Analysis and Optimization of Cryptographically Generated Addresses (CGA)
Revisiting Self-Certifying Address Generation and Verification

Joppe Bos and Onur Özen

Laboratory for Cryptologic Algorithms
École Polytechnique Fédérale de Lausanne

January 2009

Project Supervisor: Prof. Jean-Pierre Hubaux
Outline

1 Problem statement
2 CGA for IPv6
   - Outline
   - Security of CGA
   - Several Attacks
   - Efficiency
3 CGA++
   - Design goals
   - Prevent current attacks
   - Address Generation
   - Address Verification
   - Security
4 Conclusions
   - Summary
Problem outline

Problem:
The need for the nodes to be able to generate their own address and verify the ones from others without relying on any global trusted authority.
Problem: The need for the nodes to be able to generate their own address and verify the ones from others without relying on any global trusted authority.

Solution: Using self-certifying addresses, which allows hosts and domains to prove they have the address they claim to have.
Problem:
The need for the nodes to be able to generate their own address and verify the ones from others without relying on any global trusted authority.

Solution:
Using self-certifying addresses, which allows hosts and domains to prove they have the address they claim to have.

How:
The name of the object is the public-key (or, for convenience the hash of the public-key) that corresponds to that object.
Motivation

The Impact

Self-certifying addresses are widely used and standardized.
Motivation

The Impact

Self-certifying addresses are widely used and standardized.

Examples:

- CGA (Cryptographically Generated Addresses) for IPv6
  

- HIP (Host Identity Protocol)
  
Motivation

The Impact
Self-certifying addresses are widely used and standardized.

Examples:
- CGA (Cryptographically Generated Addresses) for IPv6
- HIP (Host Identity Protocol)

What is missing?
The standards do not give concrete analysis of the security properties and the efficiency of the underlying cryptologic primitives.
The focus of the project

In this project,

- One of the crucial components of the IPv6 world is targeted: CGA for address generation and verification.

CGA for IPv6 is analyzed from security and efficiency point of view.

- Security: Formalization of the security of CGA and generation of new attack models.
- Efficiency: Estimation of the cost of address generation/verification and possible attacks.

A new and more secure protocol is designed: CGA++.

- Resistant against certain type of attacks which are possible against CGA.
- Better authentication.
- Higher security when no hash extensions are used.
The focus of the project

In this project,

- One of the crucial components of the IPv6 world is targeted: CGA for address generation and verification.
- CGA for IPv6 is analyzed from security and efficiency point of view.
The focus of the project

In this project,

- One of the crucial components of the IPv6 world is targeted: CGA for address generation and verification.

- CGA for IPv6 is analyzed from security and efficiency point of view. **Security:** Formalization of the security of CGA and generation of new attack models.

- A new and more secure protocol is designed: CGA++.
The focus of the project

In this project,

- One of the crucial components of the IPv6 world is targeted: CGA for address generation and verification.

- CGA for IPv6 is analyzed from security and efficiency point of view.
  - **Security:** Formalization of the security of CGA and generation of new attack models.
  - **Efficiency:** Estimation of the cost of address generation/verification and possible attacks.

A new and more secure protocol is designed: CGA++. It is resistant against certain type of attacks which are possible against CGA and offers better authentication and higher security when no hash extensions are used.
The focus of the project

In this project,

- One of the crucial components of the IPv6 world is targeted: CGA for address generation and verification.
- CGA for IPv6 is analyzed from security and efficiency point of view. **Security:** Formalization of the security of CGA and generation of new attack models. **Efficiency:** Estimation of the cost of address generation/verification and possible attacks.
- A new and more secure protocol is designed: CGA++. 
The focus of the project

In this project,

- One of the crucial components of the IPv6 world is targeted: CGA for address generation and verification.
- CGA for IPv6 is analyzed from security and efficiency point of view. **Security:** Formalization of the security of CGA and generation of new attack models. **Efficiency:** Estimation of the cost of address generation/verification and possible attacks.
- A new and more secure protocol is designed: CGA++.  
  - Resistant against certain type of attacks which are possible against CGA
  - Better authentication
  - Higher security when no hash extensions are used
The notion of a self-certifying name is straight-forward:

The public-key itself:

![Diagram](image.png)
The notion of a self-certifying name is straight-forward:

The public-key itself:

- Advantages
  - Each user has a unique ID
The notion of a self-certifying name is straight-forward:

The public-key itself:

- **Advantages**
  - Each user has a unique ID
  - Security depends on underlying Public-Key Cryptosystem
The notion of a self-certifying name is straight-forward:

The public-key itself:

- **Advantages**
  - Each user has a unique ID
  - Security depends on underlying Public-Key Cryptosystem

- **Disadvantages**
  - Standards require short IDs
The notion of a self-certifying name is straightforward:

The public-key itself:

Advantages:
- Each user has a unique ID
- Security depends on underlying Public-Key Cryptosystem

Disadvantages:
- Standards require short IDs
- Short keys can be broken in seconds on an ordinary PC!
The notion of a self-certifying name is straight-forward:

