Our aim is to soundly tie not two (implementation and formal model) but three elements of the protocol development process: implementation, formal model and specification. By the latter, we mean a human-readable protocol description that is usually placed in RFC. The models' languages, which are based on logics or special versions of general-purpose programming languages, are not quite suitable for this task: they are either not convenient for capturing the low-level details or are firmly imperative.

Therefore, our goal is a declarative specification language that could be directly used in the RFCs to considerably enhance the degree of formalization of these documents. Yet, the specification must be automatically translatable both to the interoperable implementation and to the programs for the state-of-the-art protocol model analyzers such as ProVerif [10] and Tamarin [11].

2. Related work

There exist many formal notations for data structures: ASN.1, JSON, etc. These notations are often provided with the engines, which can automatically generate the binary data using the provided data structure definition and, in the opposite direction, automatically unpack the binary data in accordance with the definition. Such projects as CSN.1 [12], TSN.1 [13], BinPAC [14], NetPDL [15] are targeted specifically at the network protocols. While the readability of some of these notations can be suitable, their expressiveness (in the domain of cryptographic protocols) does not. We need to have behind the notation not simply a message generator/parser waiting to be embedded to some bigger program, but a generic cryptographic protocol implementation waiting for (semi-)declarative specification to adjust to specific case. Therefore, the primary challenge is to find such powerful underlying abstraction, whereas the notation would have to be naturally emerged from it.

3 Cryptographic Stack Machine Notation One

We propose an abstraction named cryptographic stack machine (abbreviated as CSM), which is a stack machine specifically tailored to the needs of cryptographic protocols. Within the proposed approach, the message definition is in fact a sequence of the CSM instructions. The instructions set is divided into "bare-metal" and "sugared" parts. The "sugared" instructions make the message definitions (which in their essence are imperative) looking declarative. The instructions set may be expanded if needed.

To reflect the fact that the declarative style of the protocol message definitions is one of the main targets, we name our notation «Cryptographic Stack Machine Notation One» (abbreviated as CMN.1) adopting the naming style of the ASN.1, CSN.1 and TSN.1 notations.
3.1 CMN.1 syntax

Below, the terms 'String', 'Integer', 'Int', 'Word8' denote the sets of strings, unlimited integers, integers ranged from 0 to 2^{32}-1 and integers ranged from 0 to 2^8-1, respectively. The curled brackets mean repetition, the square ones – optionality. The symbol '"', means comma itself, not concatenation.

\[
\text{Prog} ::= "[\{Instr,\}*\{Instr\}]",
\]

Instr ::= BareMetal | Sugared

BareMetal ::= Const Word8List | Var VarName Role VarType | V VarName | SEnc SEncAlg | Enco EncoAlg | Xor Int | ModAdd | ModMult | ModInv | Add Integer | Rev RFun | Hash HashAlg | Pad Int Word8List | Mod | ModExp | Take IntList | Split IntList | SplitE Int | ECMult | ECA pp | C | CE Len LenHdr

Sugared ::= C Prog | CE Prog | Hash HashAlg Prog | SEnc SEncAlg Prog | Enco EncoAlg Prog | Mod Prog | ModAdd Prog | ModMult Prog | ModExp Prog | ModInv Prog | ECMult Prog | ECAdd Prog | Len LenHdr Prog | Xor Prog | Add Integer Prog | Take IntList Prog | Split IntList Prog | SplitE Int Prog | Pad Int Word8List Prog | Map Prog Int Prog | Sort Int Prog | Select Int CaseList | SA Prog | WithLen LenHdr Prog | VarL Int VarName Role VarType | VL Int VarName | SelectV VarName CaseList

VarName ::= "[\{String,\}*\{String\}]",

VarType ::= Plain Int | Primary Int | Modulo Inst | UTC | ECx Inst | Sublist Prog | Choice Prog | Subset Prog | Is Prog |

