Formalization and Analysis of Ceph Using Process Algebra

Ran Li†, Nonmember, Huibiao Zhu†, Member, and Jiaqi Yin†, Nonmember

1. Introduction

With the quick proliferation of information, traditional file systems are on their way out and file system designers have looked to new architectures to facilitate system scalability. Therefore, distributed file systems came into existence. Currently, many popular distributed file systems have seen wide adoption in enterprise companies, including GFS [1], HDFS [2], Ceph [3], MooseFS [4], etc. Among them, Ceph is an emerging open-source distributed file system. Excellent performance, reliability and scalability make Ceph stand out from numerous distributed file systems [5].

Ceph consists of four main components: Client, MDS (Metadata Server), OSD (Object Storage Device), and Monitor. The Client component is used to realize the interactions between a user or an application and the Ceph. MDS stores metadata for the Ceph file system. OSD stores data, handles data replication, recovery, rebalancing, and provides some monitoring information to the Monitor. While the Monitor maintains maps of the cluster state, including the monitor map, manager map, the OSD map, the MDS map, and the CRUSH map [3]. Also, Ceph uses the Cephx authentication system which works like Kerberos to authenticate users, so that it can identify users and prevent man-in-the-middle attacks [6].

SUMMARY Ceph is an object-based parallel distributed file system that provides excellent performance, reliability, and scalability. Additionally, Ceph provides its Cephx authentication system to authenticate users, so that it can identify users and realize authentication. In this paper, we first model the basic architecture of Ceph using process algebra CSP (Communicating Sequential Processes). With the help of the model checker PAT (Process Analysis Toolkit), we feed the constructed model to PAT and then verify several related properties, including Deadlock Freedom, Data Reachability, Data Write Integrity, Data Consistency and Authentication. The verification results show that the original model cannot cater to the Authentication property. Therefore, we formalize a new model of Ceph where Cephx is adopted. In the light of the new verification results, it can be found that Cephx satisfies all these properties.

key words: Ceph, formalization, verification, process algebra

During the past few years, some research has already been carried out on Ceph. Nonetheless, the existing research focuses attention on the application of Ceph [7] or the performance analysis of Ceph [8]. To the best of our knowledge, there was only some research concerned formal verification of Ceph. Thus, we utilize the process algebra CSP to model the architecture of Ceph and use the model checker PAT to verify several inherent properties.

Process algebra CSP has been widely applied in various domains, such as OpenFlow [9], Python Software Transactional Memory architecture [10], HDFS [2], TMN protocol [11] and so on. These related work formalized models of different domains have succeeded in verifying the functional properties and security properties. Consequently, this mature method can perfectly match with our requirements for the formalization and analysis of Ceph.

In this paper, we focus on the processes of reading and writing data of Ceph, and consider the intruder’s behaviors in Ceph as well. After formalization, we verify some relative properties of the constructed model using the model checker PAT. The verification results show that the Authentication property is not satisfied. On this basis, we construct a new model of Ceph where Cephx is adopted. From the new verification results, we can believe that Cephx meets these important properties from the process algebra perspective.

The rest of this paper is organized as follows. In Sect. 2, we give an introduction to Ceph, Cephx and CSP. Section 3 is devoted to formalizing the model of Ceph based on the definitions of sets, messages, and channels. In Sect. 4, we give the assertions of the properties and analyze the verification results. Further, Sect. 5 is about the new model of Ceph which adopts Cephx. At last, we conclude our work and give the future work in Sect. 6.

2. Background

Focusing on the processes of reading and writing data, we give an overview of Ceph and Cephx in this section. Moreover, for a better understanding of the modeling in the next section, we give a brief introduction of CSP as well.

2.1 Ceph

Instead of using file allocations tables, Ceph relies on OSDs to distribute complex data accesses, update serialization, replication, reliability, and recovery to realize scalability [12]. If the client wants to read messages, it can get...
in touch with OSD directly. Besides, the processes of writing are applied in two phases for durability [13]. Figure 1 demonstrates its reading/writing processes.

- **Reading in Ceph**
  When the client requests to read data, the following sequence of actions occurs:
  1. The client first sends an open for read request to the MDS.
  2. The MDS reads the directory of the message from the OSD and returns the capability to the client.
  3. The client sends the read request with the capability to the corresponding OSD.
  4. The OSD responds if the capability is valid, the client can consequently read the message.

- **Writing in Ceph**
  When the client requests to write data, the following sequence of actions occurs:
  1. The client first sends an open for read request to the MDS.
  2. The MDS reads the directory of the message from the OSD and returns the capability to the client.
  3. The client sends the write request with the capability to the corresponding OSD and writes the object to the primary OSD.
  3.1)-(3.4) To ensure the high reliability and security of the data, Ceph has the ability to perform data replication. For replication purposes, the primary OSD identifies the secondary and tertiary OSDs, and then it writes the object to the primary OSD.
  4. When the primary OSD and replicas have been written successfully, the primary OSD returns a successful response to the client.

2.2 Cephx

To identify users and prevent man-in-the-middle attacks, Ceph uses the Cephx authentication system to authenticate users [6]. The behaviors of Cephx are similar to Kerberos [14] and Fig. 2 shows the flow of Cephx.

