Fast Integrity for Large Data

Go Yamamoto, Satoshi Oda, and Kazumaro Aoki

Information Sharing Platform Laboratories,

NTT Corporation

Overview

- An algorithm and its implementation for fast verification of data integrity.
- Delivers 6 Gbit/second on a single core of Athlon X2 @ 2GHz, 12 Gbit/second on a dual core implementation.
- Provable collision resistance under the hardness of integer factoring.
- Without random oracle / hash function.
- For large data $\sim$ 128 Mbyte or more.

Integrity Check for Various Purposes

Primitive	Type	Speed	Assumption
Poly1305-AES	MAC	1456 Mbit/GHz	AES
SHA-1	Hash	719 Mbit/GHz	SHA-1
VSH	Hash	9 Mbit/GHz	VSSR
Proposed	Hash list	3213 Mbit/GHz	Factoring

CPU type may differ for each figure.

Background

To check data integrity:

1. **TagGenerate**: **Data** $\rightarrow$ **Tag**
2. Preserve **Tag**
3. **IntegrityCheck**: $(\textbf{Data}^*, \textbf{Tag}) \rightarrow \mathtt{OK}/\mathtt{NG}$.

In some environment **IntegrityCheck** runs much frequently than **TagGenerate**. We want a fast algorithm for **IntegrityCheck**.

In the Conventional Design

IntegrityCheck is slower than **TagGenerate**. Because

TagGenerate(**Data**):

1. Compute hash function h(**Data**).

IntegrityCheck(**Data***, **Tag**):

1. **Tag**$^* \leftarrow$ **TagGenerate**(**Data***)
2. Compare **Tag** and **Tag***

We consider a new design.

A New Design: Asymmetric Integrity Check

TagGenerate(**Data**):

1. Compute hash function h(**Data**).

A New Design: Asymmetric Integrity Check

TagGenerate(**Data**):

1. Compute hash function h(**Data**).

Check Integrity without recomputing **Tag**.

IntegrityCheck(**Data***, **Tag**):

1. Take a random sample from **Data**.
2. Take a random sample from **Tag**.
3. Compare the samples.

A New Design: Asymmetric Integrity Check

TagGenerate(**Data**):

1. Compute hash function h(**Data**).

Check Integrity without recomputing **Tag**.

IntegrityCheck(**Data***, **Tag**):

1. Take a random sample from **Data**.
2. Take a random sample from **Tag**.
3. Compare the samples.

How can we realize it?

Homomorphic Hash Function

Main idea: Batch verification over a hash list.

Homomorphic Hash Function

Main idea: Batch verification over a hash list.

Probabilistic sampling from hash list.

IntegrityCheck($\mathbf{Data}^* = (x_0, x_1, \ldots x_d), \mathbf{Tag} = (y_0, y_1, \ldots, y_d)$):

1. Take a random sample $\bar{x}$ from $(x_0, x_1, \ldots, x_d)$
2. Take a random sample $\bar{y}$ from $(y_0, y_1, \ldots, y_d)$.
3. Compare the samples.

Each y_i is supposed to satisfy $y_i = h(x_i)$. How can we check that $\bar{x}$ and $\bar{y}$ relates?

Homomorphic Hash Function

Batch verification requires a homomorphic hash function.

Homomorphic Hash Function

Batch verification requires a homomorphic hash function.

Let $N = pq$, $G = \mathbb{Z}/N\mathbb{Z}$, and $g \in (\mathbb{Z}/N\mathbb{Z})^*$. Consider $h(x) = g^x \pmod N$. Then, [M78] proves below.

Homomorphic Hash Function

Batch verification requires a homomorphic hash function.

Let $N = pq$, $G = \mathbb{Z}/N\mathbb{Z}$, and $g \in (\mathbb{Z}/N\mathbb{Z})^*$. Consider $h(x) = g^x \pmod N$. Then, [M78] proves below.

