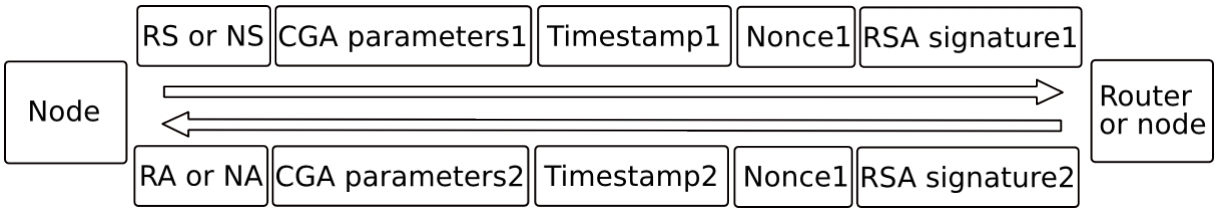

Figure 1.19: SEND options usage.

- When a node or a router sends an advertisement without being solicitated, it does not need to include a nonce. This may happen when a node changes one of its addresses (physical or CGA(IPv6 address)) or when a router wants to multicast new information about its prefixes.

1.2.4 SEND securing ND

In this section, we study how SEND secures ND against the attacks that have been detailed in the first section of this report.

1.2.4.1 NS/NA spoofing

The threat here is that a spoofed message may cause a false entry in a node's Neighbor Cache. This attack is countered thanks to the use of CGA option and RSA signature option. NS or NA messages will have to be signed to make an entry in the cache. So a node receiving a NS or NA message has to verify the identity of the sender thanks to the signature in order to update its cache. An attacker will have to realize a collision with the CGA to get a valid private key which makes him able to sign the fake NS or NA that he creates.

1.2.4.2 NUD failure

Thanks to the use of the CGA and RSA signature options, a proof of authorization to use the interface identifier of the address being solicitated is given. So only the address owner is able to responds to the NUD probes.

An attacker will not be able to spoof the identity of the leaving node because it will have to respond to NS using a message signed by the target private key which it does not have.

1.2.4.3 DAD DoS attack

In the DAD mechanism, the NA message in response to a NS message mechanism must include RSA signature option as proof of identity. The attacker will not be able to respond to a NS message because it does not possess the private key needed to generate the RSA signature.

1.2.4.4 RS/RA attacks

To counter those attacks, a router will only have to add RSA signature option to the RA message. So nodes will have to accept only signed messages coming from a router. In this case, a malicious node which wants to execute those attacks will have to get the router's private key to be able to sign its self generated RA messages.

1.2.4.5 Replay attacks

In order to counter the replay attacks, SEND introduced two options which are the timestamp for the unsolicited advertisements and nonce for solicited advertisements in order to create a challenge response protocol (concerning the nonce option).

Conclusion

In this first chapter, we have introduced the context in which CGA has been developed. In the sequel, we are going to study in details the CGA generation process, the different modifications that have been supposed to be added to CGA and its different usages.

Chapter 2

CGA generation, verification and structure

Introduction

In order to introduce a new mechanism to decentralize the association between a public key and its owner, CGAs have been introduced.

The older techniques used were all centralized with a unique entity generating certificates binding hosts to their public keys. The certificates generation entity is called the certification authority.

CGAs are IPv6 addresses where the interface identifier contains information about the public key of the host owning the address.

In this chapter we introduce the algorithms used for CGA generation, verification and the structure used to create these addresses. Then we describe the possibility of introducing some modifications in the generation process.

2.1 CGA presentation

CGAs were initially developed to be used with the SEND protocol [2]. They were used to secure neighbor discovery messages exchange [5].

Before the generation of a CGA, the host must choose a three bits length security value SEC which is used to make brute force attack against CGA generation algorithm very difficult.

Every host in a network can generate its own CGA based on two SHA-1 hashes calculus over a structure called the CGA parameter structure. The 64 leftmost bits of the first hash (hash1) are used as interface identifier with no regard to the three first bits (if we are counting from the left to the right), which correspond to the SEC value, and to seventh and eightieth bits which are the universal and group flags of an IPv6 interface identifier

(known as the u and g bits in the standard IPv6 address architecture format of interface identifiers defined in [9]). The 112 leftmost bits of the second hash value (hash2) are used

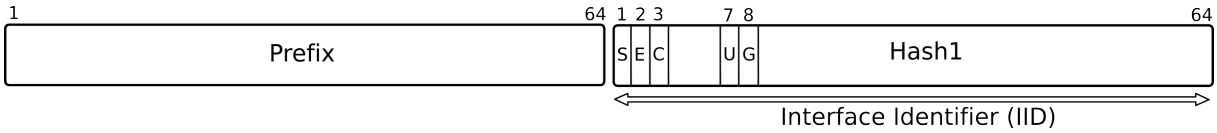

Figure 2.1: A Cryptographically Generated Address.

with SEC value to increase the complexity of CGA generation. To validate a generated address, we must verify that 16*SEC leftmost bits of hash2 are equal to zero (for example if SEC = 2, we will have to verify that 32 leftmost bits of hash2 are equal to 0, else we will have to recompute the address).

Now, we are going to study the CGA parameters structure.

2.2 CGA parameters structure

CGA parameters structure has the format giving in Figure 2.2:

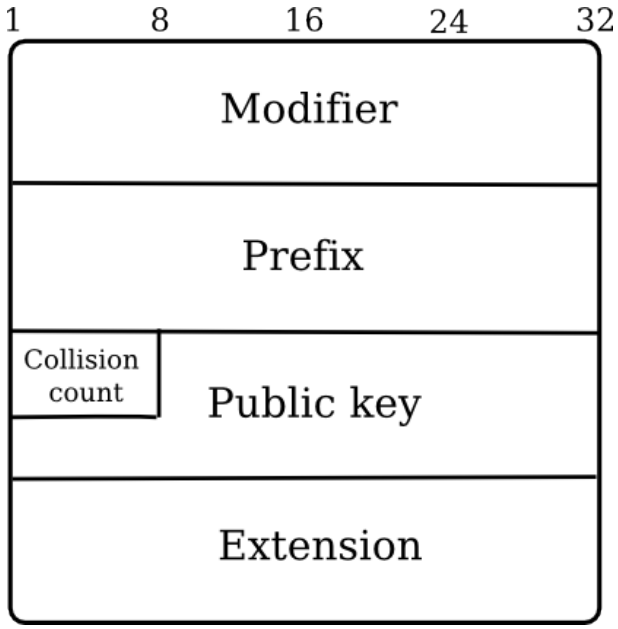

Figure 2.2: CGA parameters structure.

- *Modifier* contains a 128 bits random value. It is used during CGA generation to implement hash extension and to enhance privacy by adding randomness to the address.
- *Subnet prefix* contains the subnet value of the CGA.

- *Public key* field contains the public key of the host which is DER encoded. It should have at least a length of 384 bits. Actually the only key used in SEND is generated using RSA[1].
- The *extension* field is an optional field.

2.3 CGA generation

The CGA generation process takes as input three parameters: the public key of the host (DER encoded), the prefix of the address and the security parameter SEC. To generate a CGA, the following steps must be followed (see Figure 2.2):

1. Choose a random modifier.
2. Compute hash2 using as input to the SHA-1 function the following elements concatenated respectively from the left to the right: the modifier, nine bytes of zero (prefix field and collision count are taken equal to zero), the public key (DER encoded), and extensions. (We take only the 112 leftmost bits to form hash2)
3. Compare the 16*SEC leftmost bits of hash2 to zero: if they are all equal to zero then go to the next step, else increment the modifier value by one and go back to step 2.
4. Set the collision count to zero.
5. Compute hash1 using as input to the SHA-1 function the following elements: the final value of the modifier, the prefix, the collision count, the encoded public key and all extensions. (We take only the 64 leftmost bits to form hash1)
6. Form an interface identifier using hash1 by replacing the three leftmost bits by SEC value and by setting the u and g bits to zero (those bits are equal to the seventh and eightieth bits if we start counting from the left).
7. Form an IPv6 address using the prefix (that was provided as input) and the calculated interface identifier.
8. Perform duplicate address detection (defined in chapter 1 and [5]), if a collision is detected, increment the collision count by one and go to step 5.
9. Form the CGA parameter structure as it was defined in section 1.2.

[1]For more information about RSA key generation please view appendix B

To make it possible for mobile nodes whose subnet prefixes change frequently to use SEC values greater than zero, it has been decided not to include the subnet prefix in the input of Hash2.

The collision count must not have a value greater than two. First, it is unlikely that three collisions would occur and this precludes any DoS attack. Second, an attacker searching to match a given CGA interface identifier with its own public key can try all different values of a collision count without repeating the brute force search for the modifier value.

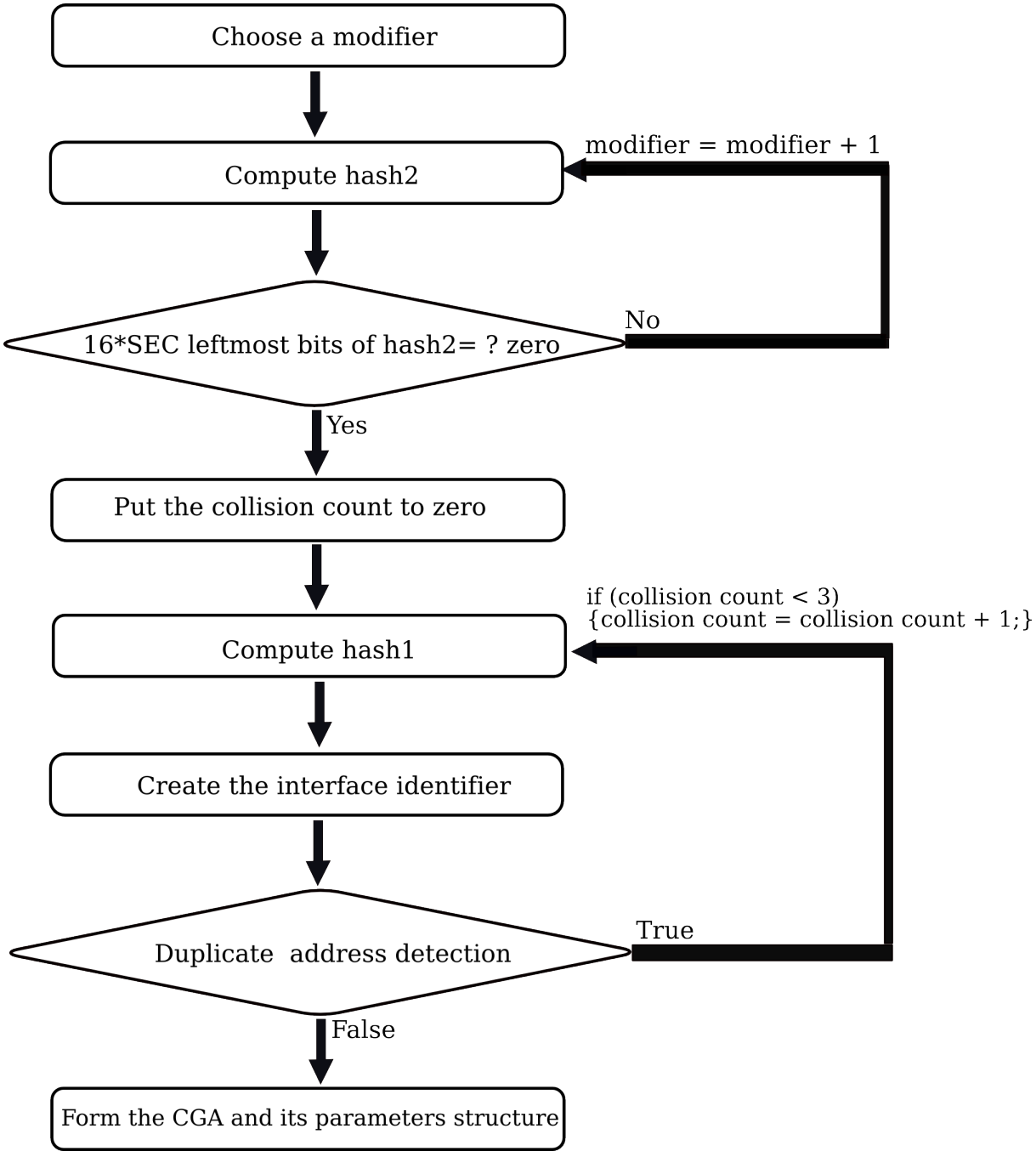

Figure 2.3: CGA generation algorithm.

2.4 CGA verification

When receiving a CGA address with its structure, the host can verify the address by executing the following process but he will not be able to detect if the packet containing the address is being replayed and will not also authenticate the sender unless the packet is signed. The following steps must be performed to check the validity of the address (see also Figure 2.3):

1. Check that the collision count is equal to 0, 1 or 2.
2. Check that the subnet prefix of the address is equal to the prefix value of the CGA structure.
3. Compute hash1 by applying SHA-1 to the CGA parameters structure.
4. Compare hash1 to the interface identifier of the address while ignoring the three first bits corresponding to SEC value and the bits corresponding to u and g.
5. Get the SEC value by taking the three leftmost bits of the interface identifier.
6. Compute hash2 by applying the SHA-1 algorithm to the CGA structure containing: the modifier, nine bytes of zero, the public key and all extensions.
7. Compare 16*SEC leftmost bits of hash2 with zero.

If the verification fails at any step, the execution of the algorithm must be stopped. Else, the verifier will have verified the address and the public key of the sender.

2.5 CGA signature

When a host wants to sign a message, he needs his private key (generated using RSA), his CGA, the corresponding CGA parameter structure and must also have a 128 bits type tag (which is a randomly chosen value, which prevents accidental type collisions with other protocols). First, it concatenates the type tag value with the message to sign (respectively from the left to the right) to form a new message which will be used next as input to the [2] signature algorithm (actually it is the only kind of signature used). Finally, it sends the message, its signature and also the CGA parameter structure.

The node that receives the previous data will have to verify the signature. So it starts by verifying the CGA using the algorithm presented in section 1.4. Then, it concatenates the message received with the type tag to form the new message which is used as input to the digest computation and finally it executes the RSA signature verification algorithm.

[2]For more information about RSA signature please view appendix B

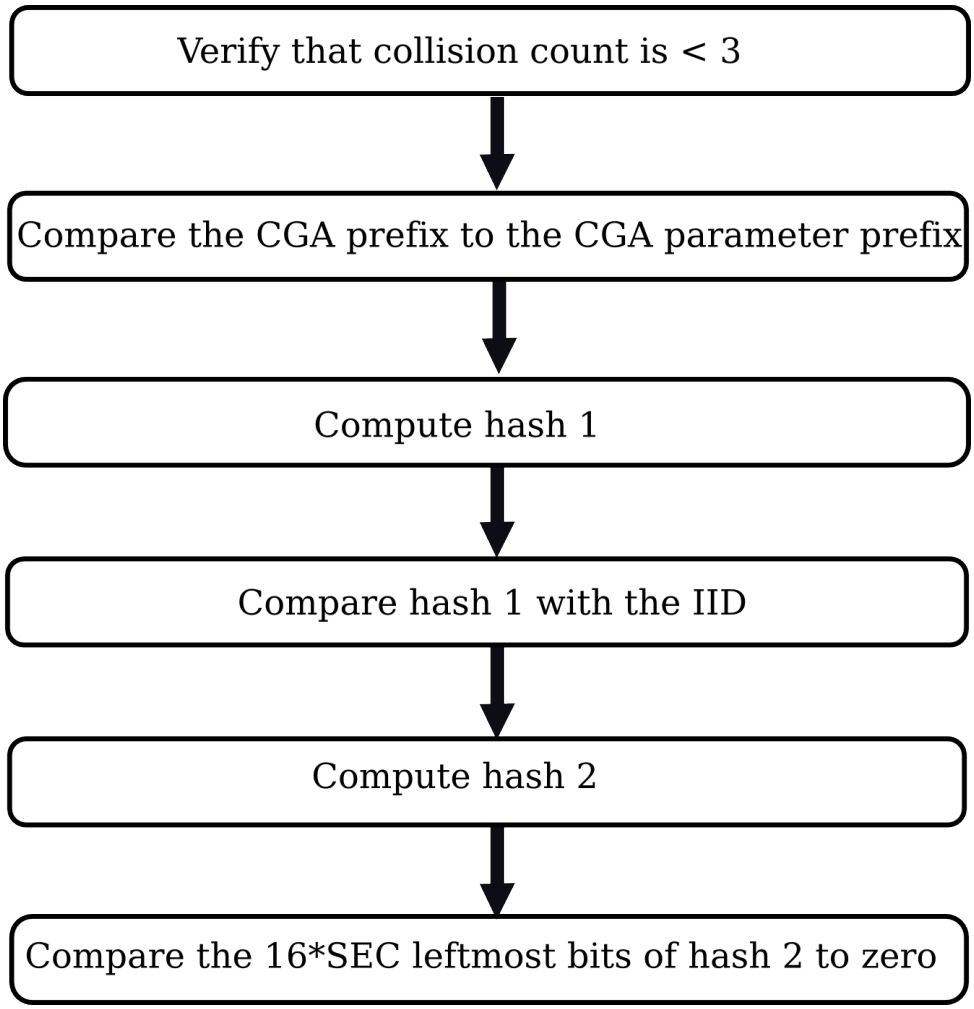

Figure 2.4: CGA verification algorithm.

The verifier must accept the signature as valid if only both the CGA verification and the signature verification succeed.

When a node transmits a message and its signature, it does not send the type tag value because it must be known by all the nodes belonging to the same network. The only type tag used actually is the SEND one.

2.6 Support for multiple hash algorithm in CGA

The hash function used when generating a CGA is SHA-1. RFC 4270 [10] presents some successful attacks against hash function. For instance, it has been verified that the property of collision free of some hash algorithms is no more reliable. It means that an attacker can find two messages that produce the same hash. So the non repudiation of a signed message is no more trustworthy.