- **Flow of Cephx**
  When Cephx is adopted, a series of actions occurs:
  1. To use Cephx, the client sends a create request.
  2)-(4) The authentication subsystem generates the secret key, stores a copy to the monitor, and sends the user’s secret key back to the client. Moreover, the monitor shares this secret key with OSD and MDS.
  5 To authenticate with the monitor, the client sends an authenticate request to the monitor.
  6 The monitor generates a session key, encrypts it with the user’s key, and passes it back to the client.
  7 The client decrypts it with the shared secret key and obtains the session key. The client then requests a ticket with this session key.
  8 The monitor generates a ticket, encrypts it with the user’s key, and passes it back to the client. After decrypting and gaining the ticket, the client uses it to connect to the OSD and the MDS.

2.3 CSP

CSP, the abbreviation of Communicating Sequential Processes, is a classic process algebra proposed by C. A. R. Hoare [15]. It offers many operators to simulate the actions of processes. The syntax of a subset of CSP is given below. Here, $P$ and $Q$ represent processes, $a$ and $b$ are actions, $c$ stands for a channel to realize the communication between processes, and $B$ is a Boolean condition.

$$P, Q ::= a \rightarrow P \mid c?x \rightarrow P \mid c!u \rightarrow P || P \mid | P □ Q \mid P \leftarrow B \rightarrow Q \mid P[[a ← b]]$$

- $a \rightarrow P$ denotes that a process behaves exactly as described by $P$ after engaging in the event $a$.
- $c?x \rightarrow P$ describes that a process receives a message through the channel $c$ and assigns the value of the message to $x$, and then behaves like process $P$.
- $c!u \rightarrow P$ means a process sends the message $u$ through the channel $c$ and then behaves like process $P$.
- $P||Q$ stands for parallel. It represents process $P$ executes in parallel with process $Q$.
- $P □ Q$ indicates that a process performs like $P$ or $Q$ and the choice is determined by the environment.
- $P \leftarrow B \rightarrow Q$ expresses conditional choice. When $B$ is true, the process behaves like $P$. Otherwise, it behaves like $Q$. 
$P[\alpha \leftarrow \beta]$ is the syntax of renaming and signifies an event $a$ is replaced by $b$ in process $P$.

### 3. Modeling Ceph

In this section, we present the formalized model of Ceph. We first give the definitions of sets, messages, and channels. Based on these definitions, we use CSP to formalize the model of Ceph.

#### 3.1 Sets, Variables, Messages, and Channels

For the sake of convenience in modeling, we introduce some sets. We define the set $\text{Req}$ of request, $\text{Resp}$ of response messages and $\text{Content}$ of data. There are three types of requests in the set of $\text{Req}$, namely: $\text{Open}$, $\text{Read}$ and $\text{Write}$. We model these requests as $\text{Enum}$ types. For response messages, it contains acknowledgment signals and we define them as $\text{Boolean}$ types in our model. The set of $\text{Content}$ contains data transferred in the system and we set them as $\text{Integer}$ types.

In our model, variables that begin with uppercase letters are responsible for sending values. They are declared as global variables in PAT and we list them in Table 1. Besides, local free variables with lowercase initials, used to store values received from other entities, are not declared in PAT.

On this basis, the messages transmitted in our model can be abstracted as below.

\[
\begin{align*}
\text{MSG}_{\text{req}} &= df \{\text{msg}_{\text{req}}, \text{req}, \text{data} \mid \text{req} \in \text{Req}, \text{data} \in \text{Content}\} \\
\text{MSG}_{\text{resp}} &= df \{\text{msg}_{\text{resp}}, \text{resp} \mid \text{resp} \in \text{Resp}\} \\
\text{MSG}_{\text{data}} &= df \{\text{msg}_{\text{data}}, \text{data} \mid \text{data} \in \text{Content}\}
\end{align*}
\]

Then, we present the channels defined to realize the communications in our model.

- Channels of processes between legal entities: $\text{ComCM}$, $\text{ComMC}$, $\text{ComCO}$, $\text{ComOC}$, $\text{ComOR}$, $\text{ComR1}$, $\text{ComR2}$, $\text{ComR0}$. 
- Channels of the intruder faking/intercepting processes: $\text{FakeCM}$, $\text{InterceptCM}$, $\text{ InterceptOC}$

#### 3.2 Overall Modeling

Since we assume that there are two replicas of OSD in our model, the overall model of Ceph contains five parallel processes, including $\text{Client}$, $\text{MDS}$, $\text{OSD}$, $\text{OSD}_{1}$, and $\text{OSD}_{2}$. $\text{Client}$, $\text{MDS}$, and $\text{OSD}$ simulate the behaviors of the client, the MDS, and the primary OSD respectively. $\text{OSD}_{1}$ and $\text{OSD}_{2}$ represent the behaviors of the secondary and tertiary OSDs. Besides, we introduce a process $\text{Intruder}$ used to fake or intercept messages in communication. We formalize the whole model as below.

\[
\text{CEPH} = df \text{Client} \parallel \text{MDS} \parallel \text{OSD} \\
\parallel \text{OSD}_{1} \parallel \text{OSD}_{2} \parallel \text{Intruder};
\]

#### 3.3 Client Modeling

We formalize the process $\text{Client}$ using the general choice $\square$ to describe the reading processes and writing processes mentioned above. The model of $\text{Client}$ is presented as below.

