Testing of cryptographic protocols based on formal specifications

Mikhail Krichanov
Institute for System Programming
Russian Academy of Sciences,
National Research University
Higher School of Economics,
Email: krichanov@ispras.ru

Alexey Karnov
Institute for System Programming
Russian Academy of Sciences,
Moscow State University,
Email: karnov@ispras.ru

Grigoriy Volkov
Institute for System Programming
Russian Academy of Sciences,
National Research University
Higher School of Economics,
Email: gdvolkov@ispras.ru

***Abstract*—The article [1] proposed a notation for the definition of a protocol message called CMN.1, which was based on an abstraction named cryptographic stack machine. That declarative specification language was constructed so as to be directly used in the "Request for Comments" (RFC) document series to considerably enhance the degree of formalization of these documents. This paper presents the results of the validation of the aforementioned notation on a wide spectrum of cryptographic protocols (SSH, EAP, QUIC, HIP, IKEv2, Noise). Limitations of the declarative style of CMN.1 are discussed and solutions to the problems, across which came the authors during the development of specifications of the cryptographic protocols, are proposed.**

***Index terms*—** cryptographic stack machine, cryptographic protocol message notation, SSH, EAP, QUIC, HIP, IKEv2, Noise.

I. Introduction

The complexity of modern software systems requires special attention to quality assurance issues. Two of the most important aspects of the quality of software systems are reliability and security of information transfer in distributed systems. The complexity of this task is primarily due to the fact that it is often impossible to avoid the transfer of information through public networks with uncontrolled access. As a solution to the problem of secure data transmission network security protocols are used, which are based on methods of strong cryptography for protection against unauthorized access to the information. Absence of vulnerabilities in implementations of security protocols is critically important. There are various reasons of occurrence of vulnerabilities:

- Vulnerability in the protocol scheme (for example, using an encryption function whose cryptographic strength is less than expected). As a result all protocol implementations become vulnerable.
- Inconsistency between the implementation of the security protocol and its specification, incorrect implementation of the encryption algorithm, what results in the specific implementation of the protocol becoming vulnerable.
- Side channels, i.e. information about physical processes occurring in the system, not reflected in the verification and testing model: system response time, energy consumption graph, electromagnetic radiation, etc., can also be the basis for attacks on a specific protocol implementation.

One of the main ways to ensure reliability and security of program systems is to use formal methods of designing, development and analysis of programs ("formal methods"). The best results can be obtained by the usage of formal methods through out the whole stages of the software system development cycle:

- collection and analysis of requirements to the software system;
- design and verification;
- implementation;
- testing.

Depending on the kind of software system and its reliability and security requirements, formal methods can be used in one or several different ways:

- construction at the design stage of formal specifications or models based on the requirements to the software system;
- verification and falsification of formal models, proof or refutation of necessary properties, in particular security properties;
- generation of the software system implementation according to its formal specification;
- usage of programming languages with an elaborate system of types and the possibility of declarative description of required program behavior (for example, with the help of assumptions and postconditions) with the support of appropriate tools;
- testing the implementation of a software system for compliance with the constructed formal specification.

There exist many ways to improve the reliability of protocol implementation:

- Using special languages to implement protocols. One can attribute to them languages, which type systems allow to describe data security restrictions (security-typed languages [2], such as Jif [3] and Flow Caml [4]) or security properties of cryptographic primitives and design solutions of the protocol. Security properties of the protocol implementation written in such a language, can

be verified. It allows to use the verified implementation of the protocol as reference (miTLS [5]).
- Testing of protocol implementation for compliance with its formal specification. This method does not impose restrictions on the way of creation of the protocol implementation, and when the quality criteria is chosen correctly, it can provide a sufficient level of reliability, although it does not guarantee the absence of vulnerabilities.

The Cryptographic Stack Machine Notation One (CMN.1) is a domain specific language, which is designed for specification of cryptographic protocols. This specification language is embedded in Haskell programming language, that is any semantically correct protocol specification in CMN.1 is a semantically correct program in Haskell language. Thus, by the specification of protocols not only CMN.1 constructions are available, but also all functionalities of the Haskell language.

The abstract executor of the protocol specification in CMN.1 is a stack machine implemented in the support library. The set of instructions of this stack machine was developed taking into account specifics of cryptographic protocols. The cryptographic stack machine aims to process protocol messages – parse incoming messages and generate outgoing ones. Moreover, the same specification can be used for both variants of work.

