Analysis and Optimization of Cryptographically Generated Addresses

Joppe W. Bos[1], Onur Özen[1], and Jean-Pierre Hubaux[2]

[1] EPFL IC IIF LACAL, Station 14, CH-1015 Lausanne, Switzerland
{joppe.bos, onur.ozen}@epfl.ch
[2] EPFL IC ISC LCA1, Station 14, CH-1015 Lausanne, Switzerland
jean-pierre.hubaux@epfl.ch

Abstract. The need for nodes to be able to generate their own address and verify those from others, without relying on a global trusted authority, is a well-known problem in networking. One popular technique for solving this problem is to use self-certifying addresses that are widely used and standardized; a prime example is cryptographically generated addresses (CGA). We re-investigate the attack models that can occur in practice and analyze the security of CGA-like schemes. As a result, an alternative protocol to CGA, called CGA++, is presented. This protocol eliminates several attacks applicable to CGA and increases the overall security. In many ways, CGA++ offers a nice alternative to CGA and can be used notably for future developments of the Internet Protocol version 6.

1 Introduction

Cryptographically generated addresses (CGA) is a technique that creates a fixed size address by hashing the address owner's public key with the help of a cryptographic hash function. This technique enables the address owner to assert address ownership by creating a relation between the address and the address owner's public/private key pair.

An example where this technique can be used is in Internet Protocol version 6 (IPv6) addresses. The 64-bits of these addresses known to be the *interface identifier* are then generated with the help of CGA as proposed by Aura [1, 2].

The main advantage of CGAs is that they are self-certified; a trusted third party or a public-key infrastructure (PKI) is not needed to generate the IPv6 address or to verify other addresses. Self-certified addresses are extensively used in protocols such as Secure Neighbor Discovery [3], Shim6 [4] and the IPv6 mobility support [5]; they offer features such as duplicate address detection and proof of address ownership.

The idea of cryptographically generated addresses first appeared in the child-proof authentication for mobile-IPv6 (CAM) protocol by O'Shea and Roe [6]; this was later improved by Nikander [7]. An alternative method was proposed by Montenegro and Castelluccia [8] under the name "statistically unique and

cryptographically verifiable addresses" (SUCV). Finally, the actual model was presented by Aura [1] and appears as an RFC [9].

The drawback of the early proposals of CGA, namely the small number of bits of the address to accommodate the result of the cryptographic hash function, is solved by Aura in [1] by using hash extensions. With this technique, the resistance of the scheme against impersonation is increased at the cost of increasing the time needed for address generation while keeping the necessary operations required for the verification constant. This is realized by scaling, with the help of a relation defined by the security parameter, the effort needed to break the system in the future due to the progress of technology. This implies an increase in the cost of address generation; the exponential growth over time of computing power compensates this increase.

To the best of our knowledge, no work has been published, besides the RFC documents and the unpublished work in [2], on the analysis of CGA since the original proposal [1]. As also observed in the original proposal, CGA is susceptible to a global time-memory trade-off attack that eliminates the effect of hash extensions in the long run at the cost of storage. Such an attack is assumed to be impractical in [1]. Moreover, due to lack of authentication, CGA is vulnerable to a replay attack where an attacking node is capable of sniffing and storing signed messages from a target node and replay them later.

In this work, we present a detailed security/efficiency analysis of CGA and a proposal to solve security problems related to self-certifying address generation and verification in CGA. This protocol, mainly based on the ideas of CGA, is called CGA++. In the analysis part, a novel security framework is provided, which enables us to evaluate the security of CGA-like schemes, including CGA++. In this design, we mitigate the global time-memory trade-off attack by reducing its effect to a specific network. Furthermore, we introduce digital signatures in order to overcome the lack of authenticity in CGA and to increase the security when no hash extensions are used. CGA++ offers an alternative to CGA and can be used in practice for future development of IPv6.

The organization of the paper is as follows. Section 2 introduces the preliminaries for IPv6 addresses including the system model and the necessary notation throughout the paper. In Section 3 and 4, we provide the specification and the analysis of CGA, respectively. We introduce the design of CGA++ in Section 5 followed by its analysis in Section 6. The compatibility and the applications of CGA++ are discussed in Section 7. We conclude the paper in Section 8.

2 Preliminaries

The objective for using CGA is to generate self-certified IPv6 addresses. For the sake of clarity, we adhere to the conventions and the notations from [10, 11].

IPv6 addresses are 128-bit data blocks constructed by the concatenation of two 64-bit words: *subnet prefix* and *interface identifier* [10]. The former is located on the most significant 64-bit, which is used to determine the nodes' location in the Internet topology. The latter, being comprised of the least significant 64-bits,

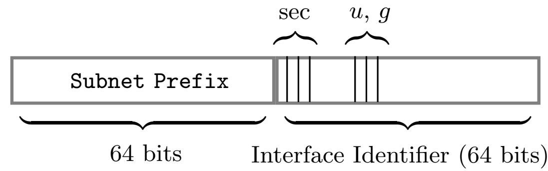

Fig. 1. *Format of an IPv6 address. The parameters sec, u and g are placed at the most significant three, seventh and eighth bit of the interface identifier, respectively.*

acts as an identity of the node whose generation process is the main target of this work. See Fig. 1 for the general overview of IPv6 addresses.