First, we take $\text{ComCM}' \text{msg}_{\text{req}}, \text{Open}. \text{Msg}_{j}$ as an example to explain the description in CSP. In the model of $\text{MDS}$, $\text{ComCM}' \text{msg}_{\text{req}}, \text{req}. \text{msg}_{j}$ is the corresponding channel input event. As introduced earlier, $\text{ComCM}$ is a channel used to deliver messages from the client to the MDS. $!$ represents output and $?$ means input. $\text{Open}$ and $\text{Msg}_{j}$ are expressions that evaluate to values (at run time), while $\text{req}$ and $\text{msg}_{j}$ are local variables that take the input values. For channel output, the compound value evaluated from both $\text{Open}$ and $\text{Msg}_{j}$ is stored in the buffer. For channel input, the element in the buffer (i.e., $\text{Open}$ and $\text{Msg}_{j}$) is retrieved and then assigned to local free variables $\text{req}$ and $\text{msg}_{j}$ respectively. To better understand what information is communicated in the model, we use $\text{msg}_{\text{req}}$ to indicate the type of the transmitted message and we omit it in PAT for sake of brevity.

The formulas before $\square$ simulate the behaviors of the client when it wants to read messages from the OSD. It first sends an “Open” request to the MDS, and then waits for the response. After receiving the response, the client

### Table 1: Global variables in Ceph model

| Variables       | Types      | Meanings                                                      |
|-----------------|------------|---------------------------------------------------------------|
| $\text{Open}$/$\text{Read}$/$\text{Write}$ | $\text{Enum}$ | Requests                                                     |
| $\text{Ack}$/\text{Ack1}$/\text{Ack2}$ | $\text{Boolean}$ | Acknowledgment signals                                        |
| $\text{Data}$   | $\text{Integer}$ | Data written to the OSD                                       |
| $\text{Msg}_{\text{OSD}}[3]$ | $\text{Integer}$ | Data stored in the primary/secondary/tertiary OSD |
| $\text{Msg}_{\text{OSD}}[1]$ | $\text{Array}$  | $i$th data stored in the primary OSD                         |
| $\text{Msg}_{\text{OSD}}[2]$ | $\text{Array}$  | $i$th data stored in the primary OSD                         |
| $\text{Capability}$ | $\text{Integer}$ | Indicate whether the message is readable or writable         |
| $\text{FData}$  | $\text{Integer}$ | Intruder’s data                                               |
| $\text{DataReadSuccessi}$ | $\text{Boolean}$ | Check whether the $i$th data can be read successfully         |
| $\text{FakeClientR}$ | $\text{Boolean}$ | Check whether the intruder can read or write data successfully |
| $\text{FakeClientW}$ | $\text{Boolean}$ | Check whether the intruder can read or write data successfully |
gains the capability and sends a “Read” request along with this capability. If the MDS receives this request and affirms this request is valid, it will send a true response. Therefore, the client can read the data from the corresponding OSD. Here, read_success(DataReadSuccessj = true) is a data operation. An event can be attached with assignments that update global variables in PAT. If the client reads the data (i.e., reaches read_success event), the variable DataReadSuccessj is assigned to true.

The rest formulas describe the behaviors of the client when it writes data to the OSD. Similarly, it sends some requests to the MDS. Here, i is used to record which location is written. Afterwards, it writes data to the primary OSD. For replication purposes, the primary OSD will write the data to replicates. After completing all write operations in replicas, the client will receive a response from OSD once the OSD confirms that the object has been stored successfully. Otherwise, it fails.

\[ \text{Client}_0 = \text{df Client}_0[\text{ComCM!}\text{msg}_0\text{.Open.Msg}_j \rightarrow \text{ComMC!}\text{msg}_0\text{.data}\text{.Capability} \rightarrow \text{ComCO!}\text{msg}_0\text{.Read.Capability} \rightarrow \text{ComOC!}\text{msg}_0\text{.Write.Capability}\{i = j;\} \rightarrow \text{ComOC!}\text{msg}_0\text{.req-response} \rightarrow \text{ComCO!}\text{msg}_0\text{.data} \rightarrow \text{write-success(DataWriteSuccess}_j = \text{true;}} \rightarrow \text{Client}_0 \]

\[ <\text{(response == true)}> \]

\[ <\text{(response \_fail \rightarrow Client}_0) > \]

\[ \text{\□ ComCM!}\text{msg}_0\text{.Open.Msg}_j \rightarrow \text{ComMC!}\text{msg}_0\text{.data}\text{.Capability} \rightarrow \text{ComCO!}\text{msg}_0\text{.Read.Capability} \rightarrow \text{ComOC!}\text{msg}_0\text{.Write.Capability}\{i = j;\} \rightarrow \text{ComOC!}\text{msg}_0\text{.req-response} \rightarrow \text{ComCO!}\text{msg}_0\text{.data} \rightarrow \text{ComOC!}\text{msg}_0\text{.req-ack} \rightarrow \text{write-success(DataWriteSuccess}_j = \text{true;}} \rightarrow \text{Client}_0 \]

\[ <\text{(ack == true)}> \]

\[ <\text{(ack\_fail \rightarrow Client}_0) > \]

\[ <\text{(response == true)}> \]

\[ <\text{(response \_fail \rightarrow Client}_0) > \]