Theorem 1 (Miller) *Under the extended Riemann Hypothesis, there exists a probabilistic polynomial-time machine that outputs factors of N if given some $x_0 \in \mathbb{Z}$ such that $h(x_0) = 1 \pmod N$ on input.*

Homomorphic Hash Function

Batch verification requires a homomorphic hash function.

Let $N = pq$, $G = \mathbb{Z}/N\mathbb{Z}$, and $g \in (\mathbb{Z}/N\mathbb{Z})^*$. Consider $h(x) = g^x \pmod N$. Then, [M78] proves below.

Theorem 1 (Miller) *Under the extended Riemann Hypothesis, there exists a probabilistic polynomial-time machine that outputs factors of N if given some $x_0 \in \mathbb{Z}$ such that $h(x_0) = 1 \pmod N$ on input.*

If it is hard to factor N, then $h(x)$ is collision resistant homomorphic function, even if x is a huge volume of data (but still in poly-size).

Batch Verification for Discrete Logs[BGR98]

Let $G = \mathbb{Z}/p\mathbb{Z}$, $y_i \in G$ and $x_i \in \mathbb{Z}/(p-1)\mathbb{Z}$ for each $i = 1, \ldots, d$. Choose $g \in G$ and fix it. Let k be the security parameter.

ShortExponetTest$((x_1, y_1), (x_2, y_2), \ldots, (x_d, y_d))$:

1. For each i, choose random $e_i \leftarrow_u \{0, 1, \ldots, 2^k - 1\}$.
2. Compute $\bar{x} = \sum_i e_i x_i \pmod{p-1}$.
3. Compute $\bar{y} = \prod_i y_i^{e_i}$.
4. Compare $\bar{y}$ and $g^{\bar{x}}$ in G. If coincides, output OK. Otherwise output NG.

ShortExponetTest outputs OK if and only if $y_i = g^{x_i}$ for all i (with negligible error probability).

Proposed TagGenerate

Consider hash list from $h(x) = g^x \pmod N$.

TagGenerate$(x_1, x_2, \ldots, x_d)$:

1. For all $i = 1, 2, \ldots, d$ compute $y_i = g^{x_i} \pmod N$.
2. Output $(y_1, y_2, \ldots, y_d)$.

Proposed TagGenerate

Consider hash list from $h(x) = g^x \pmod N$.

TagGenerate$(x_1, x_2, \ldots, x_d)$:

1. For all $i = 1, 2, \ldots, d$ compute $y_i = g^{x_i} \pmod N$.
2. Output $(y_1, y_2, \ldots, y_d)$.

This **TagGenerate** is very slow. About 1 Mbit/second on Athlon X2@ 2GHz with GMP.

Proposed TagGenerate

Consider hash list from $h(x) = g^x \pmod N$.

TagGenerate$(x_1, x_2, \ldots, x_d)$:

1. For all $i = 1, 2, \ldots, d$ compute $y_i = g^{x_i} \pmod N$.
2. Output $(y_1, y_2, \ldots, y_d)$.

This **TagGenerate** is very slow. About 1 Mbit/second on Athlon X2@ 2GHz with GMP. If the factors of N are known, the procedure gets much faster: About 400 Mbit/second (still slow?).

First Step

The first step it to replace p with $N = pq$. Let $G = \mathbb{Z}/N\mathbb{Z}$, $g \in (\mathbb{Z}/N\mathbb{Z})^*$.

ShortExponetTest$((x_1, y_1), (x_2, y_2), \ldots, (x_d, y_d))$:
Outputs OK if $y_i = g^{x_i}$ for all i, otherwise outputs NG.

1. For each i, choose random $e_i \leftarrow_u \{0, 1, \ldots, 2^k - 1\}$.
2. Compute $\bar{x} = \sum_i e_i x_i \pmod{\lambda(N)}$
3. Compute $\bar{y} = \prod_i y_i^{e_i}$.
4. Compare $\bar{y}$ and $g^{\bar{x}}$ in G. If coincides, output OK. Otherwise output NG.