Word8List ::= "[\{Word8,\}*\{Word8\}]",

IntList ::= "[\{Int,\}*\{Int\}]",

IntegerList ::= "[\{Integer,\}*\{Integer\}]",

SEncAlg ::= AES128CBC | AES256CBC …

HashAlg ::= SHA1 | SHA256 …

EncoAlg ::= SSLPad Int | B2DERInt | B2DERBits …

LenHdr ::= BE Int | LE Int | DER

CaseTy ::= Case Word8List Prog | Cases "[\{Word8List,\}*\{Word8List \}]", Prog | Case' Condition Prog | Otherwise Prog | CaseUndef Prog

CaseList ::= "[\{CaseTy,\}*\{CaseTy\}]",

Condition ::= Bytes Word8List | Equal Integer | Less Integer | More Integer | LessOrEq Integer | MoreOrEq Integer | OneOf IntegerList | Otherwise'

Role ::= Clnt | Serv | A | B | S | CA | RA | TTP …

3.2 CMN.1 semantics

CSM takes the top 3 elements of the stack as arguments: a, b, c. CSM encrypts a with b as initial vector and c as the key using symmetric encryption algorithm alg. Here and after: 1) if the stack is underflowed, CSM returns an error; 2) the last argument in the argument list is located at the top of the stack; 3) the arguments of the function are removed from the stack; 4) the result is pushed to the stack.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Arguments</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Xor'</strong> n</td>
<td>Exclusive OR. Arguments: the top n elements of the stack.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ModAdd', ModMult'</strong></td>
<td>Addition (multiplication) of a and b modulo m. List of arguments: a, b, m.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ModInv'</strong></td>
<td>Inverse of a under modulo m. List of arguments: a, m.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Add'</strong> n</td>
<td>Let a is the top element of the stack. CSM adds n to a modulo 2^(8*k), where k is the length of a in bytes.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Rev</strong> fun</td>
<td>The function that is reverse to the function fun, where fun must be one of: Enco' alg, SEnc' alg, Xor' n, ModMult', ModAdd', ModInv', Add' n.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Mod'</strong></td>
<td>Modulo operation. List of arguments: a.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ModExp'</strong></td>
<td>Modular exponentiation: a^b mod m. List of arguments: a, b, m.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Hash' alg</strong></td>
<td>CSM calculates the hash of a using algorithm alg.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Pad' n ws</strong></td>
<td>Padding of a using the bytes ws until the length of the result reaches n (n must be equal or greater than length of a). List of arguments: a.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Take' ns</strong></td>
<td>Here ns is the list of numbers. If the top element of the stack is less than the sum of the elements of ns, then CSM returns the specification error. Otherwise, CSM cuts the top element of the stack into n parts considering the numbers from the ns list as lengths of elements and pushes (from left to right) the resulting n elements onto the stack, where n is the length of the ns list. The remainder of the top element is dropped (if any).</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Split' ns</strong></td>
<td>The same as the instruction Take' ns, except that the top element of the stack must be exactly equal to the sum of the numbers from the ns list.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>SplitE' n</strong></td>
<td>Is equivalent to the instruction Split' [k,k...k], where k = len / n, where len is the length of the top element of the stack (len must be dividable by n).</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ECMult'</strong></td>
<td>Elliptic curve scalar multiplication. List of arguments: curve_id (curve identifier), x (x-coordinate), y (y-coordinate), k (the scalar). Instruction produces 2 elements of the stack: x-coordinate and y-coordinate.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ECAdd'</strong></td>
<td>Elliptic curve addition of points (x1,y1) and (x2,y2). List of arguments: curve_id (curve identifier), x1,y1,x2,y2. Instruction produces 2 elements of the stack: x-coordinate and y-coordinate.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>C' n</strong></td>
<td>Concatenation. Arguments: the top n elements of the stack.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>CE' n</strong></td>
<td>Concatenation of the equal-sized arguments.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Lessons**