Or, for convenience the hash of the public-key

![](flowchart.png)
The notion of a self-certifying name is straight-forward:

Or, for convenience the hash of the public-key

Advantages
- Short IDs are possible
The notion of a self-certifying name is straight-forward:

Or, for convenience the hash of the public-key

Public key -> Hash function -> ID

**Advantages**
- Short IDs are possible

**Disadvantages**
- Security depends on the underlying hash function
The notion of a self-certifying name is straight-forward:

Or, for convenience the hash of the public-key

- **Advantages**
  - Short IDs are possible

- **Disadvantages**
  - Security depends on the underlying hash function
  - The problems begin (new attacks)!!
CGA for IPv6

Outline
- Security of CGA
- Several Attacks
- Efficiency

Problem statement
CGA for IPv6
CGA++
Conclusions

Diagram:
- Hash2
  - SHA-1
  - Modifier
  - Zero
  - Zero
  - Subnet Prefix
  - Collision Count
  - Public Key

- SHA-1

- Hash1
  - Sec (3 bits)
  - u,g (2 bits)
  - 59 bits
  - Subnet Prefix (64 bits)
  - Interface ID (64 bits)
Set modifier to a random 128-bit value.
Address Generation

1. Set modifier to a random 128-bit value.
2. Concatenate the modifier and the encoded PK.

Outline
- Security of CGA
- Several Attacks
- Efficiency

Problem statement
- CGA for IPv6
- CGA++
- Conclusions

Title: Address Generation

Diagram:
- Hash2
  - SHA-1
  - Modifier
  - Zero
  - Subnet Prefix
  - Zero
  - Collision
  - Count
  - Public Key

- SHA-1

- Hash1
  - Sec (3 bits)
  - u,g (2 bits)
  - 59 bits

- Subnet Prefix (64 bits)
- Interface ID (64 bits)
Address Generation

1. Set *modifier* to a random 128-bit value.
2. Concatenate the *modifier* and the encoded *PK*.
3. Execute SHA-1 algorithm. The leftmost 112 bits of the result are Hash2.

**Outline**
- Problem statement
- CGA for IPv6
- CGA++
- Conclusions
- Security of CGA
- Several Attacks
- Efficiency

**Address Generation**

```
Hash2

SHA-1

Modifier
Zero
Subnet Prefix
Zero
Collision Count
Public Key

SHA-1

Hash1

Sec (3 bits) u,g (2 bits) 59 bits

Subnet Prefix (64 bits) Interface ID (64 bits)
```
Address Generation

16*sec bits are ZERO ?

1. Set \textit{modifier} to a random 128-bit value.
2. Concatenate the \textit{modifier} and the encoded \textit{PK}.
3. Execute SHA-1 algorithm. The leftmost 112 bits of the result are \textit{Hash2}.
4. Compare the $16 \times \text{Sec}$ leftmost bits of \textit{Hash2} with 0. If they are all zero, continue with Step (5). Otherwise, increment the \textit{modifier} and go back to Step (2).
Address Generation

1. Set \textit{modifier} to a random 128-bit value.
2. Concatenate the \textit{modifier} and the encoded \textit{PK}.
3. Execute SHA-1 algorithm. The leftmost 112 bits of the result are Hash2.
4. Compare the $16 \times \text{Sec}$ leftmost bits of Hash2 with 0. If they are all zero, continue with Step (5). Otherwise, increment the \textit{modifier} and go back to Step (2).
5. Set the \textit{collision count} to zero.

Subnet Prefix (64 bits)  Interface ID (64 bits)

Sec (3 bits)  u,g (2 bits)  59 bits
Address Generation

1. Set modifier to a random 128-bit value.
2. Concatenate the modifier and the encoded PK.
3. Execute SHA-1 algorithm. The leftmost 112 bits of the result are Hash2.
4. Compare the 16 × Sec leftmost bits of Hash2 with 0. If they are all zero, continue with Step (5). Otherwise, increment the modifier and go back to Step (2).
5. Set the collision count to zero.
6. Concatenate the modifier, subnet prefix, collision count and encoded PK values.
1. Set `modifier` to a random 128-bit value.
2. Concatenate the `modifier` and the encoded `PK`.
3. Execute SHA-1 algorithm. The leftmost 112 bits of the result are `Hash2`.
4. Compare the $16 \times \text{Sec}$ leftmost bits of `Hash2` with 0. If they are all zero, continue with Step (5). Otherwise, increment the `modifier` and go back to Step (2).
5. Set the `collision count` to zero.
6. Concatenate the `modifier`, `subnet prefix`, `collision count` and encoded `PK` values.
7. Execute SHA-1. The leftmost 64 bits of the result are `Hash1`.

---

**Address Generation**

16*sec bits are ZERO?