Based on the given model Client$_0$, we need to consider the behaviors of the intruder. On account of the vulnerability of the client, communication on the channels involved the client is not safe and it can be intercepted or faked by the intruder. As shown in Fig. 3, the intruder can intercept messages when the client sends messages to other entities. We achieve the description of the intruder by means of renaming [11] as below.

\[ \text{Client} = \text{df Client}_0[\text{ComCM!}\text{msg}_0\text{.Open.Msg}_j \rightarrow \text{ComCM!}\text{msg}_0\text{.req-response} \rightarrow \text{ComCM!}\text{msg}_0\text{.data}\text{.Capability} \rightarrow \text{ComMC!}\text{msg}_0\text{.data}\text{.Capability} \rightarrow \text{ComCO!}\text{msg}_0\text{.Read.Capability} \rightarrow \text{ComOC!}\text{msg}_0\text{.Write.Capability}\{i = j;\} \rightarrow \text{ComOC!}\text{msg}_0\text{.req-response} \rightarrow \text{ComCO!}\text{msg}_0\text{.data} \rightarrow \text{ComOC!}\text{msg}_0\text{.req-ack} \rightarrow \text{write-success(DataWriteSuccess}_j = \text{true;}} \rightarrow \text{Client}_0 \]

\[ <\text{(response == true)}> \]

\[ <\text{(response \_fail \rightarrow Client}_0) > \]

\[ \text{\□ InterceptCM!Client} \rightarrow \text{InterceptCO!Client} \rightarrow \text{Intruder} \]

\[ \text{Fig. 3 Client with the Intruder} \]

Here, \{[c]\} stands for the set of all communications over channel c. The first two formulas mean that the process Client performs either a ComCM!\text{msg} or InterceptCM!\text{msg} event, whenever Client$_0$ performs a corresponding ComCM!\text{msg} event. In PAT, we realize the renaming by replacing common communications with faked or intercepted communications, and combining them with common communications using the general choice \□. Since we assume that all entities are safe except for the client, so the communication from other entities to the client cannot be faked. Therefore, the ComMC!\text{msg} event in Client is performed in the same way as that in Client$_0$. The remaining three formulas have similar meanings.

### 3.4 MDS Modeling

MDS is mainly responsible for sending capability to the client, and the model of MDS is shown below.

\[ \text{MDS}_0 = \text{df ComCM!}\text{msg}_0\text{.req-ack} \rightarrow \text{ComCM!}\text{msg}_0\text{.data}\text{.Capability} \rightarrow \text{MDS}_0; \]

\[ \text{\{call(Translate, msg);\}} \rightarrow \text{ComMC!}\text{msg}_0\text{.data}\text{.Capability} \rightarrow \text{MDS}_0; \]

We use call(Fuction, p); to call a function. Function is the name of the function and p is its parameter. Here, Translate is a function defined to determine the capability. We give the details of the Translate in PAT as below.

\[ \text{\#define Translate(msg)} \]
\[ \text{if(msg == Msg0) \{Capacity = 0;\}} \]
\[ \text{else if(msg == Msg1) \{Capacity = 1;\}} \]
\[ \text{else \{Capacity = 2;\};} \]

We assume that there are three types of messages in our model, including Msg0, Msg1, and Msg2. They indicate the message is readable & writable, readable & unwritable and unreadable & unwritable, respectively.

When the communication behavior occurs, the local variable msg is assigned to the corresponding global variable (i.e., Msg0, Msg1, and Msg2). So, if the parameter msg equals to Msg0 (i.e., this message is readable & writable), Capability is updated to 0. Other cases are similar, and the MDS returns the corresponding capability to the client.

Then, we consider the actions of the intruder based on the given model MDS$_0$. As shown in Fig. 4, the intruder can intercept messages when the MDS sends messages to the client and disguise as a legal client to send fake
messages to the MDS. Similarly, the corresponding process $MDS$ is formalized using renaming as below. The first two formulas mean that the process $MDS$ performs either a $ComMC?msg$ or $InterceptMC?msg$ event, whenever $MDS_0$ performs a corresponding $ComMC?msg$ event. The next two formulas represent that when the $ComCM?msg$ event occurs in $MDS_0$, the process $MDS$ behaves like $ComCM?msg$ or $FakeCM?msg$.

$$MDS =_{df} MDS_0\{\{\begin{array}{l}
ComMC!([ComMC]) \rightarrow ComMC!([ComMC]), \\
ComMC!([ComMC]) \rightarrow \text{InterceptMC!([ComMC])}, \\
ComCM?([ComCM]) \rightarrow \text{ComCM?([ComCM])}, \\
ComCM?([ComCM]) \rightarrow \text{FakeCM?([ComCM])};
\end{array}\}\}$$

### 3.5 OSD Modeling

OSD is a physical or logical storage unit and the client can interact with OSD directly. The client first writes data to the primary OSD, and then the primary OSD writes data to the replicas. The primary OSD can be formalized as follows.

$$OSD_0 =_{df} \text{ComCO?msg}_\text{req}.req.\text{capability} \rightarrow \begin{cases}
\text{ComOC!msg}_\text{req}.true \rightarrow \\
\{\text{ComOC!msg}_\text{req}_\text{data}.msg_j \rightarrow OSD_0\}
\end{cases}$$

Once the OSD receives a “Read” request, it checks the capability and sends the message to the client if the capability is valid. While, if the request is a “Write” request and the capability is valid, the client writes the data to the primary OSD. Then, the primary OSD identifies the replicas and writes data to them. After the message has been replicated on all of the replicas, the primary OSD responds an ack to the client.

