Verifying Cryptographic Protocols: A symbolic model for cryptographic APIs for C

Gijs Vanspauwen[1] and Bart Jacobs[1]

[1]iMinds-DistriNet, KU Leuven, 3001 Leuven, Belgium.

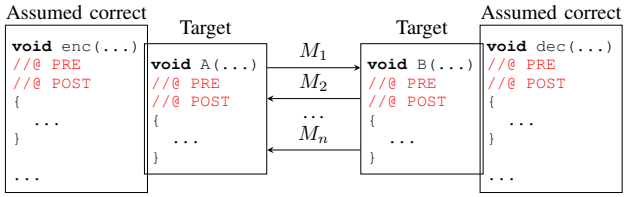

Figure 1. Overview of our approach for verifying cryptographic protocols

***Abstract*—We propose an approach for verifying cryptographic protocol implementations written in C. We statically prove the correctness of these implementations with the general purpose verifier VeriFast. More concretely we prove: memory safety, the absence of explicit and implicit information leaks, and functional correctness which includes protocol integrity. Our invariant-based approach requires an extension of the symbolic model of cryptography in order to work for protocol implementations in C written against an existing cryptographic API.**

***Index Terms*—static verification, verification of C programs, cryptographic protocols, symbolic model of cryptography, cryptographic libraries**

I. Introduction

We almost cannot imagine our everyday lives anymore without a connection to the Internet. From managing bank accounts to staying in touch with friends: we heavily rely on this massive piece of technology. In order to provide a secure environment for these day-to-day activities, web browsers and web servers they communicate with rely on cryptography.

In this paper, we describe an approach to proving the correctness of such cryptographic software written in the C programming language. More precisely, our approach (depicted in Figure 1) allows to verify cryptographic protocol implementations. Such an implementation is a software realization of a particular communication pattern that establishes a certain security goal (e.g. confidentiality of a message) using cryptographic primitives (e.g. key generation and encryption). Each protocol participant is assigned a specific role and different roles are implemented separately as, for instance, distinct C functions like the functions A and B from Figure 1. Together, these roles make up a protocol implementation. Assuming the cryptographic primitives are perfect, we prove that the implementation of such a protocol is memory safe, does not leak secrets and indeed achieves its security goals.

The approach we developed builds on top of the work from [11] and is explained in full detail in [12]. In this paper, we highlight the most important aspects and a first noteworthy property of our method is that it is similar to the approaches from [1] and [6]: use a general purpose program verifier to verify protocol implementations that are written against a trusted API containing the cryptographic primitives. First, the semantics of the functions in this API are specified through contracts (i.e. the pre- and postconditions in Figure 1) which are assumed correct. Then, the corresponding verification methodology can use these contracts to reason about the protocol implementations (refinement type checking in [1] and symbolic execution in [6]).

The cryptographic API in both [1] and [6] is designed in such a way that verification of the protocols within the symbolic model of cryptography is directly possible. In the symbolic model all messages are terms of a cryptographic algebra. The cryptographic primitives construct messages in this algebra and pairing/unpairing operators allow to compose/decompose messages. For this reason the cryptographic API in both [1] and [6] contains, besides the cryptographic primitives, functions to compose and decompose messages, and the network API accepts and returns these messages. The contracts of all the functions in the resulting API then allow to track the symbolic message content of memory regions during verification. More importantly however, the contracts of the network API enable the enforcement of a network invariant for messages allowed on the network (see [4]). This network invariant is key to proving functional correctness and security of protocol implementations.

A crucial difference with our approach is that we target protocol implementations that employ preexisting cryptographic libraries. Mbed TLS and the more widely known OpenSSL are two examples of such preexisting libraries that provide the required cryptographic functionality for writing protocol implementations. Since we target protocols that use such libraries we do not have the liberty to design the trusted API as we see fit. We are stuck with their accompanying APIs and most cryptographic libraries do not have a structured message concept such as is necessary for the approaches described in [1] and [6]. Instead, they leave it up to the protocol implementation to compose and parse messages using C buffers and C functions like `memcpy` and `memcmp`.

Although in such a setting the symbolic model could not be directly applied as in [1] or [6], we aspired to use a similar symbolic reasoning. Therefore we introduced an extension of the symbolic model of cryptography. This extended model enables the verification of protocol code that itself implements

the composing and parsing of messages. It allows to associate symbolic cryptographic values with bit strings present in memory, and to keep track of these values while composing and parsing messages. In turn, the contracts of the functions in a cryptographic API can express associations between input and output buffers to functions on the one hand, and the symbolic result of cryptographic primitives on the other hand.

The rest of this paper starts with an overview of our extended symbolic model in Section II. Then, we devote the entire Section III to the further discussion of symmetric decryption. Next, in Section IV, we discuss an exemplary verified protocol implementation of which the complete annotated source code is given in the appendix. We give some results in Section V and related work is discussed in Section VI. Finally, we conclude this paper in Section VII.

II. Extended Symbolic Model in VeriFast

While a fundamental specification of our extended symbolic model in the Coq proof assistant[1] is under development, we initially encoded it directly in VeriFast[2], a general purpose verifier for C programs. It is this encoding we discuss and the encoding depends on different VeriFast concepts. For readability, we always explain the purpose and intended meaning of the definitions shown, but refer to [12] for more details.

In the outline of our approach here, we focus on symmetric encryption and the generation of nonces and keys. The full encoding of our extended symbolic model in VeriFast is available in the `examples/crypto_with_flow` directory of the latest VeriFast release and also deals with hashes, keyed hashes, authenticated encryption, asymmetric encryption and signatures. We chose mbed TLS[3] as the preexisting cryptographic library to write verified protocol implementations against, but the same approach works for other libraries that provide similar functionality (e.g. OpenSSL).