In the address format, there are several parameters in the *interface identifier*, which have special semantics. The first parameter set is comprised of the u and g bits, located in the seventh and the eighth bit of the *interface identifier*. The combination $u = g = 1$ is unused for other purposes and suggested by Aura to indicate the use of CGA [1]. The other value in the *interface identifier* is the security parameter *sec*, a three-bit user defined parameter used to determine the length of the hash extension in the protocol. In CGA, this parameter is used to scale the relation in the hash extension. We provide the necessary notation used in the rest of the paper in Table 1.

In the original RFC [9], the proposed hash function is SHA-1 [12]. The recently discovered weaknesses of SHA-1 led to a modification of the CGA specification to enable the support of alternative hash functions [13]. Therefore, in this paper, we denote the hash function used in the protocol as $\mathcal{H}$ and assume this hash function to be ideal: it is optimally collision, preimage and second-preimage resistant.

We assume, throughout the paper, that (mobile) nodes are capable of dealing with Internet protocols and are also equipped with and capable of calculating basic cryptographic algorithms. Moreover, we assume an increase of performance of mobile nodes following Moore's law. Yet, of course, there will always be a market for low-end devices. But it is unlikely for them to be isolated and mobile, and even if they are, they will not be security sensitive.

For the attacker, we ignore the time required to generate valid public/private key pairs as these can easily be computed and stored, if necessary, in an initialization phase.

3 CGA Specification

The actual protocol for self-certified address generation and verification for IPv6 using CGA appears in RFC 3972 [9], which is based on the ideas from [1]. In this section, we recall the specification of CGA. CGA uses a technique called hash extension, which is realized by the security parameter *sec*; this parameter linearly scales the number of bits used in the hash extension by imposing $16 \times sec$ many bits to zero in the hash value denoted by Hash2.

Data	Notation	Length
IPv6 Address	IPv6	128-bit
Subnet Prefix	SP	64-bit
Interface Identifier	Interface ID	64-bit
Public-Key	K_{pub}	Variable length
Private-Key	K_{priv}	Variable length
Digital Signature	Sign	Variable length
Modifier	m	128-bit
Collision Count	CC	8-bit
Security Parameter	sec	3-bit
u,g flags	u, g	1-bit each

Table 1. *Notation and data sizes of the various fields used in this article.*

The main idea behind CGA is to trade efficiency for security. When generating a new address, a node has to satisfy certain constraints, i.e. the hash extension, which decreases the efficiency of the address generation. Because an attacker needs to do this extra work as well, the level of security increases compared to the setting where no hash extensions are used. The verification, on the other hand, requires a constant amount of time and does not suffer from an efficiency decrease. This ensures that an attacking node needs to perform all the computational work, thereby preventing the denial of service of verifiers.

Address Generation. The procedure for generating an IPv6 address using CGA is illustrated in Fig. 2 which can be described as follows.

1. Set the *modifier* to a random 128-bit value. Select the security parameter sec and set the *collision count* to zero.
2. Concatenate the *modifier*, 64 + 8 zero bits, and the encoded public-key. Execute the $\mathcal{H}$ algorithm on the concatenation. The leftmost 112 bits of the result are Hash2.
3. Compare the $16 \times sec$ leftmost bits of Hash2 with zero. If they are all zero (or if $sec = 0$), continue with Step (4). Otherwise, increment the *modifier* and go back to Step (2).
4. Concatenate the *modifier*, *subnet prefix*, *collision count* and encoded public-key. Execute the $\mathcal{H}$ algorithm on the concatenation. The leftmost 64 bits of the result are Hash1.
5. Form an *interface identifier* by setting the two reserved bits u and g in Hash1 both to 1 and the three leftmost bits to sec.
6. Concatenate the *subnet prefix* and *interface identifier* to form an 128-bit IPv6 address.
7. If an address collision with another node within the same subnet is detected, increment the *collision count* and go back to step (4). However, after three collisions, stop and report the error.

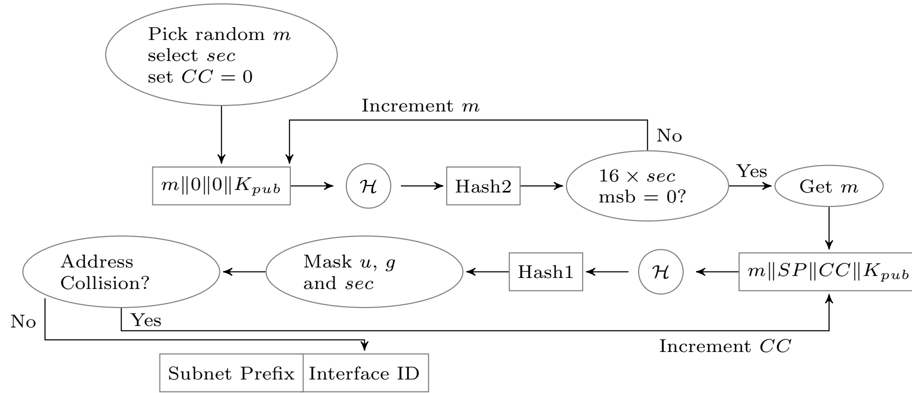

Fig. 2. *Detailed data flow of the address generation in CGA.*

Verification of Address Ownership. The verification of address ownership is realized by the execution of the following steps. Given the IPv6 address, *collision count* and the *modifier*,