The communication between the client and the OSD is at risk of being intruded as well. Therefore, the corresponding process $OSD$ is formalized using renaming which is similar to the process $MDS$.

### 3.6 Replica Modeling

The OSD replica is mainly in charge of receiving data from the primary OSD and sending confirmed responses to the primary OSD. Since $OSDR_2$ has similar behaviors with $OSDR_1$, we just present the model of $OSDR_1$ as below.

$$OSDR_1 =_{df} \text{ComOR}_\text{!msgdata}.data \rightarrow \begin{cases}
\text{MSG}_\text{OSDR1}[i] = data; ACK1 = true; \\
\text{ComR10!msg}_\text{resp}.ACK1 \rightarrow OSDR_1;
\end{cases}$$

### 3.7 Intruder Modeling

We formalize the intruder as a process to simulate the behaviors of faking or intercepting messages. The intruder can intercept the messages transmitted over the channels of the intruder. Besides, the intruder can pretend as a legal client and send fake messages to other entities. The model is formalized as below.

$$\text{Intruder} =_{df} \begin{cases}
\square_{\text{n}0,1,2} \text{FakeCO!Read.n} \rightarrow \text{InterceptOC?response} \\
\rightarrow \text{InterceptOC?msg} \\
\rightarrow (fread_success\{\text{FakeClientR} = true;\} \rightarrow \text{Intruder}) \\
\rightarrow (fmsg = Msg) \\
\rightarrow (fread_fail \rightarrow \text{Intruder}) \\
\square_{\text{n}0,1,2} \text{FakeCO!Write.n} \rightarrow \text{InterceptOC?response} \\
\rightarrow \text{FakeCO!FData} \rightarrow \text{InterceptOC?ack} \\
\rightarrow (fwrite_success\{\text{FakeClientW} = true;\} \rightarrow \text{Intruder}) \\
\rightarrow (fack = true) \\
\rightarrow (fwrite_fail \rightarrow \text{Intruder})
\end{cases}$$

If the intruder can gain the message through faked or intercepted channels, we deem that the intruder has succeeded in reading. If the intruder can write fake data (i.e., FData) to the OSD and receive an ack, this situation is considered as the success of invalid writing. We set the value of FData written by the intruder to 1000. Here, FakeClientR and FakeClientW are boolean variables defined to verify the two cases respectively. When the intruder successfully reads or writes data (i.e., the intruder performs fread_success and fwrite_success), the corresponding variable is assigned to true.

### 4. Verification

In this section, we verify some related properties of the above constructed model with the aid of the model checker PAT.
4.1 Properties of Ceph

We verify five related properties of Ceph, including Deadlock Freedom, Data Reachability, Data Write Integrity, Data Consistency and Authentication. The assertions and the corresponding descriptions of the properties are shown below.

**Property 1: Deadlock Freedom**

Deadlock is a situation where nothing further can happen. This property means that we need to ensure the model can never reach a deadlock state. We use a primitive in PAT to describe this property.

```c
#define Data_Write_Integrity !(ACK1 == false & ACK == true) || (ACK2 == false && ACK == true);
#define CEPH [] Data_Write_Integrity;
```

**Property 2: Data Reachability**

The property refers to the reachability of data, i.e., the model should ensure correct reading and writing. We apply three assertions to describe this property.

Since we assume that Msg0 and Msg1 are readable, the first assertion is valid when `DataReadSuccess0` or `DataReadSuccess1` is true.

```c
#define Read_Success (DataReadSuccess0 || DataReadSuccess1) == true;
#define CEPH reaches Read_Success;
```

Analogously, we assume that Msg0 is writable and its initial value is 0. If the writable message Msg0 equals to the value of Data, the second assertion is valid.

```c
#define Write_Success Msg0 == Data;
#define CEPH reaches Write_Success;
```

Additionally, it is necessary to guarantee Private Data Safety when a read or write operation happens. We use the LTL (Linear Temporal Logic) formula to describe this property. [ ] is the symbol of “always”. In our model, we suppose that Msg1 is unwritable and Msg2 is unreadable & unwritable. This property represents that the client can never access the unreadable data and can never modify the unwritable data.

```c
#define PrivateData_Safety (MSG_OSD[1] != Data && MSG_OSD[2] != Data && Data_ReadSuccess2 == false);
#define CEPH [] PrivateData_Safety;
```

If all the above three assertions are valid, the property of Data Reachability is fulfilled.

**Property 3: Data Write Integrity**

This property expresses the integrity of the writing process. Only when all the replicas are written successfully, the primary OSD returns a confirmation message to the client.

