Verifying Server Computation

Lea Kissner[1] and Dawn Song[1]

Carnegie Mellon University *
`leak@cs.cmu.edu, dawnsong@cmu.edu`

Abstract. In many scenarios, clients receive the results of computation which has been performed by a remote server. An example of such a setting is the third-party publishing model, in which a server answers arbitrary database queries posed by clients with data originally provided by a third party, who does not participate in the protocol. Previous work in this scenario has addressed range queries, but there has not previously existed a general solution for detecting cheating on the part of the server for arbitrary queries. We propose different two techniques for approximate proof construction to detect server misbehavior: spot checking and transformation-based checks. We develop proof constructions, using the spot-checking paradigm, for arbitrary database queries. We also offer an illustrative example of a use of the transformation-based check technique.

1 Introduction

In many applications, clients must trust the correctness of the computations performed by a server. If the server is corrupted, the integrity of these results may be compromised. We wish to reduce the amount of trust which the client is required to hold in the server by introducing checks which reduce the probability that the the client does not detect cheating in the computation by the server.

An example of such a scenario is the third-party publishing model. In this model, a *data owner* transfers the contents of a database to a *server*. This server answers arbitrary database queries from *clients*, using the data provided. We wish to allow the clients to verify that the results of their queries are correct; i.e., the results are identical to those which would have been provided by an honest data owner.

Previous work in this area provides protocols for only a small subset of the basic relational algebraic operations [4, 7]. There currently exists no general solution to this problem. We propose protocols for each of the relational algebraic operations, which allow, by composition, arbitrary SQL database queries to be performed. Our protocols are approximate, allowing the client to adjust the proof size, and accordingly the assurance level of the proof. These protocols are most appropriate in the setting where a small number of suppressions of elements from the results is not critical, or where the repercussions for the server's operator may dissuade him from being caught.

* This research was supported in part by the Center for Computer and Communications Security at Carnegie Mellon under grant DAAD19-02-1-0389 from the Army Research Office, and by gifts from Bosch, Cisco, Intel, and Matsushita Electric Works Ltd. The views and conclusions contained here are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of ARO, Bosch, Carnegie Mellon University, Cisco, Intel, Matsushita Electric Works Ltd., or the U.S. Government or any of its agencies.

We propose two different techniques for application to verification of server computation: remote spot-checking, and transformation-based checks. A remote spot-checking proof consists of requiring the server to pseudorandomly sample portions of the intermediate results or output, and to prove that they have been computed correctly. A transformation-based check consists of a server performing the requested computation on the original data, as well as performing a related computation on data which has been transformed through a function which, for the server, is infeasible to reverse. Our proof constructions for relational algebra operations are based on the remote spot-checking paradigm. We offer an illustrative example of transformation-based check constructions.

1.1 Related Work

Previous work has addressed third-party publishing of databases, but only provides proof construction for range (selection) queries [4]. Third-party publication of files was addressed, but without support for querying [2]. Proof constructions for other sorts of queries in authenticated data structures include those in search DAGs, red-black trees, and simple binary search trees [7, 1, 8].

Previous related work using spot-checking proofs with a remote prover has addressed simple statistical queries in sensor net aggregators [9]. Other work uses spot-checking to verify a number of properties in locally available data [5]. There is also work in which a prover assists the verifier in checking properties in data to which they both have access [6].

2 Preliminaries

In this section we discuss the tools and concepts which are used in this paper.

2.1 Relational Algebra

Queries performed on relational databases are typically formulated in SQL, which is translated into a composition of relational algebra operations before execution. The basic operations are:

- *Selection* $\sigma_{P,A_i,c}(S) = \{s \mid s \in S \ \wedge \ P(s,c)\}$ where $A(s)$ is the value of attribute A in element s, $\Theta \in \{=, \neq, <, >, \leq, \geq\}$, and $P(s,c) = A_i(s)\ \Theta\ c$.
  This operator is encompassed in a range query, where the result is $\{s \mid s \in S \ \wedge \ c_1 \leq A(s) \leq c_2\}$, where c_1 and c_2 may be set to be unbounded. We support nested range queries.
- *Projection* $\pi_{A_i,A_j,\ldots,A_k}(S) = \{\langle A_i(s), A_j(s), \ldots, A_k(s)\rangle \mid s \in S\}$ where $A_i, \ldots, A_k$ are distinct attributes of elements in S.
- *Cartesian product* $S \times T = \{st \mid s \in S \ \wedge \ t \in T\}$
- *Union* $S \cup T = \{u \mid u \in S \vee u \in T\}$
- *Set difference* $S - T = \{u \mid u \in S \wedge u \notin T\}$