- **Len' e** The length of the top element of the stack written in e format, where e can be one of: BE n (packing into n big-endian bytes), LE n (packing into n little-endian bytes), DER (packing using ASN.1 DER format).
- **Insert i** CSM moves the top element of the stack to the i-th position.
- **Pick i, Dup i** CSM moves (for Pick) or copies (for Dup) the i-th element of the stack to the top position.
- **Free i** CSM removes the i-th element from the stack.
- **Elem i p** CSM executes the program p using temporary empty stack and then puts in the current working stack the i-th element of temporary stack.
- **SA' n k p** CSM copies n elements from the current working stack to temporary stack, executes the program p using a new temporary stack and then inserts the resulting elements between the (k+1)-th and k-th elements of the current working stack.
- **Map' p i n** The stack must contain at least i*n elements. CSM executes the program p n times using at each iteration a new temporary stack to which the next i elements from the current working stack are moved (beginning from the depths of the stack). At each iteration the elements containing in temporary stack after execution of p are moved to the current working stack.
- **Sort' i n** CSM considers the top i*n elements of the stack as a list of n elements, where each element, in turn, is a list of i elements. CSM sorts this list of n elements comparing their first (from the depths of the stack) elements.
- **Select cs** CSM converts the list of the cases cs into the form:  
  \[ \text{Case}_1 c_1, \ldots, \text{Case}_n c_n. \]  
  If CSM finds in the list cs (from left to right) the condition c to which the top element of the stack satisfies, then it removes the top element from the stack and executes the program p. Otherwise, it returns an error.
- **M p** Macro instruction: CSM simply executes the program p.
- **L n p** Macro instruction supplemented by the total length of the resulting elements of p execution (parameter n).

**"Sugared" instructions**

- **C p, CE p, Xor p, SEnc al p, Mod p, ModMult p, ModAdd p, ModExp p, ModInv p, ECMult p, ECAdd p** CSM executes the program p using temporary empty stack and copies the resulting m elements onto the current working stack. Then it executes the "bare-metal" counterpart of the "sugared" instruction: C' m, CE' m, Xor' m, SEnc' al, Mod', ModMult', ModAdd', ModExp', ModInv', ECMult' or ECAdd'. In the end, CSM moves the resulting elements (two elements in the case of the ECMult' or ECAdd' instruction and one element in the other cases) to the current working stack.
4. Simple CMN.1-based specification language

The language presented below is simple in the sense that it doesn't capture the protocol automata in full. A specification consists of the CMN.1-based message definitions and a sequence of protocol actions with simple branching support (table 2).

<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>roles r vlist</td>
<td>Here vlist is the list of pairs of type ((V \text{ name}, is)). For each pair, the action executes the CMN.1 instruction is and includes the pair ((\text{name}, \text{val})) in a storage s var belonging to the CMN.1 instance of the role r, where (\text{val is concatenation of the resulting elements of the execution of is}).</td>
</tr>
<tr>
<td>select r pacs</td>
<td>This action provides a branching support in the same manner as the CMN.1 instruction Select is cs does. The difference between the lists cs and pacs is that (\text{acs consists of elements } \text{Case value p, whereas } \text{acs consist of elements } \text{Case value a, where a is a sequence of protocol actions.})</td>
</tr>
<tr>
<td>connect r pport addr</td>
<td>If this action is present, the specification turns into the client implementation acting as the protocol role role. The action carries out the connection to a third-party server implementation listening on the port port of the IP-address addr.</td>
</tr>
<tr>
<td>accept role port</td>
<td>The specification turns into the server implementation acting as role and listening on the port port.</td>
</tr>
<tr>
<td>printPV printPV'</td>
<td>Both actions generate the ProVerif program corresponding to the protocol events that took place at the time of the call. The first action generates a full program, the second one ignores the lengths fields of messages and related events as non-essential in order to make this program more concise and productive.</td>
</tr>
</tbody>
</table>

Bearing in mind the elegant and concise syntax of the Haskell language and advantages of embedded domain-specific languages, we integrate our CMN.1-based specification language in Haskell.