`ACK1` and `ACK2` represent the replica `OSDR1` and `OSDR2` have been written respectively. `ACK` is the confirmation message sent from the primary OSD to the client.

```c
#define CEPH [] PrivateData_Safety;
```

**Property 4: Data Consistency**

This property asks whether Ceph can always satisfy data consistency or not. This assertion is equivalent to

```c
#assert CEPH deadlockfree;
```

It means that data stored in the primary OSD should be the same as those stored in the replicas if the write operation happens.

```c
#define Data_Conistency ((ACK == false) || (MSG_OSD[0] == MSG_OSDR1[0] && MSG_OSD[0] == MSG_OSDR2[0] && MSG_OSD[1] == MSG_OSDR1[1] && MSG_OSD[1] == MSG_OSDR2[1] && MSG_OSD[2] == MSG_OSDR1[2] && MSG_OSD[2] == MSG_OSDR2[2]));
#define CEPH [] Data_Conistency;
```

**Property 5: Authentication**

This property means that the intruder fails in posing as a legal client to read or write data successfully.

```c
#define AuthenticationR FakeClientR == false;
#define CEPH [] AuthenticationR;
#define AuthenticationW FakeClientW == false;
#define CEPH [] AuthenticationW;
```

4.2 Verification Results

PAT runs on the machine with core i7 and 32G memory. The verification results of Ceph are illustrated in Fig. 5.

- **Property 1 is valid.** It indicates that our model cannot get stuck in a deadlock state and is well defined.
- **Property 2 is valid.** The first two assertions are valid, which means our model has succeeded in simulating the processes of reading and writing data in Ceph. We give the traces of the first two assertions in Fig. 6. Moreover, the third assertion is valid and it shows that Ceph guarantees Private Data Safety. Thus, Ceph meets the property of Data Reachability.
Property 3, 4 are valid. It expresses that Ceph satisfies the specification of writing and ensures Data Consistency.

Property 5 is invalid. The counterexamples returned by PAT are presented in Fig. 6. They show that the intruder can read data and write fake data (i.e., FData) successfully through faked and intercepted channels in the absence of security mechanisms. Therefore, we introduced Cephx where encryption and authentication are adopted.

5. Improvement

Since Ceph cannot satisfy the Authentication property, we formalize a new model of Ceph where Cephx is adopted in this section. We model it using similar methods and feed the improved model to PAT again.

5.1 Modeling Cephx

5.1.1 Sets, Variables, Messages, and Channels

Analogously, we first introduce some newly added sets, messages, and channels. **Ticket** is the set of tickets and **Key** is the set of keys. Besides, every entity has its name and **Name** is the set of entities’ names.

The communication flows of these entities are the same, except for the specific names, keys and tickets. Therefore, we focus on the communication flows and ignored the differences in specific messages. Further, we abstract these names, keys and tickets as enumeration types based on the similarity. We list newly added variables in Table 2.

Based on the above descriptions and the processes of Cephx, we give some more messages as follows.

\[
\begin{align*}
MSG_{key} &= \{ msg_{key.n.k} | n \in Name, k \in Key \} \\
MSG_{ndata} &= \{ msg_{ndata.E(k.data)} | k \in Key, data \in Content \} \\
MSG_{req} &= \{ msg_{req.E(k.t).req.data} | k \in Key, t \in Ticket, req \in Req, data \in Content \}
\end{align*}
\]

Here, \( E(k.msg) \) indicates that \( k \) is the key used to encrypt the message \( msg \).

Then, we give the definitions of the added channels.

- Channels of processes between legal entities: \( ComCMon, ComMonC, ComMonM, ComMMon, ComMonO, ComOMon, ComMonR1, ComR1Mon, ComMonR2, ComR2Mon \)
- Channels of the intruder faking/intercepting processes: \( FakeCMon, InterceptCMon, InterceptMonC \)

5.1.2 Overall Modeling

The new model is composed of seven parts and we give it as below. We update the processes of the client, the MDS, the OSD and the intruder. We also add Monitor to indicate the actions of the monitor in Cephx.

\[
CEPHX = \{ Monitor \| NClient \| NMDS \| NOSD \| OSDR1 \| OSDR2 \| NIntruder \}
\]
5.1.3 Monitor Modeling

When Cephx is adopted, the monitor is responsible for delivering shared secret keys to entities and authenticating the client. The model of the monitor is formalized as the process Monitor as below.

\[
\text{Monitor}_0 = \text{df } \text{ComCMon}!\text{msg}_{\text{req}}.\text{name} \rightarrow \text{ComCMon}!\text{msg}_{\text{req}}.\text{ShareKey} \rightarrow \text{ComCMonM}!\text{msg}_{\text{data}}.\text{E}(\text{sessionkey}, \text{ticketkey}) \rightarrow \text{ComCMonM}!\text{msg}_{\text{data}}.\text{E}(\text{sessionkey}, \text{ticketkey}) \rightarrow \text{ComCMonR}!\text{msg}_{\text{data}}.\text{E}(\text{sessionkey}, \text{ticketkey}) \rightarrow \text{ComCMonR}!\text{msg}_{\text{data}}.\text{E}(\text{sessionkey}, \text{ticketkey}) \rightarrow \text{Monitor}_0
\]

\[
\langle \text{req } = \text{ Create} \rangle \rightarrow \text{ComCMon}!\text{msg}_{\text{data}}.\text{E}(\text{sessionkey}, \text{ticketkey}) \rightarrow \text{ComCMon}!\text{msg}_{\text{data}}.\text{E}(\text{sessionkey}, \text{ticketkey}) \rightarrow \text{Monitor}_0
\]

\[
\langle \text{Dsessionkey } = \text{ true} \rangle \rightarrow \langle \text{key}_{\text{fail }\rightarrow \text{ NClient}_0} \rangle
\]

If the monitor receives a “Create” request from the client, it creates the shared user’s secret key, stores it, and then delivers this key to the client. The monitor also shares the secret key with the MDS and the OSD by transferring ShareKey to them.