As CGA uses the SHA-1 only to create the interface identifier, the previous kind of attacks is not feasible. “Essentially, all the current applications of CGA rely on CGA to

protect a communication between two peers from third party attacks and not to provide protection from the peer itself" [11] so there is no need to provide non repudiation property.

The need to enable multiple hash function support in CGA is motivated by the unknown needs of future CGA applications which may become susceptible to attacks against the collision free property of SHA-1. In addition, present attacks against hash functions encourage providing the possibility of using new powerful hash function. When speaking about implementing new hash algorithm with CGA, we must answer the following question: where to encode the hash function being used? Here are some proposals of encoding hash functions:

- The first idea that comes in mind is to include a new Type-Length-Value (TLV) extension to the CGA parameters structure. This hash algorithm extension contains the hash algorithm used. This kind of extension makes CGA susceptible to the downgrading or bidding down attacks. In the bidding down attack, the attacker generates the same target CGA using a weak hashing algorithm having as input the CGA parameters structure (belonging to the target) and the corresponding hash algorithm extension. Even if the target generated its address using a strong hash algorithm. A way to counter this attack is to encode the CGA hash function identifier in the address itself.

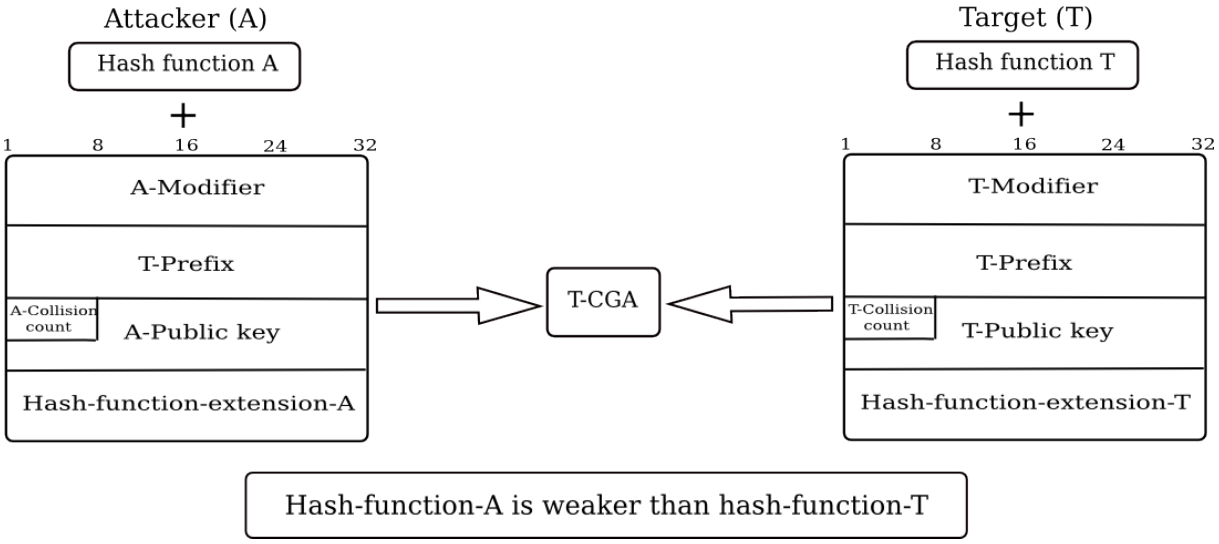

Figure 2.5: Bidding down attack.

- The second approach consists in using some bits of the interface identifier to encode the hash algorithm version. This method is rejected because it weakens the CGA generation as less hash information can be encoded in the interface identifier.

- The most interesting approach consists in using the security parameter SEC [11] to encode the hash value used. As such, SEC bits carry information about brute

force attack and hash function identifier. SEC is used to express the need of using a new hash function and to fix the amount of bits of hash2 that must be equal to zero (16*SEC leftmost bits of hash2 must be equal to zero). If the number of hash functions increases, some SEC values will be reused (for example the SEC values corresponding to the weakest hash functions). In any case, an implementation must not simultaneously support two different meanings of SEC.

2.7 CGA and DHCP

CGA is created by a node using its public key and some parameters. It is a kind of stateless address autoconfiguration. Its generation algorithm is complex and includes some operations which are specially added to increase the cost of brute force attack against this type of address.

In a CGA parameters structure, the modifier is a 128 bits random integer whose computing operation consumes many resources. It also depends on the security parameter SEC which is going to be used during the CGA generation.

The SEC value is chosen by the administrator of the network depending on the infrastructure and the level of security needed. Greater is SEC, more efficient is the provided security, but also greater is the time and power consumption for the generation algorithm. This generation function is too much consuming for mobile nodes having restricted computational capacities and power. To overcome this problem, one proposed solution is to delegate the computation of the modifier to a server which could be a DHCP server.

Using a server offers some new features such as asking nodes generating their own addresses to register. So nodes belonging to the same network are authenticated. In mobile networks where proxying is needed, a server can be used to notify end hosts about the proxying of their SEND messages. The server informs the proxy about the different CGA created and their corresponding parameters structure so it uses its private key to sign messages coming from those addresses.

In this section, we present a proposal that is based on a DHCP server helping nodes to generate CGA and providing a centralised administration [12] [13].

DHCP server can be used to:

- Inform nodes about the parameters used to conFigure a CGA (see Figure 2.6).
- Compute the modifier so the node will not have to do the exhaustive calculus. The server can compute the modifier or redirect the CGA parameters to another host which will do the computation. Next, the server can generate the CGA and send it to the requesting node (see Figure 2.6).
- Register the generated address (see Figure 2.7).

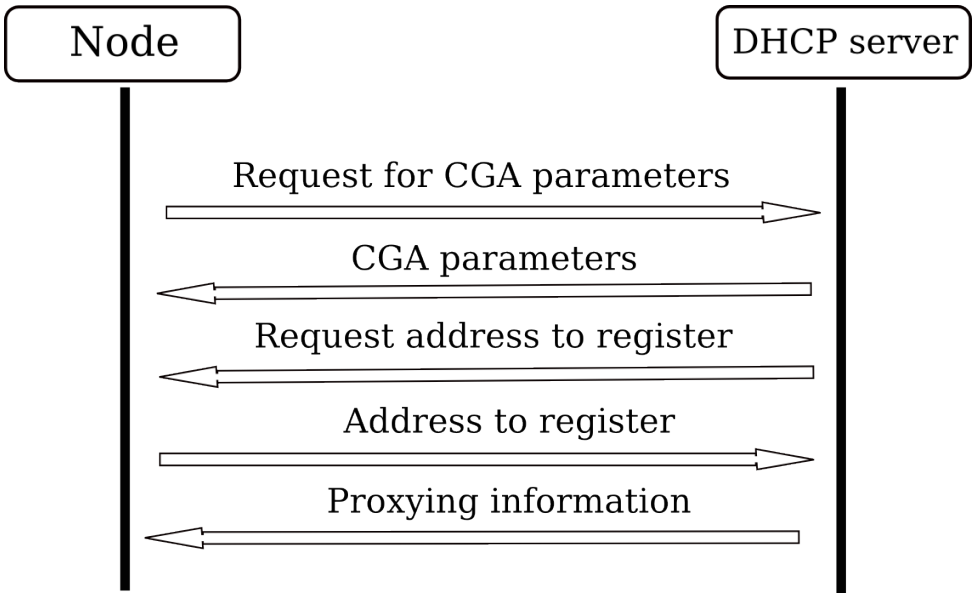

Figure 2.6: DHCP server sending CGA parameters.

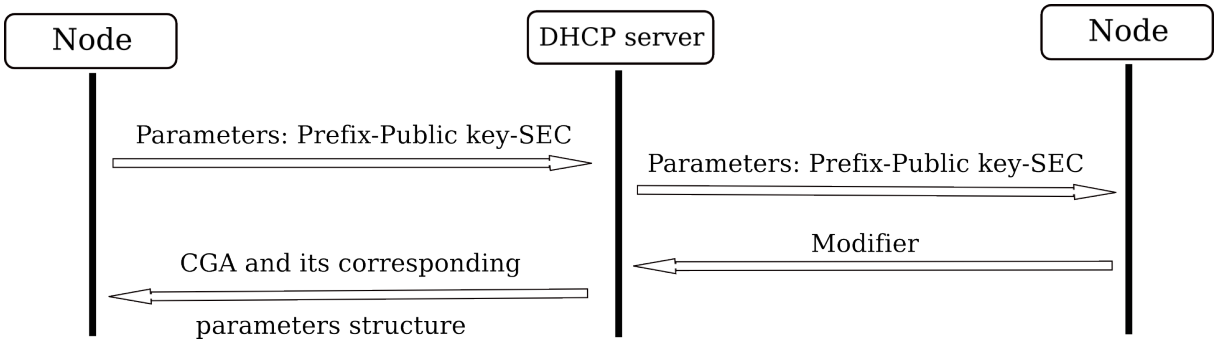

Figure 2.7: Modifier computation by the DHCP server.

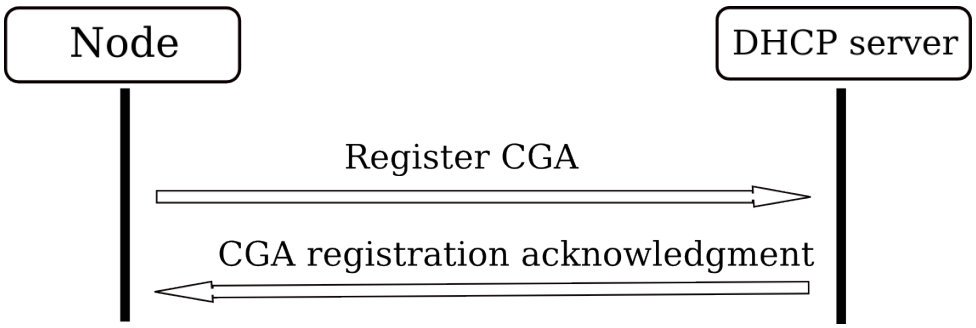

Figure 2.8: Generated address registration in the DHCP server.

- Forward proxying information.

There are some threats that appear due to the fact of using DHCP with CGA, we can list for example:

- A malicious node propagates in DHCP message a SEC value providing less security than it would be desired by the network administrator.

- A malicious node overloads the server with a number of requests for generating modifiers with different values of SEC trying to cause denial of service especially with the SEC greatest values (7 and 6). This attack is very difficult to realise thanks to the limited number of SEC values (8).
- An attacker (spoofing a DHCP server) requests from a node to generate different modifiers (when they are generated locally) using different requests containing diverse SEC values. The attacker aim is to cause a denial of service.
- A malicious node generates fake messages carrying fake information to exhaust proxy resources.
- An attacker overloads the DHCP server with address registration requests until causing denial of service.

Conclusion

In this chapter, we have seen how to generate a CGA and how to verify it. We have also explained the possible extension that has been proposed to enhance the possibility of using another hash function than SHA-1 during CGA generation. Concerning the use of a DHCP server to help nodes acquiring a CGA, it still only a proposal but it could become a reality especialy in networks using CGA with power and computational limited devices.

Chapter 3

CGA usage in multihoming and mobile networks

Introduction

Initially CGA was introduced to be used with SEND in order to secure ND. Actually CGAs are being adapted for usage in mobile networks where ND proxying is needed and in multihomed networks where a node can be connected to many internet service providers (ISP) at the same time.

In this chapter, we first introduce how CGA was adapted to be used in multihomed networks with Hash Based Addresses (HBA) [14]. Next we present the different solutions proposed to enable secure ND proxying.

3.1 CGA and HBA usage in multihoming networks

In this section we describe HBA and how to use them with CGA.

3.1.1 HBA presentation

The concept of HBA has been introduced specially for nodes belonging to multiple networks at the same time. In this case, a node has multiple addresses with multiple prefixes.

The threats that affect multihoming networks are basically redirection attacks which could be divided into two groups: hijacking attacks that consist on stealing ongoing or future communications from a victim and flooding attacks.

To counter these threats, we use HBA with the protocol SHIM6 [15].

The SHIM6 is a layer 3 approach and protocol for providing locator agility below the transport protocols so that multihoming can be provided with failover and load sharing properties. It introduces a new approach to associate locator as upper-layer identifiers. In fact, the identifiers name space corresponds to the locator selected in the initial contact

with the remote peer as the upper layer identifier (ULID). Even if it happens that the locator changes due to a failure, the upper protocol stack elements will continue to use this upper level identifier without changes.

SHIM6 is used by a host which has multiple IPv6 addresses to setup state with other hosts. This state can be used later to failover to a different locator pair. This protocol allows existing communications to continue when a site that has multiple connections to the internet experiences an outage on a subnet.

HBA binds together multiple IPv6 addresses that belong to the same node. It contains information about the different prefixes. The technique used consists on including a hash of the permitted prefixes in the low order bits of the IPv6 address. Its motivation is to provide a mapping between the ULID and the different locators.

For example, we consider a node having three ISP. It will have three addresses having three different prefixes: A, B and C. To create the different HBAs the following actions have to be executed:

1. Create a list of prefixes: L = A, B, C
2. Choose a random value called modifier.
3. Compute the interface identifier for every prefix from a hash value having as input: the value of the modifier, the prefix and L.
4. Create the IPv6 addresses as presented in Figure 3.1.

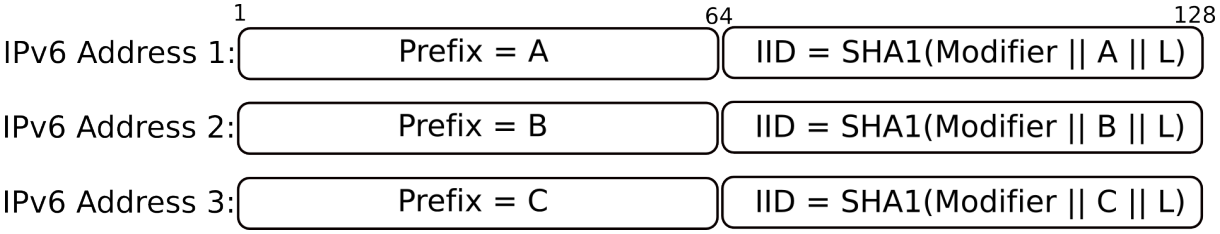

Figure 3.1: Example of three HBA.

In next sections, we detail HBA generation algorithm and then expose a solution using HBA and CGA together.

3.1.2 HBA generation algorithm

When generating a HBA these steps could be followed (this algorithm has been defined in [16] and it is used to generate only one HBA):

1. Choose a security value SEC.
2. Compute C = SHA-1 (hba ‖ SEC ‖ Subnet Prefix ‖ Modifier ‖ P_1 ‖ ... ‖ P_N), hba is a string and P1 ... PN are parameters.

3. Compute G = 64 + 20 * SEC rightmost bits of C, if the 20 * SEC leftmost bits of G are zero, go to the following step. Else, compute a new modifier and go to step 2.

4. Create the interface identifier by taking the rightmost 64 bits of C and putting the u and g bits to one and the bits 63 and 64 to SEC value.

5. Create the IPv6 address: @ = Subnet Prefix || interface identifier.

In a multihoming network the parameters ($P_1 \ldots P_N$) are remplaced by the prefixes of different ISPs.

3.1.3 HBA/CGA compatibility

The HBA technique uses the interface identifier of an IPv6 address to encode information about the multiple prefixes used by a multihomed host but it does not contain cryptographic information like CGA.

There are at least two reasons to provide HBA/CGA compatibility:

1. If HBAs are not compatible with CGAs, multihoming node will not be able to do secure neighbor discovery using SEND. HBA provides only fault tolerance (using SHIM6) but it does not provide security (using SEND).

2. CGA provides additional features that can not be achieved using HBA only: it is impossible to add new prefixes to the original set of prefixes after generating the set. So the new prefix will not be available for established communications and only new ones benefit from it.

The algorithm of HBA/CGA generation has as input the public key of the generating node and the list of prefixes belonging to different ISPs. The node using HBA/CGA tells its peers to use HBA verification when one of the addresses of its HBA/CGA set is used as locator or to use CGA verification when a new address that does not belong to the HBA/CGA set is used as locator.

The parameter introduced to bind HBA and CGA is a multi-prefix extension for CGA parameters structure.

The HBA set will be identified by a CGA parameter data structure that contains a multi-prefix extension.

Generation of a HBA is like using the CGA parameters structure but with a random number replacing the public key.

The multi-prefix extension has a TLV format (Type-Length-Value) and has the structure presented in Figure 3.2.

- The *P* bit is used in the extension to indicate whether a public key is used or not in the HBA/CGA parameters structure.

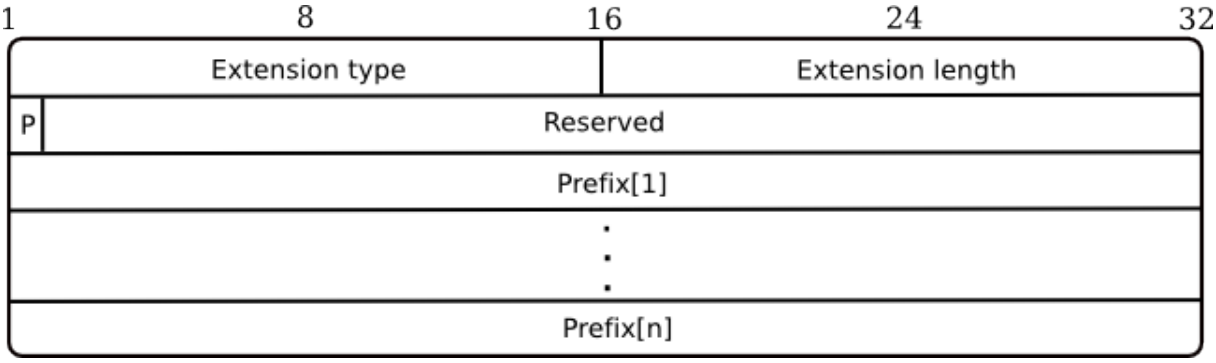

Figure 3.2: Multi-prefix extension for CGA parameters structure.