2.2 Cryptographic Tools

A *hash function* takes as input bit strings of arbitrary length and returns a fixed-length output (hash value). We require two properties of the hash function used in this paper: that it is one-way, and collision resistant. For a one-way hash function, given a hash value v, it is computationally infeasible to determine any x such that $h(x) = v$. For a collision resistant hash function it is computationally infeasible to determine any two inputs x, y such that $h(x) = h(y)$.

A *pseudo-random generator* takes as input a seed, and returns an arbitrarily long pseudo-random output which is a deterministic function of that seed. This output is computationally indistinguishable from a truly random sequence without knowledge of the seed.

Private information retrieval (PIR) is a protocol with two parties: a client and a server. The server holds a string of data, and the client holds an index i. Interacting through a PIR protocol allows the client to retrieve the data at index i without revealing i to the server. We assume polylogarithmic communication complexity, as in [3].

Commitment to Sets A *commitment* is a method to allow a party to commit to a value without immediately revealing it. He may open the commitment later in the protocol, proving that he did in fact commit to a certain value. A commitment is computationally infeasible to open to any value but the original. In our paper, we use commitments to prove membership in a set, represented by a tree. The set commitments used in this paper are an extension of Merkle hash trees [8], with the addition of a commitment to the number of elements in the tree and the depth.

We use two types of trees to represent sets: uniform-depth binary trees and red-black trees. In a uniform-depth binary tree, all data elements are at the leaves, and are all at the same depth. The first data element (if they are ordered) is placed at the leftmost leaf, the second at the next leaf to the right, and so on, so that empty leaves only occur at the $2^k - n$ rightmost leaves, where $2^{k-1} < n \leq 2^k$. Empty leaves are ignored, and should have a fixed commitment, such as 0. A red-black tree is a binary search tree on which insertions and deletions are performed so as to maintain a height of $O(\lg n)$. We place data only at the leaves.

3 Database Spot-Checking

The proofs and protocols given in this section address the third-party data scenario. In this situation a data owner signs some small commitment to his set of data elements, and gives both this and the data to a data server. He can update this set efficiently without retaining the data himself. This server answers queries from clients, and attaches a proof that the data returned in response to a given query is ”right”, both complete and correct. Completeness ensures that all of the data matching the query has been returned to the client, and correctness guarantees that the data which is returned is a subset of that given by the data owner.

Past approaches to this problem have described proofs which are nearly-exact in nature (can only be forged with negligible probability). However, these schemes do not support more than a small subset of possible queries. By making these proofs approximate, we allow for less expensive proofs on a wider variety of queries. As the error threshold and confidence level is adjustable, an explicit tradeoff can be made between proof size and exactitude. Our algorithm also gives proofs for any arbitrary relational algebra query, while past works have addressed only subsets of this space efficiently.

3.1 Data Management

The protocols referenced in this section allow a data owner to give an initial set of data to the server, add an element to the current data, and delete an element from the current data, while updating the commitments to the current set of data. The data owner must retain only a commitment to the current data and the number of elements in the current data set.

The data owner inserts all elements in each data set into a red-black tree, and commits to it as in Sec. 2.2. The data owner sends the entire set of data, and the commitment to it, to the server. Each client will need to have a current copy of this commitment as well, to verify the proofs.

Algorithms for insertion and deletion are given in [1]. Those given are persistent, so searches can be performed at any point in the past state of the database, as well as in the present. If this is not desired, obsolete nodes may be deleted. The data owner may perform these algorithms by making the requests for specific nodes from the server, who holds the data. The server may prove that all his responses to the requests are correct by giving the node and the committed path to the root of the data tree.

Uniform Sampling on Red-Black Trees The proof constructions given in this section all require sampling elements from the current data set with independent probability p. Proving that each element has been considered is straightforward if the commitment tree is a simple binary tree. We give an analogous method of sampling a red-black tree.

