Clarification of Ambiguity for the Simple Authentication and Security Layer

Farah Al-Shareefi, Alexei Lisitsa, and Clare Dixon

Department of Computer Science, University of Liverpool
Liverpool, L69 3BX, UK
{F.M.A.Al-Shareefi,lisitsa,cldixon}@liverpool.ac.uk

Abstract. The Simple Authentication and Security Layer (SASL) is a framework for enabling application protocols to support authentication, integrity and confidentiality services. The SASL was originally specified in RFC 2222, and later updated in RFC 4422, using natural language. However, due to the richness of natural language this involves ambiguities and imprecision. Whilst there is an Oracle implementation of SASL, its documentation also contains informal descriptions and under-defined specifications of the RFCs. This paper provides clarification of ambiguity in SASL using Abstract State Machines (ASMs). This clarification is based on two ASM essential notions: a ground model to capture the intended SASL behavior in an understandable way, and a refinement notion to accurately explicate the ambiguous parts of the behavior. We also show some differences between RFCs and the description of the Oracle implementation. We believe our work can serve as a basis for further implementation and for formal analysis.

Keywords: Ambiguity, Simple Authentication and Security Layer, Abstract State Machines

1 Introduction

The Simple Authentication and Security Layer (SASL) is a framework that can be used by application protocols to perform authentication, and to optionally supplement it with what is called security layer services, including integrity and confidentiality. SASL was firstly described in Requests for Comments (RFC) 2222 [14], and then in RFC 4422 update [13], using natural language. Unfortunately, the RFCs, being stated in natural language that intrinsically has associated informality and imprecision, are sometimes ambiguous. Despite that, there is an Oracle implementation of SASL [15], its documentation also includes textual explanations and some unclear specification of the RFCs. In addition, this implementation involves hidden details about the functions which are called to achieve specific tasks.

To overcome the imprecision and ambiguity problems, formal methods can be used as they are based on mathematical foundations [8]. Among these methods, we choose the Abstract State Machine (ASM) method [11], since it can be used to specify systems in a rigorous mathematical, understandable, and scalable way [6].

In this paper, the ambiguities of the SASL textual explanations in the RFCs and Oracle implementation documents, are analyzed formally using the ASM method. This is achieved by implementing two strategies of the ASM method. First, the ground model directly captures the informal SASL behavior in an understandable and concise but precise enough manner. Second, the refinement strategy allows us to precisely explicate and re-elaborate the under-defined notions in the ground model. The refined specification is written in the executable ASMETA Language (AsmetaL) [10], since it is close to the ASM mathematical concepts, and it directly permits us to test specification errors.

The main contributions of this paper are:

– clarifying the ambiguities of the SASL informal descriptions in RFCs and Oracle implementation documents clearly in terms of ASMs;
– presenting a methodology for clarifying ambiguity that starts with the RFC document to capture its informal description, via the ASM ground model, then it explicates the potential description ambiguities depending on other document sources by using ASM refinement;
– highlighting the main differences between RFCs and the Oracle documents.

The rest of this paper is organized as follows. Section 2 presents background knowledge about the SASL framework and the ASM method. Section 3 describes the ASM formal specification, and highlights how this specification elucidates the main ambiguities of SASL. Section 4 discusses our results. Section 5 presents some related work. Finally, Section 6 concludes the paper.

2 Background

In this section, we describe both the Simple Authentication and Security Layer framework and the Abstract State Machine Method.

2.1 Simple Authentication and Security Layer

The Simple Authentication and Security Layer (SASL) was initially introduced in RFC 2222 [14], and later updated in RFC 4422 [13], as a framework for providing authentication support with an optional security layer service, such as integrity or confidentiality, to connection-oriented protocols, via substitutable mechanisms. Providing these services is achieved through using a shared abstraction layer which has a structured interface between intended protocols and mechanisms. With this layer, any SASL supported protocol, such as IMAP [18], SMTP [19], etc., can exploit any SASL supported mechanism, such as PLAIN [20], DIGEST-MD5 [12], etc.

Based on RFC 2222/4422 [13,14], the client and server of the SASL protocol application launch a negotiation about the selection of a suitable mechanism, then they negotiate the authentication. Basically, the client requests to connect with the server using SASL. Then, the server replies with a list of supported authentication mechanisms. Next, the client selects the best mechanism. After that, the authentication is started by the client via sending an authentication command, which involves the selected mechanism and optionally authentication data, to the server. The authentication exchange continues until the authentication succeeds, fails, or is aborted by the client or the server. During the authentication exchange, when the selected mechanism supports the security layer, the client and server negotiate the use of a security layer. If they both agree about using it, then both sides must negotiate the maximum size for the cipher text buffer, that each side is able to receive. The RFCs specification, however, leaves open a number of questions, in particular: how the server advertises its mechanisms' list, how the client selects the best mechanism, when the client and server agree about using the security layer and how it can be used, and how they negotiate the maximum cipher text buffer size. Some of these questions relate to the ambiguity and missing details of the informal description for the API routines in the Oracle implementation documentation [15].