3.1.4 HBA/CGA generation algorithm

This algorithm is taken from the CGA generation process but it has been modified to enhance the possibility of creating many CGAs using the multi-prefix extension. It takes as input: the nodes public key, a vector of n 64 bits prefixes and a SEC value. It has the following steps (see Figure 3.3):

1. Generate the multi-prefixes extension and the vector of n 64 bits prefixes. If a public key is given then the P flag is set to one.
2. Choose a random modifier.
3. Compute hash2 using as input to the SHA-1 function the following elements concatenated respectively from the left to the right: the modifier, nine bytes of zero (prefix field and collision count set to zero), the public key (DER encoded), and extensions. (We take only the 112 leftmost bits to form hash2)
4. Compare the 16*SEC leftmost bits of hash2 to zero: if they are all equal to zero then go to the next step, else increment the modifier value by one and go back to step 3.
5. Set the collision count to zero.
6. For i from 1 to n (number of prefixes) do
   (a) Compute hash1[i] using as input to the SHA-1 function the following elements: the final value of the modifier, the prefix[i], the collision count, the encoded public key and the extension. (We take only the 64 leftmost bits to form hash1)
   (b) Form an interface identifier using hash1[i] by replacing the three leftmost bits by SEC value and by setting the u and g bits to zero (those bits are equal to the seventh and eightieth bits if we start counting from the left).
   (c) Form an IPv6 address using the prefix[i] (that was provided as input) and the calculated interface identifier.

(d) Perform duplicate address detection, if a collision is detected, increment the collision count by one and go to step 6.

(e) Form the CGA parameter structure using: the final value of the modifier, the prefix[i], the final value of the collision count, the public key and the multi-prefixes extension.

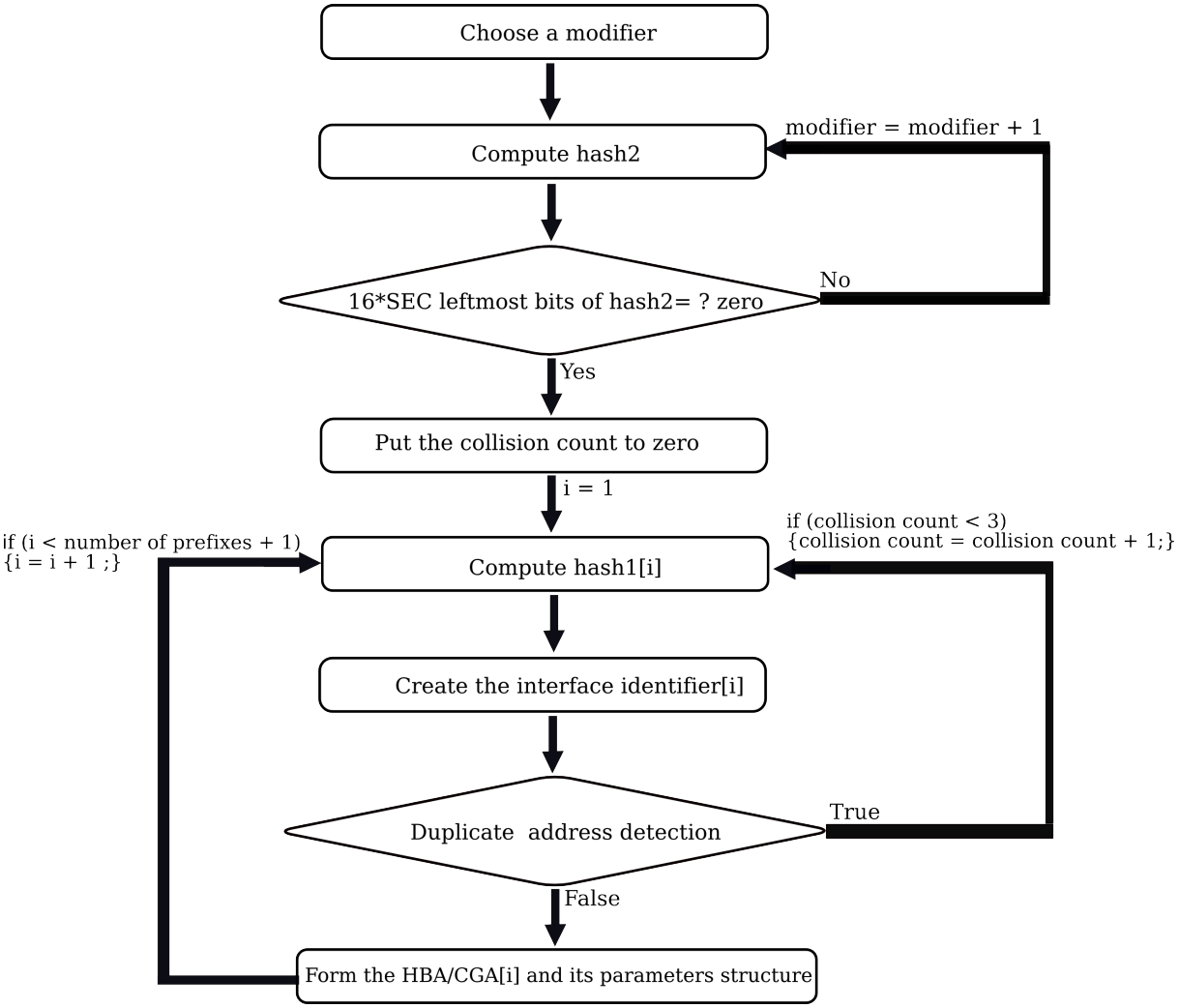

Figure 3.3: HBA/CGA generation algorithm.

3.1.5 HBA/CGA verification

To verify a HBA based on CGA, we must start by executing the verification process presented in section 2.4. This verification aims to prove the binding existing between the address and the corresponding CGA parameters structure. Such verification is useful when the goal is creating a binding between the public and the HBA.

Next, we have to verify that the HBA belongs to a HBA set associated to a given CGA parameters data structure. A HBA set is identified by a CGA parameter data structure that contains a multi-prefix extension.

To verify that a HBA(1) belongs to a HBA set associated with another HBA(2), we verify that the HBA(1) prefix is included in the prefix set defined in the multi-prefix extension, then we substitute the prefix included in the subnet prefix field (of the CGA parameters structure associated to HBA(2)) by the prefix of the HBA(1) and we perform the CGA verification process defined in section 2.4.

The inputs to the HBA verification process are a HBA and a CGA parameters structure. It is described by the following steps and Figure 3.4:

1. Verify that the HBA prefix is included in the prefix set defined by the multi-prefix extension.

2. Run the CGA verification process described in section 1.4 after changing the subnet prefix field with the prefix of the address being approved.

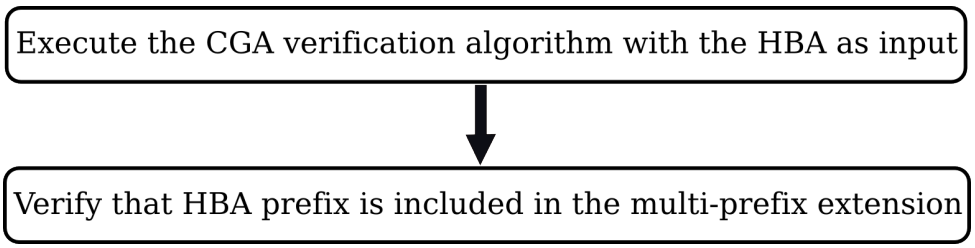

Figure 3.4: HBA/CGA verification algorithm.

3.2 CGA and ND proxying

In IPv6 mobile environment [17], it is necessary to provide ND proxying. The proxy known as Home Agent performs address defence which consists in protecting the mobile nodes address when it leaves the home link. This allows other nodes belonging to the same link to continue sending traffic to the node. It also prevents any new arriving node from claiming the mobile nodes address [18].

The ND proxy is responsible for answering NS that are addressed to a mobile node which left the link. The proxy answers using a NA message whose target address field contains the mobile node's address. However the IPv6 address and the link layer address correspond to the proxy's ones.

In this section, we take a look at neighbor discovery (ND) proxying [19]. We explain why it is not possible to use SEND to secure ND proxying. At the end, we discuss the proposed solution for securing ND.

3.2.1 Proxy behaviour

When a proxy receives an IPv6 packet, it updates its interface cache by adding an entry for the sender. The way it proceeds this packet depends on its type. It focuses on if it

negotiates link layer address or not, because only packets carrying information about link layer addresses are going to be proxied such as neighbor solicitation message packets.

When a proxy receives multicast packets on one of its interface, it forwards it unchanged on all other proxy interfaces on the same link. When it receives ordinary unicast packet, it forwards it to its interface for which the next hop address appears in the neighbor cache (if it is not locally destined).

The link layer header and the link layer address within the payload for each forwarded packet will be modified as follows:

- The source address will be the address of the outgoing interface.
- The destination address will be the address in the neighbor entry corresponding to the destination IPv6 address.
- "The link layer address within the payload is substituted with the address of the outgoing interface [19]".

3.2.2 ND proxying

Two scenarios exist for ND proxying as described below.

3.2.2.1 IPv6 mobile nodes and ND proxy

A proxy has to deliver packets to a node whether it is present or not in the home network. In mobile IPv6 network, the proxy corresponds to the home agent. It answers to NS destinated to a node which left the link. It responds using a NA message whose target address field contains the mobile node address but the IPv6 address and the link layer address correspond to the proxy ones(see the example in the Figure 3.5).

In this case, no solicitations are proxied because the proxy generates the advertisement itself. The proxy needs to override existing valid nodes cache entries (which could be protected by SEND) when a host leaves its home network. Next, it uses its link layer address to redirect the flow of packets in the tunnel connecting the home agent/proxy to the mobile node.

3.2.2.2 Bridge like ND proxies

In this case, the proxy takes place between two segments, it forwards messages while modifying their source and destination MAC addresses and rewrites their link layer address options and override flags.

The bridge like ND proxying (Figures 3.6 and 3.7) has been defined in [20]. It allows also router discovery operation based on the exchange of a RS and RA messages.

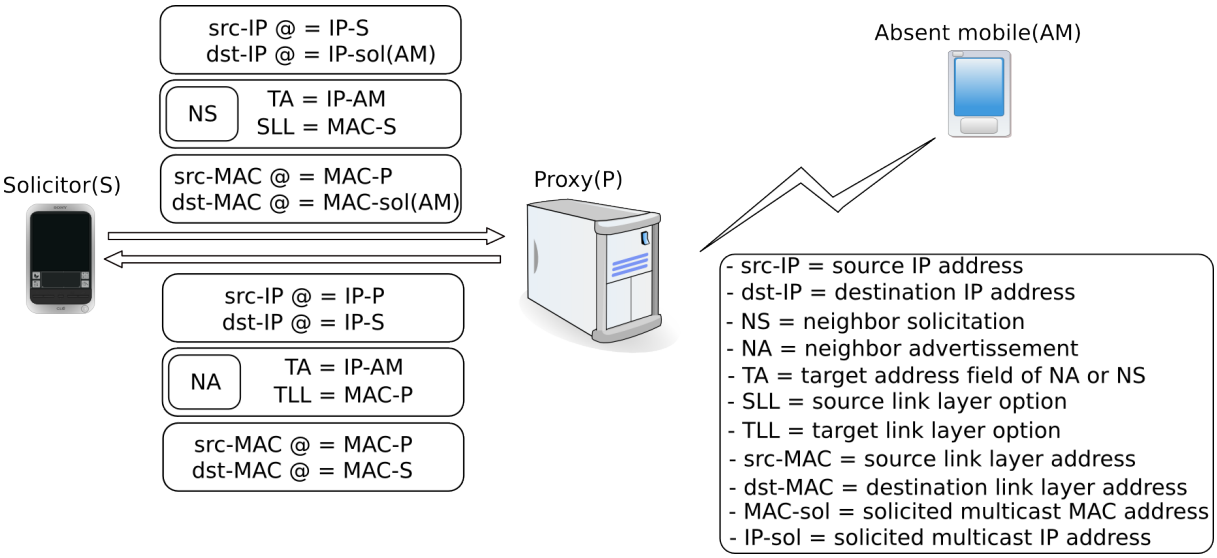

Figure 3.5: Example of ND proxying in mobile IPv6.

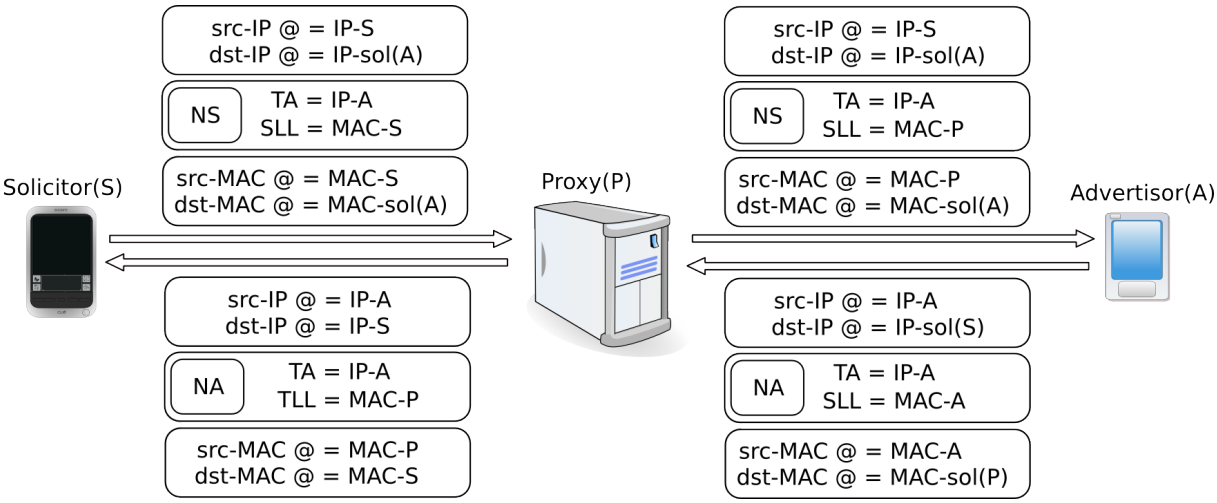

Figure 3.6: Example of bridge like ND: NS/NA messages.

3.2.3 Securing ND proxying

All proxied messages presented previously are not secured because they are based on the ND protocol which is itself insecure. So all attacks against ND are still feasible against proxied ND. In addition, some malicious nodes can spoof the proxy identity and cause denial of service attacks. For instance, an attacker which executes spoofing will not forward the messages which the proxy is supposed to forward causing unreachability between the communicating nodes.

One approach consists on using SEND messages to secure proxied ND but this can not be done without changing SEND or the proxying procedure. In fact, a proxy has not the public and private key pair used to generate the mobile node address and to sign different SEND messages. So the fact of rewriting the message (which is going to be forwarded) breaks the digital signature of the generating node.

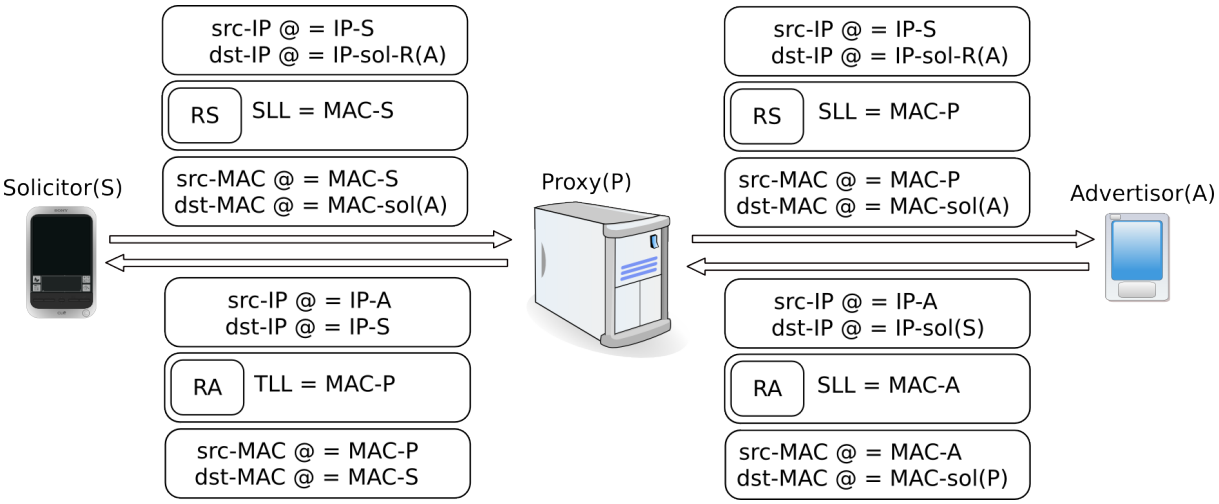

Figure 3.7: Example of bridge like ND: RS/RA messages.

In the next sections, we discuss two proposals to secure neighbor discovery proxying based on CGA.

3.2.3.1 Multi-key CGA

We have seen previously the two major ways of proxying:

- Proxying of a mobile node s home address by its home agent: in this case the mobile generates a certificate authorizing the home agent to proxy the address.
- Proxying by a bridge approach: in this case the proxy obtains a certificate from the router authorizing it to proxy.

The problem with these methods is that a querying node can discover from the certificate and its parameters whether the message is generated by a proxy or by the node itself. An attacker could find out whether or not the owner of the address was on its home link. So it can try to spoof the address while its owner is off link.

The idea proposed to provide anonymity of the signer consists in modifying SEND signature option. A signer has to use a kind of signature which will not permit the signature verifier to determinate who signed the message but it will be able only to verify the validity of the signature. The used signature scheme is called ring signature[1] [21].

In a ring signature scheme, the signer uses its own private key and the public keys of a possible group of signers that it chooses randomly (even without their agreement). The ring signature is signer-ambiguous which means that the verifier should be unable to determine the identity of the actual signer in a ring of size r with a probability greater than 1/r[21].

