Cryptographic Stack Machine Notation One

S.E. Prokopev <s.e.pr@mail.ru>
Independent researcher
Moscow, Russia

Abstract. A worthy cryptographic protocol specification has to be human-readable (declarative and concise), executable and formally verified in a sound model. Keeping in mind these requirements, we present a protocol message definition notation named CMN.1, which is based on an abstraction named *cryptographic stack machine.* The paper presents the syntax and semantics of CMN.1 and the principles of implementation of the CMN.1-based executable protocol specification language. The core language library (the engine) performs all the message processing, whereas a specification should only provide the declarative definitions of the messages. If an outcoming message must be formed, the engine takes the CMN.1 definition as input and produces the binary data in consistency with it. When an incoming message is received, the engine verifies the binary data with respect to the given CMN.1 definition memorizing all the information needed in the further actions. The verification is complete: the engine decrypts the ciphertexts, checks the message authentication codes and signatures, etc. Currently, the author's proof-of-concept implementation of the language (embedded in Haskell) can translate a CMN.1-based specifications both to the interoperable implementations and to the programs for the ProVerif protocol analyzer. The excerpts from the CMN.1-based TLS protocol specification and corresponding automatically generated ProVerif program are provided as an illustration.

Keywords: cryptographic stack machine; cryptographic protocol message notation; executable cryptographic protocol specification languages; embedded domain-specific languages; Haskell; ProVerif; TLS.

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

For citation: Prokopev S.E. Cryptographic Stack Machine Notation One. Trudy ISP RAN/Proc. ISP RAS, vol. 30, issue 3, 2018, pp. 165-182. DOI: 10.15514/ISPRAS-2018-30(3)-12

1. Introduction

The establishment of good soundness relations between cryptographic protocol implementations and their formal models is a popular research area. The existing approaches differ by the starting point of development (implementation first [1-6] or formal model first [7-9]), by the degree of cryptographic soundness of the models (symbolic [10] or computational [9]), by the presence of the formal proof of the soundness of the model-to-implementation (or vice verse) translation procedure, by implementation usability area and by other aspects.

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.

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' | ECAdd' | C' | CE' | Len' LenHdr | InsertTo Int | PickFrom Int | Dup Int | Free Int | Elem Int Prog | Map' Prog Int Int | Sort' Int Int | SA' Int Int Prog | Select' CaseList | M Prog | L Int Inst
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 Inst 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 has one main stack and varying number of temporary stacks, random-number generator, real-time clock, the storage s_var containing the values of the protocol variables (actually they don't vary in CSM) and the register s_rol containing the identifier of the protocol role (fig. 1).

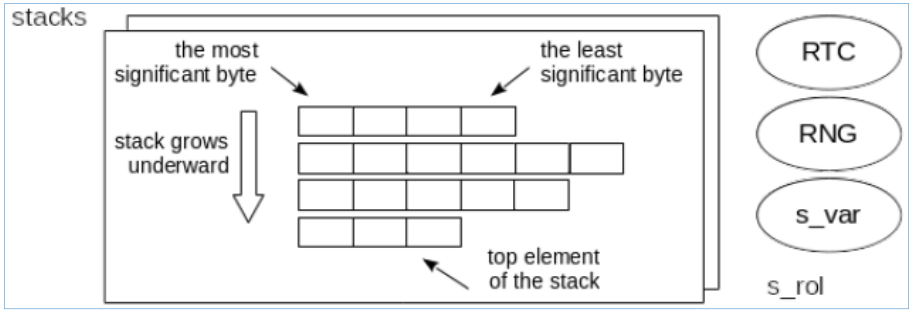

Fig. 1. Cryptographic stack machine

The language of the CSM instructions extends the line of the stack-oriented languages. It supports branching but doesn't support looping or recursing (table 1).