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Zero</th>
<th>Sec (3 bits)</th>
<th>59 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero</td>
<td>Zero</td>
<td>u,g (2 bits)</td>
<td></td>
</tr>
<tr>
<td>Subnet Prefix</td>
<td>Collision Count</td>
<td>PublicKey</td>
<td></td>
</tr>
</tbody>
</table>

SHA-1

```
Hash2
```

SHA-1

```
Hash1
```

Subnet Prefix (64 bits)  | Interface ID (64 bits)
1. Set modifier to a random 128-bit value.
2. Concatenate the modifier and the encoded PK.
3. Execute SHA-1 algorithm. The leftmost 112 bits of the result are Hash2.
4. Compare the $16 \times \text{Sec}$ leftmost bits of Hash2 with 0. If they are all zero, continue with Step (5). Otherwise, increment the modifier and go back to Step (2).
5. Set the collision count to zero.
6. Concatenate the modifier, subnet prefix, collision count and encoded PK values.
7. Execute SHA-1. The leftmost 64 bits of the result are Hash1.
8. Form an interface identifier by setting $u, g$ in Hash1 both to 1 and the three leftmost bits to Sec.
Address Generation

1. Set \( \text{modifier} \) to a random 128-bit value.
2. Concatenate the \( \text{modifier} \) and the encoded \( PK \).
3. Execute SHA-1 algorithm. The leftmost 112 bits of the result are \( \text{Hash2} \).
4. Compare the \( 16 \times \text{Sec} \) leftmost bits of \( \text{Hash2} \) with 0. If they are all zero, continue with Step (5). Otherwise, increment the \( \text{modifier} \) and go back to Step (2).
5. Set the \( \text{collision count} \) to zero.
6. Concatenate the \( \text{modifier}, \text{subnet prefix}, \text{collision count} \) and encoded \( PK \) values.
7. Execute SHA-1. The leftmost 64 bits of the result are \( \text{Hash1} \).
8. Form an \( \text{interface identifier} \) by setting \( u, g \) in \( \text{Hash1} \) both to 1 and the three leftmost bits to \( \text{Sec} \).
9. Concatenate the \( \text{subnet prefix} \) and \( \text{interface identifier} \) to form a 128-bit IPv6 address.
Address Generation

1. Set modifier to a random 128-bit value.
2. Concatenate the modifier and the encoded PK.
3. Execute SHA-1 algorithm. The leftmost 112 bits of the result are Hash2.
4. Compare the $16 \times \text{Sec}$ leftmost bits of Hash2 with 0. If they are all zero, continue with Step (5). Otherwise, increment the modifier and go back to Step (2).
5. Set the collision count to zero.
6. Concatenate the modifier, subnet prefix, collision count and encoded PK values.
7. Execute SHA-1. The leftmost 64 bits of the result are Hash1.
8. Form an interface identifier by setting $u, g$ in Hash1 both to 1 and the three leftmost bits to Sec.
9. Concatenate the subnet prefix and interface identifier to form a 128-bit IPv6 address.
10. If an address collision is detected, increment the collision count and go back to step (6). After three collisions, stop.
Assume we are not using hash extensions.

### Impersonation

Given a network, assume the addresses are generated and verified with basic CGA. Then, the number of operations required for impersonation of a specific node is $\mathcal{O}(2^{62+t})$ hash function evaluations. Here, $\mathcal{O}(2^t)$ is the required time for generating valid public/private-key pairs in terms of hash function evaluations.
Security of CGA for IPv6

Impersonation
Given a network, assume the addresses are generated and verified by CGA for IPv6 with security parameter \( sec = s \). Then, the number of operations required for impersonation of a specific node is \( O(2^{59+16 \times s}) \) hash function evaluations.
Design Choices for CGA for IPv6

Hash2

SHA−1

Modifier
Zero
Subnet Prefix
Zero
Collision Count
Public Key

SHA−1

Hash1

Sec (3 bits)
u,g (2 bits)
59 bits

Subnet Prefix (64 bits) Interface ID (64 bits)

Design Rationale

- The modifier value

Easier to generate Hash2
Subnet prefix (why zero for Hash2?)
Provide Mobility
Collision count (why zero for Hash2?)
Provide Efficiency
Public key (why not zero for Hash2?)
Prevent Stealing
Modifiers
Design Choices for CGA for IPv6

Hash2

SHA-1

Modifier
Zero
Subnet Prefix
Zero
Collision Count
Public Key

SHA-1

Hash1

Sec (3 bits)
u,g (2 bits)
59 bits

Subnet Prefix (64 bits)  Interface ID (64 bits)

Design Rationale

- The modifier value
  - Easier to generate Hash2
Design Choices for CGA for IPv6

Design Rationale

- The modifier value
  Easier to generate Hash2

- Subnet prefix (why zero for Hash2?)
Design Choices for CGA for IPv6

Design Rationale

- The modifier value
  Easier to generate Hash2

- Subnet prefix (why zero for Hash2?) Provide Mobility
Design Choices for CGA for IPv6

Design Rationale

- The modifier value
  Easier to generate Hash2

- Subnet prefix (why zero for Hash2?) Provide Mobility

- Collision count (why zero for Hash2?)