[1]Ring signature algorithm is detailed in appendix C

Using this kind of signature permits a mobile node to use its own private key and a proxy public key to sign all its proxied messages. An attacker will not be able to determinate who signed the message and consequently the state of the mobile node which he is targeting. The node uses in that case an RST (Rivest-Shamir-Truman) ring signature[21] option instead of a standard SEND RSA signature option. In addition, it will add an RST signature suboption in the CGA parameters option. So the CGA parameters option will contain the public key of the node and the public key of the proxying router in the RST signature suboption. It generates a new kind of CGA called Multi-key CGA[2] [18].

The multi-key CGA generation algorithm has the same steps than the CGA generation algorithm but it includes only one change in step 2 concerning the hash2 defined in chapter 2 section 2.3. Hash2 is computed using an SHA-1 hash over the structure formed by: the modifier, 9 bytes of zeros, a hash value having as input the nodes and routers public keys instead of the DER-encoded nodes public key and some extensions (if any).

The verification algorithm also includes one modification concerning the step 6 of the CGA verification algorithm defined in section 2.4 of the previous chapter. This change is related to hash2 computation and verification.

Before the CGA or signature verification, the verifying node must check that the routers public key in the CGA parameter option matches a certified public key from a router on the link. This ensures that the two keys used belong to two legitimate members of the group.

3.2.3.2 Authority delegation approach

Proxy neighbor discovery requires a delegation of authority on behalf of the absent address owner to the proxy. Without this authority, other devices on the link have no reason to trust an advertiser [20].

Authority delegation can be provided using two ways. First, routers that are certified by the routing authority using SEND can be also authorized and certified to proxy traffic for absent node on the link. Second, node which has CGA can sign the proxys public key and address so that it can be trusted. The certificate signed by the proxying node is passed to the proxy to use it as a proof of trust. It can be sent when sending the binding update message to the proxy (this message is used to indicate to the home agent that the node is absent). “In both methods, the original address owner’s advertisements need to override the proxy if it suddenly returns, and therefore timing and replay protection from such messages need to be carefully considered [20]”.

[2]For more information about multikey-CGA options and RST signature please refer to appendix C

Conclusion

In this chapter we have seen the two other possible usages of CGA in addition to its usage with SEND. In the following chapters we are going to study the performance of CGA based on its CPU consumption and robustness. We will also introduce a new improvement to CGA to make its generation faster.

Chapter 4

CGA security

Introduction

SEND was introduced to secure ND messages against different attacks but it seems to be itself vulnerable. In fact some attacks targeting the protocol SEND have been discovered. In addition, there are some steps in CGA generation and verification algorithms that could make this kind of address vulnerable to some attacks having as aim to make a DoS of a host or to realise a collision with the cryptographicly generated interface identifier.

In this chapter we start by discussing SEND robustness to attacks and then we expose CGA security level.

4.1 Attacks against SEND

In this section, we study some attacks that could be done against SEND.

4.1.1 ND DoS attack targeting routers

Attackers overload the router with packets for non existing addresses on the link, this kind of attack can be done using zombies like in distributed denial of service (DDoS). So the router becomes busy by answering NS for non existing addresses and is not available to legitimate nodes. A DoS is caused.

To counter this attack, clever cache management and limitation of the amount of state reserved to unresolved solicitations must be implemented in each router.

4.1.2 Replay attacks

While the timestamp delta value (defined in paragraph 5.3.4.2 of [2]) has not expired, replay attacks are still possible. In order to clarify this idea we present a successful replay attack against SEND, which is presented in [22].

The attack consists in listening all NS messages used in the DAD procedure, those messages are characterized by the use of the unspecified address as source address. They contain also the following four options: CGA, RSA signature, timestamp and nonce. To make this attack successful, the attacker only has to replay immediately the packets. In fact, they have a valid RSA signature, a correct timestamp and nonce is not verified in this case.

The timestamp is correct if the packet is replayed before the maximum timestamp delta value. DAD process lasts "DupAddrDetectTransmits*RetransTimer" milliseconds. The default values of DupAddrDetectTransmits and RetransTimer are respectively 1 and 1000. So to make the attack successful, the packet must be replayed during the next second:

Replay time < Sending time + 1 second

In this case, a target receiving a valid NS containing the address that it tried to acquire during the DAD, won't use it as address. So if the attacker replays the packets of every NS coming from the target in a DAD process, he will cause a DoS attack. One solution to counter this attack is to accept the address after three negative trials because the probability of three consecutive collisions is of the order of $1/10^{18}$ (for more details about the calculus of the probability see [22]).

4.1.3 Resource overloading targeting nodes

Some attacker tries causing a DoS attack by overloading the nodes. For example, sending to a host a large number of unnecessary CPA will make him do exhaustive computations, with exhaustive battery consumption. That will highly likely result in a denial of service. To counter this attack, we could define a limit of resources to be used when treating certificates. So when this limit is reached, packets are discarded.

4.2 CGA security

The purpose of CGAs is to prevent spoofing of existing IPv6 addresses by binding cryptographic information (the public key) to the address.

An attacker can create a new CGA using a subnet prefix and a public key (its own public key or a target one) but he will not have the possibility to use it, because as we mentioned it, CGA has been developed to be used with SEND which imposes to use CGA option with RSA signature option. So the attacker will have to sign the message that he is willing to send. To do a signature he will need a private key but it is very difficult to get the private key corresponding to a target public key especially when trying to find

a collision of the hash value hash1. In this section we discuss some CGA threats and possible vulnerabilities.

4.2.1 CGA parameters and valid addresses

For each valid CGA parameters data structure, there are 4×(SEC+1) different CGAs that match the value. In fact, the interface identifier, which has a length of 64 bits, has only 59 bits that have to be verified because we will not verify the 3 bits of SEC and the u and g bits. The action of decrementing the SEC value does not invalidate the address and u and g bits will give us four different values of valid CGAs corresponding to the same CGA structure. For example(Figure 4.1), if SEC = 2, we will verify that 32 leftmost bits of hash2 are equal to zero so indirectly we will verify also in this case that 16 leftmost bits of hash2 are equal to zero (this is the case where SEC = 1), and we validate also the case where SEC = 0. If we add the four possible values of the interface identifier introduced by u and g for every SEC value, we will have a total of 4×(SEC+1) valid CGAs corresponding to the same CGA structure.

The number 1 in the formula 4×(SEC+**1**) represents the case where SEC is equal to 0.

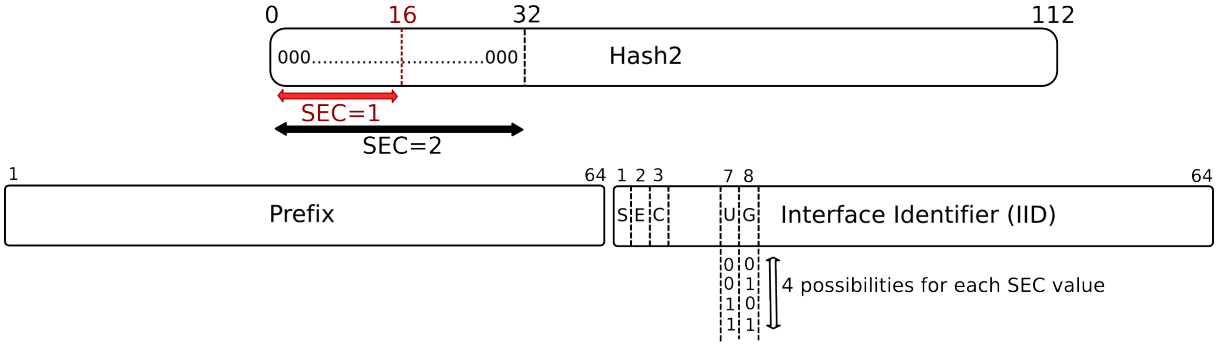

Figure 4.1: SEC, u and g bits influence on valid CGA creation.

4.2.2 CGA and signature option

CGA usage must be accompanied by the use of a signature option. In fact, CGA without a signature does not offer security. An attacker will be able to spoof other nodes addresses. It can also use a target's public key to create a new CGA and executes then the ND attacks described in section 1.1.4.

4.2.3 CGA and collisions

As computers become faster, an attempt to search a collision against the 64 bits of the interface identifier could be done by an attacker. But the use of the prefix field in the

CGA structure will help to prevent in some way this attack because the attacker will have to create different databases for each subnet prefix while attempting brute force attack.

Before talking about collision attacks we make a remind of the mathematical concept of $\bigcirc$. Let f and g be two functions taking as input a variable $n \in \mathbf{N}$, $f(n)=\bigcirc(g(n))$ means that it exists $n_0 \in \mathbf{N}$ and $C \in \mathbf{R}_+^*$ such that for every n superior to n_0 we have f(n) is inferior to C multiplied by g(n):

$f(n)=\bigcirc(g(n)) \iff \exists\ n_0 \in \mathbf{N}$ and $\exists\ C > 0$ and $n > n_0$ such that $0 \leq f(n) \leq C \times g(n)$

When we want to identify the complexity of an algorithm we try always to compare it to a mathematical functions using $\bigcirc$.

The cost of the brute force attack against the interface identifier generation algorithm depends on the identifier's length which is equal to 64 bits. Due to the fact that the 3 bits of SEC and the 2 bits u and g are not verified and have well known values the interface identifier generation cost is depending only on 59 bits. Considering the fact that every bit can have only two values 1 or 0, the brute force attack cost becomes $\bigcirc(2^{59})$.

The input to the second hash function (hash2) is modified (by changing the modifier value) until the leftmost 16×SEC bits of the hash value are zero. This increases the cost of brute force attack and the cost of the generation by a factor of $2^{(16\times SEC)}$. So the cost of creating a CGA parameters data structure that binds the attacker s public key with somebody else s address is increased from $\bigcirc(2^{59})$ to $\bigcirc(2^{(59+16\times SEC)})$.

The security parameter SEC decreases also the correlation rate between addresses created using the same initial CGA parameters structure. In fact the address created using SEC equal to 1 will have a correlation rate, relatively to the initial address which has a SEC equal to 0, greater than the one generated with SEC equal to 2. The next table gives an idea about correlation of addresses having the same parameters and inputs to CGA generation algorithm and only differs with SEC value.

SEC value	0	1	2
Correlation rate	1	0.508475	0.445763

Table 4.1: Correlation rate of addresses having same initial CGA parameters structure and different SEC values.

These values have been computed using as input to the CGA generation algorithm an RSA public key with 1024 bits length. We took 10 samples to make the average of the correlation rates. We remark that the correlation rate of a CGA having a SEC equal to 2 is inferior to the one relative to address with SEC equal to 1. So for the same initial CGA parameters structure the fact of trying a collision of two addresses by modifying only the SEC value is impossible and the collision rate of these addresses decreases in function when SEC value increases.

4.3 HBA/CGA security

The goal of HBAs (presented in 3.1) is to create a group of addresses that are securely bound, so that they can be used interchangeably when communicating with a node. If there is no secure binding between the different addresses of a node, many attacks can be executed against HBA such as redirecting the communications of a victim to an address selected by the attacker.

When using HBAs, a node using address A can redirect the communication to a new address B if and only if B belongs to the HBA set of A. If an attacker wants to redirect a communication addressed to HBA1 to an address IP-A, he has to create a CGA parameters data structure that generates a HBA set containing both HBA1 and IP-A. In order to generate the required HBA set, the attacker has to find a CGA parameters structure whose multi-prefix extension contains HBA1 and IP-A. So to realise this, the attacker needs generating brute force attack which implies the generation of multiple HBA sets with different parameters (for instance with different modifiers).

Thanks to the use of CGA, the cost of generating HBA has the order of $\bigcirc(2^{59+16\times SEC})$.

HBA can not prevent man in the middle attacks because an attacker may change addresses used in the communication by adding or removing prefixes. The attacker has to make sure that CGA parameters structure and the HBA set are changing accordingly.

Conclusion

In this chapter we presented some security aspects of the CGA. The attacks against CGA such as brute force attacks still theortical and have not been implemented yet which encourage the use of CGA in some constrained networks such as mobile networks. But when we speak about constrained networks we speak always about time. So in the next chapter we are going to study the CGA time performance.

Chapter 5

CGA performance and improvement

Introduction

In this chapter we start by presenting the tools that have been used and programs that have been developped to evaluate the CGA time performance. Then we introduce the different tests that were realised and the results that we got. In the end we present the improvements that we made in the CGA generation algorithm and SEND protocol by replacing the RSA keys and signature by an elliptic curve cryptographic (ECC) key pair and elliptic curve digital signature algorithm (ECDSA).

5.1 Algorithms

In this section, we present all the tools that we used and the different algorithms that we developped.

5.1.1 Tools presentation

While programming the CGA generation algorithm we used some interesting tools as OpenSSL library which contains all the cryptographic functions that we need as hash and signature functions. In addition we used Maemo which is an operating system for the Nokia Internet Tablet line of handheld computers. It was originally named "Internet Tablet OS". Maemo was used when adapting CGA generation algorithm to Tablet PC.

5.1.1.1 OpenSSL

The OpenSSL is an open source toolkit implementing the Secure Sockets Layer (SSL v2/v3) and Transport Layer Security (TLS v1) protocols as well as a full-strength general purpose cryptography library[23].

The version of OpenSSL that we used is 0.9.8e. We used the functions that:

- seed the pseudo random generator because it is a necessary step before an RSA key generation.
- generate and check an RSA or ECC key pair.
- generate and verify an RSA or ECDSA signature.
- generate an error message.

5.1.1.2 Maemo

The development environment for maemo running on the desktop is called maemo SDK [24]. It is only installed and run on a GNU/Linux operating system. Supported GNU/Linux distributions for maemo SDK are currently Debian and Ubuntu, but installing maemo SDK is also possible for other distributions. On other operating systems such as Windows, a VMWare image can be used to provide a working GNU/Linux environment. The maemo SDK creates a sandboxed maemo development environment on a GNU/Linux desktop system largely built on a tool called Scratchbox [25]. In most ways, this environment behaves like the operating system on the device, but with added development tools. This means that the development process is very similar to a normal desktop GNU/Linux, and the kinds of embedded development, such as cross-compiling, are handled transparently by Scratchbox.

Scratchbox is a specially packaged environment, providing the necessary tools and also isolating the development efforts from the real GNU/Linux system. Scratchbox also makes it easy to perform cross-compiling, which means building the software into a binary format that is executable in the target device.

While working inside Scratchbox, programs will be running in a changed root environment (chroot). In GNU/Linux systems, it is possible to change the part of file paths that a process will see. Scratchbox uses this mechanism on start to switch its root directory (/) to something else than the real root. This is part of the isolation technique used. Because of this, the environment is called a sandbox.

5.1.2 CGA generation and verification algorithms

We have developped the CGA generation algorithm using the C language. The generation or verification process is done using the same command but different options.

The program is composed of the following files:

- **cga.c** contains all the steps of the CGA generation algorithm and the main function.
- **cgah.c** contains all the functions[1] used in cga.c such that: RSA or ECC key generation, final modifier calculus and the CGA verification function.

[1]The list of functions that have been developped during this project is given in appendix E

- **cgah.h** is the headers file which contains all the prototypes of the previous functions instead of all the defined structures and all imported libraries.
- **hash.c** contains all the hashing functions (SHA-1, SHA-256, RIPEMD and TIGER).
- **hash.h** contains the prototypes of the hashing functions.
- **tiger.c** contains TIGER hash algortihm because it is not included in OpenSSL library.
- **tiger.h** contains TIGER algorithm prototypes.
- **sboxes.c** is used by the tiger.c source code.
- **makefile** is the installation file because it contains the compilation command *gcc* with its different arguments like *lssl, Wall...*

The command name is *cga* and it has the following options:

- **-a or --algorithm** is used to indicate which algorithm to employ when creating the key pair which is going to be used in the CGA generation process.
- **-s or --sec** is used to enter the choosen SEC value.
- **-p or --prefix** contains the prefix to be used in the address (we can enter full address and the program will take only the prefix).
- **-k or --key** indicates the key length in bits. It is equal to the modulus length when RSA is being used and to the name of the elliptic curve when ECC is used.
- **-h or --hash** is used to indicate which hash algorithm to use when computing hash1 and hash2 values. We have the choice between:RIPEMD, SHA-1, SHA-256 and TIGER [26].
- **-r or --rsa_public_key** gives the exponent to be used to compute the RSA public key.
- **-g or --generate** does not need an argument. It indicates that we are going to generate a CGA.
- **-v or --verify** does not need an argument. It indicates that we are going to verify a CGA.
- **-l or --list_curves** is used to list the possible elliptic curves that can be used and their correspondant number (which is their name).

- **-L or --List_hash_algo** lists the possible hash algorithms that could be used to compute hash 1 and 2.
- **-H or --help** explains all the options.

When using the generation command the result will be stored in three different files:

- The first file contains the created CGA.
- The second contains the CGA and its parameters structure in binary format.
- The third one contains the key used to create the CGA.

When we want to generate a CGA using an RSA public key having an exponent equal to 3, a modulus length equal to 1024, a SEC value equal to 1, abcd:: as prefix and SHA-256 as a hashing algorithm, we use the following command:
./cga -g -a rsa -k 1024 -r 3 -s 1 -h sha256 -p abcd::
To verify it, we use: *./cga -v -h sha256* and the program is asking for the name of the file where we have stored the CGA and its parameters structure then it is executing the verification algorithm over this structure using the stored CGA.

5.1.3 Test algorithms

We have changed the general algorithm presented in the previous section specially for the tests. We added some functions corresponding to RSA or ECDSA signature generation and verification because we need them in our tests. In addition, we have fixed the value of some variables such that the public key exponent to 3 or 2^{16}-1. We have also programmed a CGA generation algorithm corresponding to every hash function (SHA-1, SHA256, TIGER, RIPEMD) able to be used in the generation process. We have also integrated the CGA verification function in the same algorithm with the generation one. So our test algorithm is going to perform the following tasks:

- Create an RSA or ECC key.
- Choose a random value for the modifier and computes its final value as defined by the CGA generation algorithm.
- Create the CGA.
- Generate a RSA or an ECDSA signature.
- Verify the created CGA.
- Verify the signature.