If the monitor receives an “Authentication” request, it generates a session key SessionKey and encrypts it with the client’s secret key ShareKey. If the corresponding client can decrypt and gain the session key, the monitor will receive an encrypted request. After decrypting the encrypted request, the monitor delivers a ticket encrypted with the shared secret key to the client.

Here, DeSessionKey is a function used to determine whether the provided session key can decrypt messages. The details of the function DeSessionKey are shown below. If the Monitor receives a valid key (i.e., SessionKey) after the channel input ComCMon?msgndata.E(sessionkey, req), the Monitor can decrypt this request successfully. Otherwise, it fails.

\[
\text{DeSessionKey}(\text{sessionkey}) = \{\text{if}(\text{sessionkey }= \text{ SessionKey}) \text{ then } \text{Dsessionkey } = \text{ true} \} \text{ else } \{\text{Dsessionkey } = \text{ false} \}
\]

Similarly, we consider the actions of the intruder based on the given model Monitor. We achieve it by means of renaming and omit the details.

5.1.4 New Client Modeling

According to the processes of Cephx, we make a few modifications to the Client0 described in the above section and we present the updated model NClient0 as below.

\[
\text{NClient}_0 = \text{df } \text{ComCMon}!\text{msg}_{\text{req}}.\text{Create.C} \rightarrow \text{ComCMon}!\text{msg}_{\text{req}}.\text{key}_{\text{name.sharekey}} \rightarrow \text{ComCMon}!\text{msg}_{\text{req}}.\text{Authen.C} \rightarrow \text{ComCMon}!\text{msg}_{\text{req}}.\text{E}(\text{sessionkey, ticketkey}) \rightarrow \text{call}(\text{DeSharekey, sharekey}); \text{call}(\text{VerTicket, ticket});
\]

The updated model of MDS is shown below. The MDS can gain the shared secret key from the monitor. When it receives a request from the client, it first uses the shared secret key to decrypt the request, and then checks the validity of the ticket. If the ticket is valid, it will send a response to the client. Here, VerTicket is a function adopted to judge the validity of the ticket. This function is similar to DSessionkey, and we do not expand it here.

\[
\text{NMDS}_0 = \text{df } \text{ComCMonM}!\text{msg}_{\text{key}}.\text{name.sharekey} \rightarrow \text{ComCMonM}!\text{msg}_{\text{req}}.\text{E}(\text{ticketkey, ticket}); \text{call}(\text{VerTicket, ticket});
\]
call(Translate, msg); } →
(ComMC!msg,Capability → NMDS0)
<(Dsharekey == true ∧ Vticket == true)>
(keytkt.fail → NMDS0);

The communication between the client and the MDS is at risk of being intruded as well. Therefore, the corresponding process NMDS is formalized using renaming which is similar to the process Monitor.

5.1.6 New OSD Modeling

Similarly, the new OSD model is presented below. It receives the shared key from the monitor at the beginning. Then, the OSD checks the ticket before making a response to the client when it receives a read or write request.

\[
\text{NOSD}_0 = df \text{ComMon}O?\text{msg}\text{key.name.sharekey} \rightarrow \text{ComCO}?\text{msg}\text{req.E(sharekey.ticket).req.capability}
\]
\[
\text{call}(\text{DeSharekey, sharekey}); \text{call}(\text{VerTicket, ticket}); \]
\[
\begin{cases}
\ldots \rightarrow \left(\ldots \rightarrow \begin{cases}
\text{<(capability == 0 ∧ req == Write)}>
\ldots
\end{cases}
\end{cases}
\end{cases}
\]
\[
<(Dsharekey == true ∧ Vticket == true)>
(keytkt.fail → NOSD0);
\]

The following read & write processes are the same as described in OSD, so we omit the details. Moreover, NOSD is formalized in the same way as Monitor via renaming.

5.1.7 New Intruder Modeling

The new model of the intruder is formalized as below. It is similar to the previous model of Intruder, except that the information sent is different. The intruder attempts to read and write data with forged secret keys and tickets.

\[
\text{NIntruder} = df
\]
\[
□\text{m0,1,2} \text{FakeCO!ShareKey.FTkt.Read}.n \rightarrow \text{InterceptOC?response → InterceptOC?msg → (fread.success{FakeClientR=true;} → NIntruder)}
\]
\[
\text{<}(\text{msg == Msgn}) \rightarrow \text{(fread.fail → NIntruder)}
\]
\[
□\text{m0,1,2} \text{FakeCO!ShareKey.FTicket.Write}.n \rightarrow \text{InterceptOC?response → FakeCO!FData → InterceptOC?ack → (fwrite.success{FakeClientW=true;} → NIntruder)}
\]
\[
\text{<}(\text{ack == true}) \rightarrow \text{(fwrite.fail → NIntruder)};
\]

5.2 Verification

As illustrated in Fig. 7, the new verification results of Cephx show that all properties are satisfied. We also give the detailed verification results of the 2nd, 3rd, 7th and 8th assertions in Fig. 8.