Table 1. CSM instructions semantics

Instruction	CSM actions
"Bare-metal" instructions	
Const *bs*	CSM pushes the byte string *bs* onto the stack.
Var *s r t*	If the storage s_var contains the variable named *s*, then CSM pushes this variable value onto the stack. Otherwise, if *r* /= s_rol, CSM returns an error. Otherwise, it generates a new element of type *t*, stores its value under the name *s* in the s_var storage and puts this value in the stack. The currently defined variable types: Plain *n* – random *n* bytes; Primary *n* – random primary integer of *n*-bit length; Modulo *is* – random integer modulo *n*, where *n* is the big-endian value of the result of the instruction *is* execution; ECx *is* – random point on the curve *curve_id*, where *curve_id* is the value of the result of the instruction *is* execution; UTC – the time and date in standard UNIX 32-bit format; Sublist *p* (Choice *p*, Subset *p*) – random sublist (element, subset) of the list comprised of resulting elements of the program *p* execution; Is *p* – equivalent to Choice [C *p*].
V *s*	If the storage s_var contains the variable with name *s*, then CSM pushes the value of this variable onto the stack. Otherwise, it returns an error.
SEnc' *alg*	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.
Enco' *alg*	Encoding of *a* using algorithm *alg*. List of arguments: *a*.

Xor' *n*	Exclusive OR. Arguments: the top *n* elements of the stack.
ModAdd', ModMult'	Addition (multiplication) of *a* and *b* modulo *m*. List of arguments: *a, b, m*. Here and after: the byte strings are interpreted as integers basing on the `big endian` agreement.
ModInv'	Inverse of *a* under modulo *m*. List of arguments: *a*, *m*.
Add' *n*	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.
Rev *fun*	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*.
Mod'	Modulo operation. List of arguments: *a*.
ModExp'	Modular exponentiation: *a^b* mod *m*. List of arguments: *a,b,m*.
Hash' *alg*	CSM calculates the hash of *a* using algorithm *alg*. List of arguments: *a*.
Pad' *n ws*	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*.
Take' *ns*	Here *ns* is the list of numbers. If the length of 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).
Split' *ns*	The same as the instruction Take' *ns*, except that the length of the top element of the stack must be exactly equal to the sum of the numbers from the *ns* list.
SplitE' *n*	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*).
ECMult'	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.
ECAdd'	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.
C' *n*	Concatenation. Arguments: the top *n* elements of the stack.
CE' *n*	Concatenation of the equal-sized arguments.

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: [Case' c_1 p_1 , … , Case' c_n p_n]. If CSM finds in the list *cs* (from left to right) the condition c_i to which the top element of the stack satisfies, then it removes the top element from the stack and executes the program p_i. 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,*	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.

ECAdd *p*	
Map *q n p* Sort *n p*	CSM executes the program *p* using temporary empty stack. If *m* mod *n* /= *0*, CSM returns an error (where *m* is the number of elements of temporary stack after execution of *p*). Otherwise, it copies the resulting *m* elements onto the current working stack executes the "bare-metal" counterpart: Map' *q i n* or Sort' *i n*, where *i = m / n*.
Hash *al p*, Enco *al p*, Add *n p*, Pad *n bs p*, Len *e p*, Take *lst p*, Split *lst p*, SplitE *n p*	CSM executes the program [C *p*] using temporary empty stack and copies the resulting element onto the current working stack. After that, CSM executes the "bare-metal" counterpart of the "sugared" instruction: Hash' *al*, Enco' *al*, Add' *n*, Pad' *n bs*, Len' *e,* Take' *ls*, Split' *ls* or SplitE' *n.*
Select *is cs*	CSM tries to execute the program [C [*is*]] using temporary empty stack. If the program was successfully executed, CSM copies the resulting element onto the current working stack and executes the instruction Select' *cs*. If the execution failed (due to unknown variable), CSM checks if the list *cs* does contain the element CaseUnkno *p*. If so, CSM executes the program *p*, otherwise it returns an error.
VarL *n s r t*	Is equivalent to: L n (Var *s r t*)
VL *n s*	Is equivalent to: L n (V s)
SA *p*	Is equivalent to: SA' *1 0 p*
WithLen *e p*	Is equivalent to: M [C *p*, SA' *1 1* [Len' *e*]]
SelectV *s cs*	Is equivalent to: Select (V s) cs

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 2. Protocol actions

Action	Description
roles *rlist*	The action sets the roles participating in the protocol. Each role runs its own CSM instance.
msg *src dst p*	The message with the CMN.1 definition *p* is transferred from the role *src* to the role *dst*.