Outline
- Problem statement
- CGA for IPv6
- CGA++
- Conclusions

Outline
- Security of CGA
- Several Attacks
- Efficiency
Design Choices for CGA for IPv6

Hash2

SHA−1

Modifier  Zero  Zero  Public Key
Subnet Prefix  Collision Count

SHA−1

Hash1

Sec (3 bits)  u,g (2 bits)  59 bits
Subnet Prefix (64 bits)  Interface ID (64 bits)

Design Rationale

- The modifier value
  Easier to generate Hash2
- Subnet prefix (why zero for Hash2?) Provide Mobility
- Collision count (why zero for Hash2?) Provide Efficiency
Design Choices for CGA for IPv6

Design Rationale

- The modifier value
  Easier to generate Hash2
- Subnet prefix (why zero for Hash2?) Provide Mobility
- Collision count (why zero for Hash2?) Provide Efficiency
- Public key (why not zero for Hash2?)

Diagram:

```
<table>
<thead>
<tr>
<th>Modifier</th>
<th>Zero</th>
<th>Zero</th>
<th>Public Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subnet Prefix</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Collision Count</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Hash2

SHA-1

Hash1

Subnet Prefix (64 bits) Interface ID (64 bits)

59 bits

Sec (3 bits) u,g (2 bits)
Design Choices for CGA for IPv6

Design Rationale

- The modifier value
  Easier to generate Hash2

- Subnet prefix (why zero for Hash2?) Provide Mobility

- Collision count (why zero for Hash2?) Provide Efficiency

- Public key (why not zero for Hash2?) Prevent Stealing Modifiers
A Time-Memory Trade-Off Attack on CGA for IPv6

Generation of Hash2 is independent of the subnet prefix

Why?
A Time-Memory Trade-Off Attack on CGA for IPv6

Generation of Hash2 is independent of the subnet prefix

**Why?**

- Help mobile nodes
A Time-Memory Trade-Off Attack on CGA for IPv6

Generation of Hash2 is independent of the subnet prefix

**Why?**

- Help mobile nodes
  - They change subnet prefix often, avoid computing Hash2 over and over again
  - Mobile nodes do not have much computation power.
A Time-Memory Trade-Off Attack on CGA for IPv6

Generation of Hash2 is independent of the subnet prefix

**Why?**

- Help mobile nodes
  - They change subnet prefix often, avoid computing Hash2 over and over again
  - Mobile nodes do not have much computation power.
- This helps an attacker as well!
A Time-Memory Trade-Off Attack on CGA for IPv6

Time-Memory Trade-Off Attack

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

$$x \leq 2^{59 - \min(n_i)}$$

In other words, $x$ is independent of the security parameter $sec$. The storage requirement is $2^{33 - \min(n_i)}$ Gbyte where $\min(n_i)$ denotes the smallest value $n_i$. 
A Time-Memory Trade-Off Attack on CGA for IPv6

What is the storage size of the database?

Huge!! \(2^{59 - n_i \cdot 2^7} = 2^{33 - n_i}\) GB.

Assume a network of size \(2^{13} = 8192\) nodes
\(\Rightarrow\) 1024 TB of storage
(storing naively).

But, it is not impractical!

Solution
Obvious solution: Let \(\text{Hash2}\) depend on the current network, add the subnet-prefix to the domain of \(\text{Hash2}\).

Trade efficiency for security, more about this later.
A Time-Memory Trade-Off Attack on CGA for IPv6

What is the storage size of the database?

Huge!! \(2^{59-n_i} \cdot 2^7\) bits = \(2^{33-n_i}\) GB.
What is the storage size of the database?

Huge!! $2^{59-n_i} \cdot 2^7$ bits $= 2^{33-n_i}$ GB.
Assume a network of size $2^{13} = 8192$ nodes $\Rightarrow$ 1024 TB of storage (storing naively).
A Time-Memory Trade-Off Attack on CGA for IPv6

What is the storage size of the database?

Huge!! \(2^{59-n_i} \cdot 2^7\) bits = \(2^{33-n_i}\) GB.
Assume a network of size \(2^{13}\) = 8192 nodes \(\Rightarrow\) 1024 TB of storage (storing naively). But, it is not impractical!
What is the storage size of the database?

Huge!! $2^{59-n_i} \cdot 2^7$ bits $= 2^{33-n_i}$ GB.

Assume a network of size $2^{13} = 8192$ nodes $\Rightarrow$ 1024 TB of storage (storing naively). But, it is not impractical!

Solution

Obvious solution: Let Hash2 depend on the current network, add the subnet-prefix to the domain of Hash2. Trade efficiency for security, more about this later.
Garbage Attack

The Idea

- Ignore the work factor of generating valid public/private-key pairs.
- Impose random data as the public-key of the attacker and make use of the lack of an authentication mechanism in CGA.
Garbage Attack

The Idea

- Ignore the work factor of generating valid public/private-key pairs.
- Impose random data as the public-key of the attacker and make use of the lack of an authentication mechanism in CGA.

The Drawback

- In practice, the impact is limited since the “attacking” node would fail to present a private-key.
- It can not send authenticated message to other nodes.
Replay Attack

Replaying messages

- Select a random node in the network, sniff and store signed messages.
Replay Attack

Replaying messages

- Select a random node in the network, sniff and store signed messages.
- Ask this node to verify its address → receive the modifier value and public-key.

The attacker can not send new messages, he has no access to the private-key, but can replay the stored signed messages. Verification of these messages by other nodes will be OK!
Replay Attack

Replaying messages

- Select a random node in the network, sniff and store signed messages.
- Ask this node to verify its address → receive the modifier value and public-key.
- Then, by switching network, generate a “verifiable” address.
Replay Attack

Replaying messages

- Select a random node in the network, sniff and store signed messages.