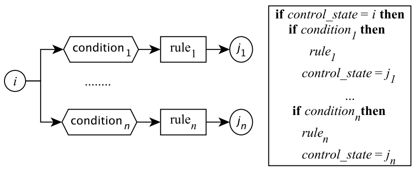
According to the Oracle implementation [15], the application communicates with the structured interface by calling a suitable API routine, which in turn calls a mechanism plug-in interface. One of these routines is: the `sasl_client_start()` which is called by the client to select the best mechanism depending on the security properties. The main properties that restrict mechanism selection are: the **security policies**, such as NOPLAINTEXT, NOACTIVE, NOANONYMOUS, etc., and the maximum **Security Strength Factor (SSF)** [15] for the client, server, and mechanism. The SSF is an integer that denotes the security layer strength. When it is zero, it indicates only authentication, if it is one, it means both authentication and integrity, while if it is greater than one, it denotes authentication, integrity, confidentiality, and at the same time the key length for encryption. Also the server calls the `sasl_listmech()` routine to obtain the mechanisms' list that satisfies the security policies.

2.2 Abstract State Machines

Abstract State Machines (ASMs) [11] were first introduced by Gurevich as a versatile machine to model any algorithm at an appropriate level of abstraction. ASMs have been developed to a practical and mathematically well-founded method for high-level system design and analysis [6]. The ASM method is constructed from three essential notions: *ASMs, a ground model, and stepwise refinement* [6].

ASMs are transition systems which are based on abstract states, to model the system's structure, and on transition rules, to model the system's dynamic behavior. The ASM states are multi-sorted first-order structures, i.e, domains of objects coming with functions and relations defined on them. The functions in ASM states can be *static*, which are never updated, *controlled*, which are updated by the machine itself, or *monitored*, which are updated by the machine's environment. ASM transition rules describe the modification of function interpretations from one state to the subsequent one. The basic transition rule is a *function update*: $f(t_1, ..., t_n) := t$. f is an arbitrary n-ary function and $t_1, ..., t_n, t$ are first-order terms, which are simultaneously updated to yield a new ASM state. There are some rule constructors, such as: **if then** (conditional rule), **par** (parallel execution of the grouped rules), **choose** (non deterministic selection), and **switch case** (extension of the conditional rule). ASMs can capture the formalization of a procedural *single-agent* and distributed *multiple agents* interacting in a synchronous and asynchronous way.

There is a specific class of ASMs called *control state ASMs* [6]. They can be employed for describing various system modes. Figure 1 shows a graphical representation of control states and the form of their transition rules.



i
condition$_1$
rule$_1$
j_1
....
condition$_n$
rule$_n$
j_n
if *control_state* = *i* **then**
if *condition$_1$* **then**
rule$_1$
control_state = *j_1*
...
if *condition$_n$* **then**
rule$_n$
control_state = *j_n*

Fig. 1: Control state ASMs

Ground model ASMs are conceptual models for capturing informal requirements of a system in a precise, concise, flexible, and understandable way. The ground model can be represented graphically using control state ASMs. From a concise ground model, by **stepwise refinement**, a more detailed model can be obtained, through changing the states definition, or the flow of operations, or both of them.

Several projects have been developed around ASMs to make them executable, such as CoreASM [1], the ASMETA[1] [9] framework, etc. In this paper, we have chosen the ASMETA framework, that includes integrated tools, in particular the ASMETA Language (AsmetaL) and the ASMETA Simulator (AsmetaS) for writing and executing ASM models [10], respectively. The AsmetaL supports encoding of ASM models which is close to the ASM mathematical concepts.

3 The Formal SASL Specification

In this section, we show how the ASM method has been used to provide formal specifications for SASL. The main aim is to clarify precisely: how the server advertises the available mechanisms, how the client selects the best mechanism, how the client determines the maximum size for the cipher buffer, and how and when the security layer is negotiated. As the SASL framework has detailed and complex behavior, we separate the SASL into three phases: the mechanism negotiation phase, the authentication negotiation phase, and the security layer negotiation phase. In each phase, we will present (if necessary) the ground model for both client and server sides, that is depicted via the control state ASM, then we will focus only on refining the rules to clarify ambiguities in the RFCs and Oracle implementation documentation[2]. Each refined rule is expressed in As-

[1] http://asmeta.sourceforge.net/.
[2] All the rules for the refined model that is based on RFC 2222/4422 are available online at https://doi.org/10.5281/zenodo.1204257, while for the refined model which

metaL. The mapping from the graphical notation of the control state ASM to the AsmetaL notation is done according to the mapping shown in Fig. 1.