set *r vvlist*	Here *vvlist* is the list of pairs of type (V *name*, *is*). For each pair, the action executes the CSM instruction *is* and includes the pair (*name, val*) in a storage s_var belonging to the CSM instance of the role *r*, where *val* is concatenation of the resulting elements of the execution of *is*.
select *r is acs*	This action provides a branching support in the same manner as the CSM instruction Select *is cs* does. The difference between the lists *cs* and *acs* is that *cs* consists of elements Case *value p*, where *p* is a CSM program, whereas *acs* consist of elements Case *value a*, where *a* is a sequence of protocol actions.
trusted *r id p*	This action takes from a trusted storage the binary data stored under the name *id* and processes these data using CMN.1 definition *p* and the CSM instance of the role *r*.
connect *r port addr*	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*.
accept *role port*	The specification turns into the server implementation acting as *role* and listening on the port *port*.
printPV printPV'	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.

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 ciphersuites 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).

```
tlsMsg m src =
   [VL 1 ["contentType",m],
    SelectV ["version","clntHello"]
       [CaseUnkno [VarL 2 ["version",m] src
                      (Choice [Const [0x03,0x03], Const [0x03,0x02],
                               Const [0x03,0x01]])],
        Otherwise [V ["version","clntHello"]]],
    WithLen (BE 2)
       [SelectV ["CCS",show src]
           [CaseUnkno [payload],
            Otherwise [payloadProtected]]]]
    where
    payload =
     SelectV ["contentType",m]
       [Case [0x14] [VarL 1 ["CCS",show src] src (Is [Const [0x01]])],
        Case [0x15] [VL 1 ["alertLevel",m],
                     VL 1 ["alertDescr",m]],
        Case [0x16] [Var ["hshkMsg",m] src (Is hshkMsg)],
        Case [0x17] [V ["dataContent",m]]]
        where
        hshkMsg =
         [VL 1 ["hshkType",m],
          WithLen (BE 3)
           [SelectV ["hshkType",m]
             [Case [0x01] clntHello,
              Case [0x02] servHello,
              Case [0x0b] servCert,
              Case [0x0c] servKeyExch,
              ...]]]
          where
          clntHello =
             [VarL 2 ["version","clntHello"] Clnt
                 (Choice [Const [0x03,0x03], Const [0x03,0x02],
                          Const [0x03,0x01]]),
              random Clnt,
              Const [0],
              WithLen (BE 2) [Var ["suites","clntHello"] Clnt
                 (Subset [Const [0x00,0x38], Const [0x00,0x32],
                          Const [0xc0,0x0a], Const [0xc0,0x09]])],
              WithLen (BE 1) [Const [0]],
              Var ["helloExt","clntHello"] Clnt (Choice [Const []])]
          servHello =
             [VarL 2 ["version","servHello"] Serv
                 (Is [V ["version","clntHello"]]),
              random Serv,