First Step

The first step it to replace p with $N = pq$. Let $G = \mathbb{Z}/N\mathbb{Z}$, $g \in (\mathbb{Z}/N\mathbb{Z})^*$. Compute $\bar{x}$ in $\mathbb{Z}$.

ShortExponetTest$((x_1, y_1), (x_2, y_2), \ldots, (x_d, y_d))$:
Outputs OK if $y_i = g^{x_i}$ for all i, otherwise outputs NG.

1. For each i, choose random $e_i \leftarrow_u \{0, 1, \ldots, 2^k - 1\}$.
2. Compute $\bar{x} = \sum_i e_i x_i$
3. Compute $\bar{y} = \prod_i y_i^{e_i}$.
4. Compare $\bar{y}$ and $g^{\bar{x}}$ in G. If coincides, output OK. Otherwise output NG.

Removing the Random Oracle

Let $R = \{(x, y) \mid y = g^x \pmod N\}$. R is *additive*: If $(x_1, y_1), (x_2, y_2) \in R$, then $(x_1 + x_2, y_1 y_2) \in R$. Apply the generic procedure to make batch process for verifications over instances from an additive NP-relation [CY07].

Batch Processing

1. Choose random $e \leftarrow_u \{0, 1, \ldots, s-1\}$
2. Compute $\bar{x} = \sum_i (e^{i-1} \pmod s) x_i$
3. Compute $\bar{y} = \prod_i y_i^{e^{i-1} \pmod s}$.
4. Run the verification process for $(\bar{x}, \bar{y})$.

Proposed Scheme

Let s be a prime number $s > 2^k$.

IntegrityCheck$((x_1, y_1), (x_2, y_2), \ldots, (x_d, y_d))$:
Outputs OK if $y_i = g^{x_i}$ for all i, otherwise outputs NG.

1. Choose random $e \leftarrow_u \{0, 1, \ldots, s-1\}$.
2. Compute $\bar{x} = \sum_i (e^{i-1} \pmod s) x_i$ in $\mathbb{Z}$.
3. Compute $\bar{y} = \prod_i y_i^{e^{i-1} \pmod s}$.
4. Compare $\bar{y}$ and $g^{\bar{x}}$ in G. If coincides, output OK. Otherwise output NG.

Collision Resistance

Uniqueness of the tag that **IntegrityCheck**(**Data**, $*$) accepts

Theorem 1 *Suppose* **Tag** *is the output of* **TagGenerate**(**Data**)*. If* **Tag**$'$ *is given on input such that* **IntegrityCheck**(**Data**, **Tag**$'$) *outputs* OK *with probability that is not negligible, and if* **Tag**$' \neq$ **Tag**, *then a probabilistic polynomial-time machine outputs factors of* N *with probability that is not negligible.*

Why fast?

...

1. Choose random $e \leftarrow_u \{0, 1, \ldots, s-1\}$.
2. Compute $\bar{x} = \sum_i (e^{i-1} \pmod s) x_i$ in $\mathbb{Z}$.
3. Compute $\bar{y} = \prod_i y_i^{e^{i-1} \pmod s}$.
4. ...

Why fast?

...

1. Choose random $e \leftarrow_u \{0, 1, \ldots, s-1\}$.
   Constant amount of random bits
2. Compute $\bar{x} = \sum_i (e^{i-1} \pmod s) x_i$ in $\mathbb{Z}$.
3. Compute $\bar{y} = \prod_i y_i^{e^{i-1} \pmod s}$.
4. ...

Why fast?

...

1. Choose random $e \leftarrow_u \{0, 1, \ldots, s-1\}$.
   Constant amount of random bits
2. Compute $\bar{x} = \sum_i (e^{i-1} \pmod s) x_i$ in $\mathbb{Z}$.
   $e^{i-1} \pmod s$ is short
3. Compute $\bar{y} = \prod_i y_i^{e^{i-1} \pmod s}$.
4. ...

Why fast?