1. Check that the *collision count* is $0, 1$ or 2 and that the *subnet prefix* is equal to the *subnet prefix* of the address. The CGA verification fails if either check fails.
2. Concatenate the *modifier*, *subnet prefix*, *collision count* and the public-key. Execute the $\mathcal{H}$ algorithm on the concatenation. The 64 leftmost bits of the result are Hash1.
3. Compare Hash1 with the *interface identifier* of the address. The differences in the two reserved bits u and g and in the three leftmost bits are ignored. If the 64-bit values differ (other than in the five ignored bits), the CGA verification fails.
4. Concatenate the *modifier*, $64 + 8$ zero bits and the public-key. Execute the $\mathcal{H}$ algorithm on the concatenation. The leftmost 112 bits of the result are Hash2.
5. Read the security parameter *sec* from the three leftmost bits of the interface identifier of the address. Compare the $16 \times sec$ leftmost bits of Hash2 with zero. If any one of these bits is nonzero, CGA verification fails. Otherwise, the verification succeeds. If $sec = 0$, verification never fails at this step.

4 Analysis of CGA

4.1 Design Rationale

In order to have a better comprehension, we explain the design rationale of CGA by going through the components inside CGA, especially the computation of Hash1 and Hash2.

Computation of Hash2. As stated by Aura in [1], the computation of Hash2 is introduced in order to gain security at the expense of efficiency. In CGA, the values that are used in Hash2 are the *modifier* and the public-key. The *subnet prefix* and the *collision count* are set to zero.

The idea is to use some common parameters in the domains of Hash1 and Hash2. As Hash2 requires special constraints, the most efficient way to satisfy this is to impose random data in the domain of Hash2. The *modifier* is used for this purpose, it allows the node to comply with the conditions imposed by the security parameter. The public-key is needed in this computation to assign this *modifier* to this node. If the public-key, or any other property specific to address generator, is not used in the computation of Hash2, an attacking node could simply send a verification request to this node and retrieve its *modifier* value. This would remove the need to compute a valid *modifier* for the attacker.

It was a design decision of Aura [1] not to include the *subnet prefix* in the computation of Hash2 for the sake of efficiency. A mobile node travels frequently from network to network and thus needs to regenerate its address over and over again. Assuming a mobile node does not have much computation power, it would be infeasible to search for a *modifier* every time it changes network. When not including the *subnet prefix* in the Hash2, a mobile node has to regenerate its Hash1, where the *subnet prefix* occurs, at the cost of only one hash function evaluation. Following the same reasoning, the *collision count* parameter is set to zero as well. This avoids computing Hash2 again when a collision of the *interface identifier* is detected after the creation of the Hash1.

Computation of Hash1. In the computation of the Hash1, all parameters are used. In the domain of Hash1, the *subnet prefix* is used in order to avoid a time-memory trade-off attack as it would be possible to store valid addresses from each network. *Collision count* is added to the domain to overcome the scenario where a duplicate address is generated. Finally, the *modifier* is used as a "proof" that the node generated a valid Hash2 and the public-key for the same reasons as in the Hash2.

4.2 Security Framework

Assume a "CGA-like" protocol design is to be assessed for security with a focus on the address generation and verification part. More precisely, a protocol is considered which makes use of two different hash function evaluations where the output of one is not used as the input for the other. A formal model can be useful for this task, especially for assessing the security of the considered protocol. Such a model is proposed in this section and is used in the remainder of this paper to state facts about the security and efficiency of such "CGA-like" protocols.

Let us denote the time needed for a node to generate an address as T_G, to verify an address as T_V and to impersonate an address as T_A. These times are stated as a function of T_1 and T_2, which denote the time to compute Hash1 and Hash2, respectively and are expressed in hash function evaluations. The number

of available bits in the address, which is the number of (truncated) output of Hash1, is denoted by l. We denote the number of bits on which we put a special condition by s, the (truncated) output of Hash2; here $l, s \in \mathbb{N}$.