              WithLen (BE 1) [Var ["sessId","servHello"] Serv (Plain 32)],
              Var ["suite","servHello"] Serv
                 (Choice [SplitE 2 [V ["suites","clntHello"]]]),
              VarL 1 ["compressAlg","servHello"] Serv
                 (Choice [Const [0x00]])]
          servCert = ...
          servKeyExch =
             [keyExchParams,
              VarL 1 ["sigHashAlg","servKeyExch"] Serv
                (Is [SelectV ["suite","servHello"]
                       [Cases [[0x00,0x32],[0x00,0x38],
                               [0xc0,0x09],[0xc0,0x0a]] [Const [0x02]]]]),
              VarL 1 ["sigAlg","servKeyExch"] Serv
                (Is [SelectV ["suite","servHello"]
                       [Cases [[0x00,0x32],[0x00,0x38]] [Const [0x02]],
                        Cases [[0xc0,0x09],[0xc0,0x0a]] [Const [0x03]]]]),
              WithLen (BE 2)
                 [mDER 0x30 [mDER 0x02 [sigPart 1],
                             mDER 0x02 [sigPart 2]]]]
              where
              keyExchParams =
                SelectV ["suite","servHello"]
                     [Cases [[0x00,0x32],[0x00,0x38]] dh,
                      Cases [[0xc0,0x09],[0xc0,0x0a]] ecdh]
                where
                dh = [WithLen (BE 2) [dhP],
                      WithLen (BE 2) [dhG],
                      WithLen (BE 2) [dhPubk Serv "servKeyExch"]]
                ecdh = ...
                sigPart i =
                  SelectV ["sigAlg","servKeyExch"]
                     [Case [0x02] [Elem i sig_dsa],
                      Case [0x03] [Elem i sig_ecdsa]]
                  where
                  sig_dsa = mSigDSA [hash,p,q,g,x,k] where
                     [p,q,g,x] = [V [x,"servCert"] | x <- ["dsaP","dsaQ",
                                                           "dsaG","dsaX"]]
                      k = Var ["dsaK","servCert"] Serv (Modulo p)
                  sig_ecdsa = ...
                  hash = ...
          ...
random src =
 C [Var ["time",show src] src UTC,
    Var ["salt",show src] src (Plain 28)]

dhP = Var ["dhP","servKeyExch"] Serv (Primary 256)
dhG = Var ["dhG","servKeyExch"] Serv (Modulo dhP)
dhX src a = Var ["dhX",a] src (Modulo dhP)
dhPubk src a = ModExp [dhG, dhX src a, dhP]
...
```

```
96     payloadProtected = ...
97
98  mDER t p = C [Const [t], WithLen DER (f t)] where
99     f 0x02 = [Enco B2DERInt p]
100    f 0x03 = [Enco B2DERBits p]
101    f _ = p
102 ...
103 mSigDSA [e,p,q,g,x,k] = [r,s] where
104    r = Mod [ModExp [g, k, p], q]
105    s = ModMult [ModAdd [ModMult [r, x, q], e, q],
106                 ModInv [k, q], q]
107 main =
108    roles [Clnt,Serv] >>=
109    -- connect Clnt 4433 0 >>= -- accept Serv 4433 >>= --
110    sendHandsh Clnt Serv [0x01] 1 >>=
111    sendHandsh Serv Clnt [0x02] 2 >>=
112    ...
113    sendHandsh Serv Clnt [0x0c] 4 >>=
114    ...
115    printPV'
116
117 sendHandsh src dst htype i ss =
118    set src [(V ["contentType",show i], Const [0x16]),
119             (V ["hshkType",show i], Const htype)] ss >>=
120    msg src dst [tlsMsg (show i) src]
```