3.1 The Mechanism Negotiation Phase

Fig. 2 shows the ground model at the client side for this phase. This figure is a direct interpretation of RFC 2222/4422. The client starts this phase by sending a request to ask the server to send its mechanisms' list. Whenever this list is received, the guard *At least one mechanism in the list is supported* checks if the client allows any mechanism in the list. If so, the client selects an acceptable mechanism from the server list and reaches the final state for this phase *Sending authentication request*. Otherwise, the client will send an abort response to the server, and waits for an abort reply from it. When the abort reply is received, the client aborts this exchange, by entering the *Abort* state.

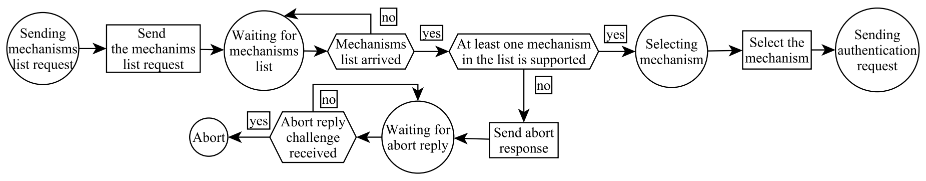

Sending mechanisms list request
Send the mechanims list request
Waiting for mechanisms list
no
Mechanisms list arrived
yes
At least one mechanism in the list is supported
yes
Selecting mechanism
Select the mechanism
Sending authentication request
no
Send abort response
Waiting for abort reply
no
Abort reply challenge received
yes
Abort

Fig. 2: Client side for mechanism selection phase - ground model

The server side for this phase is also based on RFC 2222/4422, see Fig. 3.

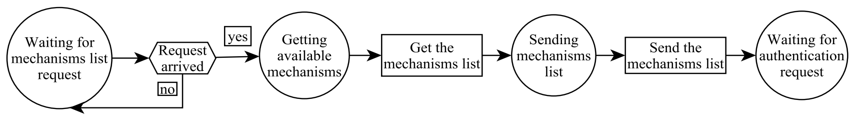

Waiting for mechanisms list request
Request arrived
yes
no
Getting available mechanisms
Get the mechanisms list
Sending mechanisms list
Send the mechanisms list
Waiting for authentication request

Fig. 3: Server side for mechanism selection phase - ground model

The ground model depicted in Fig. 3, schedules the main steps taken by the server for this phase. Initially, the server keeps waiting at the *Waiting for mechanisms list request* state until it receives a mechanism list request from the

is based on the description of Oracle implementation documentation are available at https://doi.org/10.5281/zenodo.1204242 client. When this request arrives, the server obtains the available mechanisms' list to send it to the client. After sending this list, the server goes to the final state for this phase, which is *Waiting for authentication request*.

It is not clear from Fig. 2, how the client chooses the desired mechanism. As stated by RFC 4422 [13], determining the best mechanism is the client's choice. This is specified in the `r_selectmech` rule shown in Code 1 (a). In this Code, the mechanism selection is performed in an interactive manner with the client via the monitored function `insertMechanism`. The selected mechanism should be any mechanism in the arrived mechanisms' set, which is represented by the `arrivedMechList`. An arbitrarily chosen mechanism is stored in `selMech`.

```
rule r_selectmech=
  if contains(arrivedMechList , insertMechanism) then
       selMech:=insertMechanism
  endif
```

(a) The r_selectmech rule according to RFC 4422

```
function mHasGreatestSSF($m in Mechanisms, $c in Client)=
 forall $x in arrivedMechList with
   (($x!=$m) and allin(policies($x), policies($c)) implies
                                                ssf($m)>=ssf($x))
rule r_selectmech=
  choose $m in arrivedMechList with
      allin(policies($m), policies(self)) and
      mHasGreatestSSF($m, self)=true do
          selMech:=$m
```

(b) The refined r_selectmech rule according to Oracle implementation document

Code 1: The r_selectmech rule

On the other hand, the explanation of the Oracle implementation documentation [15] states that the client selects the best mechanism, depending on the maximum mechanism SSF and client's security policy. This can be re-elaborated by refining the rule in Code 1 (a) into the one shown in Code 1 (b). In the refined rule, we added further modelling vocabulary. Precisely, let the `ssf($m)` function be the SSF value for each mechanism `$m` in the `Mechanisms` domain, and `policies($m)` be the security policies set for each mechanism `$m`, while `policies(self)` is the security policies' set for client. The 0-ary function `self` is interpreted by the client agent as itself. Each `policies` set can be one or more elements from the domain `Policies`={NOPLAINTEXT, NOANONYMOUS, NOACTIVE, MUTUALAUTH, NODICTIONARY}. The `mHas-GreatestSSF` function returns true if the selected mechanism has the greatest SSF value. The refined rule picks the best mechanism from the `arrivedMech-List`, such that the security policies set of the selected mechanism includes all the elements in the client's set, and this mechanism has SSF value, which is greater than all the SSF values of the mechanisms that their sets include the client's policies set.