1. Extend the tree so that it is a complete binary tree of uniform depth
2. Sample each leaf of this extended tree with probability p
   - If the sampled leaf is a leaf in the original red-black tree, return this node and the path to the root of the commitment
   - If the sampled leaf is not a leaf in the original red-black tree, it is in a subtree descended from a leaf v in the original tree. If this is the leftmost leaf in this subtree, then return v and the path from v to the root of the commitment.

3.2 Proof Protocol

A client **C** who wishes to make a query on the data held by **S** does so according to the `DbProof` protocol (Fig. 1). We assume that **C** already holds a current copy of the commitment to the database. In the static case, it is sufficient to have a signed commitment. However, the client must have a trusted copy of the data owner's public key. We do not address these issues in this paper.

All random numbers required for the subsequent proof constructions are obtained from the seed given by the client used as input to the pseudo-random generator. Guidelines as to relationship between the sampling probability p and the expected size of the proof returned are given in Sec. 3.7.

3.3 Range Proof Construction

The proof construction given in this section addresses the same problem as [4], that of selecting elements which fall into chosen ranges on arbitrary attributes. This operator has all the functionality of the relational algebra selection operator.

1. Client **C** sends a query in relational algebra to server **S**
2. **S** calculates the result of that query, and sends all of the set commitments to the intermediate results to **C**, as well proofs (in the form of paths to the root) for the size of each set, and the depth of the corresponding tree.
3. **C** selects a random seed for a generator and a sampling probability p and sends them to **S**
4. **S** constructs the proofs for each relational algebra operation and sends them, along with the result of the query, to **C**

C accepts if the given proofs are valid according to the verification checks given in 3.6.

Fig. 1. The `DbProof` Protocol

```
PermProof ($S$,$T$,S, T,$p$)
We assume that $S$ and $T$ are sets with hash commitments S and T
$p$ is the sampling probability

  1. for every item $i$ in $S$, with probability $p$:
     (a) Give the path from $S_i$ to the root S
     (b) Give the path from an element $j \in T$ (such that $T_j = S_i$) to the root T
```

Fig. 2. The `PermProof` Generation Algorithm

3.4 Relational Algebra Proof Construction

With the proof for nested range queries and the proofs for the other basic relational algebraic operations given below, we may construct proofs for any composition of these operations, by providing proofs for each operation. These proofs use the commitment constructed as part of the previous proof to ensure continuity of the sets involved.

– *Projection* Let U be the requested projection of S, with duplicates removed. The proof is constructed as `PermProof`(S,U,`S`, `U`, p) and `PermProof`(U,S,`U`, `S`, p).
– *Cartesian product* Let $U = S \times T$. The proof is constructed as `CartProof`(S,T,U, `S`, `T`, `U`, p).
– *Union* Let $U = S \cup T$. The proof is constructed as the concatenation of `PermProof`(S, U, `S`, `U`, p), `PermProof`(T, U, `T`, `U`, p), and `PermProof`(U, $S \cup T$, `U`, h(`S`||`T`), p).
– *Set difference* Let $U = S - T$. The proof is constructed as `DiffProof`(S, T, U, `S`, `T`, `U`, p) and `PermProof`(U, S, `U`, `S`, p).

3.5 Correctness Proof Construction

The construction of a correctness proof is done by providing a proof of set membership (in the set which the data owner committed to) for each element of the database returned. This guarantees with overwhelming probability that each data element returned was in fact derived from the original data set.

```
RangeProof ($D$,$\mathtt{D}$, $p$, $A_1$, $(\ell_1, u_1)$, ... , $A_d$, $(\ell_d, u_d)$)
We assume that $D$ is the current data set, and $\mathtt{D}$ a hash commitment to this data
The client holds $\mathtt{D}$ for verification of the proof
$p$ is the sampling probability
$(\ell_i, u_i)$ is the selected range on attribute $A_i$, $I_i$ is the remaining elements at nested range $i$
$I_0 = D$, $\mathtt{I}_0 = \mathtt{D}$

For each nested range query $i$:

  1. Sort the elements of $I_{i-1}$ according to attribute $A_i$
  2. Generate a commitment $\mathtt{I}'_{i-1}$ to the sorted set, and add this to the proof
  3. Add PermProof($I_{i-1}$, $I_{i-1}$, $\mathtt{I}_{i-1}$, $\mathtt{I}'_{i-1}, p$) and PermProof($I_{i-1}$, $I_{i-1}$, $\mathtt{I}'_{i-1}$, $\mathtt{I}_{i-1}, p$) to the proof
  4. Generate a committment $\mathtt{I}_i$ to the elements of $I_{i-1}$ in the range $\langle \ell_d, u_d \rangle$
  5. Add the index in the sorted set of the first element in the range $\langle \ell_d, u_d \rangle$
  6. Add PermProof($I_i, I_{i-1}, \mathtt{I}_i, \mathtt{I_{i-1}}', p$)
  7. Sample elements of $I_{i-1} - I_i$ with probability $p$, returning a set membership proof for each
```