The previous algorithm can be launched for example by the following command:
./cga -a rsa -k 1024 -s 0 -p abcd::

5.2 Tests and results

In this section we are going to discuss the results that we have found when testing CGA generation using RSA key. subsectionTesting context To evaluate the CGA generation and verification time, we have used an assembly code that evaluates the CPU usage time of a function. Processor time is different from actual wall clock time because it does not include any time spent waiting for Input/Output or when some other process is running.

Let us suppose that we are going to evaluate a function duration. The assembly code gets the number of ticks before the function is starting and gets the number of ticks at the end of the function from the processor internal counter (a tick is a system clock pulse). The latter is incremented at every clock pulse since the system is booted. Then we calculate the number of ticks relative to this function by computing the difference of the two previously returned values. Finally we devide the function's ticks number by the processor frequency that we get from the *cpuinfo* under *proc* directory. The assembly mnemonic used to get the ticks counter value is RDTSC: Read Time-Stamp Counter.

To make the CPU usage time evaluation the most accurate as possible, we start the test in what is called "single mode". "Single mode"helps avoiding that CPU time consumed by some other (interrupting) programs is taken into account into the measurement.

All the next presented results are the average of ten thousands samples.

5.2.1 Computer results

Now we are going to present the different results that we found. In all this chapter, when we speak about RSA key length, we are talking about the modulus key length.

5.2.1.1 CGA generation time when SEC = 0 or 1

In this paragraph, we discuss the different generation results related to CGA generation when the CGA security value SEC is equal to 0 or 1. The tests have been done in a Pentium 4 with a 2593 MHz clock frequency.

SEC value	0			
RSA key length (bits)	1024	2048	3072	7680
CGA generation time (sec)	0.163964	1.055813	3.457668	92.610627
SEC value	1			
RSA key length (bits)	1024	2048	3072	7680
CGA generation time (sec)	0.281018	1.194061	3.601473	92.899951

Table 5.1: CGA generation time using RSA key (in seconds).

The CGA generation time increases in function of the SEC value. It is due to the fact that for SEC values different from zero, we need to compute the final modifier value. In

fact, in chapter two, when we detailed the CGA generation algorithm, we said that the final modifier value is computed by incrementing the initial modifier, which is a 128 bits random number, by one and computing hash2 over the CGA parameters structure until finding that the 16 × SEC leftmosts bits of hash2 are equal to zero. So when SEC is equal to 1, we computed the final modifier value by incrementing the initial one by one until verifying that the 16 leftmosts bits of hash2 are equal to zero.

In Table 5.2 we present the CPU consumption time needed to generate an RSA key. Including the key generation time evaluation helps in studying the possibility of creating a new key pair every time a new CGA is going to be generated. In fact CGA specification [3] does not suppose that we create a new key when generating a new address. However creating a new key when generating a new address helps in protection against brute force and collision attacks where an attacker aims to get the same target's interface identifier. The attacker tries to find a collision on the target key pair using CGA parameters structure or to find the target private key through brute force attack. Generating a new key pair increases the complexity of those attacks especially when we suppose also that CGA are generated periodically.

SEC value	0			
RSA key length (bits)	1024	2048	3072	7680
RSA key generation time (sec)	0.163959	1.055806	3.457661	92.610627
SEC value	1			
RSA key length (bits)	1024	2048	3072	7680
RSA key generation time (sec)	0.163784	1.034586	3.398753	92.511586

Table 5.2: RSA key generation time (in seconds).

Here we remark only that the key generation time increases with the key length this is logical referring to RSA key generation algorithm.

We remark also that we presented two RSA key generation time, one for SEC equal to 0 and one for SEC equal to 1. It is due to the fact that we did the test with SEC equal to 0 before those with SEC equal to 1. We had the possibility to use the same generated key to compute a CGA one time with SEC equal to 0 and the other time with SEC equal to 1. The interest of doing every test separately is to verify that the average value over the ten thousands samples will be around the same value. It is a way to verify that the generation times that we got are valid.

To compare the key generation time and the address generation time presented in Table 5.1 in the case of sec equal to 0, we can use Table 5.3.

We can deduce that CGA generation time is very similar to the RSA key generation time when SEC is equal to 0. In fact, the global CGA generation time is equal to the sum of: RSA key generation time, final modifier generation time (which is null in our case because SEC is equal to 0 and the hash1 computation time). So in this case the difference

RSA key length (bits)	1024	2048	3072	7680
CGA generation time (sec)	0.163964	1.055813	3.457668	92.610627
RSA key generation time (sec)	0.163959	1.055806	3.457661	92.610616

Table 5.3: Comparison between CGA generation and RSA key generation times when SEC = 0 (given in seconds).

between the CGA generation time and the RSA key generation time will be the hash1 computation time. The table 5.4 confirms the result when SEC is equal to 1.

RSA key length (bits)	1024	2048	3072	7680
Final modifier generation time (sec)[row 2]	0.117230	0.159470	0.202715	0.388356
RSA key generation time (sec)[row 3]	0.163784	1.034586	3.398753	92.511586
row 2 + row 3 (sec)	0.281014	1.194056	3.601468	92.899942
CGA generation time (sec)	0.281018	1.194061	3.601473	92.899951

Table 5.4: Final modifier generation time influence in CGA generation time when SEC = 1(in seconds).

We deduce also from Table 5.4 that the final modifier generation time increases with the key length.

As a general conclusion to the previous tables, the CGA generation time is almost equal to the time of the key generation and the final modifier value generation when SEC value is less than 2.

5.2.1.2 CGA verification

CGA verification time is very low as depicted in Table 5.5.

SEC value	0			
RSA key length (bits)	1024	2048	3072	7680
RSA key generation time (sec)	0.000004	0.000005	0.000005	0.000009
SEC value	1			
RSA key length (bits)	1024	2048	3072	7680
RSA key generation time (sec)	0.000007	0.000008	0.000009	0.000016

Table 5.5: CGA verification time (in seconds).

We remark that CGA verification time increases with key length and SEC values. In fact CGA verification is composed of two hashs (1 and 2) computation over CGA parameters structure. When the length of parameters structure increases (with key length or modifier usage), the hash generation time increases too.

5.2.1.3 CGA generation time when SEC = 2

In Table 5.6 we present 10 samples of a CGA generation time when SEC is equal to 2. The tests have been done in a pentium 4 computer having a 2992.697 MHz CPU frequency.

	CGA generation time (sec)		Final modifier generation time (sec)	
Key length (bits)	1024	2048	1024	2048
Test 1	9865.558780	22621.214970	9865.460238	22620.832288
Test 2	1320.573976	663.083762	1320.510563	662.108090
Test 3	1677.093357	6733.043070	1677.975211	6732.396501
Test 4	5484.136953	14405.533876	5484.024933	14404.837514
Test 5	1596.026161	7531.830884	1596.891070	7530.149370
Test 6	10603.389679	6035.378657	10603.245553	6034.884386
Test 7	27795.709438	5271.830884	27795.534816	5270.497470
Test 8	365.078389	20920.210429	365.973911	20919.778124
Test 9	7506.891760	23172.781460	7506.88426	23171.981597
Test 10	1785.414634	2302.037400	1785.328066	2301.223732

Key length (bits)	3072	7680	3072	7680
Test 1	8630.801918	8284.639799	8629.184128	8238.988784
Test 2	1760.865971	519.085431	1759.786742	305.810149
Test 3	25472.998112	20702.312855	25472.038909	20609.590118
Test 4	3556.567461	5424.954255	3552.404819	5384.277676
Test 5	14877.220154	3186.761698	14876.418296	3001.611502
Test 6	2573.097311	13630.129774	2570.434226	13527.742317
Test 7	13345.505099	13557.981281	13344.499211	13557.981281
Test 8	4319.385108	6570.039450	4316.371351	6540.853194
Test 9	8943.805630	1598.470785	8945.085357	1496.811630
Test 10	5391.707009	24259.096903	5388.195878	23969.804519

Table 5.6: CGA generation time variation when SEC = 2 (in seconds).

In those tests, we have not used the tick counter to evaluate functions duration because we could not find a large enough variable to store the counter's ticks number and the use of classical variable causes a memory overflow. So we have not evaluated a CPU time but we evaluated the real generation time using the *time* function of the *time C library*. The real generation time is greater than the CPU time by some seconds (generally 1 or 2).

We remark in this case that the CGA generation time is almost equal to the final modifier generation time. In cases where the RSA key length becomes greater than 7680, it becomes almost equal to the modifier generation duration and the key computation time. The difference between the generation time and the final modifier generation time is equal to the key generation time and hash1 computation time.

It is clear that CGA generation with SEC equal to 2 takes a great amount of time. That is why actually we only did 10 tests samples and we are only next studying the cases where SEC is equal to 0 or 1. The cases where SEC are equal or greater to 2

depends on computer performance and could become interesting with the next generation of processors.

5.2.1.4 RSA signature generation and verification

In this section we present the results concerning the RSA signature generation and verification. We supposed that we are going to create the RSA signature option relative to a NS message. So we randomply generated a message of same length than NS messages provided in section 1.2.2.2 (RSA signature option).

The tests have been done in a pentium 4 computer having a 2593.685 MHz CPU frequency.

RSA key length (bits)	1024	2048	3072	7680
RSA signature generation time (sec)	0.004585	0.022217	0.053573	0.609111
RSA signature verification time (sec)	0.000070	0.000167	0.000322	0.001425

Table 5.7: RSA signature generation and verification time (in seconds).

RSA signature generation time increases with the key length and its totally logical due to the fact that the signature calculus depends on the modulus length.

5.2.1.5 Hash function impact on CGA generation time

We have tried in this test to change the hash function SHA-1 used in CGA generation algorithm with other hash functions like: SHA-256, RIPEMD and TIGER. We evaluated the final modifier generation CPU time when generating a CGA with a SEC equal to 1. This time is mostly influenced by hash functions because it depends on hash2 and SEC values.

The tests have been done in a pentium 4 computer having a 2400.005 MHz CPU frequency.

Hash function / RSA key length (bits)	SHA-1	SHA-256	RIPEMD	TIGER
1024	0.121706	0.495153	0.215673	0.270183
2048	0.167530	0.790370	0.312232	0.429265
3072	0.213916	1.029315	0.417027	0.589410

Table 5.8: Hash function influence on final modifier generation time (sec).

We remark that the most efficient algorithm concerning the modifier generation time is SHA-1. RIPEMD presents also interesting values. It is also clear that the modifier generation time depends on the key length as the hash is computed over the key.

5.2.2 Tablet PC results

The Tablet PC that we used is a Nokia N800 with an ARMv6-compatible processor. The N800 is not a phone, more a Internet Tablet that allows the user to browse the Internet and communicate using Wi-Fi networks or with mobile phone via Bluetooth. It was developed as the successor to the Nokia 770. It includes FM and Internet radio, an RSS news reader, image viewer and a media player for audio and video files. It has 388.54 MHz as CPU frequency.

We used the function *gettimeofday* from *time library* to compute the generation times because we could not use the assembly code to get the CPU time. It computes the global generation time (including CPU time consumption of other background proccessus). We compiled the CGA generation algorithm in the scratchbox using the *static* argument for *gcc* compiler because we have not OpenSSL installed in the Tablet PC.

The Table 5.9 presents the CGA generation time, the final modifier computation time and the key generation time.

SEC value	0			
RSA key length (bits)	384	512	1024	2048
CGA generation time (sec)	0.473715	0.694189	2.902132	18.004494
RSA key generation time (sec)	0.473634	0.694106	2.902089	18.004389
Final modifier generation time (sec)	0	0	0	0
SEC value	1			
RSA key length (bits)	384	512	1024	2048
CGA generation time (sec)	1.586903	1.739357	4.379279	19.910244
RSA key generation time (sec)	0.472734	0.688536	2.954937	18.017241
Final modifier generation time (sec)	1.114100	1.050757	1.424263	1.892901

Table 5.9: CGA generation time computed on a Nokia N800.

We remark in this case that the CGA generation time is greater than the generation time found on a Pentium 4. It is due to the lack of computation ressources on the tablet PC. We conclude also that actually it is not interesting to use CGA in PDA and Tablets PC.

We remark also from the previous Figure that the final modifier generation time is greater when being calculated on an N800. It is 10 times greater than a final modifier computation on a computer. It is in direct relation with the processor's capabilities.

5.3 CGA improvement with ECC

In this section we discuss the solution that we proposed to enhance the CGA generation algorithm performance and which consists in using ECC. First we are giving some introduction to ECC and ECDSA. Then we introduce the modification of the SEND protocol

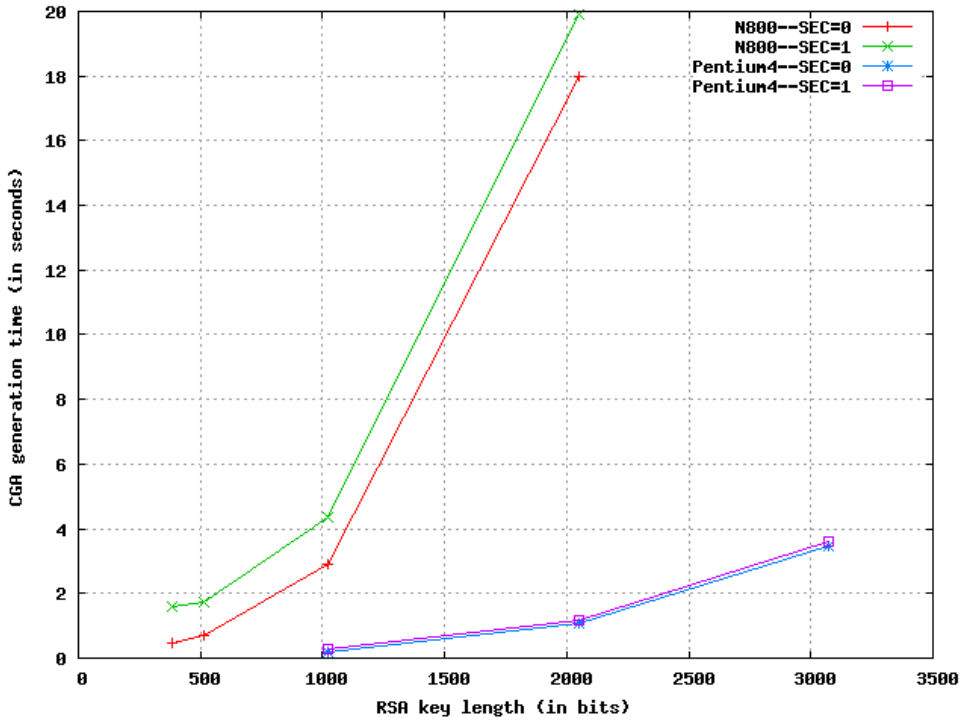

Figure 5.1: Processor influence on CGA generation time.

to support our solution. Finally we present the generation time results that we found using ECC.

5.3.1 ECC and ECDSA

Elliptic curves (EC) are defined over a finite field, generally a prime or a binary field designed by $\mathbf{F_p}$ or $\mathbf{F_{2^p}}$ where p and 2^p represent the number of elements of the field[2].

An EC is defined by Weirstrass equation[27] over a finite field **F**:

$$E: y^2+a_1xy+a_3y = x^3+a_2x^2+a_4x+a_6, \text{ where } a_1,a_2,a_3,a_4,a_6 \in \mathbf{F}$$

The EC group is an abelian group having an addition operation as binary operation and ∞ as identity element. The EC points addition is easier to understand graphically and is defined as follows: let P and Q be two distincts points belonging to an EC named E defined over a field **F**. The sum R' of P and Q is obtained by drawing a line through P and Q which will intercept E in a third point R. R' is the reflection of R relatively to X-axis.

ECC enabled devices require less storage, less power, less memory, and less bandwidth than other systems. This permits implementation of cryptography in platforms that are constrained, such as wireless devices, handheld computers, smart cards, and thin-clients. It also provides a big win in situations where efficiency is important.

[2]for more information about fields theory please refer to appendix D

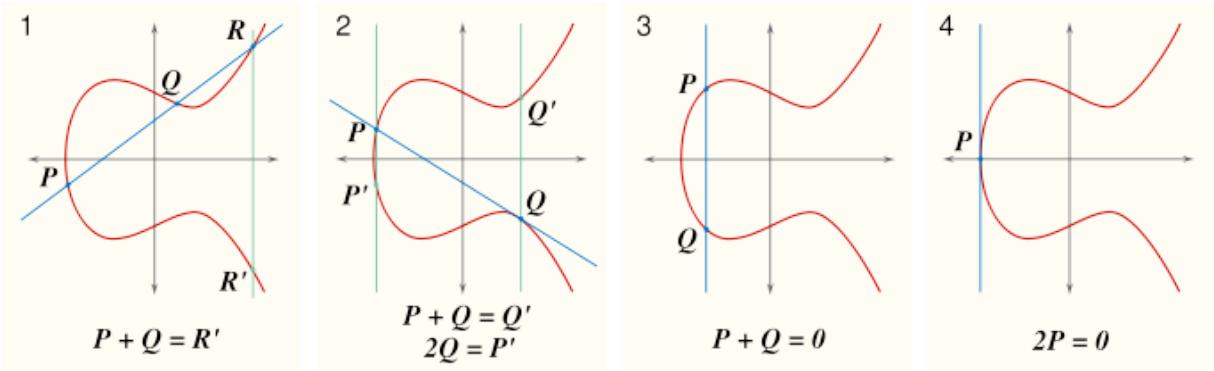

Figure 5.2: Elliptic curve points addition.

For example, the current key-size recommendation for legacy public schemes is 2048 bits. A vastly smaller 224 bits ECC key offers the same level of security. This advantage only increases with security level. For example, a 3072 bits legacy key and 256 bits ECC key are equivalent, something that will be important as stronger security systems become mandated and devices get smaller.