In this work we tried to find out, how useful the proposed domain specific language could be for specification of cryptographic protocols. We chose 6 sufficiently complicated cryptographic protocols (SSH, EAP, QUIC, HIP, IKEv2, Noise), which support several methods of authentication, key generation and encryption, and wrote specifications for them. We implemented protocols not in their whole complexity but only their handshake parts, i.e. packet exchange responsible for authentication, generation of keys and selection of cipher suite.

II. Protocols Overview

A. EAP

Extensible Authentication Protocol (EAP) is an authentication framework which supports multiple authentication methods [6]. EAP typically runs directly over data link layers such as Point-to-Point Protocol (PPP) [7] or IEEE 802 [8], without requiring IP [9]. EAP provides its own support for duplicate elimination and retransmission, but is reliant on lower layer ordering guarantees. Fragmentation is not supported within EAP itself; however, individual EAP methods may support this.

EAP was designed for network access authentication, where IP layer connectivity may not be available. Since EAP does not require IP connectivity, it provides just enough support for the reliable transport of authentication protocols, and no more.

EAP is a lock-step protocol which only supports a single packet in flight. As a result, EAP cannot efficiently transport bulk data, unlike transport protocols such as TCP [10] or SCTP [11]. While EAP provides support for retransmission, it assumes ordering guarantees provided by the lower layer, so out of order reception is not supported.

EAP architecture uses the following roles for network nodes:

- authenticator — The end of the link initiating EAP authentication.
- peer — The end of the link that responds to the authenticator.
- backend authentication server — A backend authentication server is an entity that provides an authentication service to an authenticator.
- EAP server — The entity that terminates the EAP authentication method with the peer. In the case where no backend authentication server is used, the EAP server is part of the authenticator. In the case where the authenticator operates in pass-through mode, the EAP server is located on the backend authentication server.

One of the advantages of the EAP architecture is its flexibility. EAP is used to select a specific authentication mechanism, typically after the authenticator requests more information in order to determine the specific authentication method to be used. Rather than requiring the authenticator to be updated to support each new authentication method, EAP permits the use of a backend authentication server, which may implement some or all authentication methods, with the authenticator acting as a pass-through for some or all methods and peers.

The general scheme of the protocol looks like this. A client (peer) requests access to some resource by addressing to the access system (authenticator). The authenticator sends the request with client's data to the EAP server. The EAP server requests additional data from the client. The message exchange between the client and the EAP server continues until the selected authentication method succeeds or fails. The authenticator, based on the authentication result obtained from the EAP server, decides whether to grant the client access to the requested resource or not. Thus, the authenticator acts as an intermediary between the client and the EAP server.

The first communication channel between client and authenticator can be either wired or wireless. Quite often a mobile device acts as a client and an access point acts as an authenticator. EAP packets between the client and the authenticator are forwarded by encapsulating EAP packets in the lower layer protocol, such as PPP when using point-to-point channels, or EAPOL (EAP over LAN) on IEEE 802 networks [12].

The second channel between the authenticator and the EAP server is usually wired, but in some cases (such as in roaming scenarios) additional message forwarding objects may be placed between them. The above scheme allows for different variations. For example, the EAP server itself may be located on both the authenticator and the dedicated host. The second option allows to simplify access management to several shared resources, what is very important in a number of cases.

Many network devices usually have quite limited resources and, for example, cannot store information about a large

number of users in memory. In addition, the number of users can change regularly and a single database is required. In such a system, the authenticator of a particular resource must support only the basic functionality of the EAP protocol. And realization of all authentification methods and the database with the information on all users and their access rights reside in a uniform subsystem - an authentification server. The transfer of EAP packets between the back-end authentication server and the client is done by encapsulating EAP packets into the Authentication, Authorization and Accounting (AAA) protocol, which is used between the authenticator and the back-end authentication server. RADIUS [13] and Diameter [14] are usually used as AAA protocols.

At the same time, however, the network traffic increases. A combined scheme is also possible, where the authenticator supports some authentication methods, while other methods use a dedicated server. Also EAP server may not store data for authentication of clients and address for them to an external database. Thus one of the important properties of EAP is independence from the mode of operation of the authenticator, i.e. any EAP method works identically in all aspects regardless of whether the authenticator works in relay mode or not.