Fig. 2. CMN.1-based specification of the TLS protocol (an excerpt)

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.

```
1 free c: channel.
2 ...
3 fun ModExp(bitstring,bitstring,bitstring): bitstring.
4 const dhG_servKeyExch: bitstring [data].
5 const dhP_servKeyExch: bitstring [data].
6   equation forall x:bitstring,y:bitstring;
7     ModExp(ModExp(dhG_servKeyExch,x,dhP_servKeyExch),y,dhP_servKeyExch) =
8         ModExp(ModExp(dhG_servKeyExch,y,dhP_servKeyExch),x,dhP_servKeyExch).
9 fun ModAdd(bitstring,bitstring,bitstring):bitstring.
10  equation forall a0:bitstring,a1:bitstring;
11    ModAdd(a0,a1,dhP_servKeyExch) = ModAdd(a1,a0,dhP_servKeyExch).
12  equation forall a0:bitstring,a1:bitstring;
13    ModAdd(a0,a1,dsaP_servCert) = ModAdd(a1,a0,dsaP_servCert).
```



```
14  reduc forall a0:bitstring,a1:bitstring,a2:bitstring;
15    Rev0ModAdd(ModAdd(a0,a1,a2),a1,a2) = a0.
16  reduc forall a0:bitstring,a1:bitstring,a2:bitstring;
17    Rev1ModAdd(a0,ModAdd(a0,a1,a2),a2) = a1.
18 fun ModInv(bitstring,bitstring):bitstring.
19  reduc forall a0:bitstring,a1:bitstring; Rev0ModInv(ModInv(a0,a1),a1) = a0.
20 fun HashSHA1(bitstring):bitstring.
21 fun Mod(bitstring,bitstring):bitstring.
22 fun EncoB2DERInt(bitstring):bitstring.
23  reduc forall a0:bitstring; Rev0EncoB2DERInt(EncoB2DERInt(a0)) = a0.
24 const xnull: bitstring [data].
25 const x0038: bitstring [data].
26 ...
27 let processClnt =
28  new time_Clnt: bitstring;
29  new salt_Clnt: bitstring;
30  let v17 = (time_Clnt,salt_Clnt) in
31  new suites_clntHello: bitstring;
32  let v25 = (x0303,v17,x00,suites_clntHello,x00,xnull) in
33  let hshkMsg_1 = (x01,v25) in
34  let v11 = (x16,x0303,hshkMsg_1) in
35  out(c,v11);
36  in(c,v37:bitstring);
37  let (=x16,=x0303,hshkMsg_2:bitstring) = v37 in
38  let (=x02,v48:bitstring) = hshkMsg_2 in
39  let (=x0303,v42:bitstring,sessId_servHello:bitstring,
40    =x0038,compressAlg_servHello:bitstring) = v48 in
41  let (time_Serv:bitstring,salt_Serv:bitstring) = v42 in
42  ...
43  in(c,v180:bitstring);
44  let (=x16,=x0303,hshkMsg_4:bitstring) = v180 in
45  let (=x0c,v217:bitstring) = hshkMsg_4 in
46  let (v193:bitstring,=x02,=x02,v214:bitstring) = v217 in
47  let (=dhP_servKeyExch,=dhG_servKeyExch,v190:bitstring) = v193 in
48  let (=x30,v211:bitstring) = v214 in
49  let (v206:bitstring,v210:bitstring) = v211 in
50  let (=x02,v203:bitstring) = v206 in
51  let v196 = Rev0EncoB2DERInt(v203) in
52  let (=x02,v207:bitstring) = v210 in
53  let v202 = Rev0EncoB2DERInt(v207) in
54  let v198 = (v17,v42,v193) in
55  let v199 = HashSHA1(v198) in
56  let v223 = ModInv(v202,dsaQ_servCert) in
57  let v224 = ModMult(v199,v223,dsaQ_servCert) in
58  let v226 = ModExp(dsaG_servCert,v224,dsaP_servCert) in
59  let v225 = ModMult(v196,v223,dsaQ_servCert) in
60  let v227 = ModExp(v132,v225,dsaP_servCert) in
61  let v230 = ModMult(v226,v227,dsaP_servCert) in
62  if v196 = Mod(v230,dsaQ_servCert) then
63  in(c,v237:bitstring);
64  ...
65 let processServ =
66  in(c,v10:bitstring);
```

```
69  let (=x0303,v14:bitstring,=x00,suites_clntHello:bitstring,
70    =x00,helloExt_clntHello:bitstring) = v22 in
71  let (time_Clnt:bitstring,salt_Clnt:bitstring) = v14 in
72  new time_Serv: bitstring;
73  new salt_Serv: bitstring;
74  let v38 = (time_Serv,salt_Serv) in
75  new sessId_servHello: bitstring;
76  if x0038 = Split2_2_2_2_1(suites_clntHello) then
77  let v44 = (x0303,v38,sessId_servHello,x0038,x00) in
78  let hshkMsg_2 = (x02,v44) in
79  let v34 = (x16,x0303,hshkMsg_2) in
80  out(c,v34);
81  ...
82  new dhX_servKeyExch: bitstring;
83  let v203 = ModExp(dhG_servKeyExch,dhX_servKeyExch,dhP_servKeyExch) in
84  let v206 = (dhP_servKeyExch,dhG_servKeyExch,v203) in
85  new dsaK_servCert: bitstring;
86  let v210 = ModExp(dsaG_servCert,dsaK_servCert,dsaP_servCert) in
87  let v211 = Mod(v210,dsaQ_servCert) in
88  let v218 = EncoB2DERInt(v211) in
89  let v221 = (x02,v218) in
90  let v212 = ModMult(v211,dsaX_servCert,dsaQ_servCert) in
91  let v213 = (v14,v38,v206) in
92  let v214 = HashSHA1(v213) in
93  let v215 = ModAdd(v212,v214,dsaQ_servCert) in
94  let v216 = ModInv(dsaK_servCert,dsaQ_servCert) in
95  let v217 = ModMult(v215,v216,dsaQ_servCert) in
96  let v222 = EncoB2DERInt(v217) in
97  let v225 = (x02,v222) in
98  let v226 = (v221,v225) in
99  let v229 = (x30,v226) in
100 let v232 = (v206,x02,x02,v229) in
101 let hshkMsg_4 = (x0c,v232) in
102 let v194 = (x16,x0303,hshkMsg_4) in
103 out(c,v194);
104 ...
105 process
106 ((!processClnt) | (!processServ))
```