- Ask this node to verify its address $\rightarrow$ receive the modifier value and public-key.
- Then, by switching network, generate a “verifiable” address.
- The cost for an attacker to generate an address becomes only $O(1)$ instead of $O(2^{16 \times sec})$. 
Replay Attack

Replaying messages

- Select a random node in the network, sniff and store signed messages.
- Ask this node to verify its address → receive the modifier value and public-key.
- Then, by switching network, generate a “verifiable” address.
- The cost for an attacker to generate an address becomes only $O(1)$ instead of $O(2^{16 \times sec})$.
- The attacker cannot send new messages, he has no access to the private-key, but can replay the stored signed messages. Verification of these messages by other nodes will be OK!
Use the Collision Count

Impersonation with collision count > 0

- The attacker can search networks for nodes with a nonzero collision count.
- Use the valid modifier/public-key of this node (with collision count set to zero) to generate an existing address.
Use the Collision Count

Impersonation with collision count $> 0$

- The attacker can search networks for nodes with a nonzero collision count.
- Use the valid modifier/public-key of this node (with collision count set to zero) to generate an existing address.
- Nevertheless, the probability of having a collision in the addresses is low.
Use the Collision Count

Impersonation with collision count $> 0$

- The attacker can search networks for nodes with a nonzero collision count.
- Use the valid modifier/public-key of this node (with collision count set to zero) to generate an existing address.
- Nevertheless, the probability of having a collision in the addresses is low.
- 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 at the same time.
## Efficiency of CGA for IPv6

<table>
<thead>
<tr>
<th>sec</th>
<th># calls</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$2^1 \times 16$</td>
<td>$2^{1 \times 16} - 18.5$ seconds</td>
</tr>
<tr>
<td>2</td>
<td>$2^2 \times 16$</td>
<td>$2^{2 \times 16} - 18.5$ seconds</td>
</tr>
<tr>
<td>3</td>
<td>$2^3 \times 16$</td>
<td>$2^{3 \times 16} - 18.5$ seconds</td>
</tr>
<tr>
<td>4</td>
<td>$2^4 \times 16$</td>
<td>$2^{4 \times 16} - 18.5$ seconds</td>
</tr>
<tr>
<td>5</td>
<td>$2^5 \times 16$</td>
<td>$2^{5 \times 16} - 18.5$ seconds</td>
</tr>
<tr>
<td>6</td>
<td>$2^6 \times 16$</td>
<td>$2^{6 \times 16} - 18.5$ seconds</td>
</tr>
<tr>
<td>7</td>
<td>$2^7 \times 16$</td>
<td>$2^{7 \times 16} - 18.5$ seconds</td>
</tr>
</tbody>
</table>

Table: The Required Time to Generate an Address with CGA for IPv6 for Different Security Parameters.
Efficiency of CGA for IPv6

Table: The Required Time to Generate an Address with CGA for IPv6 for Different Security Parameters.

<table>
<thead>
<tr>
<th>sec</th>
<th># calls</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$2^{1\times16}$</td>
<td>$2^{1\times16-18.5}$ seconds</td>
</tr>
<tr>
<td>2</td>
<td>$2^{2\times16}$</td>
<td>$2^{2\times16-18.5}$ seconds</td>
</tr>
<tr>
<td>3</td>
<td>$2^{3\times16}$</td>
<td>$2^{3\times16-18.5}$ seconds</td>
</tr>
<tr>
<td>4</td>
<td>$2^{4\times16}$</td>
<td>$2^{4\times16-18.5}$ seconds</td>
</tr>
<tr>
<td>5</td>
<td>$2^{5\times16}$</td>
<td>$2^{5\times16-18.5}$ seconds</td>
</tr>
<tr>
<td>6</td>
<td>$2^{6\times16}$</td>
<td>$2^{6\times16-18.5}$ seconds</td>
</tr>
<tr>
<td>7</td>
<td>$2^{7\times16}$</td>
<td>$2^{7\times16-18.5}$ seconds</td>
</tr>
</tbody>
</table>

Conclusion: Setting the security parameter to zero or one are currently practical values.
CGA++, Design goals

Make CGA++
CGA++, Design goals

Make CGA++

- without hash extensions at least as secure as basic-CGA
CGA++, Design goals

Make CGA++

- without hash extensions at least as secure as basic-CGA
- resistant against the TMTO attack
  - What is the impact on the efficiency (especially for mobile nodes)?
CGA++, Design goals

Make CGA++

- without hash extensions at least as secure as basic-CGA
- resistant against the TMTO attack
  - What is the impact on the efficiency (especially for mobile nodes)?
- resistant against the garbage/collision count/replay attack
**CGA++, Design goals**

Make CGA++

- without hash extensions at least as secure as basic-CGA
- resistant against the TMTO attack
  - What is the impact on the efficiency (especially for mobile nodes)?
- resistant against the garbage/collision count/replay attack
- capable of authentication inside the verification part
CGA++, Design goals

Make CGA++

- without hash extensions at least as secure as basic-CGA
- resistant against the TMTO attack
  - What is the impact on the efficiency (especially for mobile nodes)?
- resistant against the garbage/collision count/replay attack
- capable of authentication inside the verification part
- as similar to CGA as possible
The Time-Memory Tradeoff Attack

Fix

Add the subnet prefix to the computation of the Hash2 value.
The Time-Memory Tradeoff Attack

Fix
Add the subnet prefix to the computation of the Hash2 value.

Efficiency
- Current practical security parameters are: $sec \in \{0, 1\}$.
- No change when $sec = 0$.
- When $sec = 1$, the time for address renewal is increased from 1 hash to $1 + 2^{16}$ hash evaluations.
  → Address renewal costs the same as address generation.
The Time-Memory Tradeoff Attack

**Fix**

Add the subnet prefix to the computation of the Hash2 value.

**Efficiency**

- Current practical security parameters are: $sec \in \{0, 1\}$.
- No change when $sec = 0$.
- When $sec = 1$, the time for address renewal is increased from 1 hash to $1 + 2^{16}$ hash evaluations.