Other important properties of EAP protocol are independence from the environment, independence from the method and independence from the cipher suite. EAP protocol was originally developed for usage with the Point-to-Point Protocol (PPP) [7]. Later it came of use for access authentication in wired networks IEEE 802 [12] and wireless networks IEEE-802.11 [15] and IEEE-802.16e [16]. It is also used as a method of authentication in Internet Key Exchange Protocol version 2, IKEv2 [17].

Thus, one of the purposes of EAP creation was to ensure that its methods function properly above any underlying protocol, i.e. EAP methods should not use any downstream information, such as MAC addresses, during their execution. Method independence means that by providing relay mode, an authenticator can support any method implemented on the partner and server, not just locally implemented methods.

Essentially, independence from cipher suites provides independence from the environment. Since the cipher suites of different underlying protocols differ, independence from the environment requires that the key material being exported has sufficient length and entropy to handle any cipher suite. With EAP authentication, different environments can be used simultaneously and, as a result, EAP will be executed through different communication protocol stacks.

B. SSH

Secure Shell (SSH) is a protocol for secure remote login and other secure network services over an insecure network [18]. It consists of three major components:

- The Transport Layer Protocol [19] provides server authentication, confidentiality, and integrity. It may optionally also provide compression. The transport layer will typically be run over a TCP/IP connection, but might also be used on top of any other reliable data stream.
- The User Authentication Protocol [20] authenticates the client-side user to the server. It runs over the transport layer protocol.
- The Connection Protocol [21] multiplexes the encrypted tunnel into several logical channels. It runs over the user authentication protocol.

The client sends a service request once a secure transport layer connection has been established. A second service request is sent after user authentication is complete. This allows new protocols to be defined and coexist with the protocols listed above.

The connection protocol provides channels that can be used for a wide range of purposes. Standard methods are provided for setting up secure interactive shell sessions and for forwarding ("tunneling") arbitrary TCP/IP ports and X11 [22] connections.

The SSH transport layer [19] is a secure, low level transport protocol. It provides strong encryption, cryptographic host authentication, and integrity protection. Authentication in this protocol level is host-based; this protocol does not perform user authentication. A higher level protocol for user authentication can be designed on top of this protocol.

The protocol has been designed to be simple and flexible to allow parameter negotiation, and to minimize the number of round-trips. The key exchange method, public key algorithm, symmetric encryption algorithm, message authentication algorithm, and hash algorithm are all negotiated.

C. QUIC

QUIC (Quick UDP Internet Connections) is a multiplexed and secure general-purpose transport protocol [23] that provides:

- Stream multiplexing
- Stream and connection-level flow control
- Low-latency connection establishment
- Connection migration and resilience to network address translation (NAT) rebinding
- Authenticated and encrypted header and payload

QUIC uses UDP as a substrate to avoid requiring changes to legacy client operating systems and middleboxes. QUIC authenticates all of its headers and encrypts most of the data it exchanges, including its signaling, to avoid incurring a dependency on middleboxes.

QUIC relies on a combined cryptographic and transport handshake to minimize connection establishment latency. QUIC provides reliable, ordered delivery of the cryptographic handshake data. QUIC packet protection is used to encrypt as much of the handshake protocol as possible. TLS [24] acts as a security component of QUIC. TLS 1.3 provides critical latency improvements for connection establishment over previous versions. Absent packet loss, most new connections can be established and secured within a single round trip; on subsequent connections between the same client and server, the client can often send application data immediately, that is, using a zero round trip setup.

After completing the TLS handshake, the client will have learned and authenticated an identity for the server and the server is optionally able to learn and authenticate an identity for the client. TLS supports X.509 [25] certificate-based authentication for both server and client.

TLS provides two basic handshake modes of interest to QUIC:

- A full 1-RTT handshake in which the client is able to send application data after one round trip and the server immediately responds after receiving the first handshake message from the client.
- A 0-RTT handshake in which the client uses information it has previously learned about the server to send application data immediately.

Rather than a strict layering, these two protocols are co-dependent: QUIC uses the TLS handshake; TLS uses the reliability, ordered delivery, and record layer provided by QUIC. At a high level, there are two main interactions between the TLS and QUIC components:

- The TLS component sends and receives messages via the QUIC component, with QUIC providing a reliable stream abstraction to TLS.
- The TLS component provides a series of updates to the QUIC component, including (a) new packet protection keys to install (b) state changes such as handshake completion, the server certificate, etc.