On the server side in Fig 3, getting the available mechanisms' list needs elucidation. As indicated by RFC 4422 [13], the server just advertises the available mechanisms' list. This is specified in the `r_getmechs` rule shown in Code 2 (a). In this code, let the `mList($c, self)` be a set of the advertised mechanisms which will be sent to the `$c` client. The `saslmechs(self)` set contains one or more SASL mechanisms for server use. The server, in the `r_getmechs` rule, will simply make a copy of all the elements in the `saslmechs(self)` set and pass it to the `mList($c, self)`, which is initially empty set, to represent the advertised mechanisms' list.

```
rule r_getmechs($c in Client)=
  mList($c,self):=saslmechs(self)
```

(a) The r_getmechs rule according to RFC 4422

```
rule r_getmechs($c in Client)=
  let ($i=0) in
    while $i<size(saslmechs(self)) do
      seq
        let($m=at(asSequence(saslmechs(self)), iton($i))) in
          if (exist $p in policies(self) with
             contains(policies($m), $p)=true) then
               mList($c,self):=including(mList($c,self),$m)
          endif
        endlet
        $i:=$i+1
      endseq
  endlet
```

(b) The refined r_getmechs rule according to Oracle implementation document

Code 2: The r_getmechs rule

Obtaining the available mechanisms' list is described in the Oracle implementation documentation [15], as "The server can call sasl_listmech() to get a list of the available SASL mechanisms that satisfy the security policy". In this quoted statement, it is not obvious whether there is a specific policy for the SASL mechanisms and what is meant to satisfy this policy. In the Java security guide provided by Oracle [16], it says that there is a particular policy set for each SASL mechanism, such as the {NONANONYMOUS}, {NOPLAINTEXT, NOACTIVE, NODICTIONARY}, and {NONANONYMOUS, NOPLAINTEXT} for the PLAIN, EXTERNAL, and DIGEST-MD5 mechanisms, respectively. As an attempt to understand the exact meaning of 'satisfy the security policy', we analyse the server's reply (sending the available mechanisms' list to the client) in some SASL mechanism examples. For instance, in the DIGEST-MD5 mechanism example [12], the server sends the {PLAIN, DIGEST-MD5} list. We can see that these two mechanisms share the NONANONYMOUS policy. This means that the server adopts the NONANONYMOUS policy and it sends the mechanisms which satisfy this policy. Similarly, in the EXTERNAL mechanism example [13], the server sends the {DIGEST-MD5, EXTERNAL}. Again, these mechanisms in the list share the NOPLAINTEXT policy, which is supported by the server. Accordingly, the `r_getmechs` rule in Code 2 (a) can be refined into the rule in Code 2 (b).

In the refined `r_getmechs` rule, the server gets a mechanism from the `saslmechs` for the server use, which satisfies the following condition: the policy set for this mechanism contains a policy of the server's policies set. In other words, the policy set for every mechanism in the advertised mechanisms' list supports at least one server's policy.

3.2 The Authentication Negotiation Phase

This phase is the longest phase in SASL. As a result, we divide the ground model for both client and server into two parts: one for achieving an initial step in this phase, and one for performing the later step(s). The number of the later steps is determined by the selected mechanism. Due to space restrictions we do not provide all of these constructed models[3]. The ground model for the client agent of the initial and later step(s) includes (if necessary) the `Get response` rule to get the required authentication data to the server. While, the ground model for the server agent of the initial and later step(s) includes (if necessary) the `Get challenge` rule to get the required authentication data to the client.

One underspecified aspect of this phase, is the negotiation about using the security layer and the maximum cipher buffer size, which are involved in the `Get response` and `Get challenge` rules on the client and server sides, respectively. In RFC 4422 [13], it was stated that when the selected mechanism supports a security layer, then a negotiation about using this layer must be carried out, but how this negotiation takes place is not defined. However RFC 2222 [14] defines this by stating that the negotiation includes exchanging a **bit-mask** (1: no security layer, 2: integrity, and 4: privacy), which corresponds to a security layer level. This bit-mask defines the unstated privacy service and ignores the confidentiality service.