6. Conclusion and Future Work

In this paper, we have modeled the basic architecture of
Ceph with the classic process algebra CSP. Then, we utilized the model checker PAT to check some related properties of the two formalized models, including Deadlock Freedom, Data Reachability, Data Write Integrity, Data Consistency, and Authentication. The verification results indicate that our formalized model is well defined and it succeeds in simulating the processes of Ceph. Moreover, we can safely draw a conclusion that Ceph caters to Data Write Integrity and Data Consistency.

Unfortunately, it can be found that Ceph cannot guarantee the Authentication property. Hence, we formalized a new model of Ceph where Cephx authentication is adopted. The new verification results imply that Cephx realizes authentication, i.e., the intruder fails to fake as a client.

In the future, we will dive into more performance issues over Ceph. For example, we will introduce the heartbeat mechanism to our formalized model. Additionally, some other security problems of Ceph will be focused on.

Acknowledgements

This work was partly supported by the National Key Research and Development Program of China (Grant No. 2018YFB2101300), and the National Natural Science Foundation of China (Grant Nos. 61872145, 62032024).

References

[1] B. Li, M. Wang, Y. Zhao, G. Pu, H. Zhu, and F. Song, “Modeling and verifying google file system,” 2015 IEEE 16th International Symposium on High Assurance Systems Engineering, pp.207–214, IEEE, 2015.
[2] W. Xie, H. Zhu, X. Wu, S. Xiang, J. Guo, and P.C. Vinh, “Modeling and verifying hdfs using process algebra,” Mobile Networks and Applications, vol.22, no.2, pp.318–331, 2017.
[3] S.A. Weil, “Ceph: reliable, scalable, and high-performance distributed storage,” Ph.D. thesis, University of California, Santa Cruz, 2007.
[4] X. Shi, Y. Ji, H. Xie, and Y. Lu, “A prefetching mechanism based on moosefs,” International Conference on Trustworthy Computing and Services, pp.146–153, Springer, 2013.
[5] S.A. Weil, S.A. Brandt, E.L. Miller, D.D. Long, and C. Maltzahn, “Ceph: A scalable, high-performance distributed file system,” Proc. 7th Symposium on Operating systems design and implementation, pp.307–320, 2006.
[6] Ceph, “Ceph storage,” https://ceph.io/ceph-storage/, accessed March 31, 2021.
[7] C.T. Yang, E. Kristiani, Y.T. Wang, G. Min, C.H. Lai, and W.J. Jiang, “On construction of a network log management system using elk stack with ceph,” The Journal of Supercomputing, pp.1–17, 2019.
[8] D. Gudu, M. Hardt, and A. Streit, “Evaluating the performance and scalability of the ceph distributed storage system,” IEEE International Conference on Big Data, Washington, DC, USA, Oct. 27-30, 2014, pp.177–182, 2014.
[9] S. Xiang, H. Zhu, X. Wu, L. Xiao, M. Bonsangue, W. Xie, and L. Zhang, “Modeling and verifying the topology discovery mechanism of openflow controllers in software-defined networks using process algebra,” Science of Computer Programming, vol.187, p.102343, 2020.
[10] A. Liu, H. Zhu, M. Popovic, S. Xiang, and L. Zhang, “Formal analysis and verification of the pstm architecture using csp,” Journal of Systems and Software, vol.165, p.110559, 2020.

[11] G. Lowe and B. Roscoe, “Using csp to detect errors in the tmn protocol,” IEEE Trans. Softw. Eng., vol.23, no.10, pp.659–669, 1997.
[12] A.W. Leung and E.L. Miller, “Scalable security for large, high performance storage systems,” Proc. second ACM workshop on Storage security and survivability, pp.29–40, 2006.
[13] C. Maltzahn, E. Molina-Estolano, A. Khurana, A.J. Nelson, S.A. Brandt, and S. Weil, “Ceph as a scalable alternative to the hadoop distributed file system,” login: The USENIX Magazine, vol.35, pp.38–49, 2010.
[14] J. Kohl, C. Neuman, et al., “The kerberos network authentication service (v5),” Tech. Rep., RFC 1510, Sept. 1993.
[15] C.A.R. Hoare, “Communicating sequential processes,” Communications of the ACM, vol.21, no.8, pp.666–677, 1978.

Ran Li is currently a Ph.D. candidate in Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, Shanghai. Her research interests contain Hoare logic, formal methods and process algebra.

Huibiao Zhu is currently a professor in East China Normal University. He earned his Ph.D. degree in formal methods from London South Bank University, London, in 2005. During these years, he has studied various semantics and their linking theories for Verilog, SystemC, web services and probability system. He was the Chinese PI of the Sino-Danish Basic Research Center IDEA4CPS.

Jiaqi Yin is currently a Ph.D. candidate in Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, Shanghai. His research interests contain formal methods, mobile edge computing, cloud computing and process algebra.