  $\rightarrow$ Address renewal costs the same as address generation.

As we have seen address generation takes $\approx 0.2$ second on a modern CPU. Assume a mobile node is five times slower, it still takes $\leq 1$ second. For future hash extensions, we participate on the fact that mobile nodes will get faster as well.
Authentication

Idea

Introduce authentication inside the verification process. “Bind” the modifier value, subnet prefix and collision count to this public/private key pair.
Authentication

Idea

Introduce authentication inside the verification process. “Bind” the modifier value, subnet prefix and collision count to this public/private key pair.

How?

Sign these values in the computation of the Hash1 value.

This prevents almost all instances of the garbage/collision count/replay attack.
Idea

Introduce authentication inside the verification process. “Bind” the modifier value, subnet prefix and collision count to this public/private key pair.

How?

Sign these values in the computation of the Hash1 value.

This prevents almost all instances of the garbage/collision count/replay attack.

- Replay attack will only work when copying a current address in the same network → easy to detect.
Problem statement
CGA for IPv6
CGA++
Conclusions

Design goals
Prevent current attacks

**Address Generation**

Address Verification

Security

**Hash2**

H

<table>
<thead>
<tr>
<th>Public Key</th>
<th>Subnet Prefix (SP)</th>
<th>Modifier (m)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Public Key

Sign(SP,m,Collision Count)

H

**Hash1**

Sec (3 bits)

u,g (2 bits)

59 bits

Subnet Prefix (64 bits) | Interface ID (64 bits)
Address Generation

Design goals
Prevent current attacks

Address Generation
Address Verification
Security

Choose security parameter \( sec \in \{0, \ldots, 7\} \).
Choose security parameter $sec \in \{0, \ldots, 7\}$. 
Set the modifier value to a random 128-bit value.
**Address Generation**

1. Choose security parameter $\text{sec} \in \{0, \ldots, 7\}$.
2. Set the *modifier* value to a random 128-bit value.
3. Concatenate the *modifier*, *subnet prefix* and the encoded *public-key*. Execute the hash algorithm on the concatenation. Continue until $16 \times \text{sec}$ bits are zero by increasing the value of modifier.
Address Generation

1. Choose security parameter $sec \in \{0, \ldots, 7\}$.
2. Set the modifier value to a random 128-bit value.
3. Concatenate the modifier, subnet prefix and the encoded public-key. Execute the hash algorithm on the concatenation. Continue until $16 \times sec$ bits are zero by increasing the value of modifier.
4. Set the collision count value to zero.
5. Sign the modifier, collision count and subnet prefix with the private-key corresponding to used public-key.
Choose security parameter $sec \in \{0, \ldots, 7\}$.

2. Set the modifier value to a random 128-bit value.

3. Concatenate the modifier, subnet prefix and the encoded public-key. Execute the hash algorithm on the concatenation. Continue until $16 \times sec$ bits are zero by increasing the value of modifier.

4. Set the collision count value to zero.

5. Sign the modifier, collision count and subnet prefix with the private-key corresponding to used public-key.

6. Concatenate the encoded public-key and the signature values. Execute the hash algorithm on the concatenation. The most significant 64 bits of the result are $Hash1$. 

Subnet Prefix (64 bits) | Interface ID (64 bits)
Address Generation

1. Choose security parameter $sec \in \{0, \ldots, 7\}$.
2. Set the modifier value to a random 128-bit value.
3. Concatenate the modifier, subnet prefix and the encoded public-key. Execute the hash algorithm on the concatenation. Continue until $16 \times sec$ bits are zero by increasing the value of modifier.
4. Set the collision count value to zero.
5. Sign the modifier, collision count and subnet prefix with the private-key corresponding to used public-key.
6. Concatenate the encoded public-key and the signature values. Execute the hash algorithm on the concatenation. The most significant 64 bits of the result are Hash1.
7. Form an interface identifier by setting the two reserved bits in Hash1 both to 1 and three bits to sec.
Address Generation

1. Choose security parameter $\text{sec} \in \{0, \ldots, 7\}$.
2. Set the $\text{modifier}$ value to a random 128-bit value.
3. Concatenate the $\text{modifier}$, $\text{subnet prefix}$ and the encoded $\text{public-key}$. Execute the hash algorithm on the concatenation. Continue until $16 \times \text{sec}$ bits are zero by increasing the value of modifier.
4. Set the $\text{collision count}$ value to zero.
5. Sign the $\text{modifier}$, $\text{collision count}$ and $\text{subnet prefix}$ with the $\text{private-key}$ corresponding to used $\text{public-key}$.
6. Concatenate the encoded $\text{public-key}$ and the $\text{signature}$ values. Execute the hash algorithm on the concatenation. The most significant 64 bits of the result are $\text{Hash1}$.
7. Form an $\text{interface identifier}$ by setting the two reserved bits in $\text{Hash1}$ both to 1 and three bits to $\text{sec}$.
8. Concatenate the $\text{subnet prefix}$ and $\text{interface identifier}$ to form an 128-bit $\text{IPv6}$ address.
9. If an address collision is detected, increment the $\text{collision count}$ and go back to step (5) (up to 3 collisions).
Address Verification

Given the IPv6 address, the signature and the public-key of the node:

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