The following subsections will introduce our extended symbolic model of cryptography step-by-step. We start in Subsection II-A with a small overview of the entire approach. Then, in Subsection II-B, we discuss the definitions to track principal identities during symbolic execution. Each function implementing (part of) a protocol role will need such an identity as a permission to invoke specific other functions from the trusted API (e.g. the network functions). When a protocol participant misbehaves, this permission is revoked in order to enforce the security guarantees of our model. Next we show how verified protocol code can actually send bytes on the network in Subsection II-C. For this we chose the fairly standard (i.e. comparable to the `sys/socket.h` header from POSIX) network API of mbed TLS and augmented it with VeriFast contracts to specify its semantics. In Subsection II-D we discuss all the definitions required to give meaningful contracts to cryptographic primitive implementations and we conclude that subsection by showing contracts of the relevant mbed TLS primitives. It's also discussed there that we call the symbolic representation of the result of a cryptographic primitive, a cryptogram. That result could be a secret (e.g. a generated session key) and the caller of a cryptographic primitive does not automatically have the permission to read from the corresponding memory region. This prevents secret information from leaking into regular program variables or into the program's control flow. How to prove that such a memory region does not contain any secrets to get read permissions to it, is explained in Subsection II-E. We use an invariant-based approach for these proofs, very similar to [1], [4] and [6]. The main difference is that we enforce this invariant on readable memory regions, instead of on messages on the network. Invariants are also the key mechanism to prove security properties of protocol implementations. Subsection II-F concludes this section with a discussion of our attacker model.

[1]The Coq proof assistant: https://coq.inria.fr/

[2]VeriFast: http://distrinet.cs.kuleuven.be/software/verifast

[3]mbed TLS: https://tls.mbed.org/

A. Overview of the verification approach

The verification of a cryptographic protocol implementation written against our trusted annotated API comprises the following steps, not necessarily in this order:

- augment each function implementing a protocol role with a contract that encodes the expected security properties
- write a main function that runs an undetermined number of protocol instances and our attacker implementation in parallel resulting in a single application
- interactively verify the entire application with VeriFast

The result of a successful verification is a protocol that can withstand any attacker that has complete access to the network and can produce any message using the same cryptographic API as the honest protocol participants. All the steps of our approach heavily depend on the definitions, function prototypes and contracts that make up our trusted cryptographic API. In the rest of this section we discuss all its essential parts.

B. Tracking identities of principals

Each function that implements a protocol role will need an identity to invoke specific functions from the trusted cryptographic API. This identity is not only a permission to use the network functions (see Subsection II-C), to generate random values (see Subsection II-D) or to perform decryption (see Section III), it also allows us to link random values to the identity of the creator. During the verification of, according to our model, misbehaving protocol code, some of these permissions can be revoked in order to prevent the code from undermining its own security goals.

The definitions for tracking principal identities during verification are shown in Listing 1. For clarity, the different permissions that an identity comprises are defined separately as the predicates `network_permission`, `random_permission` and `decryption_permission`. A principal identity then is defined in terms of these permissions, as a chunk of the predicate `principal`. Its first argument is the sequence number in the line of generated identities to ensure that each identity is unique. The second argument keeps track of how many random values are generated for the principal. The predicate `principals` is used to keep track of

```
/*@
predicate network_permission(int principal;);
predicate random_permission(int principal, int values;);
predicate decryption_permission(int principal;);
predicate principal(int principal, int values;) =
  network_permission(principal) &*&
  random_permission(principal, values)  &*&
  decryption_permission(principal)
;
predicate principals(int count);

require_module principals_mod;
lemma void principals_init();
  requires module(principals_mod, true);
  ensures  principals(0);

lemma int principal_create();
  requires principals(?count);
  ensures  principal(result, 0) &*& result == count + 1 &*&
           principals(result);
@*/
```

Listing 1. Tracking principal identities and corresponding permissions

```
//@ import_module principals_mod;
void role1()
//@ requires principal(?p1, ?random_values);
//@ ensures  principal(p1, ?new_random_values);
{
  //@ open principal(p1, random_values);
  /* ... */
  //@ close principal(p1, _);
}
```

Listing 2. Illustration of using the definitions for principal identities

how many identities are generated thus far as indicated by its only argument. To retrieve the permission for generating identities (i.e. a chunk of the predicate `principals`), the lemma `principals_init` should be invoked. The precondition of this lemma uses the module system of VeriFast (notice the `require_module` declaration) to ensure that the lemma can only be invoked once in order to prevent the recycling of identities. How exactly this module system works is not explained here, but illustrative examples can be found in the latest VeriFast release. With the permission for generating identities, one can start creating new identities by invoking the lemma `principal_create`.

Listing 2 illustrates these definitions with the implementation of a single protocol role `role1`. The contract of `role1` reflects that this implementation of a protocol role needs access to a principal identity. The main function that launches this protocol role, has to use the lemmas `principals_init` and `principal_create` to generate that identity. Important to note here is that each piece of protocol code should only have access to one principal identity. Otherwise a misbehaving piece of protocol code which gets some of its identity permissions revoked (e.g. to prevent it from further using the network API, see Subsection II-C), can still fall back to another identity. This would make our encoding of the extended symbolic model in VeriFast unsound and is therefore not allowed in our approach. Although this could be enforced using function type definitions, we rely on a simple manual check for now.

C. The network API

An extract from the annotated network API is shown in Listing 3. The C functions shown are obtained from the

```
/*@
inductive socket_status =
 | bound_to_port | socket_init | connected;

predicate net_status(int socket, list<char> address,
                     int port, socket_status status);
@*/
int net_connect(int *socket, const char *host, int port);
/*@ requires integer(socket, _) &*&
             [?f]option_string(host, ?h); @*/
/*@ ensures  integer(socket, ?s) &*&
             [f]option_string(host, h) &*&
             result != 0 ? true :
             net_status(s, h, port, socket_init); @*/

int net_set_block(int socket);
//@ requires net_status(socket, ?h, ?p, socket_init);
/*@ ensures  result != 0 ? true :
             net_status(socket, h, p, connected) ; @*/

int net_send(void *socket, const char *buf, size_t len);
/*@ requires network_permission(?id) &*&
             integer(socket, ?s) &*&
             net_status(s, ?ip, ?port, connected) &*&
             len <= MAX_SIZE &*&
             [?f1]chars(buf, len, ?cs); @*/
/*@ ensures  network_permission(id) &*&
             integer(socket, s)  &*&
             net_status(s, ip, port, connected) &*&
             [f1]chars(buf, len, cs); @*/

int net_recv(void *socket, char *buf, size_t len);
/*@ requires network_permission(?id) &*&
             integer(socket, ?s) &*&
             net_status(s, ?ip, ?port, connected) &*&
             chars(buf, len, _) &*& len <= MAX_SIZE; @*/
/*@ ensures  network_permission(id) &*&
             integer(socket, s)  &*&
             net_status(s, ip, port, connected) &*&
             chars(buf, len, _) &*& result <= len; @*/

void net_close(int socket);
//@ requires net_status(socket, _, _, _);
//@ ensures  true;
```

Listing 3. Extract from the annotated network API

network API of mbed TLS and they form a classical socket API. Specifications added to these C functions ensure that verified code uses them in a correct fashion. For brevity, only the functions to establish a connection at client-side are shown.