On the other hand, in the explanation of the Oracle implementation documentation [15], the SSF value (0: authentication, 1: authentication and integrity, and > 1: authentication, integrity, confidentiality and the key length), is used instead of a bit-mask. However, it is not clear how the client and server agree about using a security layer.

The Java security guide provided by Oracle [16] states that the selected mechanism, when its SSF value is greater than or equal to 1, tells the server to send its supported **Quality of Protection** (QOP) list, which includes one or more items from the following: **auth** (authentication), **auth-int** (authentication and integrity), and **auth-conf** (authentication, integrity, and confidentiality). Later,

[3] The full ground models are available online at https://doi.org/10.5281/zenodo.1200216

the client selects a protection value from this list according to its SSF value, and sends it to the server. The server verifies that the client's protection value is within its list, to save the session SSF value which is equivalent to the client's protection value. The saved SSF value represents the agreed security layer service. However, in this guide, there is insufficient detail about how the client and server determine the maximum buffer size when they agree about using the confidentiality service.

In the DIGEST-MD5 SASL mechanism example [12], it was stated that when the server sends its supported maximum buffer size (if desired), the client will check the availability of buffer size value in the received challenge. If it exists, the client will determine that the buffer size for this session is equal to subtracting 16 bytes from the minimum size of the received one and the client's supported one. If it is not available, the client will determine that the buffer size is equal to the default value 65536. Following this description, we present in Code 3 the specification of how the client determines the maximum buffer size.

```
if contains(receivCh(self), "maxbuf")=true then
  choose $max in Maxbuf with
  eq(at(receivCh(self), iton(indexOf(receivCh(self), "maxbuf")+1)),
  toString($max)) do
      if $max<maxBuf(self) then
         maxBufDetermined:=$max-16
      else
         maxBufDetermined:=maxBuf(self)-16
      endif
else
   maxBufDetermined:=65520
endif
```

Code 3: Specifying the maximum buffer size in the client side

In Code 3, the `receivCh(self)` is a sequence of String that represents the received challenge from the server, the integer domain `Maxbuf` contains the following possible values for the buffer's size {65535, 131071, 262143, 16777215}, and the `maxBuf(self)` is the client's maximum buffer size. First of all, the client checks if the `receivCh(self)` contains the server's maximum buffer size, to calculate the buffer size, or to set it to the default value. At the calculation, the client chooses an integer value from the `Maxbuf` domain, since the `receivCh(self)` is a sequence of String, which is equal to the string value contained in the `receivCh(self)`. Then the chosen value is compared with the client's buffer size to determine the buffer size, which is stored in `maxBufDetermined`.

3.3 The Security Layer Negotiation Phase

This phase is an optional phase. Performing this phase depends on the negotiation in the previous phase. This negotiation includes exchanging a bit-mask according to RFC 2222 [14], while it includes exchanging SSF value according to the Oracle documentation [15]. As we stated previously that the bit-mask does not define the confidentiality service, we specify this phase relying on the Oracle implementation documentation [15], as well as the RFC 4422 [13]. Furthermore, the specification for integrity and confidentiality protected messages are based on the RFC 2831 for the DIGEST-MD5 SASL mechanism [12], because the RFC 2222/4422 and the Oracle implementation documentation do not illustrate this specification. We annotate the main information for specifying this phase in Fig. 4.