Fig. 3. The corresponding ProVerif program (an excerpt)

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 *exp*. *EQ* is initialized with the equation (*exp,bs*).
>
> **Step 2**. For every new equation (*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 Enco, SEnc, Xor, ModMult, ModAdd, ModInv or Add functions) or be a complex group operation taking into account other equations from *EQ* (e.g. for 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 *exp* are known, the engine calculates the value of *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. cSymExec *p* – The engine executes the program *p* symbolically and returns the descriptor of the generated symbolic expression.
2. cCalc *d* – The engine calculates the value of the symbolic expression with descriptor *d*.
3. cGetVal *d* – The engine returns the value of the symbolic expression with descriptor *d*.
4. cSetVal *d bs* – The engine assigns the value bs to the symbolic expression with descriptor d.
5. cVerify *d bs* – The engine verifies the byte string *bs* with respect 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. cEvent *ev* – The engine logs the event *ev* (i.e. the environment can insert additional events into the engine log).
7. 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

[1]. S. Chaki and A. Datta. Aspier: An automated framework for verifying security protocol implementations. In Proceedings of the Computer Security Foundations Workshop, 2009, pp. 172–185.

[2]. J. Goubault-Larrecq and F. Parrennes. Cryptographic protocol analysis on real C code. In Proceedings of the 6th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'05), Lecture Notes in Computer Science, vol. 3385, 2005, pp. 363–379.

[3]. Mihhail Aizatulin, Andrew D. Gordon, and Jan Jurjens. Extracting and verifying cryptographic models from C protocol code by symbolic execution. In Proc. of the 18th ACM Conference on Computer and Communications Security (CCS'11), 2011, pp. 331–340.

[4]. Nicholas O'Shea. Using Elyjah to analyse Java implementations of cryptographic protocols. In Proc. of the Joint Workshop on Foundations of Computer Security, Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security (FCS-ARSPA-WITS'08), 2008.

[5]. Karthikeyan Bhargavan, Cedric Fournet, Andrew Gordon, and Stephen Tse. Verified interoperable implementations of security protocols. ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 31, no. 1, 2008.

[6]. Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. Verified Models and Reference Implementations for the TLS 1.3 Standard Candidate. In Proc. of the IEEE Symposium on Security and Privacy, 2017.

[7]. Matteo Avalle, Alfredo Pironti, Riccardo Sisto, and Davide Pozza. The JavaSPI framework for security protocol implementation. In Proc. of the 6th International Conference on Availability, Reliability and Security (ARES'11), 2011, pp. 746–751.

[8]. David Cade, Bruno Blanchet. From Computationally-Proved Protocol Specifications to Implementations and Application to SSH. Available at: http://prosecco.gforge.inria.fr/personal/dcade/CadeBlanchetJoWUA13.pdf, accessed 10.06.2018.

[9]. A. Delignat-Lavaud et al. Implementing and Proving the TLS 1.3 Record Layer. In Proc. of the 2017 IEEE Symposium on Security and Privacy (SP), 2017, pp. 463-482.

[10]. Bruno Blanchet. Automatic Verification of Security Protocols in the Symbolic Model: the Verifier ProVerif. In Foundations of Security Analysis and Design VII, FOSAD Tutorial Lectures, Lecture Notes in Computer Science, vol. 8604, 2014, pp. 54-87.

[11]. Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. 2017. Source files and annotated RFC for TLS 1.3 analysis. (2017). Available at: https://tls13tamarin.github.io/TLS13Tamarin/, accessed 10.06.2018.

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

[13]. Transfer Syntax Notation One (TSN.1). Available at: http://www.protomatics.com/tsn1.Html, accessed 10.06.2018..

[14]. The BinPAC language. Available at: https://www.bro.org/sphinx/components/binpac/README.html, accessed 10.06.2018..

[15]. Mario Baldi, Fulvio Risso. NetPDL: An Extensible XML-Based Language for Packet Header Description. Computer Networks, vol, 50, issue 5, 2006, pp. 688-706.