To establish a connection, a client first has to call the function `net_connect`. If successful (i.e. the result is zero), the client receives a chunk of the predicate `net_status`. As the postcondition of `net_connect` indicates, the third argument of this chunk will be `socket_init`. This reflects the fact that the initialization of the socket is not complete after only calling `net_connect`, since the communication type still has to be set to blocking or non-blocking. So the final step in initializing a client socket is calling `net_set_block`. For simplicity we only support blocking communication. After calling `net_set_block` the client receives a chunk of the form `net_status(_, _, _, connected)` and he can start sending and receiving messages via the corresponding socket using the functions `net_send` and `net_receive`. The contracts of the send and receive function require the caller to have the permission to use the network API and a completely initialized socket. The actual messages sent and received are simply character buffers described by VeriFast's usual `chars` chunks. Finally, the function `net_close` allows one to close a socket.

```
/*@
inductive cryptogram =
  | cg_nonce          (int principal, int i)
  | cg_symmetric_key  (int principal, int i)
  | cg_encrypted      (int principal, int i,
                       list<char> pay, list<char> iv);
@*/
```

Listing 4. Cryptograms are the results of cryptographic computations

D. Contracts for cryptographic primitives

In this subsection we introduce all the definitions that are required to specify the contract of a cryptographic primitive. At the end we also give contracts for two cryptographic primitives selected from mbed TLS:

- `havege_random` (nonce and key generation)
- `aes_crypt_cfb128` (encryption and decryption)

a) Cryptograms: Before we can specify a contract for some cryptographic primitive, we need a symbolic representation of the result of cryptographic computations. We call these results cryptograms, i.e. instances of the inductive datatype `cryptogram` from Listing 4. As already mentioned, we focus here on symmetric encryption and the generation of random nonces and keys. So in this scope `cryptogram` needs only three constructors, while more constructors are necessary in the full encoding of our extended symbolic model.

The first two constructors both represent random values and we make a distinction between symmetric keys and nonces for clarity[4]. Each of these two constructors has two parameters and a `cg_nonce(p,i)` or `cg_symmetric_key(p,i)` cryptogram symbolizes the ith random value generated by principal `p`. Symmetric encrypted messages then, are represented by the constructor `cg_encrypted` and this constructor has four parameters. The first two parameters serve to identify the key that was used and the third parameter records the plaintext. The fourth parameter finally, allows for the same plaintext to be encrypted with the same key to different ciphertexts. This is why an initialization vector is used in encryption and so the parameter `iv` corresponds to the initialization vector that was chosen to create the ciphertext.

It is instructive to contrast this definition of `cryptogram` with the definition of messages in a regular symbolic model. Such a definition is shown in Listing 5 as the inductive datatype `msg`. A clear similarity between these two definitions is the representation of nonces and symmetric keys. An obvious difference is that there is no pairing operator to compose two cryptograms in our extended symbolic model while `msg` has the constructor `msg_pair` to compose messages. Another obvious difference is that `cryptogram`, in contrast to `msg`, has no `msg_data` constructor because plain messages simply have the type `list<char>` in the encoding of our extended symbolic model. The constructor `msg_encrypted` and the corresponding constructor `cg_encrypted` are very similar except that they differ in the type for their payloads. In the definition of `msg` this type is `msg` itself and so the definition of `msg` is truly recursive. The definition of `cryptogram` on the other hand is not recursive since the payload in the constructor `cg_encrypted` has the type `list<char>`.

[4] It also prevents keys from being used as nonces and vice versa.

```
/*@
inductive msg =
  | msg_data            (list<char> raw_data)
  | msg_pair            (msg fst, msg snd)
  | msg_nonce           (int principal, int i)
  | msg_symmetric_key   (int principal, int i)
  | msg_auth_encrypted  (int principal, int i,
                         msg payload, list<char> iv);
@*/
```

Listing 5. Messages in a classic symbolic model: NOT part of our approach

```
//@ fixpoint list<char> chars_for_cg(cryptogram cg);
```

Listing 6. Character representation of a cryptogram

b) Character representation of cryptograms: In the contract for a cryptographic primitive, we want to link the contents of a C buffer to some symbolic result produced by that cryptographic primitive (i.e. a cryptogram). As a first step we define the pure function `chars_for_cg` in Listing 6. This function returns the exact character representation for a given cryptogram. It is initially completely unspecified, but its function values are determined during symbolic execution with VeriFast by the postconditions of the cryptographic primitive implementations. Suppose the primitive for random value generation outputs, in a C buffer, the character representation `cs` of a symbolic cryptogram `key` as Figure 2 illustrates. Then we know `chars_for_cg(key)` is equal to `cs`.

We chose to give the function `chars_for_cg` an injectivity and surjectivity property for each kind of cryptogram as these properties simplify the rest of the approach. Surjectivity means that any list of characters can, for exampls, be interpreted as some unspecified `cg_nonce`. Injectivity means that if, for example, the character representation of two keys is equal, then they must be the same key or a cryptographic collision occurred. The definitions for these properties are not shown due to space constraints.

c) Memory regions containing secrets: The result of a cryptographic primitive could be a secret (e.g. a session key), and we want to prevent leaking secrets to the attacker. For this reason we treat C buffers where cryptographic primitives store their results in a special way. Permissions of C buffers are normally tracked by chunks of the predicate `chars` as declared in Listing 7. A chunk `chars(buffer, n, cs)` for example, indicates that there is a valid allocated memory region starting at the address `buffer` with a size of `n` bytes and with content `cs`. VeriFast grants the owner of such a chunk the permission to read and write the indicated memory region. We now introduce the very similar predicate `crypto_chars`. The difference with the `chars` predicate is that memory regions expressed by a `crypto_chars`

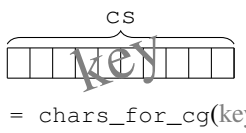

Figure 2. Illustration of the intended meaning of `chars_for_cg`

```
/*@
inductive crypto_chars_kind = | normal | secret;

predicate chars(char *buffer, int n; list<char> cs);
predicate crypto_chars(crypto_chars_kind kind,
                       char *buffer, int n; list<char> cs);
@*/
```

Listing 7. Chars vs. crypto_chars: possible confidentiality of memory

```
/*@
lemma void crypto_chars_to_chars(char *array, int n);
  requires [?f]crypto_chars(?kind, array, n, ?cs) &*&
           col || kind == normal;
  ensures  [f]chars(array, n, cs);

lemma void chars_to_crypto_chars(char *array, int n);
  requires [?f]chars(array, n, ?cs);
  ensures  [f]crypto_chars(normal, array, n, cs);

lemma void chars_to_secret_crypto_chars(char *array, int n);
  requires [?f]chars(array, n, ?cs);
  ensures  [f]crypto_chars(secret, array, n, cs);
@*/
```

Listing 8. Some conversions between chars and crypto_chars

chunk contain possibly secret cryptographic information. Since `crypto_chars` is a custom predicate (i.e. it is not part of the predefined predicates known to VeriFast), the corresponding memory region is neither readable nor writable directly.