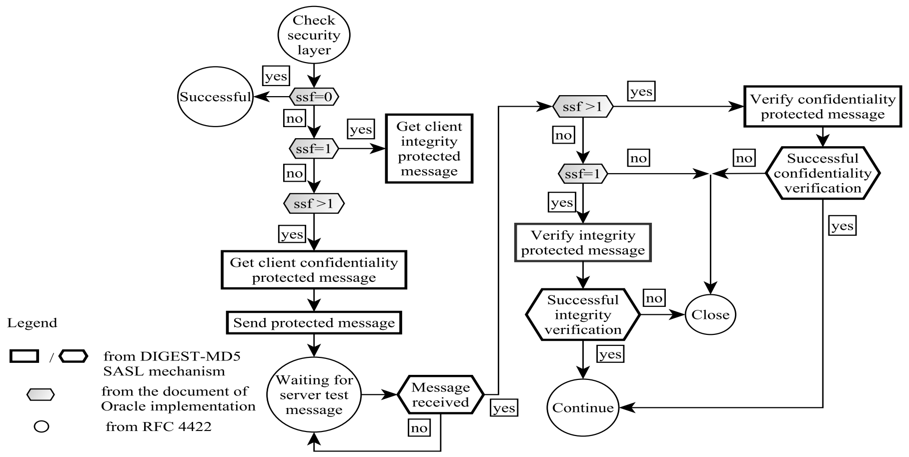

Check security layer
yes
Successful
ssf=0
no
ssf=1
yes
Get client integrity protected message
no
ssf >1
yes
Get client confidentiality protected message
Send protected message
Waiting for server test message
Message received
no
yes
ssf >1
yes
Verify confidentiality protected message
no
ssf=1
no
no
Successful confidentiality verification
yes
Verify integrity protected message
Successful integrity verification
no
Close
yes
Continue
Legend
from DIGEST-MD5 SASL mechanism
from the document of Oracle implementation
from RFC 4422

Fig. 4: Client side for security layer negotiation phase - ground model

Fig. 4 illustrates the ASM ground model for negotiating the security layer service on the client side. The client starts this phase by checking the SSF value, that was agreed by both client and server in the authentication phase. If this value is zero, then the client will reach the final state *Successful*. This state indicates that the client has been authenticated successfully, and there is no security layer. If the SSF value is one, this means that the subsequent protocol messages must be integrity protected. Therefore, the flow goes to execute the *Get client integrity protected message*, to obtain a test message that appends with the computed Message Authentication Code (MAC) for the message sequence number and the message itself [12]. While if the SSF value is greater than one, then the following protocol messages must be confidentiality protected (encrypted). As a result, the client executes the *Get client confidentiality protected message*, to encrypt a test message together with its computed MAC [12]. The encryption is done according to the selected cipher, which is one of the following: rc4-40 (40 bit key), rc4-56 (56 bit key), rc4 (128 bit key), and aes-ctr (128 bit key). Later, the client sends the protected message to the server, and changes its state to *Waiting for server test message*. Whenever the client receives a protected message from the server, it will check the agreed SSF value. When this value is greater than one, the client will perform confidentiality verification (decrypt the message, compute the MAC and compare it with the received one). While, if the SSF value is one, the client will perform integrity verification (compute the MAC and compare it with the received one). In case that the verification succeeds, the client reaches the final state *Continue*, which means the client can continue the interactions after SASL. Whereas, the client terminates the connection with the server and changes its state to *Close*, when the verification fails.

As the ground model in Fig. 4, clearly shows how the client uses a security layer service, and how the SSF value guides the client to determine whether a security layer has been negotiated, we do not show the refinement for this model. Furthermore, we do not present the server ground model for this phase, since it is similar to the client one, except that the server keeps waiting for a protected message from the client before sending its message.

As in this phase we need to encrypt and decrypt the message with regards to a suitable cipher, we specify the encryption and decryption actions in an abstract manner based on the ASM features, see Code 4.

```
dynamic abstract domain CipherText
controlled key: CipherText -> String
controlled plainMsg: CipherText -> Seq(String)
controlled plainText: Seq(String)
controlled cipher: CipherText
controlled method: CipherText -> String
rule r_encrypt($msg in Seq(String), $key in String, $method in String)=
  choose $e in CipherText with plainMsg($e)=$msg do
    cipher:=$e
  ifnone
      extend CipherText with $ciph do
       par
         plainMsg($ciph):=$msg
         key($ciph):=$key
         method($ciph):=$method
         cipher:=$ciph
       endpar
rule r_decrypt($cipher in CipherText, $key in String, $method in
                                                            String)=
  choose $ciphtext in CipherText with ($ciphtext=$cipher) and
     (key($ciphtext)=$key) and (method($ciphtext)=$method) do
       plainText:=plainMsg($ciphtext)
  ifnone
       plainText:=""
```

Code 4: Abstract specification for encryption and decryption

In Code 4, first, we introduce the specification signature. The `CipherText` is an infinite domain for the cyphertext. The unary function `key` represents the key for a given `CipherText` element. The nullary function name `plainMsg` is a sequence of Strings for the plain text. The `cipher` is an element of `CipherText` domain. The `method` is a cipher method that has been used to encrypt the plain message.

After the signature we specify two rules: the `r_encrypt` rule for converting the presented plain message into an encrypted one using the determined key and method, and the `r_decrypt` rule which transforms the encrypted message into a plain text one using a specific key and method. The `r_encrypt` rule, firstly, chooses an element in the `CipherText` domain, such that the plain text for this element is equal to the given message. This element represents the cyphertext for the message. When choosing an element returns nothing (the presented message has not been encrypted previously), this rule will generate a new cyphertext, given its plain text, key, and method, by extending the `CipherText` domain. While `r_decrypt` rule choose a cyphertext item from the `CipherText` domain, in such a way that this item equals to the given cyphertext, to return the plain text of this item. If there is no such item, this rule will return the empty string.