Address Generation. Address generation for a legitimate node is not expected to exceed $2^s \cdot T_2$ in order to meet the conditions of Hash2, plus T_1 to generate the address. The cost of address generation T_G is therefore:

$$T_G = 2^s \cdot T_2 + T_1. \tag{1}$$

Address Verification. To verify an address, the conditions on Hash2 need to be validated, representing a duration T_2. If this validation is successful, the address needs to be checked in time T_1. The time needed for verification, T_V, is

$$T_V = T_1 + T_2. \tag{2}$$

Impersonation. In order to impersonate a node, an attacker can proceed in two ways: by first satisfying the constraints on Hash1 and then on Hash2 or vice versa. Beginning with Hash1, the attacker must first perform a second-preimage attack on Hash1, which is expected to take no more than $2^l \cdot T_1$ hash function evaluations to find another valid parameter set to match the *interface identifier*. Once fulfilled, the conditions on Hash2 for the generated *modifier* should be satisfied, which happens with probability 2^{-s} for an ideal hash function. The total cost C_{A,H_1} for impersonation, when beginning with Hash1, becomes $C_{A,H_1} = (2^l \cdot T_1 + T_2) \cdot 2^s$.

Starting from Hash2, the conditions on Hash2 are met at a cost of up to $2^s \cdot T_2$ hash function evaluations. Next, Hash1 is created and verified if it collides with the target address. The probability of hitting this specific Hash1 is 2^{-l}. Therefore, the total cost C_{A,H_2}, when beginning with Hash2, becomes $C_{A,H_2} = (2^s \cdot T_2 + T_1) \cdot 2^l$. An attacker can choose either of these values in order to minimize his attack cost. Hence, the time for impersonation, T_A, in a generic model becomes

$$T_A = \min(C_{A,H_1}, C_{A,H_2}) = \min((2^l \cdot T_1 + T_2) \cdot 2^s, (2^s \cdot T_2 + T_1) \cdot 2^l). \tag{3}$$

4.3 Security of CGA

In order to assess the security of CGA, we begin with discussing the basic principles of cryptographic hash functions: collision, preimage and second-preimage resistance. This allows us to evaluate certain bounds in order to attack the scheme, assuming that the underlying cryptographic hash function has no known weaknesses. With the help of the birthday- and the brute-force attack, finding a collision and a (second) preimage require $\mathcal{O}(\sqrt{2^n})$ and $\mathcal{O}(2^n)$ hash function evaluations, as the digest size n tends to infinity, respectively.

A collision attack is not very powerful in this setting as it means being able to generate two nodes with the same address without having any control over the generated address. The preimage attack is equivalent to the second preimage attack since all the domain parameters of the hash function are known. The second preimage-attack is the most powerful attack model for this setting. It is equivalent to being able to generate another valid CGA parameter from a given address: i.e. impersonation. This attack model and its cost are treated in the following where the proof follows directly from the cost needed to perform a second-preimage attack when not using hash extensions (i.e. $sec = 0$) and our security framework for the case $sec > 0$ (cf. Section 4.2).

Lemma 1. *Given a network, assume the addresses are generated and verified by CGA with security parameter sec. Then, the number of operations required for the impersonation of a specific node is*

$$T_A = \begin{cases} 2^{59} & \text{if } sec = 0, \\ 2^{59+16\times sec} + 2^{16\times sec} & \text{if } 1 \leq sec \leq 3, \\ 2^{59+16\times sec} + 2^{59} & \text{otherwise.} \end{cases}$$

hash function evaluations.

This shows that the resistance of CGA against impersonation is mainly dominated by the increasing values of the security parameter sec.

A Time-Memory Trade-Off Attack on CGA. As observed in [1], a time-memory trade-off attack can be mounted on CGA in order to impersonate a node. Details of the time and memory complexities of this attack are not stated in [1] and they are assumed to be impractical. The following lemma explains this more specifically.

Lemma 2. *Given a number of $k > 0$ networks each of size approximately 2^{n_i} nodes, for $0 < i \leq k$, assume the nodes use CGA for address generation and verification. Using a time-memory trade-off attack, an attacker needs at most T calls to the hash function and comparisons of the hash-values in order to impersonate one random node in the network of size 2^{n_i}. When the number of attacks $A \to \infty$, the number of calls T per attack is asymptotically bounded by*

$$T \leq 2^{59-min(n_i)}. \tag{4}$$

In other words, T is independent of the security parameter sec. The storage requirement is $2^{33-min(n_i)}$ gigabytes, where $min(n_i)$ denotes the smallest value n_i.

Proof. Assume a database is given with valid *modifier* values m_j, $j \in \mathbb{N}$, such that the condition on Hash2 is satisfied. When targeting a network of size 2^{n_i}, a second-preimage for Hash1 of a random node is expected after 2^{59-n_i} hash function evaluations. The cost C, the number of hash function evaluations to create the database of *modifier* values, is expected to take no more than $C =$

$2^{59+16\times sec-n_i}$. The database is independent of the used *subnet prefix* and can be computed once and used for all subsequent attacks in the future. Then, the average cost T per attack becomes $T = 2^{59-n_i} + \frac{2^{59+16\times sec-n_i}}{A}$. By selecting the smallest network size among n_i, which maximizes the attack cost, the number of hash evaluations becomes asymptotically, when the number of attacks go to infinity, $T \leq 2^{59-\min(n_i)}$. The storage cost is $128 \cdot 2^{59-\min(n_i)}$ bits which corresponds to $2^{33-\min(n_i)}$ gigabytes. □

Note that this attack cannot be used to impersonate a specific node. Instead, it can be used to impersonate a random node in the network. In order to illustrate the required storage for such an attack, assume an attacker wants to impersonate an address of a random node in a network of size 2^{16}, this requires $2^{33-16} = 2^{17}$ gigabytes = 128 terabyte of storage. This is significant but not infeasible.