The Table 5.10 presents the equivalence between RSA and ECC key length in terms of security level.

RSA key length (bits)	ECC key length (bits)
1024	163
2048	224
3072	256
7680	384
15360	512

Table 5.10: RSA and ECC key length equivalence in security level.

5.3.1.1 ECC key generation

Let E be an elliptic curve defined over a finite field **F**. Let P be a point in E, and suppose that P has prime order n. Then the cyclic subgroup of E generated by P is: $\langle P \rangle = \{ \infty, P,2P,3P, \ldots,(n-1)P\}$.

The equation of the elliptic curve E, and the point P and its order n, are the public domain parameters. A private key is an integer d that is selected uniformly at random from the interval [1, n-1], and the corresponding public key is Q = dP.

The problem of determining d given the domain parameters P and Q is the elliptic curve discrete logarithm problem (ECDLP).

5.3.1.2 ECDSA generation and verification

ECDSA is the elliptic curve analogue of the Digital Signature Algorithm (DSA). The signature is done as following:

1. select k $\in$ [1,n-1]
2. compute kP=(x_1,y_1) and convert x_1 to an integer x_2
3. compute r = x_2 mod n. If r = 0 then go to step 1
4. compute e = h(m)
5. compute s = k^{-1}(e+dr) mod n. If s = 0 then go to step 1
6. form the signature (r,s)

The signature verification inputs are: the signed message m, the hash function h, the public key Q, the signature (r,s), the point P and its order n. Its steps are:

1. verify that r and s are integers in the interval [1, n-1]
2. compute e = h(m)
3. compute w = s^{-1} mod n
4. compute u_1 = ew mod n and u_2 = rw mod n
5. compute X = u_1P + u_2Q
6. if X = ∞ reject the signature
7. convert the x-coordinate x_1 of X to an integer x_2
8. compute v = x_2 mod n
9. if v = r then accept the signature else reject it

The proof that signature verification works is: if a signature (r,s) on a message m was indeed generated by the legitimate signer, then:
s = k^{-1}(e+dr) mod n. Rearranging gives:
k = s^{-1}(e+dr) = s^{-1}e+s^{-1}rd = we+wrd = u_1 +u_2d mod n
Thus X = u_1P +u_2Q = (u_1 +u_2d)P = kP, and so v = r as required.

5.3.2 SEND options adaptation to ECDSA

In order to use ECC and ECDSA with the protocol SEND we have proposed:

- to use one of the three bits of SEC to indicate to the remote peer whether RSA or ECC are used as cryptographic algorithm for key generation and signature generation and verification. Thanks to the fact that only the three first value (0,1 and 2) of SEC are actually used because of the complexity of the final modifier computation if SEC is greater than 2, we can use the leftmost bit of SEC as an algorithm identifier. For example when it is equal to 0, we use RSA key pair and signature and in the case it is equal to 1 we use ECC key pair and ECDSA.

SEC value	Meaning
000	RSA key pair and RSA signature algorithm are used and SEC=0
001	RSA key pair and RSA signature algorithm are used and SEC=1
010	RSA key pair and RSA signature algorithm are used and SEC=2
100	ECC key pair and ECDSA signature algorithm are used and SEC=0
101	ECC key pair and ECDSA signature algorithm are used and SEC=1
110	ECC key pair and ECDSA signature algorithm are used and SEC=2

Table 5.11: SEC bits usage to indicate cryptographic algorithm choice.

- to replace the RSA DER encoded key by an ECC octets encoded key. This encoding is done by the OpenSSL library using the *i2o_ECPublicKey* function. It converts integer to octets string. In OpenSSL library we have not found a DER encoding function for ECC public keys. So we used the function that was defined as equivalent to *i2d_RSAPublicKey* in the case where a RSA public key is used.

- to keep the same RSA signature option presented in 1.2.2.2. In fact, the signature option contains the signature to be verified. This verification depends on the public key presented in the CGA parameters structure. This public key is verified during the CGA verification which is always done before the signature verification. During the CGA verification we can guess which signature algorithm is used if we use the previous approach consisting in associating the leftmost SEC value to the algorithm type. In our case we have two algorithms so one bit is enough but if we want to use a third algorithm, we will have to use a second bit of the interface identifier.

5.3.3 Computer results

In this section, we present the different results that we found when using the ECC while generating a CGA. We used the following EC key lengths: 163, 224, 256, 384 and 571. We used 571 instead of 512 bits because we did not find any implementation of ECC on

the OpenSSL library with a key length equal to 512 bits. So we used a key length of 571 bits which offers better security than 512 bits.

5.3.3.1 CGA generation time when SEC = 0 or 1

All the following tests have been done in a pentium 4 computer having a 2593.685 MHz CPU frequency. The Table 5.12 presents the different values of ECC CPU generation time when SEC is equal to 0 or 1.

SEC value	0				
ECC key length (bits)	163	224	256	384	571
CGA generation time (sec)	0.006449	0.012602	0.012622	0.020802	0.111246
SEC value	1				
ECC key length (bits)	163	224	256	384	571
CGA generation time (sec)	0.096317	0.108551	0.106154	0.135056	0.224526

Table 5.12: CGA generation time using ECC key (in seconds).

We remark that the generation time increases with the key length. It depends on the key generation CPU time.

In the case where SEC is equal to 1, we find that the average generation time for a 256 bits key is less than to the one relative to a 224 bits. It is due to the fact that the final modifier is generated using random information. If we make the difference between the 2 generation times we find 0.108551-0.106154=0.002397 secondes. We multiply this results by 10000 (because we used 10000 samples when computing the average), we find 23,97 secondes. So we need only that the sum of all the 10000 samples generation time concerning the 224 bits key exceeds the sum of all the 10000 samples generation time concerning the 256 bits key by 23,97 secondes to find the result. It is due to 2 reasons:the first one is that the final modifier generation time depends on the randomness of the modifier itself and the second one is that the two key lengths are very close so the generation time will be close too.

In all the next Figures, we use the notation:

Symbol	RSA key length (bits)	ECC key length (bits)
1	1024	163
2	2048	224
3	3072	256
4	7680	384
5	15360	571

Table 5.13: RSA and ECC key length equivalence.

For example a RSA key length of 2048 bits is represented on the x axis by the value 2 which represents too an ECC key length of 224 bits.

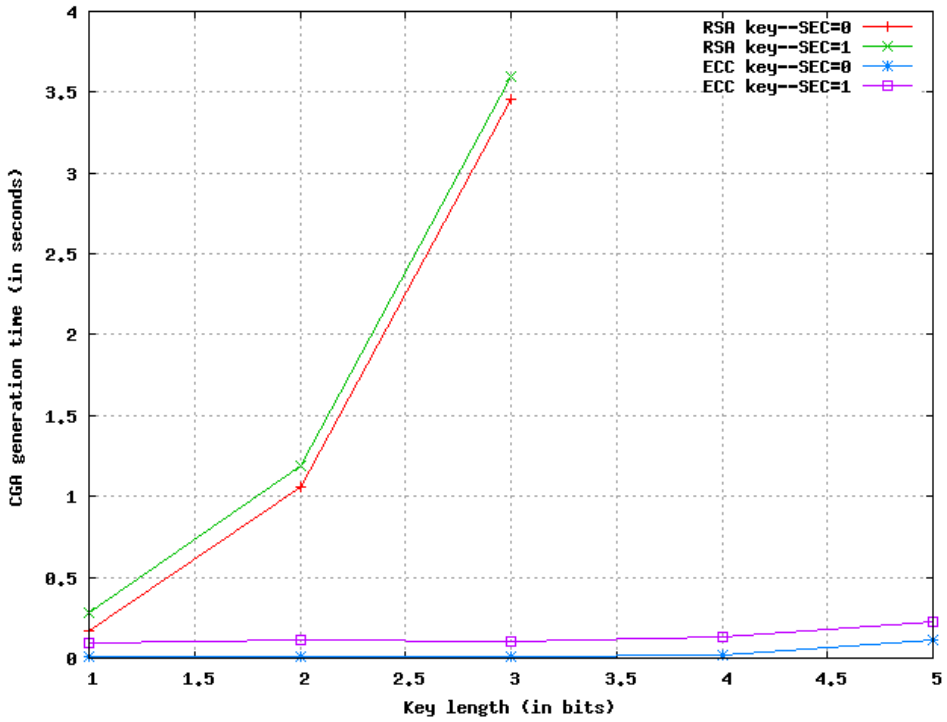

Figure 5.3: RSA and ECC CGA generation time comparison.

We remark that at the same level of security, CGA CPU generation time is more interesting when using ECC. It is 10 to 100 times inferior to the one relative to RSA key. The maximum generation time of the CGA based ECC (corresponding to a key length of 571 bits) is inferior to the minimum generation time of CGA based RSA (corresponding to a key length of 1024 bits).

Even for the final modifier generation time and the key generation time, we remark that the generation times are always lower in the case when ECC keys are used. The Table 5.14 contains the final modifier and the key generation times when ECC is used for SEC equal to 1.

ECC key length (bits)	163	224	256	384	571
Final modifier generation time (sec)	0.089857	0.096027	0.093472	0.114177	0.113482
ECC key generation time (sec)	0.006458	0.012521	0.012679	0.020876	0.113482

Table 5.14: Final modifier and key generation times when using ECC (in seconds).

We notice that the final modifier computation time is lower when ECC is used. This makes the ECC more interesting to replace RSA because for the same level of security

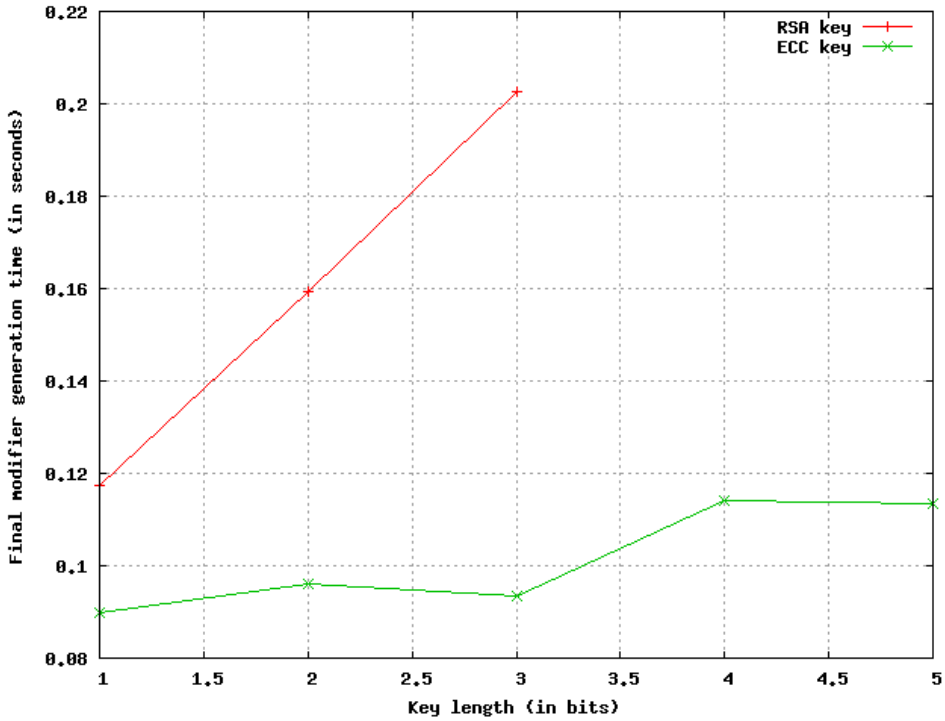

Figure 5.4: RSA and ECC final modifier generation time comparison.

we will generate CGA more quickly. ECC is faster because we use key having smaller length. In fact CGA generation is based on the use of hash function over the parameters structure which contains the public key. So longer the public key is, longer will be the CGA generation time.

5.3.3.2 CGA based ECC verification

The Table 5.15 presents CGA verification time results when being used with ECC.

SEC value	0				
ECC key length (bits)	163	224	256	384	571
ECC key generation time (sec)	0.000003	0.000004	0.000003	0.000004	0.000004
SEC value	1				
ECC key length (bits)	163	224	256	384	571
ECC key generation time (sec)	0.000004	0.000005	0.000005	0.000005	0.000007

Table 5.15: CGA based ECC verification time (in seconds).

In comparison to Table 5.5, we remark for the same level of security that CGA verification time is lower when the address is generated using ECC.

5.3.3.3 ECDSA signature generation and verification

We realised this experience keeping the hypothesis of section 5.2.2.3. The Table 5.16 resumes the results that we found:

ECC key length (bits)	163	224	256	384	571
ECDSA signature generation time (sec)	0.002231	0.004396	0.004460	0.007365	0.037403
ECDSA signature verification time (sec)	0.004379	0.005249	0.005352	0.008850	0.074777

Table 5.16: ECDSA signature generation and verification time.

ECDSA verification takes always more time than generation.

The Figure 5.5 shows that ECDSA is quicker than RSA when signing but not when verifying.

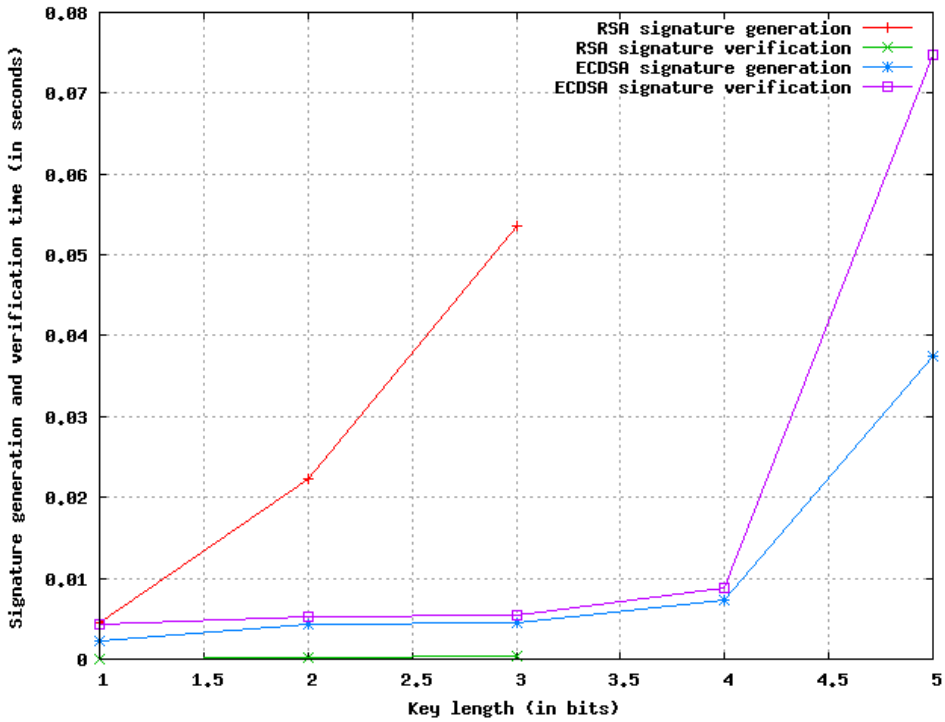

Figure 5.5: RSA and ECC signature generation and verification times comparison.

We notice that the ECDSA verification time is always higher than RSA one. But the difference between signature verification and generation for ECDSA and RSA makes ECDSA more interesting. Considering that difference we will gain more time than in the case when RSA is used.

5.3.4 Tablet PC results

The Table 5.17 contains the results found when testing the CGA generation algorithm on the N800. We have studied like for RSA the two cases where SEC is equal to 0 or 1.

SEC value	0				
ECDSA key length (bits)	163	224	256	384	571
CGA generation time (sec)	0.088265	0.137280	0.151946	0.302067	0.981322
ECDSA key generation time (sec)	0.088172	0.137182	0.151850	0.301966	0.981217
Final modifier generation time (sec)	0	0	0	0	0
SEC value	1				
ECDSA key length (bits)	162	224	256	384	571
CGA generation time (sec)	1.053433	1.179489	1.208950	1.634632	2.300124
ECDSA key generation time (sec)	0.087542	0.136879	0.151900	0.297508	0.980466
Final modifier generation time (sec)	0.965827	1.042549	1.056985	1.337056	1.319589

Table 5.17: CGA generation time computed on a Nokia N800.

We notice that even for CGA based on ECC, the generation time on a Tablet PC is still important. But at least we still can use ECC keys with SEC=0 to generate these addresses. In fact for keys having length of 163, 224 and 256 bits the generation time is inferior to 0.15 seconds which is interesting in comparison with the results found when using RSA. So for Mobile IPv6 we can propose the use of CGA with ECC keys.

In addition, in order to decrease the risk of collisions, we propose also to create periodically a CGA based on a newly generated key. The period must be well chosen to avoid that an attacker could find a collision or realize a brute force attack against the actual address. We propose also in mobile environment that a node should change its address after changing the access point to which it is connected.

For the values of SEC equal to 1 or 2, the node can use a temporary address having SEC equal to 0 until it finishes computing its true CGA with a higher SEC value.

We remark from the Figure 5.6 that even in Tablets PC, ECC is still more performant than RSA. For this reason, ECC cryptography must be taken in consideration in future SEND and CGA specification.

Conclusion

CGA have been defined to be used with RSA. But actually to offer an interesting level of security RSA key must have a length equal or greater to 1024 bits. This makes the use of CGA with RSA with constrained devices more difficult.