4 Results and Discussion

The main aim of this paper is to provide clarification of ambiguities in SASL using ASMs. Our methodology starts with reflecting the textual description in RFCs, using ground model notion, then it re-elaborates this description using other document sources by exploiting the refinement notion. Table 1 outlines the main ambiguities that have been investigated, and the source documents for both the ambiguity itself and its formal clarified specification.

From Table 1, we can see the following:

(1) selection of the best mechanism is ambiguous in RFC 4422 [13], as it just states that the client selects the best one. We try to elucidate this using the description of the Oracle implementation [15], which states that the client selects the best mechanism with the maximum SSF, and according to its security policy;
(2) advertising the available mechanisms' list is not clear in both RFC 4422, which only states that the server advertises the list, and the Oracle implementation, which states that the server advertises the mechanisms that satisfy the security policy. We convert the informal description of Oracle into a formal one, based on analysing the server reply in the document sources shown in Table 1. We conclude that satisfying the security policy means at least one server's policy must be supported by every mechanism in the advertised list;
(3) determining the maximum buffer size is under-defined in RFC 4422 [13] and the Oracle implementation. For explicating that, we use the explanation that is provided by the DIGEST-MD5 SASL mechanism [12];
(4) using the security layer in RFC 2222 needs more explication, as it states that using this layer relates to the agreed bit-mask, which does not consider the confidentiality service. Therefore, we rely on the Oracle implementation, that uses the SSF instead of a bit-mask, to show when this layer is used. Also, we rely on the DIGEST-MD5 SASL mechanism [12], to show how the client and server negotiate this layer.

This paper shows how the ASM formalism is valuable in clarifying the ambiguity, especially with its ground model and the refinement notions. The ground model can first capture the informal specification in understandable way and at the desired level of details. Then, it can be evolved via stepwise refinement into a precise and enhanced mathematical specification.

Table 1: The source document for each ambiguity and its formal clarified specification

No.	The ambiguity	The document source for ambiguity	The clarified specification	The document source for clarification
1	The client selects the best mechanism	RFC 4422 [13]	Code 1 (b)	Oracle implementation document [15]
2	The server advertises the available mechanisms' list	RFC 4422 [13], and Oracle implementation document [15]	Code 2 (b)	DIGEST-MD5 SASL mechanism [12], Oracle implementation document [15], and its Java security guide [16]
3	Determining the maximum cipher text buffer size	RFC 4422 [13], and Oracle implementation document [15]	Code 3	DIGEST-MD5 SASL mechanism [12]
4	How and when the security layer is negotiated	RFC 2222 [14]	Ground model in Fig. 4	Oracle implementation document [15],DIGEST-MD5 SASL mechanism [12], and RFC 4422 [13]

As we construct a formal specification and provide links between it and informal or underdefined resources, we could prove properties of the development specification using the ASMETA framework in a similar way to [2].

We present an executable AsmetaL specification for private key encryption and decryption, in an abstract style.

In our ASM specification, the timing aspects for SASL are not considered, since neither of RFC 2222/4422 and Oracle implementation documents give specification for that.

5 Related Work

Our work elucidates ambiguities in the informal description for SASL, based on the ASM method. Therefore, we will now discuss other work related to either elucidating ambiguity or to the ASM method.

In [4], the ASM formalism is used to get a formal model of the Kerberos Authentication System which is based on the Needham and Schroeder authentication protocol. The formal model is used as a basis to locate the minimum assumptions to guarantee the correctness of the system and to analyse its security weaknesses.

In [7], the ASM ground models of a content adaptation system employed for the interactions between different client devices and the Cloud, is presented. This work is extended in [3], by refining the initial model into a more detailed one, through focusing on the interactions between the client and the middleware server to retrieve information relating to the client's device. Furthermore, the modelling process has been supported by validation and verification activities which are integrated within the ASMETA framework.

In [17], abstract encryption and decryption is specified using the language AsmL. This specification is based on the object-oriented features and constructs, and thus it diverts from the theoretical model of ASMs.

The researchers in [5] use Higher-order logic (HOL4) to develop a rigorous post-hoc specification for TCP, UDP, and the Sockets API, that reflects the behavior of different implementations, include: FreeBSD 4.6, Linux 2.4.20-8, and Windows XP SP1. They validate their specification against several thousand traces captured from these implementations, to test whether they meet this specification. This paper is notable in the context of our work as its authors are motivated by increasing clarity and precision over ambiguous informal specifications of the RFC, that may result in inconsistent implementations. In this paper, we do not consider validating that the implementation meets the specification. We focus on clarifying ambiguities in the RFC description, and on elucidating uncertainty in the textual explanation of the implementation. Furthermore, our specification is expressed using the ASM method, because it is accessible, as it requires a minimum of notational coding, unlike HOL4, which requires extensively annotating the mathematical definitions side-by-side with informal specification [5].