Authentication. One of the known limitations of CGA, as mentioned in [1], is the possibility for an attacking node to sniff and store signed messages from a target node. Once this is done, the attacker obtains the public-key and the *modifier*; with these values it can create a valid address using a different *subnet prefix*. Sending forged, correctly signed messages is computationally infeasible but by replaying the sniffed messages, an attacker can mislead legitimate nodes by convincing them that he owns an address.

This type of attack can also be used to generate an address that already exists in the network. That is, for a specific security parameter *sec*, the attacker can collect many valid *modifier* and public-keys together with a certain amount of signed messages from these nodes. Next, a *subnet prefix* is selected, and with the help of the stored values, a search is started for a hit in one of the addresses. This helps to reduce the complexity of the impersonation attack.

Another instance of such an attack is to search for nodes with a non-zero *collision count*. In the address generation of CGA, the nodes generate an address and look for a collision in the network. If there is a node with the generated address, the new node has to generate a new address by increasing the *collision count*. Hence, the attacker can look for a non-zero *collision count* in the network and use the valid *modifier* and public-key of this node to generate an existing address in the network with *collision count* zero. This helps the attacker to generate a duplicate addresses; he could even replay signed messages. Nevertheless, the probability of having a collision in the addresses is low and this attack fully depends on the non-zero *collision count*. Still, as the mobility property leads to the need to generate new addresses for the nodes while travelling from one network to another, the probability of address collision increases.

4.4 Efficiency of CGA

In CGA, the address generation time T_G is equal to $T_G = 2^{16\times sec} + 1$, which is dominated by the security parameter, assuming $sec > 0$, whereas the address verification time T_V is constant, namely equal to two. To illustrate the actual

computational demand, we make the optimistic assumption that a node has computing power comparable to a modern CPU used in a workstation. Our simple implementation of CGA, which uses the open source library OpenSSL [14], computes approximately $2^{18.5}$ digests/sec/CPU on a modern workstation (AMD64). The estimated time needed to comply with Hash2 requirements are provided in the following table

sec value	1	2	3	4	5	6
Required Time	0.2 secs	3.2 hrs	24 yrs	$1.6 \cdot 10^6$ yrs	$1.0 \cdot 10^{11}$ yrs	$6.8 \cdot 10^{15}$ yrs

These results correspond with the performance results from [1]. This table shows that the required time for generating a valid address with a high security parameter *sec* is currently impractical. However, it will be feasible in the future due to the exponential growth in the computational capabilities of the nodes. A possible solution to the efficiency problem for the current use of larger *sec* values is to generate these values off-line or search them in parallel, just as presented in the time-memory trade-off attack.

5 CGA++ Specification

Design Rationale. The main design rationale behind CGA++ follows from the fact that even if CGA offers a good protocol for self-certified address generation and verification, it has some limitations. Therefore, our main goal is to fix these weaknesses without losing too much efficiency. Considering the adoption and the extensive future use of CGA, one of our main goals is to adhere as closely as possible to CGA, thus offering an easy transition from CGA to CGA++.

As mentioned, a global time-memory trade-off attack is feasible at the cost of memory. In order to prevent this global attack the first obvious modification is to include the *subnet prefix* in the computation of Hash2. The verifier should make sure to check full IPv6 addresses and not the so-called link-local addresses as specified in IPv6 [10]. This has some efficiency loss; nevertheless, we believe this to be tolerable. Furthermore, an extra authentication mechanism is introduced by using digital signatures inside the verification process, preventing nearly all the mentioned attacks against CGA. This has the additional advantage that when no hash extensions are used ($sec = 0$) the security of the protocol is increased, compared to CGA. As a result, we propose a more secure, easy to adopt and compact alternative to CGA.

Address Generation. The general procedure of generating IPv6 address using CGA++ is depicted in Fig. 3 (note the similarities with Fig. 2). It can be described as follows.

1. Choose security parameter sec $\in \{0, \ldots, 7\}$. Set the *modifier* to a random 128-bit value and set the *collision count* to zero.