Chunks of the predicate `crypto_chars` come in two flavors and what flavor a chunk belongs to depends on its first argument. A `crypto_chars` chunk where the first argument is `normal` is equivalent to a regular `chars` chunk. This means it can be converted to a `chars` chunk and back with the lemmas `crypto_chars_to_chars` and `chars_to_crypto_chars` from Listing 8. In normal circumstances (i.e. if no cryptographic collision occurs), a `crypto_chars` chunk where the first argument is `secret` can only be converted to a `chars` chunk after one has proven that its content is not secret (see Subsection II-E). This restriction prevents actual secret information from leaking into regular program variables or into the program's control flow. Such a `secret crypto_chars` chunk can also be converted to a `chars` chunk with the lemma `crypto_chars_to_chars` if a cryptographic collision occurs (i.e. if `col` equals `true`), which is fine as we prove all protocol properties "up to a collision". The lemma `chars_to_secret_crypto_chars` finally, converts any `chars` chunk to a `secret crypto_chars` chunk.

We now can establish a link between a symbolic cryptogram and an actual C buffer using the predicate with the overloaded name `cryptogram` defined in Listing 9. The body of this definition reads as follows: "The possibly secret memory region of length `n` starting at address `buffer` is not

```
/*@
fixpoint bool cg_is_generated(cryptogram cg);

predicate cryptogram(char* array, int n, list<char> cs,
                     cryptogram cg) =
  crypto_chars(secret, array, n, cs) &*&
  cs == chars_for_cg(cg) && cg_is_generated(cg);
@*/
```

Listing 9. Expressing the cryptographic content of a C buffer

```
struct havege_state{ /*...*/ };
typedef struct havege_state havege_state;
//@ predicate havege_state(havege_state *s) = true /*...*/;
//@ predicate havege_state_initialized(havege_state *s);

void havege_init(havege_state *havege_state);
//@ requires havege_state(havege_state);
//@ ensures  havege_state_initialized(havege_state);

void havege_free(havege_state *havege_state);
//@ requires havege_state_initialized(havege_state);
//@ ensures  havege_state(havege_state);
```

Listing 10. Initializing and freeing a context for random value generation

```
/*@ predicate random_request(int principal, int info,
                             bool key_request) = true; @*/
//@ fixpoint int cg_info(cryptogram cg);

int havege_random(void *state, char *output, size_t len);
/*@ requires [?f]havege_state_initialized(state) &*&
             random_request(?p, ?info, ?key_request) &*&
             random_permission(p, ?count) &*&
             chars(output, len, _) &*& len >= MIN_SIZE; @*/
/*@ ensures  [f]havege_state_initialized(state) &*&
             random_permission(p, count + 1) &*&
             result == 0 ?
               cryptogram(output, len, ?cs, ?cg) &*&
               info == cg_info(cg) &*& key_request ?
                 cg == cg_symmetric_key(p, count + 1)
               :
                 cg == cg_nonce(p, count + 1)
             : chars(output, len, _);  @*/
```

Listing 11. A cryptographic primitive to generate random values

only correctly allocated, its contents `cs` is also the character representation of the generated cryptogram `cg`." The concept of generated cryptograms is important for the contract of `memcmp` discussed in Subsection II-E.

d) A primitive for random value generation: The annotated primitive we chose for random value generation is `havege_random`. It implements HAVEGE[5] random number generation and is shown in Listing 11. To invoke this primitive, an initialized `havege_state` structure is required. One can initialize such a structure with the function `havege_init` from Listing 10. The contract of this function takes a `havege_state` chunk and transforms it into a `havege_state_initialized` chunk. Once all necessary random values are generated, the initialized structure can be discarded via the function `havege_free`, also from Listing 10.

With an initialized `havege_state` structure one can start generating random values. As the precondition of `havege_random` in Listing 11 indicates, also a permission to generate random values is required. This permission takes the form of a chunk of the predicate `random_permission` as discussed in Subsection II-B. The caller of `havege_random` must also provide a `random_request` chunk primarily to pass the ghost arguments `info` and `key_request`. Since we differentiate between keys and nonces, the caller has to indicate if he wants to generate a key or a nonce. He can do exactly this with the parameter `key_request`. The `info` parameter

[5] HAVEGE: HArdware Volatile Entropy Gathering and Expansion, http://www.irisa.fr/caps/projects/hipsor/

```
struct aes_context{ /*...*/ };
typedef struct aes_context aes_context;

//@ predicate aes_context(aes_context *c) = true /*...*/;
/*@ predicate aes_context_initialized(aes_context *ctx,
                                      int p, int c);@*/

int aes_setkey_enc(aes_context *ctx, const char *key,
                   unsigned int keysize);
/*@ requires [?f]cryptogram(key, ?size, ?cs, ?cg) &*&
               keysize == size * 8 &*&
               cg == cg_symmetric_key(?p, ?c) &*&
             aes_context(ctx) &*&
             (keysize == 128 || keysize == 192
                             || keysize == 256); @*/
/*@ ensures  [f]cryptogram(key, size, cs, cg) &*&
             result == 0 ?
               aes_context_initialized(ctx, p, c)
             :
               aes_context(ctx); @*/

void aes_free(aes_context *ctx);
//@ requires aes_context_initialized(ctx, _, _);
//@ ensures  aes_context(ctx);
```