QUIC carries TLS handshake data in CRYPTO frames, each of which consists of a contiguous block of handshake data identified by an offset and length. Those frames are packaged into QUIC packets and encrypted under the current TLS encryption level. As with TLS over TCP, once TLS handshake data has been delivered to QUIC, it is QUIC's responsibility to deliver it reliably. Each chunk of data that is produced by TLS is associated with the set of keys that TLS is currently using.

One important difference between TLS records (used with TCP) and QUIC CRYPTO frames is that in QUIC multiple frames may appear in the same QUIC packet as long as they are associated with the same encryption level. For instance, an implementation might bundle a Handshake message and an ACK for some Handshake data into the same packet. QUIC takes the unprotected content of TLS handshake records as the content of CRYPTO frames. TLS record protection is not used by QUIC. QUIC assembles CRYPTO frames into QUIC packets, which are protected using QUIC packet protection. TLS also provides QUIC with the transport parameters that the peer advertised during the handshake.

D. IKEv2

Version 2 of the Internet Key Exchange (IKE) protocol [17] performs mutual authentication between two parties and establishes an IKE Security Association (SA) that includes shared secret information that can be used to efficiently establish SAs for Encapsulating Security Payload (ESP) [26] or Authentication Header (AH) [27] and a set of cryptographic algorithms to be used by the SAs to protect the traffic that they carry.

IKE normally listens and sends on UDP. Since UDP is a datagram (unreliable) protocol, IKE includes in its definition recovery from transmission errors, including packet loss, packet replay, and packet forgery. IKE is designed to function so long as (1) at least one of a series of retransmitted packets reaches its destination before timing out; and (2) the channel is not so full of forged and replayed packets so as to exhaust the network or CPU capacities of either endpoint. Even in the absence of those minimum performance requirements, IKE is designed to fail cleanly (as though the network were broken). IKEv2 itself does not have a mechanism for fragmenting large messages.

E. HIP

Host Identity Protocol (HIP) [28] allows consenting hosts to securely establish and maintain shared IP-layer state, allowing separation of the identifier and locator roles of IP addresses, thereby enabling continuity of communications across IP address changes. The protocol is designed to be resistant to denial-of-service (DoS) and man-in-the-middle (MitM) attacks. When used together with another suitable security protocol, such as the Encapsulated Security Payload (ESP) [26], it provides integrity protection and optional encryption for upper-layer protocols, such as TCP and UDP.

In HIP, public cryptographic keys, of a public/private key pair, are used as Host Identifiers (HI), to which higher layer protocols are bound instead of an IP address. By using public keys (and their representations) as host identifiers, dynamic changes to IP address sets can be directly authenticated between hosts, and if desired, strong authentication between hosts at the TCP/IP stack level can be obtained. A system can have multiple identities, some 'well known', some unpublished or 'anonymous'. A system may self-assert its own identity, or may use a third-party authenticator like DNS Security (DNSSEC) [30], Pretty Good Privacy (PGP) [33], or X.509 [25] to 'notarize' the identity assertion. If the private key is possessed by more than one node, the Identity can be considered to be a distributed one.

A hashed encoding of the HI, the Host Identity Tag (HIT), is used in protocols to represent the Host Identity. The HIT is 128 bits long and has the following three key properties:

- it is the same length as an IPv6 address and can be used in address-sized fields in APIs and protocols;
- it is self-certifying (i.e., given a HIT, it is computationally hard to find a Host Identity key that matches the HIT);
- the probability of HIT collision between two hosts is very low.

The HIP base exchange is a two-party cryptographic protocol used to establish communications context between hosts. The base exchange is a Sigma-compliant [29] four-packet exchange. The first party is called the Initiator and the second party the Responder. The four-packet design helps to make HIP DoS resilient. The protocol exchanges Diffie-Hellman keys in the 2nd and 3rd packets, and authenticates the parties

in the 3rd and 4th packets. Additionally, the Responder starts a puzzle exchange in the 2nd packet, with the Initiator completing it in the 3rd packet before the Responder stores any state from the exchange. The Responder can remain stateless and drop most spoofed 3rds because puzzle calculation is based on the Initiator's Host Identity Tag. The idea is that the Responder has a (perhaps varying) number of precalculated 2nd packets, and it selects one of these based on the information carried in the 1st.

Finally, HIP is designed as an end-to-end authentication and key establishment protocol, to be used with Encapsulated Security Payload (ESP) and other end-to-end security protocols. The base protocol does not cover all the fine-grained policy control found in Internet Key Exchange (IKE) [17] that allows IKE to support complex gateway policies. Thus, HIP is not a replacement for IKE.