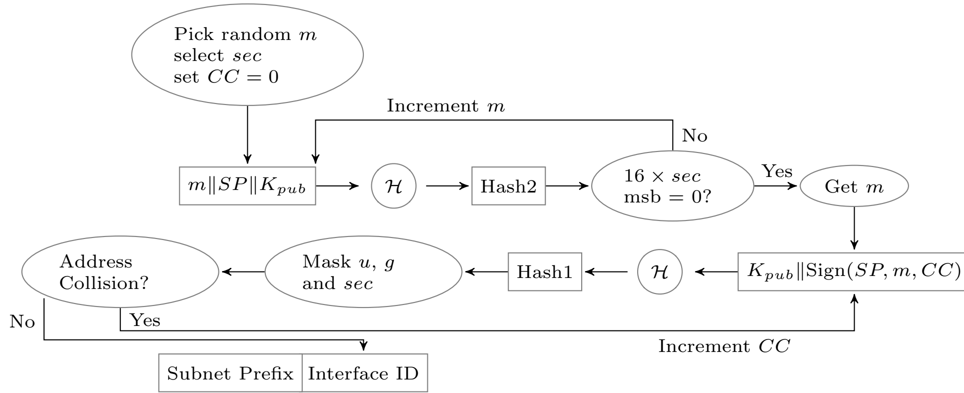

Fig. 3. *Detailed data flow of the address generation in CGA++.*

2. Concatenate the *modifier*, *subnet prefix* and the encoded public-key. Execute the hash algorithm on the concatenation. Check the most significant $16 \times sec$ bits of the result. Continue until $16 \times sec$ bits are zero by incrementing the *modifier*.
3. Sign the *modifier*, *collision count* and *subnet prefix* with the private-key corresponding to the public-key used.
4. Concatenate the encoded public-key and the signature. Execute the hash algorithm on the concatenation. The most significant 64 bits of the result are Hash1.
5. Form an *interface identifier* by setting the two reserved bits in Hash1 both to 1 and three bits to sec.
6. Concatenate the *subnet prefix* and *interface identifier* to form an 128-bit IPv6 address.
7. If an address collision is detected, increment the *collision count* and go back to step (3). However, after three collisions, stop and report the error.

The address generation of a node begins with satisfying the constraints in the hash extension as in CGA. The *collision count* is omitted, instead of being set to zero, which makes the input to the hash function smaller. Once this is satisfied, the address owner signs the *subnet prefix*, *modifier* and the *collision count* with his private-key. The public-key is concatenated to the signature and the corresponding interface identifier is obtained by hashing this concatenation.

Verification of Address Ownership. After the address generation has been performed, the verification of the address ownership is realized by the execution of the following steps. Given the IPv6 address, the signature and the public-key of the node,

1. Verify the signature and obtain the *modifier*, *collision count* and *subnet prefix*.

(a) *Benchmark results, in cycles, of the RSA public-key signature system.*

x-bit signature	Generate a key pair	Sign 59 bytes	Verify 59 bytes
512	$3.9 \cdot 10^7$	$1.1 \cdot 10^6$	$5.3 \cdot 10^5$
768	$8.0 \cdot 10^7$	$2.0 \cdot 10^6$	$6.0 \cdot 10^5$
1,024	$1.4 \cdot 10^8$	$2.9 \cdot 10^6$	$7.0 \cdot 10^5$
1,536	$3.2 \cdot 10^8$	$6.7 \cdot 10^6$	$1.0 \cdot 10^5$
2,048	$6.8 \cdot 10^8$	$1.2 \cdot 10^7$	$1.3 \cdot 10^5$

(b) *Benchmark results of the SHA-1 hash function.*

Hashing x bytes	Cycles per byte	Cycles per message
8	137.75	1,102
64	25.62	1,640
576	10.14	5,841
1,536	8.91	13,686
4,096	8.45	34,611

Table 2. *Measurements taken from ECRYPT benchmarking of cryptographic systems [15]. These median results are from runs on a AMD Athlon 64 X2 (2.0 GHz).*

2. Check that the *collision count* is 0, 1, or 2 and that the *subnet prefix* is equal to the *subnet prefix* of the address (not the link-local address but the full IPv6 address). The CGA++ verification fails if either check fails.
3. Read the security parameter *sec* from the three leftmost bits of the *interface identifier* of the address (*sec* is an unsigned 3-bit integer).
4. Concatenate the *modifier*, *subnet prefix* and the encoded public-key. Execute the hash algorithm on the concatenation. Check if the most significant $16 \times sec$ bits of the result are zero. The CGA++ verification fails if the check fails.
5. Concatenate the encoded public-key and the signature. Execute the hash algorithm on the concatenation and compare the output with the *interface identifier*. The differences in the two reserved bits and three bits for *sec* are ignored. If the 64-bit values differ (other than in the five ignored bits), the CGA++ verification fails.

The address verification starts with the usual checks, similar to CGA, in the IPv6 address of the node to be verified. The signature is verified, then the *modifier*, *collision count* and *subnet prefix* are extracted. Note that, compared to CGA, CGA++ does extra authenticity checks using the signature of the address generator; in order to verify an address only the signature, public-key and the address are needed.

6 Analysis of CGA++

6.1 Security of CGA++

We analyze CGA++ in a similar fashion as we did for CGA in Section 4. With the help of digital signatures, we eliminate the lack of authentication in the verification process. Including the *subnet prefix* in both domains of Hash1 and Hash2 reduces the scope of a time-memory trade-off attack to a specific network. The following lemma introduces the computational demand for impersonation, again the proof follows from our security framework (cf. Section 4.2).