Fig. 3. The `RangeProof` Generation Algorithm

```
CartProof ($S$,$T$,$U$,$\mathtt{S}$, $\mathtt{T}$,$\mathtt{U}$,$p$)
We assume that $S$,$T$, and $U$ are sets with hash commitments $\mathtt{S}$, $\mathtt{T}$, and $\mathtt{U}$
$p$ is the sampling probability

  1. for every pair of items $i_S \in S$, $i_T \in T$ with probability $p$:
     (a) Give the path from $S_{i_S}$ to the root $\mathtt{S}$ (if it has not already been given)
     (b) Give the path from $T_{i_T}$ to the root $\mathtt{T}$ (if it has not already been given)
     (c) Give the path from an element $j \in U$ (such that $U_j = (i_S, i_T)$) to the root $\mathtt{U}$
```

Fig. 4. The `CartProof` Generation Algorithm

3.6 Proof Validation

For a proof given by a server to be considered valid, it must pass all validation checks given in this section.

- The set commitments given in the `DbProof` Protocol are the same as those used in the proof returned.
- All set membership proofs are valid, with the proofs giving a valid path to the root commitment, with the correct tree depth. For details, see Sec. 2.2.
- The commitments used for sets do not vary between subproofs – if a set S is the result of a relational algebraic operation and is used as input to another operation, the same commitment is used.
- No sampled element is duplicated within any set. Unless a union operation has taken place , the hash commitment to the element should remain constant when that element is sampled more than once.
- The correct elements have been sampled. As the client supplied the seed from which all pseudo-random numbers used in the proofs are generated, the client may reproduce these numbers himself and determine if the elements from each set which are returned as

```
DiffProof ($S$,$T$,$U$,S, T,U, $p$)
We assume that $S$ and $T$ are sets with hash commitments S and T
$p$ is the sampling probability
An element $i$ has value $i_A$ at attribute $A$

 1. for every item $i$ in $S$, with probability $p$:
    (a) Give the path from $S_i$ to the root S
    (b) If $i \in T$ then
          i. Give the path from an element $j \in T$ (such that $T_j = i$) to the root T
        else
          i. Let $\{A_1, \ldots, A_k)$ be a minimal set of attributes such that no member of $T$ is equivalent to
             $i$ on these attributes
             Include RangeProof($T$,T, $A_1$,($i_{A_1}$,$i_{A_1}$), ..., $A_k$,($i_{A_k}$,$i_{A_k}$))
         ii. Give the path from an element $j \in U$ (such that $U_j = S_i$) to the root T
```

Fig. 5. The DiffProof Generation Algorithm

"sampled" are in fact those which should have been sampled, according to the generated pseudo-random stream.

3.7 Analysis

For each proof construction given above, we give an expected proof size and simple error bounds on incorrect inclusions and exclusions from the result set. Actual errors should be detected much more readily by many of the tests given in the last section, but are difficult to account for in determining an error bound for a single operation.

Note that the expected size of a proof of membership in a set S is $\lg |S|$.