As an illustration, we present an excerpt from the CMN.1-based specification of the TLS protocol (fig. 2; note that the order of declarations can be arbitrary in the Haskell language). A specification, which serves as source for this excerpt, comprises about 500 lines (the total for client and server) covering substantial part of the TLS v.1.2 protocol including four cipher suites and X.509 certificates support and excluding extensions and renegotiations. The specification turned into the implementation (see the actions connect and accept in the table 2) was successfully tested for interoperability with the OpenSSL v.1.0.2o tool (both in the client and server roles).
5. Translation to the ProVerif program

The ProVerif program presented in the fig. 3 was generated automatically from the above specification (it is a console output of the call printPV; see the line 115 in the fig. 2). This program corresponds to the protocol trace based on the ciphersuite TLS-DHE-DSS-WITH-AES-256-CBC-SHA. The program passed the ProVerif compiler checks without warnings. The events and queries of interest have to be inserted manually because CMN.1-based specifications do not contain such information.

```c
14 reduce forall a0:bitstring,a1:bitstring,a2:bitstring;
15 16 reduce forall a0:bitstring,a1:bitstring,a2:bitstring;
17 18 fun ModAdd([ModAdd(a0,a1,a2),a2,a2],a0).a0;
19 20 fun Hash(SHA1(bitstring,bitstring));
21 22 fun Mod(bitstring,bitstring,bitstring);
23 24 fun EncodeD2E7Int(bitstring,bitstring);
25 26 fun forall a0:bitstring,bitstring,bitstring,bitstring;
27 28 fun EncodeD2E7Int(bitstring,bitstring);
29 30 fun HashInt(HashInt(HashInt(a0,a1)),a1) = a0.
31 32 fun Hash(SHA1(bitstring,bitstring));
33 34 const nul: bitstring = [data].
35 36 const x0038: bitstring = [data].
```

The ProVerif program presented in the fig. 3 was generated automatically from the above specification (it is a console output of the call printPV; see the line 115 in the fig. 2). This program corresponds to the protocol trace based on the ciphersuite TLS-DHE-DSS-WITH-AES-256-CBC-SHA. The program passed the ProVerif compiler checks without warnings. The events and queries of interest have to be inserted manually because CMN.1-based specifications do not contain such information.
6. Engine implementation details

The engine implements the functionality that is significantly more powerful than the CSM machine presented in the section 3. The engine does not execute the CMN.1-notated programs as straightforward as CSM does. It executes the programs symbolically: the elements of the stack are not byte strings but symbolic expressions. This well-known technique allows the engine to fully take over the task of verification of the incoming messages using the same CMN.1-definitions that are used in the direct task of message generation. The verification is complete: the engine decrypts the ciphertexts, checks MACs and signatures, etc. Throughout a protocol execution, the engine accumulates the generated symbolic expressions, their values, lengths and types. It uses this information to generate or verify the protocol messages in the future.

In addition, the engine logs such events as calculations of the values of the symbolic expressions and applications of the rewriting rules. This information can be used by the engine’s environment to extract symbolic traces and convert them to the programs for symbolic verifiers, e.g. ProVerif (as was presented in the previous section).

The scheme of the verification is as follows. Let the byte string \( bs \) is considered by the engine as a protocol message with the CMN.1 definition \( p \). Let \( EQ \) is a set variable containing equations, i.e. pairs of type (symbolic expression, byte string). The engine implements the verification procedure as follows.

**Step 1.** The engine executes the program \( p \) symbolically resulting the symbolic expression \( \text{exp} \). \( EQ \) is initialized with the equation \((\text{exp},bs)\).

**Step 2.** For every new equation \((\text{exp},bs)\) from \( EQ \), until neither of Step 2.1 or Step 2.2 can be applied anymore:

**Step 2.1.** The engine tries to apply a rewriting rule to this equation. This rule can be a simple inversion (for \( \text{Enco} \), \( \text{SEnc} \), \( \text{Xor} \), \( \text{ModMult} \), \( \text{ModAdd} \), \( \text{ModInv} \) or \( \text{Add} \) functions) or be a complex group operation taking into account other equations from \( EQ \) (e.g. for \( \text{Split} \)). The application of the rule produces one or several new equations, which are inserted in \( EQ \). If some rule was applied, the engine returns to the beginning of the Step 2. Otherwise, it goes to the Step 2.2.