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. Нацеливаясь на эти требования, в статье предложена нотация CMN.1, предназначенная для описания сообщений криптографических протоколов и основанная 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 авторской 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 протокола базовый 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

нотации CMN.1 и возвращает вердикт об их соответствии друг другу, извлекая и запоминая при этом все содержащиеся в сообщении данные, необходимые для формирования или верификации следующих сообщений протокола. Процесс верификации является полным: базовый модуль осуществляет расшифрование, проверку кодов аутентификации сообщений и значений цифровой подписи и т.д. Текущая реализация языка включает функции трансляции спецификаций в исполняемый код, совместимый с существующими программными реализациями протоколов, а также функции конвертации спецификаций в программы на входном языке анализатора протоколов ProVerif. В качестве иллюстрации приводятся выдержки из CMN.1-спецификации протокола TLS и соответствующей ей автоматически сгенерированной программы для ProVerif.

Ключевые слова: язык спецификаций криптографических протоколов; нотация сообщений криптографических протоколов; криптографическая стековая машина; встроенные предметно-ориентированные языки; Haskell; ProVerif; TLS

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

Для цитирования: Прокопьев С.Е. Нотация криптографической стековой машины версии один. Труды ИСП РАН, том 30, вып. 3, 2018 г., стр. 165-182 (на английском языке). DOI: 10.15514/ISPRAS-2018-30(3)-12

Список литературы

[1]. S. Chaki and A. Datta. Aspier: An automated framework for verifying security protocol implementations. In Proceedings of the Computer Security Foundations Workshop, 2009, pp. 172–185.

[2]. J. Goubault-Larrecq and F. Parrennes. Cryptographic protocol analysis on real C code. In Proceedings of the 6th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'05), Lecture Notes in Computer Science, vol. 3385, 2005, pp. 363–379.

[3]. Mihhail Aizatulin, Andrew D. Gordon, and Jan Jurjens. Extracting and verifying cryptographic models from C protocol code by symbolic execution. In Proc. of the 18th ACM Conference on Computer and Communications Security (CCS'11), 2011, pp. 331–340.

[4]. Nicholas O'Shea. Using Elyjah to analyse Java implementations of cryptographic protocols. In Proc. of the Joint Workshop on Foundations of Computer Security, Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security (FCS-ARSPA-WITS'08), 2008.

[5]. Karthikeyan Bhargavan, Cedric Fournet, Andrew Gordon, and Stephen Tse. Verified interoperable implementations of security protocols. ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 31, no. 1, 2008.

[6]. Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. Verified Models and Reference Implementations for the TLS 1.3 Standard Candidate. In Proc. of the IEEE Symposium on Security and Privacy, 2017.

[7]. Matteo Avalle, Alfredo Pironti, Riccardo Sisto, and Davide Pozza. The JavaSPI framework for security protocol implementation. In Proc. of the 6th International Conference on Availability, Reliability and Security (ARES'11), 2011, pp. 746–751.

[8]. David Cade, Bruno Blanchet. From Computationally-Proved Protocol Specifications to Implementations and Application to SSH. Режим доступа: http://prosecco.gforge.inria.fr/personal/dcade/CadeBlanchetJoWUA13.pdf, дата обращения 10.06.2018.

[9]. A. Delignat-Lavaud et al. Implementing and Proving the TLS 1.3 Record Layer. In Proc. of the 2017 IEEE Symposium on Security and Privacy (SP), 2017, pp. 463-482.

[10]. Bruno Blanchet. Automatic Verification of Security Protocols in the Symbolic Model: the Verifier ProVerif. In Foundations of Security Analysis and Design VII, FOSAD Tutorial Lectures, Lecture Notes in Computer Science, vol. 8604, 2014, pp. 54-87.

[11]. Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. 2017. Source files and annotated RFC for TLS 1.3 analysis. (2017). Режим доступа: https://tls13tamarin.github.io/TLS13Tamarin/, дата обращения 10.06.2018.

[12]. Concrete Syntax Notation One (CSN.1). Режим доступа: http://csn1.info, дата обращения 10.06.2018.

[13]. Transfer Syntax Notation One (TSN.1). Режим доступа: http://www.protomatics.com/tsn1. Html, дата обращения 10.06.2018.

[14]. The BinPAC language. Режим доступа: https://www.bro.org/sphinx/components/binpac/README.html, дата обращения 10.06.2018..

[15]. Mario Baldi, Fulvio Risso. NetPDL: An Extensible XML-Based Language for Packet Header Description. Computer Networks, vol, 50, issue 5, 2006, pp. 688-706.