Listing 12. Initializing and freeing a context for symmetric encryption

allows to associate some custom information with the resulting cryptogram by choosing the function value of `cg_info`. Function values of `cg_info` are important when proving authentication properties of protocol implementations (see Section IV). The last requirement imposed by the precondition then, is a correctly allocated output buffer. If the result of a call to `havege_random` is successful (i.e. the return value is zero), then the postcondition ensures that the content of the output buffer is linked to the proper cryptogram and that this cryptogram has the correct information associated with it. Note that in the postcondition of `havege_random` the second argument of the `random_permission` chunk is incremented to ensure that all generated random values are linked with a distinct symbolic cryptogram.

e) An encryption and decryption primitive: We chose the AES cipher[6] for symmetric encryption and decryption. The C functions selected from mbed TLS that are sufficient to encrypt and decrypt with this cipher are: `aes_setkey_enc`, `aes_free` and `aes_crypt_cfb128`. As was the case for for generating random values, also for `aes_crypt_cfb128` some C structure must first be initialized. More specifically an `aes_context` structure must be initialized with a correctly generated key using the function `aes_setkey_enc` from Listing 12 and after its usage the function `aes_free` frees such a structure. Note that the precondition of `aes_setkey_enc` enforces that the length of the key is 128 bits, 192 bits or 256 bits as is required by the AES cipher.

The C function `aes_crypt_cfb128` from Listing 13 implements both encryption and decryption. Which of these two operations is performed is determined by the value provided for the parameter `mode`: `AES_ENCRYPT` or `AES_DECRYPT`. The next thing the precondition requires is a chunk of the predicate `aes_context_initialized`, an initialization

[6] Advanced Encryption Standard (FIPS PUB 197, http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf)

```
#define AES_ENCRYPT 1
#define AES_DECRYPT 0

int aes_crypt_cfb128(aes_context *ctx, int mode,
                     size_t len, size_t *iv_off, char *iv,
                     const char *input, char *output);
/*@ requires
      mode == AES_ENCRYPT || mode == AES_DECRYPT &*&
      aes_context_initialized(ctx, ?p1, ?c1) &*&
      crypto_chars(?iv_kind, iv, 16, ?iv_cs) &*&
      u_integer(iv_off, 0) &*& chars(output, len, _) &*&
      mode == AES_ENCRYPT ?
      (
        random_permission(?p2, ?c2) &*&
        iv_cs == chars_for_cg(cg_nonce(p2, c2)) &*&
        [?f]crypto_chars(?kind, input, len, ?in_cs) &*&
          len >= MIN_SIZE &*&
        ensures
        (
          aes_context_initialized(ctx, p1, c1) &*&
          random_permission(p2, c2 + 1) &*&
          [f]crypto_chars(kind, input, len, in_cs) &*&
          crypto_chars(secret, iv, 16, _) &*&
          u_integer(iv_off, _) &*&
          result != 0 ?
            chars(output, len, _)
          :
            cryptogram(output, len, _, ?cg) &*&
            cg == cg_encrypted(p1, c1, in_cs, iv_cs)
        )
      ) : (
        decryption_pre(true, ?gar_in, ?p2, ?s, ?in_cs) &*&
        [?f]cryptogram(input, len, in_cs, ?cg) &*&
          cg == cg_encrypted(?p3, ?c3,
                             ?out_cs3, ?iv_cs3) &*&
        ensures
        (
          aes_context_initialized(ctx, p1, c1) &*&
          [f]cryptogram(input, len, in_cs, cg) &*&
          u_integer(iv_off, _) &*&
          crypto_chars(?kind, output, len, ?out_cs) &*&
          crypto_chars(secret, iv, 16, _) &*&
          decryption_post(true, ?gar_out,
                          p2, s, p1, c1, out_cs) &*&
          gar_out == (gar_in || p1 != p3 || c1 != c3 ||
                      iv_cs != iv_cs3) &*&
          result != 0 || gar_out ?
            kind == normal
          :
            kind == secret && out_cs == out_cs3
        )
      ); @*/
//@ ensures true;
```

Listing 13. A cryptographic primitive for encryption and decryption

vector (IV) with offset zero[7] and an output buffer of size `length`. The remainder of the requirements in the precondition and the entire postcondition depends on the `mode`.

If `aes_crypt_cfb128` is invoked for encryption, the first thing that the precondition further requires is the permission to generate random values and this has to do with the 16-byte IV. Since it is important to randomly generate a fresh IV for each encryption, the contract enforces this using that permission. More specifically the assertion `iv_cs == chars_for_cg(cg_nonce(p2, c2))` forces the buffer `iv` to contain a freshly generated nonce and the increment of `c2` in the postcondition prevents the same nonce from being

[7] We chose to annotate the stream cipher primitive `aes_crypt_cfb128` as if it was a primitive for encrypting a single message of any size, without allowing the updated initialization vector to be used for a subsequent encryption. The advantage in doing so, instead of annotating e.g. the cipher block chaining primitive `aes_crypt_cbc`, is that the contracts do not have to deal with the complexity of padding (although this is perfectly possible).

```
/*@
lemma void crypto_chars_split(char *array, int i);
  requires [?f]crypto_chars(?k, array, ?n, ?cs) &*&
           0 <= i &*& i <= n;
  ensures  [f]crypto_chars(k, array, i, ?cs1) &*&
           [f]crypto_chars(k, array + i, n - i, ?cs2) &*&
           cs1 == take(i, cs) &*& cs2 == drop(i, cs) &*&
           cs == append(cs1, cs2);

lemma void crypto_chars_join(char *array);
  requires [?f]crypto_chars(?k, array, ?n, ?cs1) &*&
           [f]crypto_chars(k, array + n, ?n0, ?cs2);
  ensures  [f]crypto_chars(k, array, n + n0, ?cs) &*&
           cs == append(cs1, cs2);
@*/
```

Listing 14. Splitting and joining possibly confidential memory regions

reused. The remaining part of the precondition describes the required input buffer. The postcondition for encryption returns all the updated permissions and links the output buffer to the correct `cg_encrypted` cryptogram. Note that the buffer for the IV has become a `secret crypto_chars` chunk. Indeed, as its updated contents is correlated with the key and input to `aes_crypt_cfb128`, it could contain secret data[8].

The part of the contract of `aes_crypt_cfb128` for decryption is very analogous. Ignoring the `decryption_pre` and `decryption_post` for now and assuming the ghost variable `gar_in` is `true`, the contract for decryption encodes the following property: "if decryption was successful and if the correct IV and key were provided and if no cryptographic collision occurred, then the output buffer contains the payload of the presented encrypted message". There is however one important issue that arises for unauthenticated encryption and that is how to handle successful decryption with the wrong key or IV. The mechanism we devised for this is discussed in the next section and it uses the predicates `decryption_pre` and `decryption_post` which are defined there.