RSA x-bit key	Signature time	Log_2 of the signature time	Verification time	Log_2 of the verification time
512	707	9.5	35	5.1
768	1338	10.4	40	5.3
1024	1910	10.9	47	5.5
1536	4432	12.1	69	6.1
2048	7812	12.9	89	6.4

Table 3. *Signature and verification time expressed in SHA-1 hash function evaluations for different RSA key sizes.*

Lemma 3. *Given a network, assume the addresses are generated and verified by CGA++ with security parameter sec. Let S denote the time required to compute a signature expressed in hash function evaluations and assume $S < 2^{16}$. Then, the number of required hash function evaluations needed for impersonation of a specific node is*

$$T_A = \begin{cases} 2^{59} \cdot (1+S) & \text{if } sec = 0 \\ 2^{59+16\times sec} + 2^{59}(1+S) & \text{if } sec > 0. \end{cases}$$

6.2 Attack Costs and the Efficiency of CGA++

In order to make a comparison with CGA, the timing results to measure the computational cost of signing messages in terms of hash function evaluations from ECRYPT Benchmarking of Cryptographic Systems (eBACS) [15] are used. From now on, we assume the use of the RSA [16] public-key signature scheme because this is the default in the RFC [9]. Note that CGA++ is independent from the signature scheme used. The benchmark data regarding measurements of the signature scheme are stated in Table 2(a) for different key sizes and the benchmarks on the same architecture using the SHA-1 hash digest are stated in Table 2(b).

Address Generation. Due to the use of digital signatures, moving from CGA to CGA++, T_1 increases from 1 to $(1+S)$ and T_2 remains equal to 1. This increase in time is only significant when $sec = 0$ as $T_G = 2^{16\times sec} + S + 1$. Assuming the node uses a 1024-bit RSA key, the time increases from one hash function evaluations in CGA to $T_1 \approx 2^{10.9}$ hash function evaluations in CGA++ (see Table 3). For $sec > 0$, the time increase is negligible as this is dominated by the time required to compute the hash extensions.

Address Renewal. In CGA++, the address renewal time is equivalent to the time needed for address generation to resist the global time-memory trade-off attack. This is a drawback compared to the constant amount of time needed by CGA. Assume a mobile node does not have much computation power, say

five times less compared to a more powerful machine. The address renewal time is less than a second when using $sec = 1$. When $sec > 1$, the values of sec which are currently impractical (cf. Section 4.4), we anticipate the fact that the performance of mobile nodes (capable of performing cryptographic operations) will increase accordingly in the future following Moore's law, which would reduce the efficiency problem significantly (cf. Section 2).

Address Verification. In both CGA and CGA++, address verification takes a constant amount of hash function evaluations. In CGA, $T_V = 1$, whereas in CGA++ this amount is increased with a signature verification: $T_V = 1 + S$. Fortunately, the signature verification time is shorter compared to the time needed to sign a message, but it still consumes the same number of CPU cycles as 47 hash function evaluations when using 1024-bit RSA keys, see Table 3. This constant increase is tolerable considering the efficiency of evaluating hash functions in practice.

Impersonation. The time needed for impersonation in CGA++ is roughly equivalent to the time needed in CGA when $sec > 1$, not taking the possibility of mounting the time-memory trade-off attack into account for CGA, see Lemma 1 and 3. The increase of time needed for impersonation, due to the use of digital signatures, becomes negligible with respect to the hash extension time. However, when no hash extensions are used, the digital signature time is significant. Assuming the use of 1024-bit RSA keys, an attacker would need 2^{59} hash function evaluations using CGA, whereas this value increases to $2^{69.9}$ in CGA++.

6.3 Comparison of CGA++ with CGA

Table 4 summarizes the comparison between CGA and CGA++. The overall efficiency decreases when moving from CGA to CGA++. The time needed to generate a new and verify a current address is increased by a constant amount of time, whereas the time needed to renew an address increases exponentially when hash extensions are used. The security of CGA++ is improved compared to CGA. The global time-memory trade-off attack is no longer possible, increasing the security level against impersonation attacks. Moreover, an additional authentication mechanism is introduced by using digital signatures inside the protocol. The constant amount of loss in efficiency and gain of security, when no hash extensions are used, are due to the additional computation needed for signing and verifying digital signatures.