6 Conclusion and Future Work

We have provided the ASM specifications that elucidate ambiguities in the SASL framework. We have focused on the ambiguous parts in RFC 2222/4422 and Oracle implementation documents, including mechanism selection, providing mechanisms' list, defining when and how the security layer can be used, and determining the maximum cipher buffer size.

We have showed how the comprehensible specification has been achieved based on two ASM notions: a ground model and stepwise refinement. The ground model enabled us to reflect the desired behavior, which is explained in RFCs, in an understandable way. While the stepwise refinement helped us to explicate the ambiguous part of the desired behavior in an accurate way, using other document sources to inform us.

We convert the informal specification into formal one by expressing it in the ASM formalism, which is mathematically well-defined, precise, and easily understood.

To further our research we are planning to consider the security of the SASL, to show whether the SASL specification is secure. We intend to use a suitable security analysis technique to elicit security requirements for the SASL and to verify them at the verification level.

Acknowledgments

The third author was partially supported by the EPSRC funded RAI Hub FAIR-SPACE (EP/R026092/1).

References

1. The CoreASM Project. http://www.coreasm.org/
2. Al-Shareefi, F., Lisitsa, A., Dixon, C.: Abstract state machines and system theoretic process analysis for safety-critical systems. In: Brazilian Symposium on Formal Methods. pp. 15–32. Springer (2017)
3. Arcaini, P., Holom, R.M., Riccobene, E.: ASM-based formal design of an adaptivity component for a Cloud system. Formal Aspects of Computing 28(4), 567–595 (2016)
4. Bella, G., Riccobene, E.: Formal analysis of the Kerberos authentication system. Journal of Universal Computer Science 3(12), 1337–1381 (1997)
5. Bishop, S., Fairbairn, M., Norrish, M., Sewell, P., Smith, M., Wansbrough, K.: Engineering with logic: HOL specification and symbolic-evaluation testing for TCP implementations. pp. 55–66. ACM Press (2006)
6. Börger, E., Stärk, R.: Abstract State Machines: A Method for High-Level System Design and Analysis. Springer, Heidelberg (2003)
7. Chelemen, R.M.: Modeling a web application for cloud content adaptation with ASMs. In: Cloud Computing and Big Data (CloudCom-Asia), International Conference on. pp. 44–55. IEEE (2013)
8. Froome, P., Monahan, B.: The role of mathematically formal methods in the development and assessment of safety-critical systems. Microprocessors and Microsystems 12(10), 539–546 (1988)
9. Gargantini, A., Riccobene, E., Scandurra, P.: Model-driven language engineering: The ASMETA case study. In: Software Engineering Advances. ICSEA. The Third International Conference on. pp. 373–378. IEEE (2008)
10. Gargantini, A.M., Riccobene, E., Scandurra, P.: A metamodel-based language and a simulation engine for abstract state machines. J. Univ. Comput. Sci. 14(12), 1949–1983 (2008)
11. Gurevich, Y.: Evolving algebras 1993: Lipari guide. In: Specification and Validation Methods, pp. 9–36. Oxford University Press (1995)
12. Leach, P., Newman, C.: Using Digest Authentication as a SASL Mechanism. RFC 2831 (2000)
13. Melnikov, A., Zeilenga, K.: Simple Authentication and Security Layer (SASL). RFC 4422 (2006)
14. Myers, J.: Simple Authentication and Security Layer (SASL). RFC 2222 (1997)
15. Oracle: Writing applications that use SASL. In: Developer's Guide to Oracle Solaris® 11 Security, chap. 7, pp. 126–148. Oracle (2012)
16. Oracle: Java SASL API Programming and Deployment Guide. In: Java Platform, Standard Edition Security Developers Guide, chap. 10, pp. 21–28. Oracle (2016)
17. Rosenzweig, D., Runje, D., Slani, N.: Privacy, abstract encryption and protocols: an ASM model-part I. In: Abstract State Machines 2003. pp. 372–390. Springer (2003)
18. Siemborski, R., Gulbrandsen, A.: IMAP Extension for Simple Authentication and Security Layer (SASL) Initial Client Response. RFC 4959 (2007)
19. Siemborski, R., Melnikov, A.: SMTP Service Extension for Authentication Initial Client Response. RFC 4954 (2007)
20. Zeilenga, K.: The PLAIN Simple Authentication and Security Layer (SASL) Mechanism. RFC 4616 (2006)