E. Reasoning about buffers that contain secrets

As already explained, the result of a cryptographic primitive is written in a buffer described by a `crypto_chars` chunk. Before we explain how to prove that the conversion from a `secret crypto_chars` to a `chars` chunk is allowed, we show how to combine possibly secret memory regions into a single memory region described by a `crypto_chars` chunk, which is necessary to construct a payload for encryption.

a) Composing possibly secret memory regions: As explained in the VeriFast Tutorial [8], two separately tracked adjacent character buffers can be merged together with a lemma called `chars_join` and any tracked character buffer can be split into two with the lemma `chars_split`. Listing 14 shows the `crypto_chars` counterparts of these lemmas.

A first step in creating a single memory region from different arbitrary parts then, is to create `crypto_chars` chunks for the desired memory regions with the lemma `crypto_chars_split`. The next step is to copy these parts to adjacent memory regions before they can finally be

[8] This is actually a crude measure. If both the key and input are not secret, the resulting encrypted message and IV are also not secret. The complete encoding of our extended symbolic model takes this fact into account.

```
void memcpy(void *dst, void *src, size_t count);
/*@ requires chars(dst, count, ?cs) &*&
             [?f]crypto_chars(?kind, src, count, ?cs0); @*/
/*@ ensures  crypto_chars(kind, dst, count, cs0) &*&
             [f]crypto_chars(kind, src, count, cs0); @*/
```

Listing 15. A contract for `memcpy`

```
/*@
fixpoint list<cryptogram> cgs_in_chars(list<char> cs);

lemma_auto void cg_constraints(cryptogram cg);
  requires true;
  ensures  cons(cg, nil) == cgs_in_chars(chars_for_cg(cg));
@*/
```

Listing 16. Exact cryptographic information in a character buffer

merged with the lemma `crypto_chars_join`. In many C programs, and also here, this copying is done with the function `memcpy` from the C standard library. Listing 15 shows a contract for `memcpy` that can handle `crypto_chars` chunks.

b) Tracking cryptographic content: To prove that a composed buffer described by a `crypto_chars` chunk does not contain any secrets, we need a way to track the cryptographic content of that buffer while composing it. We already introduced the pure function `chars_for_cg` to express the cryptographic content of a buffer in which some primitive wrote its result. Suppose now we have the situation depicted in Figure 3. To describe the cryptographic content of a buffer composed of bits and pieces of buffers in one-to-one correspondence with a cryptogram, the function `chars_for_cg` seems insufficient. Therefore we introduce the function `cgs_in_chars` from Listing 16, which identifies for each list of characters the minimal set of cryptograms of which cryptographic information is present in that list. Although the situation depicted in Figure 3 is somewhat contrived, a dishonest principal can decide to leak pieces of a secret in this way. Moreover, during the parsing of messages it is not known which cryptographic information is present, and the function `cgs_in_chars` allows to express relationships between different parts of a message. Also defined in Listing 16, lemma `cg_constraints` expresses the property that the cryptographic information in the character representation of a cryptogram is simply that cryptogram itself.

Now suppose we have the situation depicted in Figure 4. There, the concatenation of two buffers each containing a key is split somewhere at random. We cannot use the function `cgs_in_chars` to say that both resulting buffers contain information about both keys. At least one of the two resulting buffers will contain information about just one of the keys. We introduce the function `cgs_in_chars_bound` to deal with this situation. It identifies valid upper bounds

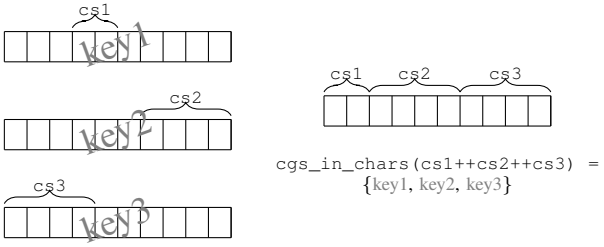

Figure 3. Illustration of the intended meaning of `cgs_in_chars`

```
/*@
fixpoint bool cgs_in_chars_bound(list<char> cs,
                                 list<cryptogram> cgs);

lemma void cgs_in_chars_bound(list<char> cs);
  requires true;
  ensures  true == cgs_in_chars_bound(cs, cgs_in_chars(cs));
@*/
```

Listing 17. Upper bounds on cryptographic information

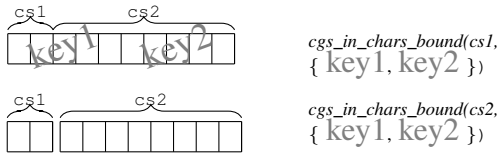

Figure 4. Illustration of the intended meaning of `cgs_in_chars_bound`

on the exact cryptographic information exposed by a list of characters. The only lemma in Listing 17, which is also named `cgs_in_chars_bound`, expresses that the exact cryptographic content of a C buffer is also a valid upper bound. Lemmas similar to the ones in Listing 14, but not shown, allow to track this upper bound while splitting and joining C buffers.

c) Converting `secret crypto_chars` to `chars`: Our main mechanism to prove functional correctness of a protocol implementation is an invariant as in [1], [4] and [6]. Instead of enforcing this invariant on messages on the network, we enforce this invariant on readable memory regions, i.e. memory regions tracked by a `chars` chunk and thus containing no secrets. Since the network API discussed in Subsection II-C only accepts memory regions described by `chars` chunks, our invariant indirectly holds for all the messages on the network. On top of that, our invariant also holds for any memory that can influence the control flow of a protocol implementation. Hence, even through implicit channels, only memory regions for which the invariant holds are visible to the attacker.

For each verified protocol implementation a custom invariant is required that specifies what cryptographic information is public or, equivalently, non-secret. In our approach cryptograms are the symbolic representation of cryptographic information, so an invariant has to be defined in terms of which cryptograms are public. Listing 18 shows a trivial, but not very useful, invariant defined as a the predicate `example_pub`.