**Step 2.2.** If the values of all the arguments of the top operation of the symbolic expression \( \text{exp} \) are known, the engine calculates the value of \( \text{exp} \). If this value is equal to \( bs \), the engine removes the equation from \( EQ \). Otherwise, it returns the message verification error.

The engine knows about the equality \((a b)c = (a c)b\) and analogous equality for the elliptic curve scalar multiplication, so Diffie-Hellman key exchange and ElGamal asymmetric encryption do not ask for special treatment. Yet the engine uses specific rewrites for expressions relevant to the DSA and ECDSA algorithms or to their relatives.

The calls exported by the engine are presented below.

1. \( \text{cSymExec} p \) – The engine executes the program \( p \) symbolically and returns the descriptor of the generated symbolic expression.
2. \( \text{cCalc} d \) – The engine calculates the value of the symbolic expression with descriptor \( d \).
3. \( \text{cGetVal} d \) – The engine returns the value of the symbolic expression with descriptor \( d \).
4. \( \text{cSetVal} d bs \) – The engine assigns the value \( bs \) to the symbolic expression with descriptor \( d \). If verification has failed, it returns an...
error, otherwise, it returns the superfluous remainder of the byte string \( bs \) (if present).

6. \( \text{cEvent ev} \) – The engine logs the event \( ev \) (i.e. the environment can insert additional events into the engine log).

7. \( \text{cGetLog} \) – The engine returns content of its log.

7. Conclusion

We presented cryptographic protocol message notation (named CMN.1) based on the instruction set of a stack machine specifically tailored to the needs of cryptographic protocols (named cryptographic stack machine, or CSM). The principles of implementation of the protocol specification language based on this notation also presented. Within such an approach, specifications are executable and also translatable to the programs for symbolic verifiers, such as ProVerif. The readability of CMN.1-notated specifications is brought in the court of public opinion.

In addition, the validation of the proposed notation on a wider spectrum of cryptographic protocols is needed. The validation will certainly cause minor additions to the notation (at least regarding cryptographic key types) without affecting currently defined CSM instructions. Taking into account the fact that the author's proof-of-concept implementation of the core language library (the engine) comprises only 700 lines of the Haskell code (excluding cryptographic primitives), it seems logical to provide in the future a formal description of the engine's algorithm and, basing on it, a proof of the soundness of the ProVerif-translation procedure.

References


[12]. Concrete Syntax Notation One (CSN.1). Available at: http://csn1.info, accepted 10.06.2018.


Нотация криптографической стековой машины версии один

С. Е. Прокопьев <s.e.pr@mail.ru>
г. Москва

Аннотация. Хорошая спецификация криптографического протокола должна легко восприниматься человеком (быть декларативной и лаконичной), быть исполнимой и пройти процедуру формальной верификации в некоторой адекватной модели. Начинаясь на эти требования, в статье предложена нотация CMN.1, предназначенная для описания спецификаций криптографических протоколов и основываясь на вычислительной абстракции под названием криптографическая стековая машина (CSM). Статья описывает синтаксис и семантику CMN.1, а также представляет результаты разработки языка спецификаций криптографических протоколов, построенного на основе данной нотации и встроенного в язык Haskell. В авторском реализации вся обработка сообщений инкапсулирована внутри базового библиотечного модуля, в то время как спецификация должна лишь дать декларативные определения этих сообщений. При формировании исходящего сообщения протокол базового модуль берет описание данного сообщения в нотации CMN.1 и возвращает фрагмент данных, сгенерированный по этому описанию. При обработке входящего сообщения базовый модуль берет поступивший фрагмент данных и описание ожидаемого сообщения в
Ключевые слова: язык спецификаций криптографических протоколов; нотация криптографической стековой машины; встроенные предметно-ориентированные языки; Haskell; ProVerif; TLS

DOI: 10.15514/ISPRAS-2018-30(3)-12

Список литературы