F. Noise

Noise [34] is a framework for crypto protocols based on Diffie-Hellman key agreement. Noise can describe protocols that consist of a single message as well as interactive protocols. The Noise framework supports handshakes where each party has a long-term static key pair and/or an ephemeral key pair. A Noise protocol sends a fixed sequence of handshake messages based on a fixed set of cryptographic choices. In some situations the responder needs flexibility to accept or reject the initiator's Noise protocol choice, or make its own choice based on options offered by the initiator.

NoiseSocket [35] provides an encoding layer for the Noise Protocol Framework. NoiseSocket can encode Noise messages and associated negotiation data into a form suitable for transmission over reliable, stream-based protocols such as TCP. The NoiseSocket framework allows the initiator and responder to negotiate a particular Noise protocol.

NoiseSocket doesn't specify the contents of negotiation data, since different applications will encode and advertise protocol support in different ways. NoiseSocket just defines a message format to transport this data, and APIs to access it.

III. CMN.1 Description

The article [1] presented a notation for the definition of a protocol message called CMN.1, namely the syntax and semantics of CMN.1 and the principles of implementation of the CMN.1-based **executable** protocol specification language. The article also proposed 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) look **declarative**.

The core language library (the CSM) 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 CSM takes the CMN.1 definition as input and produces the binary data in consistency with it. When an incoming message is received, the CSM verifies the binary data with respect to the given CMN.1 definition memorizing all the information needed in the further actions.

CMN.1-based specification language is integrated in Haskell language, in order to take advantage of the elegant and concise Haskell syntax.

CSM implementation details

Components of the cryptographic state machine:

express is a storage of symbolic expressions: a set of pairs *(j,t)*, where *j* is a unique descriptor of the expression (positive integer), *t* is a symbolic expression of the form *T opcode [j1,...,jn]*, where *opcode* is the code of some operation of the machine, *j1,...,jn* are descriptors of some other expressions from the storage of ***express***;

groups is a storage of expression groups: a set of pairs *(p,ks)*, where *p* is the number of the group (integer), *ks* is a set of descriptors included into the group; two descriptors are included into one group if the byte values of their expressions coincide;

lengths is a storage of byte value lengths of expressions: a set of pairs *(p,n)*, where *p* is the number of the expression group, *n* is the length of the byte value of the expression (an integer number not exceeding $2^{28} - 1$);

values is a storage of byte values of expressions: a set of pairs *(p,bs)*, where *p* is the number of the expression group, *bs* is a byte string;

myRole is a register that stores the identifier of the protocol role that this copy of the machine is running;

myNames is a storage of personal names of the subject of the protocol who owns this copy of the machine: a set of *ks* descriptors; each *k* in *ks* corresponds to an expression whose byte value is one of the names (e.g. in ASCII encoding or X.509 DN format);

step is a register that stores a two-dimensional machine step number: a pair of positive integers *(d,z)*, where *d* is the session number, *z* is the message number in the session;

events is an event log (list of pairs of the type *(ev,j)*, where *ev* is the event identifier , *j* is the descriptor, to which the event refers);

defaults is a storage of expressions used by default in case of non-computability (a descriptor is computable if the machine storages (***values*** and others) have all the information needed to calculate the byte value of the given descriptor) of the main expression (a set of pairs *(j, (k, r))*, where *j* is the descriptor of the main expression, *k* is the descriptor of the "default" expression, *r* is the role of the protocol member (Clnt, Serv, etc.), for which the default expression applies;

stacks, number of which can vary, that are dynamically created and destroyed during the operation of the cryptographic machine; a stack element is a descriptor of a symbolic expression;

At first the programs are executed symbolically: the elements of the stack are not byte strings but symbolic expressions, and then the final bytes strings are calculated.

This well-known technique allows 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: ciphertexts are decrypted, MACs and signatures are checked, etc. Throughout a protocol execution, the generated symbolic expressions are accumulated with their values, lengths and types. This information is used to generate or verify the protocol messages in the future.