When initializing our cryptographic library the custom defined invariant must be provided via the lemma `public_invariant_init` from Listing 19. An invocation of this lemma results in a `public_invar` chunk. All functions implementing a protocol role should receive a fraction of this chunk as a necessary permission to call the lemma `public_crypto_chars`, also from Listing 19, that transforms all `crypto_chars` to `chars` chunks. However, the lemma `public_crypto_chars` also requires a proof that the content of the provided buffer does not contain any secret cryptographic information. This proof takes the form of a chunk of the predicate `public_generated(pub)` which is an instance of the predicate constructor `public_generated` for the invariant `pub` (see [8] for an explanation of predicate constructors). To create such a chunk finally, one must prove that each cryptogram of which information is present in the buffer, was generated by a primitive and satisfies the invariant.

```
/*@
predicate example_pub(cryptogram cg) = switch (cg) {
  case cg_nonce(p0, c0): return true /*...*/;
  case cg_symmetric_key(p0, c0): return true /*...*/;
  case cg_encrypted(p0, c0, pay0, iv0): return true /*...*/;
};
@*/
```

Listing 18. Example of an invariant definition

```
/*@
require_module public_invariant_mod;

predicate public_invar(predicate(cryptogram) pub);

lemma void public_invariant_init(predicate(cryptogram) pub);
  requires module(public_invariant_mod, true);
  ensures  [_]public_invar(pub);

predicate_ctor public_generated(predicate(cryptogram) pub)
                               (list<char> cs) =
  [_]dummy_foreach(cgs_in_chars(cs), pub) &*&
  true == forall(cgs_in_chars(cs), cg_is_generated);

lemma void public_crypto_chars(char *array, int n);
  requires [_]public_invar(?pub) &*&
           [?f]crypto_chars(_, array, n, ?cs) &*&
           [_]public_generated(pub)(cs);
  ensures  [f]chars(array, n, cs);
@*/
```

Listing 19. Applying a protocol-specific invariant

d) Comparing secret memory regions: Memory regions that are described by a `crypto_chars` chunk cannot be read. However, some protocols need to compare possibly secret memory regions. Consider for example a protocol where a secret nonce is generated for freshness. One participant generates this value and sends it in an encrypted form to another principal. In some later stage of the protocol the first principal receives an encrypted message and needs to check that it contains the original value as part of its payload. To do this he needs to compare possibly secret memory regions.

Our approach allows to compare possibly secret memory regions via the standard library function `memcmp`. The contract for this function is shown in Listing 20. It encodes the trivial semantics of the result being equal to zero if and only if the two input buffers have the same content. If however, the input buffers were not equal and one of the input buffers was a `secret`, then the network permission that the precondition requires is revoked. This prevents badly implemented protocol implementations from leaking secrets to the network by guessing their own secrets through the function `memcmp` and making them readable.

The predicate `memcmp_secret` finally, is used to ensure that the probability of a badly implemented protocol leaking a secret via `memcmp`, is exponentially small in the size of that secret. For each of the provided buffers described by a secret `crypto_chars` chunk, the contract of `memcmp` requires a chunk of the predicate `memcmp_secret`. Such a chunk can only be produced if the corresponding buffer exactly contains a cryptogram that was generated with one of the annotated

```
/*@
predicate memcmp_secret(char* buffer, int i,
                        list<char> cs, cryptogram cg) =
  i == length(cs) && cs == chars_for_cg(cg) &&
  cg_is_generated(cg)
;
@*/

int memcmp(char *array, char *array0, size_t i);
/*@ requires network_permission(?principal) &*&
             [?f1]crypto_chars(?k1, array, ?n1, ?cs1) &*&
               (k1 == normal ? true :
                memcmp_secret(array, i, cs1, _)) &*&
             [?f2]crypto_chars(?k2, array0, ?n2, ?cs2) &*&
               (k2 == normal ? true :
                memcmp_secret(array0, i, cs2, _)) &*&
             i <= n1 &*& i <= n2; @*/
/*@ ensures  [f1]crypto_chars(k1, array, n1, cs1) &*&
             [f2]crypto_chars(k2, array0, n2, cs2) &*&
             true == ((result == 0) ==
                      (take(i, cs1) == take(i, cs2))) &*&
             result != 0 &&  (k1 == secret ||
                              k2 == secret) ?
               true : network_permission(principal); @*/
```

Listing 20. Contract for `memcmp`

```
void zeroize(char *buffer, int size);
//@ requires crypto_chars(_, buffer, size, _);
//@ ensures  chars(buffer, size, _);
```

Listing 21. A function to clear secrets from memory

primitives and if that cryptogram is compared in its entirety.

e) Clearing secret memory regions: VeriFast checks that at the end of each function you have freed all allocated memory (or have passed the ownership to some called function) and that at the end of each block, all memory regions allocated on the stack are present in the symbolic heap before they are deallocated. A buffer that contains a secret is described by a `crypto_chars` chunk, but to release allocated memory (both in the heap and on the stack), it needs to be converted to a `chars` chunk. The function `zeroize` from Listing 21 allows a protocol implementation to erase its generated secrets from memory once the protocol is finished. The fact that this is necessary is reasonable and it can be considered good practice to clear all secrets from memory after protocol completion.

F. The attacker model

Following other symbolic models, our attacker has complete access to the untrusted network. He can grab any message from the network and put any message on there that he can produce using the same cryptographic API as the honest principals. To allow for the attacker to send anything he can produce with our cryptographic API, he must have the following capabilities:

- Send a part of a message he finds on the network
- Send the concatenation of any two messages he finds
- Leak his own generated keys and nonces
- Encrypt or decrypt a message from the network with a key he finds on the network and send the result

We encoded these capabilities as lemma function types (for more information see [8]). Listing 22 illustrates such a capability, more precisely the capability of the attacker to leak his own keys. The contract expresses that if a principal is `bad` (which indicates it is an identity of the attacker), the invariant should hold for the cryptograms representing his keys. This allows

```
/*@
typedef lemma void bad_key(predicate(cryptogram) pub,
                           predicate() ppred)
                          (cryptogram key);
  requires  ppred() &*& key == cg_symmetric_key(?p, _) &*&
            true == bad(p);
  ensures   ppred() &*& [_]pub(key);
@*/
```

Listing 22. Example of an attacker capabitility

```
/*@
predicate invariant_constraints(predicate(cryptogram) pub,
                                 predicate() pred) =
  is_bad_key_is_public(_, pub, pred) /* &*& ... */;
@*/

void attacker();
/*@ requires [_]public_invar(?pub) &*&
             invariant_constraints(pub, ?ppred) &*&
             ppred() &*& principals(?c1); @*/
/*@ ensures  invariant_constraints(pub, ppred) &*&
             ppred() &*& principals(?c2) &*& c2 > c1; @*/
```

Listing 23. The attacker implementation as a C function

the attacker implementation to convert his own generated keys from `crypto_chars` chunks to `chars` and send them on the network. The custom predicate `ppred` can be used to give some extra protocol-specific facts to prove the lemma.