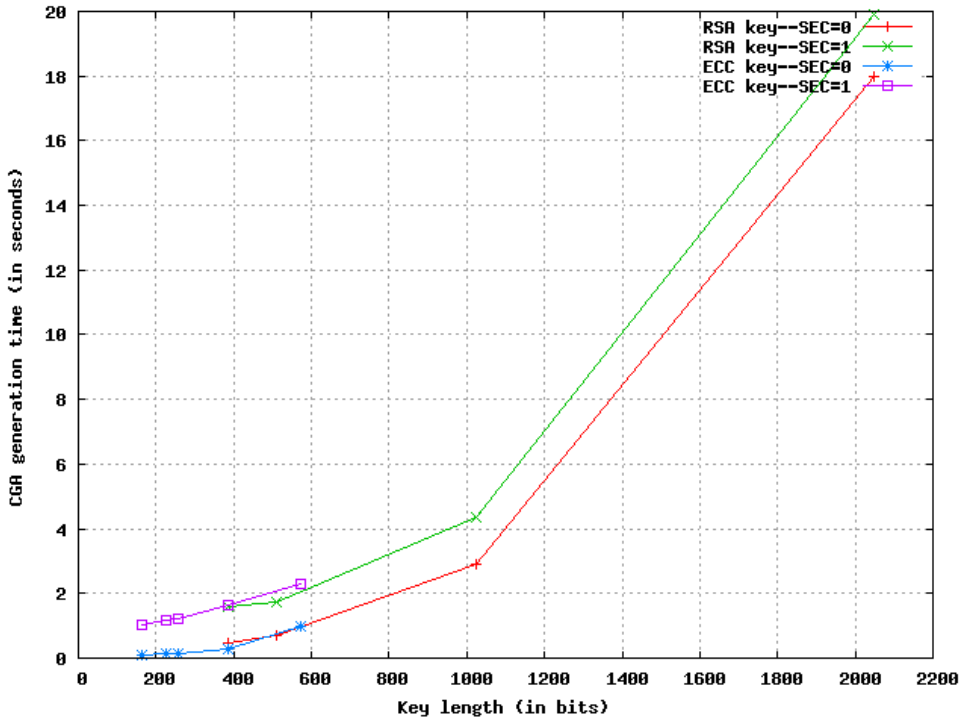

Figure 5.6: RSA and ECC CGA generation time comparison.

Thanks to the improvement we suggested with ECC usage, CGA performances have been enhanced. The number of experimental measurements we did prove that the usage of ECC is an important breakthrough for introduction of CGA into constrained devices like PDA and Tablets PC. In addition it increases CGA algorithms efficiency in wired networks.

Conclusions

CGA represents a good solution to address authentication in multi-homing and mobile networks. Thanks to the use of SEND options, secure proxying become easier. In addition, the use of Hash Based Addresses (HBA) with SHIM6 protocol become more secure because CGA solved all authentication problems.

The proposed CGA generation tests showed that in wired networks, it is very interesting to use both RSA and ECC keys when choosing SEC value equal to 0. However, when SEC equals to 1, we found that ECC key pairs are more performing than RSA keys. We also concluded that CGA generation depends on the processor performance. New methods of address computation must be introduced to make the generation process faster and to enhance the use of higher SEC values because a collision in CGA generation when SEC is equal to 0 or 1 is not computationally difficult. One way to avoid temporarily collision threats is to create periodically a new CGA based on a new public key. In this case, the use of ECC will be more interesting because a CGA with a 512 bits ECC key is generated at the same time as a CGA with a 1024 bits RSA key but offers a higher level of security.

The experiments conducted on Tablet PC results showed that the use of CGA in wireless networks is still difficult. This is due to the fact that CGA generation time exceeded one second which is very constraining in this kind of networks where time is very expensive. In our tests with Nokia N800, we remarked that the use of CGA is possible only with SEC equal to 0 with ECC keys having a length less than 256 bits. So, this makes the use of CGA currently impossible in mobile networks due to the scarcity of computational resources.

Currently, CGAs are used only with SEND. There is no specification talking about the use of these addresses with another protocol. SEND is the only protocol that defined the needed options to secure ND and to use CGA as a decentralized authentication mechanism. Even the use of CGA with SHIM6 protocol depends on SEND options. But it will be interesting to see other protocols using this kind of address based on their own specifications and options. With the spread of IPv6 usage, it become imperative to provide security mechanism but a number of questions should first be investigated: (a) will CGA usage be the best one? (b) are CGA going to be the best crypto-based identifier? (c) will CGA generation time and usage in mobile networks be improved?

Appendix A

IPv6

The length of network addresses emphasize a most important change when moving from IPv4 to IPv6. IPv6 addresses are 128 bits long, whereas IPv4 addresses are 32 bits; where the IPv4 address space contains roughly 4 billion addresses, IPv6 has enough room for 3.4×10^{38} unique addresses.

IPv6 addresses are typically composed of two logical parts: a 64-bit network prefix, and a 64-bit interface identifier, which is either automatically generated from the interface's MAC address or assigned sequentially.

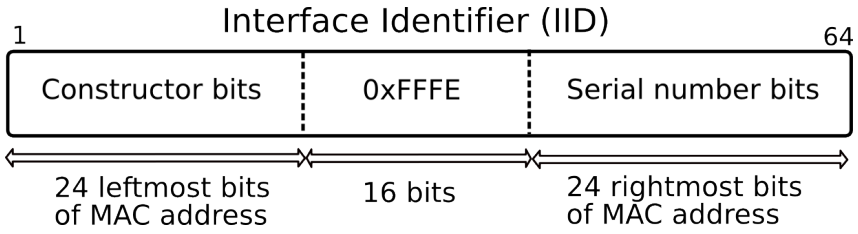

Figure A.1: Interface identifier creation using MAC address.

IPv6 addresses are classified into three types [28]:

- **IPv6 Unicast addresses**: They are similar to the unicast addresses in IPv4: a single address identifying a single interface. There are four types of unicast addresses:
  - Global unicast addresses, which are conventional, publicly routable address, just like conventional IPv4 publicly routable addresses.
  - Link-local addresses are akin to the private, non-routable addresses in IPv4 (10.0.0.0/8, 172.16.0.0/12, 192.168.0.0/16). They are not meant to be routed, but confined to a single network segment.
  - Unique local addresses are also meant for private addressing, with the addition of being unique, so that joining two subnets does not cause address collisions.

  - Special addresses are loopback addresses, IPv4-address mapped spaces, and 6-to-4 addresses for crossing from an IPv4 network to an IPv6 network.

- **Multicast addresses**: Multicast in IPv6 is similar to the old IPv4 broadcast address: a packet sent to a multicast address is delivered to every interface in a group. The IPv6 difference is it's targeted instead of annoying every single host on the segment with broadcast blather, only hosts who are members of the multicast group receive the multicast packets. IPv6 multicast addresses are routable.

Address	Application	Description
FF01::1	all nodes	node local hosts group
FF02::1	all nodes	link local hosts group
FF01::2	all routers	node local routers group
FF02::2	all routers	link local routers group
FF02::1:0	DHCP server	link local DHCP server group
FF02::1:xxxx:xxxx	solicited node	address created using a node unicast address

Table A.1: IPv6 multicast addresses.

- **Anycast**: An anycast address is a single address assigned to multiple nodes. A packet sent to an anycast address is then delivered to the first available node.

Appendix B

RSA key generation, encryption and signature

We present in this appendix RSA cryptography basics. We start by reminding RSA key generation algorithm then we present RSA encryption and signature algorithms.

B.1 RSA key generation

RSA key generation follows the next steps:

1. Generate two large random (and distinct) primes p and q.
2. Compute n = pq and φ(n) = (p-1)(q-1).
3. Select a random integer e, $1 < e < \varphi(n)$, such that gcd(e, φ(n)) = 1.
4. Compute the unique integer d, $1 < d < \varphi(n)$, such that ed = 1 (mod φ(n)).

The public key is (n, e) and d is the private key.

B.2 RSA encryption and decryption

Let suppose that Bob is going to encrypt a message to Alice.

1. Bob represents the message as an integer m in the interval [0, n-1].
2. Computes $c = m^e \bmod n$.
3. Sends the ciphertext c to Alice.

To decrypt the message Alice computes:
$c^d \bmod n = m^{ed} \bmod n = m^{(1+k(n-1))} \bmod n = m \bmod n$. Fermat theorem is used in the last transition.

B.3 RSA signature generation and verification

We suppose that the hash function is H and Bob wants to generate an RSA signature of the message m and send it to Alice.

1. Bob computes h = H(m) and represents it as an integer in the interval [0, n-1].
2. Using its private key d, he computes $s = h^d \bmod n$.
3. Bob sends s and m to Alice.

The verification process is done as follows:

1. Alice obtains Bob's public key (e, n).
2. She computes $h_1 = H(m)$.
3. She recovers $h = s^e \bmod n = h^{ed} \bmod n = h \bmod n$.
4. She compares if $h = h_1$ the signature is valid and the message m is accepted else it is rejected.

Appendix C

Ring signature and multikey CGA

The ring signature algorithm that we introduce now is defined in section 4.2 of [18]. It uses the RST signature scheme defined in [21]. This kind of signature is used with multikey CGA in mobile network when talking about proxying. Its principal advantage is that it provides anonymity

The inputs of the algorithm are:

- The message to sign.
- The M-CGA type tag.

The hypotheses are:

- The digest is called DIGEST-F(m)
- The SHA1 digest produces a d-bit string.
- Let $\oplus$ denotes the XOR function.
- Let E() be an encryption scheme that uses d-bit keys and has b-bit input and output.
- The public keys in the RST ring signature scheme are exactly the same as public keys in RSA. Specifically $pk_i = (N_i, e_i)$, where N_i is a large (e.g., 1024-bits) composite integer that is the product of two large prime numbers p_i and q_i and where e_i is an integer that is relatively prime to $(p_i\text{-}1)\times(q_i\text{-}1)$.
- Let b be an integer such that $2^b > 2^t \times N_i$ for all i.
- Let pk_i be the public key of the "real" signer.

The signature generation algorithm has the following steps:

1. Set symmetric encryption key k to be DIGEST-F(m)
2. Pick a random b-bit string v

3. For j from 1 to n (except j is not equal to i):
   (a) Pick random b-bit string x_j
   (b) Compute (q_j, r_j) such that $x_j = q_j \times N_j + r_j$ for r_j in [0, N_j]
   (c) Compute $y'_j = x_j^{e_j} \bmod N_j$ for y'_j in [0, N_j]
   (d) Set $y_j = q_j \times N_j + y'_j$
   (e) Go to Step 3.(a) if y_j is greater than or equal to 2^b
4. Compute y_j such that:
   $E(k)(y_n \oplus E(k)(y_{n-1} \oplus E(k)(\ldots \oplus E(k)(y_1 \oplus v)\ldots))) = v$
5. Compute (q_i, r_i) such that: $y_i = q_i \times N_i + r_i$ for r_i in [0, N_i]
6. Compute $x'_i = y_i^{1/e_i} \bmod N_i$ for x'_i in [0, N_i]
7. Set $x_i = q_i \times N_i + x'_i$
8. Go to Step 3 if x_i is greater than or equal to 2^b
9. Output the ring signature (x_1, ..., x_n, v)

The verification has the following steps:

1. Set symmetric encryption key k to be DIGEST-F(m)
2. For j from 1 to n:
   (a) Compute (q_j, r_j) such that: $x_j = q_j \times N_j + r_j$ for r_j in [0, N_j]
   (b) Compute $y'_j = x_j^{e_j} \bmod N_j$ for y'_j in the interval [0, N_j]
   (c) Sets $y_j = q_j \times N_j + y'_j$
3. Confirm that: $E(k)(y(n) \oplus E(k)(y(n\text{-}1) \oplus E(k)(\ldots \oplus E(k)(y(1) \oplus v)\ldots))) = v$.

C.1 RST ring signature suboption

The RST ring signature suboption is added to CGA parameter option defined by SEND. It has the following structure:

- *Type* is TBA1.
- *Length* contains the length of the suboption.
- *Public key length* contains the public key length in bytes.
- *Router certified public key* contains the public key of the router.

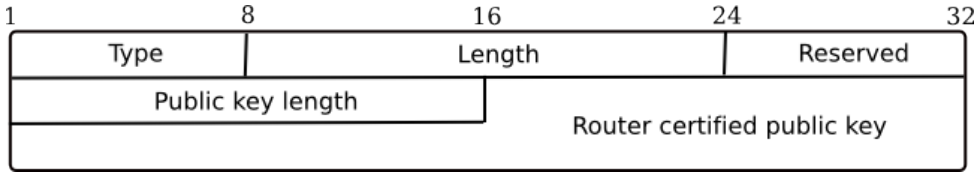

Figure C.1: RST ring signature suboption.

C.2 RST ring signature option

In order to use Multi-key CGA with SEND, a new option has to be added to replace the RSA signature option. This option has the following format:

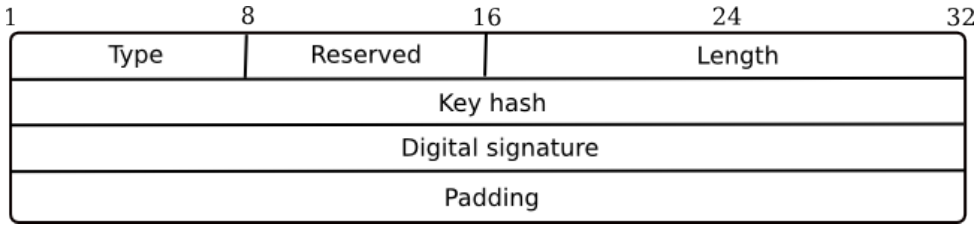

Figure C.2: RST signature option.

- *Type* is TBA2.
- *Key hash* contains the most significant 128 leftmost bits of SHA-1 hash of the public keys of the router and the address owner.
- *Digital signature* contains the signature computed using the senders private key and the public keys included in the CGA parameter option.

C.3 Secure proxy mobility option

When a mobile node leaves the home link, the home agent is responsible for proxying the address. If the address is a Multi-key CGA, the home agent can perform the proxying in a secure manner[18].

When the mobile node wants to ask for proxying from the home agent, it sends him a binding update message to bind its home address to a new care of address. It includes a Secure Proxy Mobility option containing information about the public keys used while creating the Multi-key CGA. If the certified public key of the proxy does not correspond to the home agents one, the agent responds with binding acknowledgement signalling the error.

The Secure Proxy Mobility option is presented in Figure C.3.

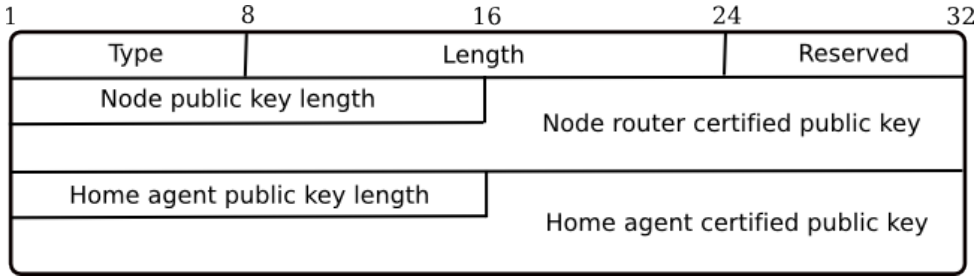

Figure C.3: Secure proxy mobility option.

These are the options that must be added in order to make multikey-CGA usage possible.

Appendix D

Mathematical concepts and ECC

We are going first to introduce some mathematical concepts [27] that are used in elliptic curve cryptogrpahy. Then we present other ECC encrypting algorithms.

D.1 Group

An abelian group (G,$\diamond$) consists of a set $\mathbf{G}$ with a binary operation $\diamond : \mathbf{G} \times \mathbf{G} \longrightarrow \mathbf{G}$ satisfying the following properties:

- Associativity: a $\diamond$ (b $\diamond$ c) = (a $\diamond$ b) $\diamond$ c for all a, b, c in $\mathbf{G}$.
- Existence of an identity: there exists an element e in $\mathbf{G}$ such that a $\diamond$ e =e $\diamond$ a =a for all a in $\mathbf{G}$.
- Existence of inverses: For each a in $\mathbf{G}$, there exists an element b in $\mathbf{G}$, called the inverse of a, such that a $\diamond$ b = b $\diamond$ a = e.
- Commutativity: a $\diamond$ b = b $\diamond$ a for all a, b in $\mathbf{G}$.

The group operation is usually called addition (+) or multiplication ($\times$). In the first instance, the group is called an additive group, the (additive) identity element is usually denoted by 0, and the (additive) inverse of a is denoted by -a. In the second instance, the group is called a multiplicative group, the (multiplicative) identity element is usually denoted by 1, and the (multiplicative) inverse of a is denoted by a^{-1}. The group is finite if $\mathbf{G}$ is a finite set, in which case the number of elements in $\mathbf{G}$ is called the order of $\mathbf{G}$.

For example, let p be a prime number, and let $\mathbf{F_p}$ = {0,1,2, ..., p-1} denote the set of integers modulo p. Then ($\mathbf{F_p}$,+), where the operation + is defined to be addition of integers modulo p, is a finite additive group of order p with (additive) identity element 0. Also, ($\mathbf{F_p^*}$, $\times$), where $\mathbf{F_p^*}$ denotes the nonzero elements in $\mathbf{F_p^*}$ and the operation $\times$ is defined to be multiplication of integers modulo p, is a finite multiplicative group of

order p-1 with (multiplicative) identity element 1. The triple ($\mathbf{F}_\mathbf{p}^*$, +, ×) is a finite field, denoted generally as $\mathbf{F}_\mathbf{p}^*$.

Now, if **G** is a finite multiplicative group of order n and g in **G**, then the smallest positive integer t such that $g^t = 1$ is called the order of g; such a t always exists and is a divisor of n. The set $\langle g \rangle = \{g^i : 0 \leq i \leq t\text{-}1\}$ of all powers of g is itself a group under the same operation as **G**, and is called the cyclic subgroup of **G** generated by g. Analogous statements are true if **G** is written additively. In that instance, the order of g in **G** is the smallest positive divisor t of n such that t.g = 0, and $\langle g \rangle = \{i \times g : 0 \leq i \leq t\text{-}1\}$. Here, t.g denotes the element obtained by adding t copies of g. If **G** has an element g of order n, then **G** is said to be a cyclic group and g is called a generator of **G**.