- *Range* A range selection is performed on S, which eliminates all but the members of set T.
  - $E[\text{proof size}] = 2p|S|\lg|S| + p(|S| - |T|)\lg|S| + p|T|(\lg|S| + \lg|T|) = 3p(|S|\lg|S| + |T|\lg|T|)$
  - $\Pr[x \text{ exclusions are detected}] = 1 - (1-p)^x$
  - $\Pr[y \text{ inclusions are detected}] = 1 - (1-p)^y$
- *Projection* A projection is performed on S.
  - $E[\text{proof size}] = 2p|S|\lg|S|$
  - $\Pr[x \text{ exclusions are detected}] = 1 - (1-p)^x$
  - $\Pr[y \text{ inclusions are detected}] = 1 - (1-p)^y$
- *Cartesian product* A Cartesian product of S_1 and S_2 results in T. Note that to pass the size validity check either there must be a large number of exclusions or inclusions (which is relatively easy to detect), or for each inclusion there must be an exclusion, and vice versa. We give error bounds for this second, less easily detectable, case.
  - $E[\text{proof size}] = p|T|(\lg|T| + \lg|S_1| + \lg|S_2|)$
  - $\Pr[x \text{ exclusions are detected}] = 1 - (1-p)^{2x}$
  - $\Pr[y \text{ inclusions are detected}] = 1 - (1-p)^{2y}$
- *Union* A union of sets S_1 and S_2 results in T.

  - E[proof size] $= 2p|S_1|(\lg|S_1| + \lg|T|) + 2p|S_2|(\lg|S_2| + \lg|T|)$
  - $\Pr[x$ exclusions are detected$] = 1 - (1-p)^x$
  - $\Pr[y$ inclusions are detected$] = 1 - (1-p)^y$
- *Set difference* The set difference $S_1 - S_2$ results in T. Note that earlier undetected inclusions in S_2 translate to undetected exclusions here.
  - E[proof size] $= p|T|(\lg|T| + \lg|S_1|) + p\left(1 - \frac{|T|}{|S_1|}\right)(\lg|S| + \lg|T|) + p\frac{|T|}{|S_1|}(\lg|S_1| + |S_2|\lg|S_2| + |\text{included range proof}|)$
  - $\Pr[x$ exclusions are detected$] = 1 - ((1-p) + p(1-p))^x = 1 - (1-p^2)^x$
  - $\Pr[y$ inclusions are detected$] = 1 - (1-p)^y$

Choosing a Sampling Rate During the `DbProof` protocol, the server commits to the size of each intermediate result. Let a be the sum of the sizes of the intermediate results, with the exception that where a Cartesian product operation is to take place, the size of the output is added in place of the input size. Let b be the size of the largest intermediate result . (Because the size of the Cartesian product of two sets is in general very large, the expected size of the proof is $O(p|S|^2 \lg|S|)$, which may be written $O(p|T|\lg|T|)$ where T is the result of the operation.) The size of the entire proof is thus expected to be $O(pa\lg b)$. If a desirable proof size is $O(f(a)\lg b)$ for some function f, $p = O\left(\frac{f(a)}{a}\right)$. The assurance rate that any given level of server dishonesty for any given set of operations is easily calculated given the analysis above and p.

4 Transformation-Based Checks

We give an example of transformation-based checks in this section, largely for illustrative purposes. The concept of a transformation based proof is as follows:

1. The data owner calculates some transformation f on his data D. This transformation should be difficult to reverse for the server.
2. The data owner sends both D and $f(D)$ to the server.
3. A client (who may be the data owner) makes a query on the data.
4. The server computes the answer to the query on D, and does a check computation on $f(D)$. This may involve repeating the same computation, or calculating a different function.
5. The server returns the answer to the query to the client. The results of the check computation may be returned with the query results (generally in the case of a different check function and an unparameterized computation), or may be requested through PIR by the client, to avoid leaking information about the transformation used by the data owner.

The proof and protocol given in this section address the third-party data scenario in which the data queried is not a set of elements, but a graph. A data owner signs a small commitment to the graph, and to a transformation of the graph. He then sends these signed commitments to the server, along with the graph and transformation. The server answers queries from clients about different properties of the graph.

Our proofs are intended for a specific situation: where the graph is sufficiently complex or uniform that given both a weighted graph G and a graph G' which is the result of applying a random isomorphism σ on the vertices of G, the adversary can determine only a small portion of σ. We provide a mechanism for querying the server for the shortest weighted path between any two given vertices. We also make the assumption that each client knows both σ and a description of the vertices of G, and does not collaborate with the server. Perhaps the most useful application for this is where the data owner is also the client.

4.1 Data Management

This protocol allows the data owner **D** to give a graph G on n nodes with edges $E(G)$, a permuted graph G', and a signed commitment to these graphs to the server. All clients must hold π, `G`, and `G'` (as defined in Fig. 6).