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

1) CSM executes the program *p* symbolically resulting the symbolic expression *exp*. *EQ* is initialized with the equation *(exp,bs)*.
2) For every new equation *(exp,bs)* from *EQ*, until neither of Step 2.a or Step 2.b can be applied anymore:
   a) CSM 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.b.
   b) If the values of all the arguments of the top operation of the symbolic expression *exp* are known, CSM calculates the value of *exp*. If this value is equal to *bs*, CSM removes the equation from *EQ*. Otherwise, it returns the message verification error.

CMN.1 details

A CMN.1 program is a sequence of CSM instructions. The language of the CSM instructions supports branches, but not loops or recursion.

Examples of the instructions are listed below:

```
Ins ::=
InsFunctions | InsStackManip |
InsStepDepen | InsAttrib | InsSugared

InsFunctions ::=
FunReversible | FunNonrevers |
iConc Int | C [Word8] |
Vk String Step | iK String Step KeyTy |
iSelect [CaseTy] | iSubset Int |
iChoice Int | iPack [Int] |
iOptional | Undef

FunReversible ::=
iEncrypt EncryptCtxTy |
iEncode EncodeCtxTy | iPad PadTy |
iModAdd | iModMult | iModInv |
iAdd Integer | iReverse |
iWithTerm [Word8] | iXor Int |
Rev FunReversible

FunNonrevers ::=
iHash HashCtxTy | iMAC MACCtxTy |
iTakeFirst | iTakeLast |
iLength LenHdr | iMod | iModExp |
iECMult | iECAdd | iTakeElem Int SplitTy |
iSplitElem Int SplitTy | iPadWith [Word8] Int

InsStepDepen ::=
Vi String | Ki String KeyTy [Ins] |
All Ins [With [(Ins, Condition')]] |
Prev Ins | Last Ins

InsAttrib ::=
Len Int Ins | Default Ins Ins |
DefaultFor Role Ins Ins | DefaultForSrc Ins Ins |
Is Ins Ins

InsSugared ::=
W String | V String | Vr String Role |
Vri String Role | LastVi String | LastVri String |
PrevVi String | PrevVri String |
K String KeyTy [Ins] |
Kr String Role KeyTy [Ins] |
Kri String Role KeyTy [Ins] |
Kk String Step KeyTy [Ins] | M [Ins] | B [Ins] |
Hash HashCtxTy [Ins]] |
Encrypt EncryptCtxTy [Ins] |
Pad [Ins] PadTy Ins | ModAdd [Ins] |
Ins // Ins | Ins ## Ins | ...
```

IV. Results

For each of these protocols, formal specifications in the CMN.1 notation have been developed that provide for secure connections. The correctness of these specifications was checked by establishing secure connections with well-known implementations of corresponding protocols. The protocol implementations used in our project were selected based on their availability only.

FreeRADIUS is the only multifunctional free implementation that has outlived many competitors [37]. The given implementation is positioned by the developer as the most widespread RADIUS server which advantages include free distribution, an open source code, the developed functionality, support of numerous methods of EAP authentication.

Dropbear is a relatively small SSH server and client [36]. It runs on a variety of POSIX-based platforms. Dropbear is open source software.

strongSwan is an IKE daemon [38] with full support for IKEv1 and IKEv2. It is open source software and runs on a variety of POSIX-based platforms.

ngtcp2 project is an effort to implement QUIC protocol. It is available at GitHub [39].

OpenHIP is a free, open source implementation of the Host Identity Protocol (HIP) for use on multiple operating systems [40].

Noise-C is a plain C implementation of the Noise Protocol, intended as a reference implementation [41].

The servers were installed on Debian 10.

In the course of specifications development, the following limitations of the basic notation and support library have been revealed:

1) Lack of implementations of some hash functions and encryption methods used in cryptographic protocols.
2) Lack of support for some kinds of padding and integer encoding.
3) Lack of support for bitwise operations.
4) Lack of UDP support.
5) Lack of support for lists or arrays.
6) The architecture of the stack machine makes it difficult to perform calculations, which can not be described in terms of CMN.1. In such cases we have to operate with raw bytes, but the machine does not give access to the byte values of instructions at the time of packet generation or verification. Such a restriction harms the main goal of this notation, namely the declarative protocol description, because a programmer has to overcome arising problems in a very imperative style(e.g. solution to a puzzle in HIP).
7) CMN.1 specifies only structure of the packets but offers nothing for specification of order, in which packets should be exchanged.
8) Lack of support for error handling or "asymmetrical" instructions (i.e., instructions whose work would depend on whether the packet is being generated or parsed) leads to the fact that many calculations (e.g., digital signature verification) have to be performed in the main body of the messages exchanging program. This is detrimental to the declarative nature of the description.
9) CSM inner State is too huge. The machine tries to perform all the necessary computations at the very moment when it receives or sends a packet, making it difficult to decompose the task.
10) CMN.1 was integrated in Haskell, in order to use its functionality, but all CSM instructions are of the same data type. It makes us think that the abilities of the static type control of the Haskell compiler are underused.
11) Lack of support for multiple application of the specification or its part to parsing the received packet. In QUIC protocol, there are situations when several QUIC packets are sent in one UDP packet and when several TLS packets are sent in one QUIC packet. Since all specification variables are either global or tied to the sequence number of the packet generation or parse function call, to parse such nested packets one has to call the packet parse function, catch CSM error telling you that there is some data left not parsed, put this data back into the CSM buffer and call the packet parse function again and so several times.
12) When the data used for encryption, hash and keys are changed several times in one message, it is problematic to use CSM variables to store them. Such a variable can only have one value at each step of the stack machine, and the number and nature of such changes varies from message to message and from schema to schema, making it nearly impossible to use an indexed set of variables of the same type. Nevertheless, the actual state containing the keys, hash and other data, one has to transfer from one message to another. Such imperative nature of the protocol (e.g., Noise) does not match well with the capabilities of CMN.1. When creating a protocol specification, one has to store relevant data (current state of the protocol) in the form of a set of CMN.1 instructions, using the Haskell language tools, and in each message the entire chain of calculations must be built anew to obtain the actual keys and hash.

The first three limitations were not fundamental, as the architecture of the support library allows to add supplementary cryptographic functions. The forth limitation was also not that difficult to overcome and to add UDP support.

Limitation number 5 requires significant improvement of the support library, as it affects the basic principles of its design. The point is that the support library allows to extract only one value of a given variable from one message (the variable's value is always a byte sequence). And if you have a list of fields, the length of which is unknown in advance, you need a previously unknown number of variables in the specification to extract data from these fields. One of the possible solutions to this problem is to introduce variables which values are lists of byte sequences; lengths of lists will be determined by the number of fields in the message.

REFERENCES

[1] 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

[2] Aslan Askarov and Andrei Sabelfeld. *Security-typed languages for implementation of cryptographic protocols: A case study.* Proc. of ESORICS 2005, Milan, Italy, Sept. 12-14, 2005. LNCS.

[3] Jif — a security-typed programming language. [Online]. Available: http://www.cs.cornell.edu/jif/

[4] Flow Caml. [Online]. Available: http://www.normalesup.org/∼simonet/soft/flowcaml/

[5] miTLS: A Verified Reference Implementation of TLS. [Online]. Available: http://mitls.org/

[6] IETF RFC 3748. B. Aboba, et al., Extensible Authentication Protocol (EAP). June 2004. [Online]. Available: https://tools.ietf.org/html/rfc3748

[7] IETF RFC 1661. W. Simpson. The Point-to-Point Protocol (PPP). July 1994. [Online]. Available: https://tools.ietf.org/html/rfc1661

[8] IEEE Standard 802, Institute of Electrical and Electronics Engineers, "Local and Metropolitan Area Networks: Overview and Architecture", 1990.

[9] IETF RFC 791, Internet Protocol, September 1981. [Online]. Available: https://tools.ietf.org/html/rfc791

[10] RFC 793, J. Postel, "Transmission Control Protocol", STD 7, September 1981. [Online]. Available: https://tools.ietf.org/html/rfc793

[11] RFC 2960, R. Stewart, et al., "Stream Control Transmission Protocol", October 2000. [Online]. Available: https://tools.ietf.org/html/rfc2960