```
<table>
<thead>
<tr>
<th>Public Key</th>
<th>Subnet Prefix (SP)</th>
<th>Modifier(m)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Public Key</td>
<td>Sign(SP,m,Collision Count)</td>
<td></td>
</tr>
</tbody>
</table>
```

```
Hash2

H

H

Hash1

Sec (3 bits)  59 bits  u,g (2 bits)

Subnet Prefix (64 bits)  Interface ID (64 bits)
```
Address Verification

Given the IPv6 address, the signature and the public-key of the node

Verification

1. Verify the signature and obtain the modifier, collision count and subnet prefix values.
2. Check that the collision count value is 0, 1, or 2 and that the subnet prefix value is equal to the subnet prefix.
Address Verification

Given the IPv6 address, the signature and the public-key of the node

**Verification**

1. Verify the signature and obtain the modifier, collision count and subnet prefix values.
2. Check that the collision count value is 0, 1, or 2 and that the subnet prefix value is equal to the subnet prefix.
3. Read the security parameter sec. 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.
Address Verification

Given the IPv6 address, the signature and the public-key of the node

Hash2

H

Public Key  Subnet Prefix (SP)  Modifier(m)

Public Key  Sign(SP,m,Collision Count)

H

Hash1

Sec (3 bits)  u,g (2 bits)  59 bits

Subnet Prefix (64 bits)  Interface ID (64 bits)

Verification

1. Verify the signature and obtain the modifier, collision count and subnet prefix values.
2. Check that the collision count value is 0, 1, or 2 and that the subnet prefix value is equal to the subnet prefix.
3. Read the security parameter sec. Concatenate the modifier, subnet prefix and the encoded public-key. Execute the hash algorithm on the concatenation. Check if the most significant 16 × sec bits of the result are zero.
4. Concatenate the encoded public-key and the signature. Execute the hash algorithm on the concatenation and compare the output with the interface identifier. Differences in the two reserved bits and three bits for sec are ignored.
Security of CGA++

Impersonation

Given a network, assume the addresses are generated and verified by CGA++ with hash extension with security parameter $sec$. Let $S$ denote the time 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 $O(2^{59} \cdot (1 + S))$ when $sec = 0$ and $O(2^{59+16\times sec})$ when $sec \geq 1$. 
## Efficiency of CGA++

<table>
<thead>
<tr>
<th>RSA $x$-bit key</th>
<th>Signature time</th>
<th>$\log_2$ of the signature time</th>
<th>Verification time</th>
<th>$\log_2$ of the verification time</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>707</td>
<td>9.5</td>
<td>35</td>
<td>5.1</td>
</tr>
<tr>
<td>768</td>
<td>1338</td>
<td>10.4</td>
<td>40</td>
<td>5.3</td>
</tr>
<tr>
<td>1024</td>
<td>1910</td>
<td>10.9</td>
<td>47</td>
<td>5.5</td>
</tr>
<tr>
<td>1536</td>
<td>4432</td>
<td>12.1</td>
<td>69</td>
<td>6.1</td>
</tr>
<tr>
<td>2048</td>
<td>7812</td>
<td>12.9</td>
<td>89</td>
<td>6.4</td>
</tr>
</tbody>
</table>

**Table:** *The Signature and Verification Time Expressed in Hash Function (SHA-1) Evaluations for Different RSA Key Sizes.*
Efficiency of CGA++

<table>
<thead>
<tr>
<th>RSA x-bit key</th>
<th>Signature time</th>
<th>Log$_2$ of the signature time</th>
<th>Verification time</th>
<th>Log$_2$ of the verification time</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>707</td>
<td>9.5</td>
<td>35</td>
<td>5.1</td>
</tr>
<tr>
<td>768</td>
<td>1338</td>
<td>10.4</td>
<td>40</td>
<td>5.3</td>