1. **D** generates a random permutation π on $[1, \ldots, n]$. Let G' be the result of applying π to the vertices of G
2. **D** Commit to $E(G)$ and $E(G')$, where each element representing (u, v) with weight w is a tuple (u, v, w). Let these commitments be denoted `G` and `G'`
3. **D** sends G, G', `G`, and `G'` to **S**

Fig. 6. Graph data setup protocol

4.2 Proof Construction

The construction of the proof is simple. If the shortest path between the queried vertices u and v is $u, x_0, x_1, \ldots, x_k, v$, then the server proves that there in fact exists such a path by referencing the commitment. For each edge (x, y) the server includes a path to the root `G`.

4.3 Proof Verification

The proof supplied by the server proves the existence of claimed path between the queried vertices u and v, but does not prove that it is the shortest such path. By retrieving the shortest path between $\pi(u)$ and $\pi(v)$ in the permuted graph G' via PIR, the server gains no knowledge of π. The client can then check that the two paths are identical; if they are not, the server is clearly cheating.

Note that the server does not know which path is being queried in the transformed graph. Thus if the server wishes to cheat, it must both guess the right path to cheat on, and cheat correctly on that path. It must therefore guess all of $\pi(u), \pi(x_0), \pi(x_1), \ldots, \pi(x_k), \pi(v)$. The upper bound on the probability of this is dependent on the particular graph being used, but this is, by the assumptions given for the protocol, close to randomly guessing each vertex transformation. The assurance of correctness can be adjusted by checking the answer returned by the server on δ proportion of queries.

To verify the proof given by the server of the path $u, x_0, \ldots, x_k, v$, the client **C**

1. checks that every edge given exists in the commitment

2. checks that no edge appears more than once
3. with δ probability, performs PirCheck$(\pi, u, x_0, \ldots, x_k, v)$ (Fig. 7)

```
PirCheck$(\pi, u, x_0, \ldots, x_k, v)$

  1. C sends a request to S to prepare a table with the shortest path between every pair of vertices
  2. C uses PIR to retrieve the shortest path between $u$ and $v$

C accepts if the path retrieved through PIR is $u, x_0, \ldots, x_k, v$
```

Fig. 7. The PirCheck Protocol

5 Conclusions

Preventing misbehavior on the part of servers who perform computation for clients is a difficult problem, both in the general case, and in many specific ones. We propose two techniques for designing approximate proof constructions for specific computations: remote spot checking and transformation-based checks. We give proof constructions based on the remote spot checking paradigm for arbitrary database queries in the third-party publishing model. We also present an illustrative example of transformation-based checks.

References

1. Aris Anagnostopoulos, Michael T. Goodrich, and Roberto Tamassia. Persistent authenticated dictionaries and their applications. *Lecture Notes in Computer Science*, 2200:379–??, 2001.
2. R. Anderson. The eternity service, 1996.
3. Christian Cachin, Silvio Micali, and Markus Stadler. Computationally private information retrieval with polylogarithmic communication. *Lecture Notes in Computer Science*, 1592:402–??, 1999.
4. Premkumar Devanbu, Michael Gertz, Charles Martel, and Stuart G. Stubblebine. Authentic data publication over the internet. *J. Comput. Secur.*, 11(3):291–314, 2003.
5. Funda Ergun, Sampath Kannan, S. Ravi Kumar, Ronitt Rubinfeld, and Mahesh Viswanathan. Spot-checkers. In *ACM Symposium on Theory of Computing*, pages 259–268, 1998.
6. Funda Ergun, Ravi Kumar, and Ronitt Rubinfeld. Fast approximate pcps. In *Proceedings of the thirty-first annual ACM symposium on Theory of computing*, pages 41–50. ACM Press, 1999.
7. C. Martel, G. Nuckolls, P. Devanbu, M. Gertz, A. Kwong, and S. Stubblebine. A general model for authenticated data structures. Technical report, 2001. CSE-2001.
8. Ralph C. Merkle. A certified digital signature. In *Proceedings on Advances in cryptology*, pages 218–238. Springer-Verlag New York, Inc., 1989.
9. Bartosz Przydatek, Dawn Song, and Adrian Perrig. SIA: Secure information aggregation in sensor networks. In *ACM SenSys 2003*, Nov 2003.