[12] IEEE Standard 802.1X-2010 - IEEE Standard for Local and metropolitan area networks–Port-Based Network Access Control, 2010.
[13] IETF RFC 3579. B. Aboba and P. Calhoun. RADIUS (Remote Authentication Dial In User Service) Support For Extensible Authentication Protocol (EAP). September 2003. [Online]. Available: https://tools.ietf.org/html/rfc3579
[14] IETF RFC 4072. Eronen, et al., Diameter Extensible Authentication Protocol (EAP) Application. August 2005. [Online]. Available: https://tools.ietf.org/html/rfc4072
[15] IEEE Standard 802.11-2007, Institute of Electrical and Electronics Engineers, "Standard for Local and metropolitan area networks - specific requirements – part 11: Wireless LAN Medium Access Control and Physical Layer specifications", 2007.
[16] IEEE Standard 802.16e-2005, Physical and Medium Access Control Layers for Combined Fixed and Mobile Operation in Licensed Bands. December 2005.
[17] IETF RFC 7296. C. Kaufman, et al., Internet Key Exchange Protocol Version 2 (IKEv2). October 2014. [Online]. Available: https://tools.ietf.org/html/rfc7296
[18] RFC 4251, T. Ylonen and C. Lonvick, Ed., "The Secure Shell (SSH) Protocol Architecture", DOI 10.17487/RFC4251. January 2006. [Online]. Available: https://tools.ietf.org/html/rfc4251
[19] RFC 4253, T. Ylonen and C. Lonvick, Ed., "The Secure Shell (SSH) Transport Layer Protocol", January 2006. [Online]. Available: https://tools.ietf.org/html/rfc4253
[20] RFC 4252, T. Ylonen and C. Lonvick, Ed., "The Secure Shell (SSH) Authentication Protocol", January 2006. [Online]. Available: https://tools.ietf.org/html/rfc4252
[21] RFC 4254, T. Ylonen and C. Lonvick, Ed., "The Secure Shell (SSH) Connection Protocol", January 2006. [Online]. Available: https://tools.ietf.org/html/rfc4254
[22] R. Scheifler, *X Window System : The Complete Reference to Xlib, X Protocol, Icccm, Xlfd, 3rd edition.*, Digital Press, ISBN 1555580882, February 1992.
[23] Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", draft-ietf-quic-transport-22 (work in progress), July 2019. [Online]. Available: https://tools.ietf.org/html/draft-ietf-quic-transport-22
[24] Thomson, M., Ed. and S. Turner, Ed., "Using Transport Layer Security (TLS) to Secure QUIC", draft-ietf-quic- tls-22 (work in progress), July 2019. [Online]. Available: https://tools.ietf.org/html/draft-ietf-quic-tls-22
[25] RFC 5280, D. Cooper, et al., "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", DOI 10.17487/RFC5280, May 2008. [Online]. Available: https://tools.ietf.org/html/rfc5280
[26] RFC 4303, S. Kent, "IP Encapsulating Security Payload (ESP)", December 2005. [Online]. Available: https://tools.ietf.org/html/rfc4303
[27] RFC 4302, S. Kent, "IP Authentication Header", December 2005. [Online]. Available: https://tools.ietf.org/html/rfc4302
[28] RFC 5201, R. Moskowitz, et al., "Host Identity Protocol", April 2008. [Online]. Available: https://tools.ietf.org/html/rfc5201
[29] Krawczyk, H., *SIGMA: The 'SIGn-and-MAc' Approach to Authenticated Diffie-Hellman and Its Use in the IKE-Protocols*, in Proceedings of CRYPTO 2003, pages 400-425, August 2003.
[30] RFC 4033, R. Arends, et al., "DNS Security Introduction and Requirements", March 2005. [Online]. Available: https://tools.ietf.org/html/rfc4033
[31] RFC 4034, R. Arends, et al., "Resource Records for the DNS Security Extensions", March 2005. [Online]. Available: https://tools.ietf.org/html/rfc4034
[32] RFC 4035, R. Arends, et al., "Protocol Modifications for the DNS Security Extensions", March 2005. [Online]. Available: https://tools.ietf.org/html/rfc4035
[33] RFC 4880, J. Callas, et al., "OpenPGP Message Format", November 2007. [Online]. Available: https://tools.ietf.org/html/rfc4880
[34] Perrin, T., The Noise Protocol Framework, July 2018. [Online]. Available: https://noiseprotocol.org/noise.html
[35] Ermishkin, A. and T. Perrin, The NoiseSocket Protocol, March 2018. [Online]. Available: https://noisesocket.org/spec/noisesocket/
[36] Dropbear. [Online]. Available: https://matt.ucc.asn.au/dropbear/dropbear.html
[37] FreeRADIUS. [Online]. Available: https://freeradius.org/
[38] strongSwan. [Online]. Available: https://www.strongswan.org/
[39] ngtcp2. [Online]. Available: https://github.com/ngtcp2/ngtcp2
[40] OpenHIP. [Online]. Available: https://sourceforge.net/projects/openhip/
[41] Noise-C. [Online]. Available: https://github.com/rweather/noise-c