</tr>
<tr>
<td>1024</td>
<td>1910</td>
<td>10.9</td>
<td>47</td>
<td>5.5</td>
</tr>
<tr>
<td>1536</td>
<td>4432</td>
<td>12.1</td>
<td>69</td>
<td>6.1</td>
</tr>
<tr>
<td>2048</td>
<td>7812</td>
<td>12.9</td>
<td>89</td>
<td>6.4</td>
</tr>
</tbody>
</table>

Table: The Signature and Verification Time Expressed in Hash Function (SHA-1) Evaluations for Different RSA Key Sizes.

Benchmark based on information taken from:

### Problem statement

**CGA for IPv6**

**CGA++**

**Conclusions**

**Summary**

<table>
<thead>
<tr>
<th></th>
<th>CGA</th>
<th>CGA++</th>
</tr>
</thead>
<tbody>
<tr>
<td>Security without hash extensions</td>
<td>59 bits</td>
<td>69.9 bits</td>
</tr>
<tr>
<td>Time to generate a new address when $s = 0$</td>
<td>$1$</td>
<td>$1 + 2^{10.9}$</td>
</tr>
<tr>
<td>Time to generate a new address when $s &gt; 0$</td>
<td>$2^{16 \times s} + 1$</td>
<td>$2^{16 \times s} + 1 + 2^{10.9}$</td>
</tr>
<tr>
<td>Time to verify an address when $s = 0$</td>
<td>$1$</td>
<td>$1 + 2^{5.5}$</td>
</tr>
<tr>
<td>Time to verify an address when $s &gt; 0$</td>
<td>$2$</td>
<td>$2 + 2^{5.5}$</td>
</tr>
<tr>
<td>Impersonation complexity when $s = 0$</td>
<td>$O(2^{59})$</td>
<td>$O(2^{69.9})$</td>
</tr>
<tr>
<td>Impersonation complexity when $s &gt; 0$</td>
<td>$O(2^{59})$</td>
<td>$O(2^{59 + 16 \times s})$</td>
</tr>
<tr>
<td>Time to renew the address when moving to a different network when $s = 0$</td>
<td>$1$</td>
<td>$1 + 2^{10.9}$</td>
</tr>
<tr>
<td>Time to renew the address when moving to a different network when $s &gt; 0$</td>
<td>$1$</td>
<td>$2^{16 \times s} + 1 + 2^{10.9}$</td>
</tr>
<tr>
<td>Authentication mechanism inside the verification protocol</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Table:** *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.*
Conclusions

Project

We

- analyzed the security of CGA
Conclusions

Project

We

- analyzed the security of CGA
- developed a security model for CGA-like protocols
  (in the report, not in the presentation)
Conclusions

Project

We

- analyzed the security of CGA
- developed a security model for CGA-like protocols (in the report, not in the presentation)
- introduced an improved version of CGA: CGA++
# Conclusions

## Project

We

- analyzed the security of CGA
- developed a security model for CGA-like protocols (in the report, not in the presentation)
- introduced an improved version of CGA: CGA++

**Advantages**

- Higher security when no hash extensions are used
- Introduced authentication in the verification process
- Resistant against replay/garbage/collision count attacks
- Resistant against global time-memory trade-off attack
Conclusions

Project

We

- analyzed the security of CGA
- developed a security model for CGA-like protocols (in the report, not in the presentation)
- introduced an improved version of CGA: CGA++

Advantages

- Higher security when no hash extensions are used
- Introduced authentication in the verification process
- Resistant against replay/garbage/collision count attacks
- Resistant against global time-memory trade-off attack

Disadvantage

- Efficiency: Renewal of addresses is as expensive as address generation
Conclusions

Experience

- Network security ≠ cryptography
- We learned to think more like network engineers
  - consider the performance of mobile nodes
  - design a protocol while taking the constraints of the current standards into account
- a good experience to see how crypto is applied in practice
Main references