...

1. Choose random $e \leftarrow_u \{0, 1, \ldots, s-1\}$.
   Constant amount of random bits
2. Compute $\bar{x} = \sum_i (e^{i-1} \pmod s) x_i$ in $\mathbb{Z}$.
   $e^{i-1} \pmod s$ is short
3. Compute $\bar{y} = \prod_i y_i^{e^{i-1} \pmod s}$.
   Short simultaneous exponentiations
4. ...

Implementation

We implemented the proposed algorithms on a standard PC with AMD64 processor.

CPU	Athlon64x2 3800+
Clock frequency	2.0 GHz
L1 cache	16 KB / core
L2 cache	512 KB / core
Chipset	NVIDIA nForce 410 MCP
Main memory	DDR SDRAM 400 MHz 512 MB$\times$2
OS	SuSE Linux 10.1 for AMD64
Compiler	gcc 4.1.0 -O3
Library	gmp 4.2.1

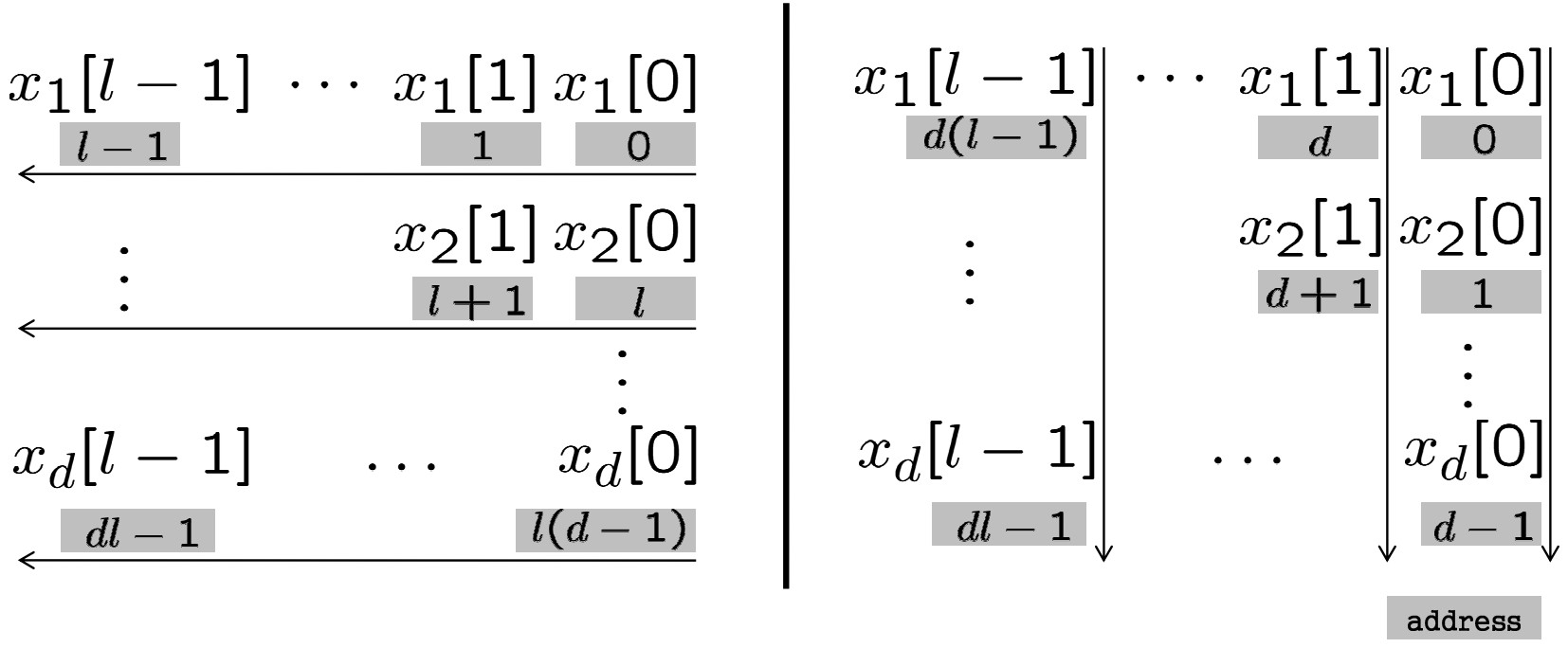
Parameters

Choose parameters for 80-bit security.