D.2 Finite fields

Fields are abstractions of familiar number systems (such as the rational numbers Q, the real numbers R, and the complex numbers C) and their essential properties. They consist of a set **F** together with two operations, addition (denoted by +) and multiplication (denoted by ×), that satisfy the usual arithmetic properties:

- (**F**,+) is an abelian group with (additive) identity denoted by 0.
- ($\mathbf{F}^*$, ×) is an abelian group with (multiplicative) identity denoted by 1.
- The distributive law holds: (a+b) × c = a × c+b × c for all a, b, c in **F**.

If the set **F** is finite, then the field is said to be finite.

D.2.1 Field operations

A field **F** is equipped with two operations, addition and multiplication. Subtraction of field elements is defined in terms of addition: for a, b in **F**, a-b = a +(-b) where (-b) is the unique element in **F** such that $b + (-b) = 0$ (-b is called the negative of b). Similarly, division of field elements is defined in terms of multiplication: for a,b in **F** with b= 0, a÷b = a × b^{-1} where b^{-1} is the unique element in **F** such that b × $b^{-1} = 1$ (b^{-1} is called the inverse of b).

D.2.2 Existence and uniqueness

The order of a finite field is the number of elements in the field. There exists a finite field **F** of order q if and only if q is a prime power, i.e., q = p×m where p is a prime number called the characteristic of **F**, and m is a positive integer. If m = 1, then F is called a prime field. If m ≥ 2, then **F** is called an extension field. For any prime power q, there is essentially only one finite field of order q; informally, this means that any two finite fields

of order q are structurally the same except that the labelling used to represent the field elements may be different. We say that any two finite fields of order q are isomorphic and denote such a field by $\mathbf{F_q}$.

D.2.3 Prime fields

Let p be a prime number. The integers modulo p, consisting of the integers {0,1,2, ..., p-1} with addition and multiplication performed modulo p, is a finite field of order p. We denote this field by $\mathbf{F_p}$ and call p the modulus of $\mathbf{F_p}$. For any integer a, a mod p shall denote the unique integer remainder r, $0 \leq r \leq p-1$, obtained upon dividing a by p; this operation is called reduction modulo p.

D.2.4 Binary fields

Finite fields of order 2^m are called binary fields or characteristic-two finite fields. One way to construct $\mathbf{F_{2^m}}$ is to use a polynomial basis representation. Here, the elements of $\mathbf{F_{2^m}}$ are the binary polynomials (polynomials whose coefficients are in the field $\mathbf{F_2} = \{0, 1\}$) of degree at most $m - 1$:
$\mathbf{F_{2^m}} = \{a_{m-1}z^{m-1}+a_{m-2}z^{m-2} +\ldots+a_2z^2+a_1z+a_0 : a_i \text{ in } \{0, 1\}\}$ An irreducible binary polynomial f(z) of degree m is chosen (such a polynomial exists for any m and can be efficiently found). Irreducibility of f(z) means that f(z) can not be factored as a product of binary polynomials each of degree less than m. Addition of field elements is the usual addition of polynomials, with coefficient arithmetic performed modulo 2. Multiplication of field elements is performed modulo the reduction polynomial f(z). For any binary polynomial a(z), a(z) mod f(z) shall denote the unique remainder polynomial r(z) of degree less than m obtained upon long division of a(z) by f(z); this operation is called reduction modulo f(z).

D.3 Elliptic curve group

An elliptic curve E over a field $\mathbf{F}$ is defined, as we said it in section 5.3.1, by the following equation: E: $y^2+a_1xy+a_3y = x^3+a_2x^2+a_4x+a_6$, where $a_1,a_2,a_3,a_4,a_6 \in \mathbf{F}$ and $\Delta \neq 0$
Δ is the discriminant of E and is defined as follows:
$\Delta = -d_2^2\, d_8-8d_3^4-27{d_2}^6 +9d_2d_4d_6$
$d_2 = a_1^2+4a_2$
$d_4 = 2a_4+a_1a_3$
$d_6 = a_3^2+4a_6$
$d_8 = a_1^2a_6+4a_2a_6-a_1a_3a_4+a_2a_3^2-a_4^2$

Here are some remarks concerning E:

- We say that E is defined over $\mathbf{F}$ because the coefficients a_1, a_2, a_3, a_4, a_6 of its defining equation are elements of $\mathbf{F}$. $\mathbf{F}$ is called the underlying field. Note that if E is defined over $\mathbf{F}$, then E is also defined over any extension field of $\mathbf{F}$.
- The condition $\Delta \neq 0$ ensures that the elliptic curve is smooth, that is, there are no points at which the curve has two or more distinct tangent lines.
- The point ∞ is the only point on the line at infinity that satisfies the projective form of the Weierstrass equation.

D.4 Elliptic curve cryptography

We present in this section the other EC cryptographic algorithms that have not been described in chapter 5. We remind that ECC key generation is done as follows:
Let E be an elliptic curve defined over a finite field $\mathbf{F_p}$. Let P be a point in $E(\mathbf{F_p})$, and suppose that P has prime order n. Then the cyclic subgroup of $E(\mathbf{F_p})$ generated by P is: $\langle P \rangle = \{\infty, P, 2P, 3P, \ldots, (n-1)P\}$.

The prime p, the equation of the elliptic curve E, and the point P and its order n, are the public domain parameters. A private key is an integer d that is selected uniformly at random from the interval [1, n-1], and the corresponding public key is Q = dP.

The problem of determining d given the domain parameters P and Q is the elliptic curve discrete logarithm problem (ECDLP).

D.4.1 ElGamal elliptic curve algorithm

We present in this subsection the encryption and decryption procedures for the elliptic curve analogue of the basic ElGamal encryption.

The encryption inputs are: elliptic curve domain parameters (p, E, P, n), public key Q, plaintext m. It is done following the next steps:

1. Represent the message m as a point M in $E(\mathbf{F_p})$.
2. Select $k \in [1, n-1]$.
3. Compute $C_1 = kP$.
4. Compute $C_2 = M + kQ$.
5. Return(C_1, C_2).

The decryption inputs are: domain parameters (p, E, P, n), private key d, ciphertext (C_1, C_2).

It is done by computing $M = C_2 - dC_1$, and then extracting m from M. In fact $M = C_2 - dC_1 = M + kQ - d\,kP = M + kQ - kQ = M$.

D.4.2 Elliptic Curve Diffie-Hellman (ECDH)

This well known algorithm is quite important in modern protocols as a key exchange and can be adopted for ECC:
Consider two parties Alice and Bob willing to exchange a common secret key without making this one known to a passive eavesdropper. Both have agreed to a common and publicly known curve E over a finite field eg $\mathbf{F_p}$ as well as to a base point P.

- Alice randomly chooses k_A, $0 < k_A < p$ and Bob accordingly k_B, $0 < k_B < p$. k_A is considered as Alice's private key, k_B is Bob's private key.
- Alice computes her public key: $Q_A = k_A P$, Bob does: $Q_B = k_B P$.
- Alice sends Q_A to Bob, Bob sends Q_B to Alice.
- Alice can now compute the shared secret for her and Bob by secret $= k_A Q_B$ and Bob also by secret $= k_B Q_A$.

An eavesdropper knows only Q_A and Q_B but is not able to compute the secret from that.

D.4.3 Elliptic Curve Discrete Logarithm Problem (ECDLP)

The problem of determining the private key d given the domain parameters P and Q is the elliptic curve discrete logarithm problem (ECDLP). It represents the analogue of the discrete logarithm problem which consists in finding y that verify $x = g^y \bmod(n)$ knowing x, g and n.

Appendix E

Algorithm functions

The different functions that have been used in CGA generation and verification algorithms are:

- *int init_modifier(parameters *p)*: chooses an initial random modifier. It takes as argument the parameters structure which is defined as follows:

```
typedef struct parameters {
                            uint8_t modifier[MODIFIER_LENGTH];
                            uint8_t pref[PREFIX_LENGTH];
                            uint8_t cc;
                            unsigned char *derkey;
                            uint32_t derlong;
                            uint8_t iid[IID_LENGTH];
                            uint8_t cga[CGA_LENGTH];
                           }parameters;
```

- *void init_error_lists(void)*: initializes the error messages list of Openssl.
- *void check_error(char *erreur)*: stores error message in the string '*erreur*'.
- *int PRNG_seed(int i,char *erreur)*: seeds Openssl pseudo random generator with *i* bytes.
- *int rsa_generation(RSA **rsa,int modulus,int pubk,char *erreur)*: creates a RSA key and store it in *rsa*. The key will have a modulus length equal to *modulus* bits and *pubk* as its public exponent.
- *int rsa_signature(uint8_t *hash,uint8_t **sign,unsigned int *signlong,RSA **rsa,char *erreur)*: generates an RSA signature.

- *int rsa_verification(uint8_t *hash,uint8_t **sign,unsigned int signlong,RSA **rsa,char *erreur)*: does an RSA signature verification.
- *void init_param_rsa(parameters *p,RSA **rsa,int modulus,int pubkey,int rand,char *erreur)*: initializes all RSA key parameters.
- *RSA *RSA_create_key(RSA **rsa,RSA **rsa_init,unsigned long e_value,void (*callback)(int,int,void *), void *cb_arg)*: creates an RSA key using another RSA key parameters.
- *int RSA_create_key_ex(RSA *rsa,RSA *rsa_init,BIGNUM *e_value, BN_GENCB *cb)*: is used in the previous function.
- *int pub_key_to_der(RSA **rsa,uint8_t **derkey,char *erreur)*: converts the public key to a DER encoded key.
- *int eckey_generation(EC_KEY **eckey,int nid,char *erreur)*: generates an ECC key.
- *int ecdsa_signature(uint8_t *hash,uint8_t **ecsign,unsigned int *signlong,EC_KEY **eckey,char *erreur)*: generates an ECDSA signature.
- *int ecdsa_verification(uint8_t *hash,uint8_t **ecsign,unsigned int signlong,EC_KEY **eckey,char *erreur)*: verifies an ECDSA signature.
- *void init_param_ec(parameters *p,EC_KEY **eckey,int nid,int rand,char *erreur)*: initializes ECC key parameters.
- *int pub_key_to_oc(EC_KEY **eckey,uint8_t **oskey,char *erreur)*: transforms ECC public key to octets string format.
- *void sha_hash(uint8_t *message,unsigned long msg2signlong,uint8_t *hash,char *erreur)*: computes a SHA1 hash.
- *void sha_hash2(uint8_t *message,unsigned long msg2signlong,uint8_t *hash2,char *erreur)*:computes a SHA1 hash and deduces hash2 value.
- *void sha_hash1(uint8_t *message,unsigned long msg2signlong,uint8_t *hash1,char *erreur)*: computes a SHA1 hash and deduces hash1 value.
- *void ripemd_hash(uint8_t *message,unsigned long msg2signlong,uint8_t *hash,char *erreur)*: computes a RIPEMD hash.
- *void ripemd_hash2(uint8_t *message,unsigned long msg2signlong,uint8_t *hash2,char *erreur)*: computes a RIPEMD hash and deduces hash2 value.

- *void ripemd_hash1(uint8_t *message,unsigned long msg2signlong,uint8_t *hash1,char *erreur)*: computes a RIPEMD hash and deduces hash1 value.
- *void tiger_hash2(uint8_t *message,unsigned long msg2signlong,uint8_t *hash2,char *erreur)*: computes a TIGER hash and deduces hash2 value.
- *void tiger_hash1(uint8_t *message,unsigned long msg2signlong,uint8_t *hash1,char *erreur)*: computes a TiGER hash and deduces hash1 value.
- *void sha256_hash(uint8_t *message,unsigned long msg2signlong,uint8_t *hash,char *erreur)*: computes a SHA256 hash.
- *void sha256_hash2(uint8_t *message,unsigned long msg2signlong,uint8_t *hash2,char *erreur)*: computes a SHA256 hash and deduces hash2 value.
- *void sha256_hash1(uint8_t *message,unsigned long msg2signlong,uint8_t *hash1,char *erreur)*: computes a SHA256 hash and deduces hash1 value.
- *void print_hash_list(void)*: prints the hash algorithms list.
- *void print_parameters(parameters *p,uint8_t *hash1,uint8_t *hash2,uint8_t sec)*: prints CGA parameters and hash values (1 and 2).
- *void print_cga(parameters *p)*: prints only CGA parameters.
- *void print_curves_list(void)*: prints the list of tha available curves.
- *void print_help(void)*: prints the help.
- *void print_verif_param(parameters *p)*: prints the parameters used during the CGA verification.
- *void fprint_parameters(FILE *fp,parameters *p,uint8_t *hash1,uint8_t *hash2,uint8_t sec)*: stores CGA parameters into a file.
- *void init_prefix_s2h(char *prefixe,parameters *p)*: transforms the keybord entered prefix from a string to an hexadecimal table.
- *void convert_16_8(parameters *p,uint16_t *tab)*: converts the keybord from words of 2 bytes to one bytes word.
- *void setbits(parameters *p,uint8_t sec)*: sets SEC, u and g bits to 0.
- *void incr_mod(parameters *p)*: increments the modifier by 1.
- *int cga_cmp_hash2_sec(uint8_t *hash2, uint8_t sec)*: compares the 16 × SEC bits of hash2 to 0.

- *uint8_t * cga_param_hash2(parameters *p,unsigned long *msg2signlong)*: forms the message used to compute hash2.
- *uint8_t * cga_param_hash1(parameters *p,unsigned long *msg2signlong)*: forms the message used to compute hash1.
- *int pad_len_cgaopt(parameters *p)*: determines the padding length to be added to the message which is going to be signed.
- *uint8_t *msg_to_sign(parameters *p,unsigned long *msg2signlong)*:forms the message going to be signed.
- *void create_iid(parameters *p,uint8_t * hash1,uint8_t sec)*: creates the interface identifier.
- *void create_cga(parameters *p)*: creates the CGA.
- *int verify_cga(parameters *p,char *erreur)*: verifies the CGA when SHA1 is used as hash algorithm.
- *int verify_cga_sha256(parameters *p,char *erreur)*: verifies the CGA when SHA256 is used as hash algorithm.
- *int verify_cga_tiger(parameters *p,char *erreur)*: verifies the CGA when TIGER is used as hash algorithm.
- *int verify_cga_ripemd(parameters *p,char *erreur)*: verifies the CGA when RIPEMD is used as hash algorithm.

Bibliography

[1] P. Nikander, J. Kempf, and E. Nordmark. IPv6 Neighbor Discovery (ND) Trust Models and Threats. RFC 3756 (Informational), May 2004.

[2] J. Arkko, J. Kempf, B. Zill, and P. Nikander. SEcure Neighbor Discovery (SEND). RFC 3971 (Proposed Standard), March 2005.

[3] T. Aura. Cryptographically Generated Addresses (CGA). RFC 3972 (Proposed Standard), March 2005. Updated by RFCs 4581, 4982.

[4] S. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6) Specification. RFC 2460 (Draft Standard), December 1998. Updated by RFC 5095.

[5] T. Narten, E. Nordmark, W. Simpson, and H. Soliman. Neighbor Discovery for IP version 6 (IPv6). RFC 4861 (Draft Standard), September 2007.

[6] A. Conta and S. Deering. Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification. RFC 2463 (Draft Standard), December 1998. Obsoleted by RFC 4443.

[7] J. Postel. Internet Protocol. RFC 791 (Standard), September 1981. Updated by RFC 1349.

[8] D. Plummer. Ethernet Address Resolution Protocol: Or Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware. RFC 826 (Standard), November 1982. Updated by RFC 5227.

[9] R. Hinden and S. Deering. Internet Protocol Version 6 (IPv6) Addressing Architecture. RFC 3513 (Proposed Standard), April 2003. Obsoleted by RFC 4291.

[10] P. Hoffman and B. Schneier. Attacks on Cryptographic Hashes in Internet Protocols. RFC 4270 (Informational), November 2005.

[11] M. Bagnulo and J. Arkko. Support for Multiple Hash Algorithms in Cryptographically Generated Addresses (CGAs). RFC 4982 (Proposed Standard), July 2007.

[12] I.Van Beijnum. Interactions between CGA and DHCPv6: draft-van-beijnum-cga-dhcp-interaction-00, November 2007.

[13] S.Xia S.Jiang and A.Garcia Martinez. Requirements for configuring cryptographically generated addresses (CGA) and overview on RA and DHCPv6 based approaches: draft-jiang-sendcgaext-cga-config-01, January 2007.

[14] M.Bagnulo. Hash Based Addresses (HBA): draft-ietf-shim6-hba-05, December 2007.

[15] M.Bagnulo. Shim6: Level 3 Multihoming Shim Protocol for IPv6: draft-ietf-shim6-proto-08, May 2007.

[16] J.Laganier. Thesis: Decentralised security for internet: cryptographic identity based propostions, 2007.

[17] D. Johnson, C. Perkins, and J. Arkko. Mobility Support in IPv6. RFC 3775 (Proposed Standard), June 2004.

[18] J. Kempf. Secure IPv6 Address Proxying using Multi-Key Cryptographically Generated Addresses (MCGAs): draft-kempf-cgaext-ringsig-ndproxy-00, August 2007.

[19] D. Thaler, M. Talwar, and C. Patel. Neighbor Discovery Proxies (ND Proxy). RFC 4389 (Experimental), April 2006.

[20] G. Daley and J.M. Combes. Secure neighbor discovery proxy problem statement: draft-daley-csi-sndp-prob-00, May 2008.

[21] A.Shamir RL.Rivest and Y.Tauman. How to leak a secret.

[22] T.Cheneau amd J.M.Combes. Article: An attack against SEND, February 2008.

[23] OpenSSL official site: http://www.openssl.org/.

[24] Maemo official site: http://maemo.org/.

[25] Scratchbox official site: http://www.scratchbox.org/.

[26] Tiger: A Fast New Cryptographic Hash Function (Designed in 1995): http://www.cs.technion.ac.il/ biham/Reports/Tiger/.

[27] A.Menezes D.Hankerson and S.Vanstone. *Guide to elliptic curve cryptography.* Springer, 2004.

[28] C.Schroder. Understand IPv6 Addresses: http://www.enterprisenetworkingplanet.com/netsp/article.php/3633211.