7 Compatibility and Applications

To facilitate its adoption, it is desirable to design a protocol that is compatible with the current schemes. Hence, when designing CGA++, one design criterion was to adhere to CGA as closely as possible. CGA offers features for protocols that require self-certified address generation and verification, where the

	CGA	CGA++
Time to generate a new address when $s = 0$	1	$1 + 2^{10.9}$
Time to generate a new address when $s > 0$	$2^{16 \times s} + 1$	$2^{16 \times s} + 1 + 2^{10.9}$
Time to verify an address when $s = 0$	1	$1 + 2^{5.5}$
Time to verify an address when $s > 0$	2	$2 + 2^{5.5}$
Impersonation time when $s = 0$	2^{59}	$2^{69.9}$
Impersonation time when $s > 0$	2^{59}	$2^{59+16 \times s} + 2^{69.9}$
Time to renew the address when moving to a different network when $s = 0$	1	$1 + 2^{10.9}$
Time to renew the address when moving to a different network when $s > 0$	1	$2^{16 \times s} + 1 + 2^{10.9}$
Resistance against the global time-memory trade-off attack	No	Yes
Authentication mechanism inside the verification protocol	No	Yes

Table 4. *Comparison between CGA and CGA++ for IPv6 using a* 1024*-bit RSA key. All timings are expressed in hash function evaluations. The parameter sec* = *s is the security parameter used for hash extensions.*

nodes are assumed to be capable of signing messages as they are equipped with public/private-key pairs. Therefore, our main contribution to the current design, using digital signatures, does not harm the compatibility of CGA++ because the rest of the protocol is nearly the same as CGA.

The Secure Neighbor Discovery [3], Shim6 [4] and IPv6 mobility support protocol [5] are the main protocols using CGA. The common feature of these protocols is to use CGA to prove address ownership and continue to sign additional data with the corresponding private key of the CGA. This is supported by CGA++ as well.

8 Conclusion

In this work, we have presented a detailed security/efficiency analysis of CGA together with a proposal to solve some security problems and limitations related to self-certifying address generation and verification in CGA. This new protocol, which is very similar to and based on the ideas of CGA, is called CGA++. The global time-memory trade-off attack, which eliminates the effect of hash extensions in the long run for CGA, is no longer possible. CGA++ has an efficiency drawback in that the address renewal costs as much as address generation. However, we believe that this is tolerable; the computational capabilities of mobile nodes (able to perform cryptographic operations) increase with the progress of technology and the current used hash extension values are still practical. As another improvement, we have introduced the use of digital signatures in the address generation and verification process, which provides authentication in

the protocol and eliminates the effect of replay attacks. Although this leads to an increase in time required for address generation and verification, it increases the security of the system, especially when no hash extensions are used. We believe that, in many ways, CGA++ is a nice practical alternative to CGA, e.g. in IPv6.

Acknowledgments. We would like to thank Tuomas Aura for sharing detailed information related to CGA. We would also like to thank the anonymous reviewers of ISC 2009 for their insightful and helpful comments.

References

1. Aura, T.: Cryptographically Generated Addresses (CGA). In: ISC. Volume 2851 of LNCS. (2003) 29–43
2. Aura, T., Roe, M.: Strengthening Short Hash Values. http://research.microsoft.com/en-us/um/people/tuomaura/misc/aura-roe-submission.pdf
3. Arkko, J., Kempf, J., Zill, B., Nikander, P.: SEcure Neighbor Discovery (SEND). RFC 3971, IETF, http://www.ietf.org/rfc/rfc3971.txt (March 2005)
4. Nordmark, E., Bagnulo, M.: Multihoming L3 Shim Approach. http://tools.ietf.org/html/draft-nordmark-multi6dt-shim-00.txt (July 2005)
5. Johnson, D., Perkins, C., Arkko, J.: Mobility Support in IPv6. RFC 3775, IETF, http://www.ietf.org/rfc/rfc3775.txt (June 2004)
6. O'Shea, G., Roe, M.: Child-proof Authentication for MIPv6 (CAM). Computer Communication Review **31**(2) (2001) 4–8
7. Nikander, P.: A Scalable Architecture for IPv6 Address Ownership, Internet Draft (2001)
8. Montenegro, G., Castelluccia, C.: Statistically Unique and Cryptographically Verifiable (SUCV) Identifiers and Addresses. In: NDSS, The Internet Society (2002)
9. Aura, T.: Cryptographically Generated Addresses (CGA). RFC 3972, IETF, http://www.ietf.org/rfc/rfc3972.txt (March 2005)
10. Hinden, R., Deering, S.: Internet Protocol Version 6 Addressing Architecture. RFC 4291, IETF, http://www.ietf.org/rfc/rfc4291.txt (February 2006)
11. Hinden, R., Deering, S., Nordmark, E.: IPv6 Global Unicast Address Format. RFC 3587, IETF, http://www.ietf.org/rfc/rfc3587.txt (August 2003)
12. National Institute of Standards and Technology: Secure hash standard. FIPS 180-1, NIST (April 1995)
13. Bagnulo, M., Arkko, J.: Support for Multiple Hash Algorithms in Cryptographically Generated Addresses (CGAs). RFC 4982, IETF, http://www.ietf.org/rfc/rfc4982.txt (July 2007)
14. OpenSSL: The Open Source Toolkit for SSL/TLS. http://www.openssl.org/ (2008)
15. Bernstein, D.J., Lange, (editors), T.: eBACS: ECRYPT Benchmarking of Cryptographic Systems. http://bench.cr.yp.to (accessed 7 January 2009)
16. Rivest, R., Shamir, A., Adleman, L.: A method for obtaining digital signatures and public key cryptosystems. Communications of the ACM (February 1978) 42–111