k	80
$\|N\|$	1024 bit
$L = \|x_i\|$	1024 * 1024 bit
s	a random 80 bit prime number

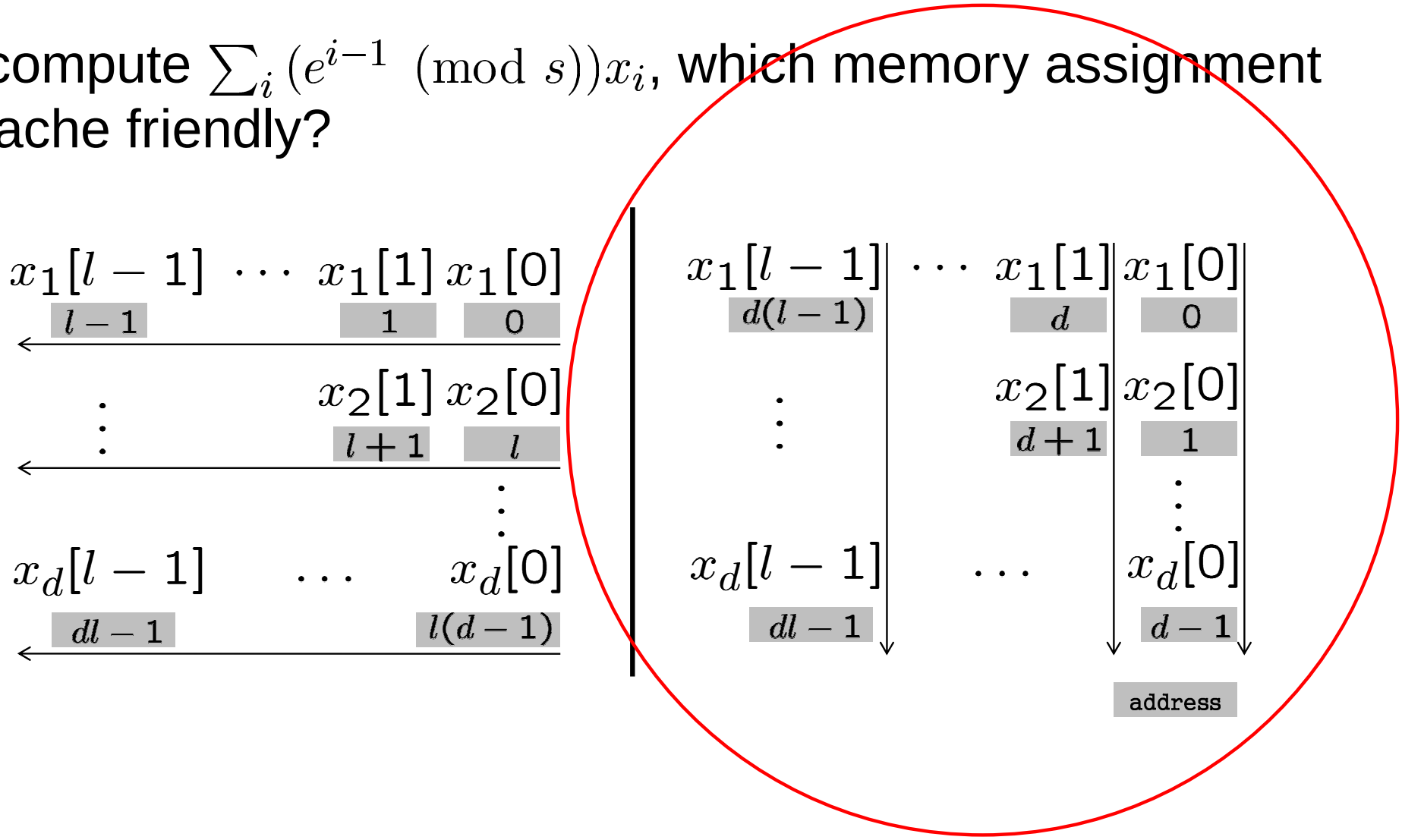
Optimization

To compute $\sum_i (e^{i-1} \pmod s) x_i$, which memory assignment is cache friendly?



Optimization

To compute $\sum_i (e^{i-1} \pmod s) x_i$, which memory assignment is cache friendly?



Test Result(1)

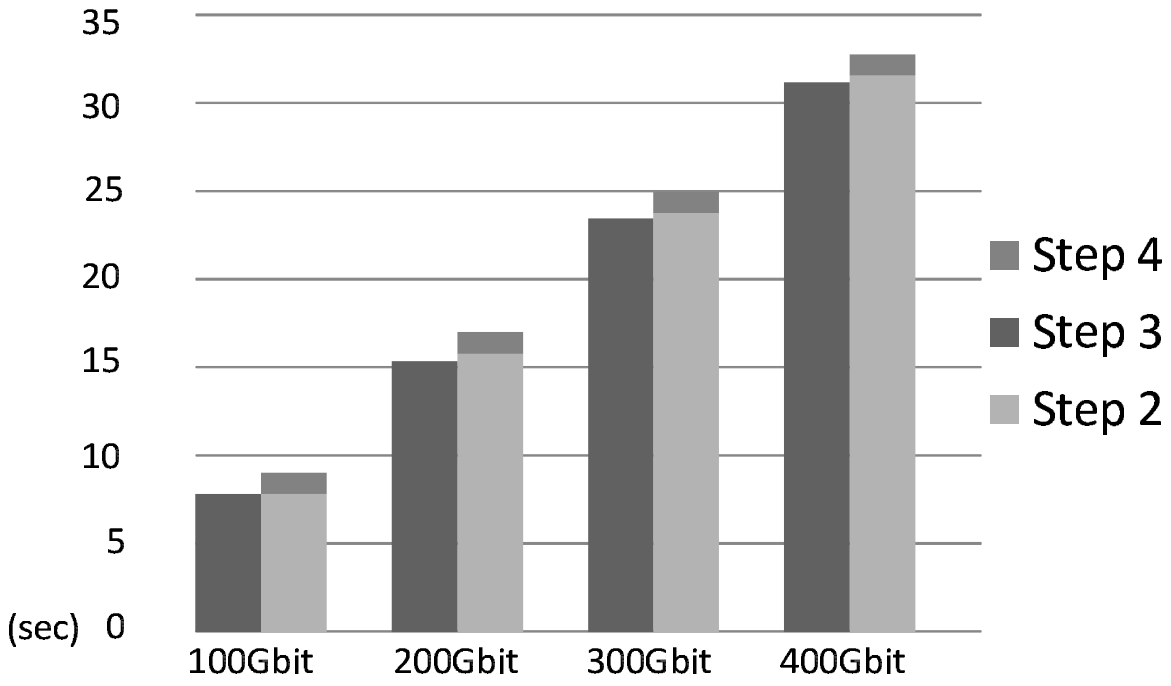
Single-thread implementation:

Message size (Gbit)	100	200	300	400
Verify (sec)	16.86	32.36	48.42	63.94

6.26 Gbit/second, 3.13 bit/cycle.

Test Result(2)

Multi-thread implementation:



12.5 Gbit/second throughput.

Conclusion

- A collision-resistant scheme is proposed. Efficient integrity check.
- Integrity checking over hast lists can be faster than tag generation.
- Implementation result indicates about 6 Gbit/second with single-thread, 12 Gbit/second with multi-thread.
- More homomorphic hash functions?