To ensure that a verified protocol is capable of withstanding any attack from the attacker (which is the ultimate goal of the entire approach), the invariant must be closed under attacker actions, i.e. the attacker must have all the capabilities previously mentioned. This can be shown by writing an implementation for each lemma function type that represents an attacker capability and producing a chunk of the predicate `invariant_constraints` from Listing 23. With such a chunk the attacker implementation can safely be run in parallel with the verified protocol implementation.

III. Decryption with the Wrong Key or IV

At the end of Subsection II-D, we mentioned the issue of specifying the semantics of unauthenticated decryption with the wrong key or initialization vector (IV). The explanation of the predicates `decryption_pre` and `decryption_post` used in the contract of `aes_crypt_cfb128` in Listing 13 that deal with this situation was postponed till here. Before we precisely define these predicates, it is instructive to know what their high-level purpose is. Together with some auxiliary definitions, they conspire to encode the following observation:

Successful unauthenticated decryption does not assure that the provided key and IV are correct. If however, during some protocol run, one expects the payload to have a specific structure after decryption[9] *and it turns out that this expectation is fulfilled, then the encrypted message must have been created with the provided key and IV or a cryptographic collision occurred.*

[9] For example, the expected structure of a payload could be that it starts with some known value.

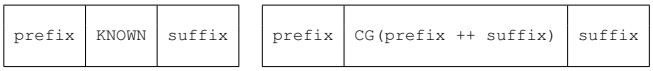

Figure 5. Illustration of possible structures of a payload

```
/*@
inductive structure =
  | known_value(int offset, list<char> known)
  | cryptogram_with_payload(int offset, int length);

predicate has_structure(list<char> cs, structure s) =
  exists(pair(?pref, ?suff)) &*& switch(s){
    case known_value(offset, known): return
      cs == append(pref, append(known, suff)) &*&
      length(known) >= MIN_SIZE &*& length(pref) == offset;
    case cryptogram_with_payload(offset, length): return
      exists(?cg) &*& cg_payload(cg) == some(?pay) &*&
      cs == append(pref, append(chars_for_cg(cg),suff)) &*&
      length >= MIN_SIZE &*& length(pay) >= MIN_SIZE &*&
      length == length(chars_for_cg(cg)) &*&
      length(pref) == offset &*& pay == append(pref, suff);
  };
@*/
```

Listing 24. Describing the structure of some payload

This observation makes sense as for any secure cipher it should be very hard to construct (without using encryption) a ciphertext that decrypts to a payload with a specific structure.

A. Expressing the expected structure of an encrypted payload

While other interpretations for the concept of structure are possible, we chose the one illustrated in Figure 5. Here, two messages are depicted, each with a different kind of structure:

1) The message contains a known value: `KNOWN`.
2) Some part of the message is the character representation of a cryptogram with as payload the concatenation of the rest of the message (i.e. `CG(prefix ++ suffix)`).

A protocol participant that is about to perform unauthenticated decryption can express both these kinds of expectations about the resulting payload using the inductive datatype `structure` from Listing 24. To actually prove that a specific payload has some structure after decryption, the predicate `has_structure` can be used. This predicate precisely encodes the requirements for a character list `cs` to have the structure `s`. Some minimal lengths are required to ensure that guessing a structure is infeasible. If one is able to close a chunk of the predicate `has_structure` with arguments `cs` and `s`, then one knows that the list `cs` has the structure `s`.

B. Decrypted payloads that have the expected structure

If a protocol participant sees that a decrypted payload fulfills his expectations, then it must have been encrypted with the same key and IV or a cryptographic collision occurred. To encode this observation we introduce the definitions from Listing 25. A chunk of the predicate `garbage` should be returned in the postcondition of any unauthenticated decryption function (and thus in the body of the predicate `decryption_post`, see further). As its name suggests, it should only be returned in the symbolic execution branch of the postcondition where the wrong key or IV was provided. After decryption and once proven that the involved payload has the expected structure (which can only be proven if the

```
/*@
predicate garbage(bool sym, int principal, structure s,
                  int p_k, int c_k, list<char> cs_out);

lemma void no_garbage(char *b, int n, structure s);
  requires garbage(?sym, ?p, s, ?p_k, ?c_k, ?cs) &*&
           col ? true : [_]has_structure(cs, s);
  ensures  decryption_permission(p) &*& true == col;
@*/
```

Listing 25. Dealing with a decrypted payload having the expected structure

```
/*@
predicate decryption_pre(bool sym, bool gar, int p,
                         structure s, list<char> cs_in) =
  !gar ? decryption_permission(p) :
         garbage(sym, p, s, _, _, ?cs_out) &*&
         exists(pair(?prefix, ?suffix)) &*&
         cs_out == append(prefix, append(cs_in, suffix));

predicate decryption_post(bool sym, bool gar, int p,
                          structure s, int p_key, int c_key,
                          list<char> cs_out) =
  !gar ? decryption_permission(p) :
         garbage(sym, p, s, p_key, c_key, cs_out);
@*/
```

Listing 26. Predicates `decryption_pre` and `decryption_post`

C code actually checks this after decryption), the lemma `no_garbage` allows to prove that a cryptographic collision indeed occurred in that symbolic execution branch[10].

C. Decryption permission

As discussed in Subsection II-B, a principal identity contains amongst others the permission to perform unauthenticated decryption. Attentive readers already saw this permission popping up in the postcondition of `garbage` from Listing 25. This is because after a decryption with the wrong key or IV, the permission should be revoked until one proves that the decrypted payload has the expected structure. So the precondition of any unauthenticated decryption function should require this permission (and thus it should be present in the body of `decryption_pre`). The reason for temporarily revoking this permission is to prevent a principal from decrypting a message twice, in which case he can use the result of the first decryption to formulate an expectation for the second.

D. Bodies of `decryption_pre` and `decryption_post`

Listing 26 shows the definitions of `decryption_pre` and `decryption_post`. The first parameter of both predicates indicates whether the predicates are used for symmetric or asymmetric decryption. An initial decryption with `aes_crypt_cfb128` must be done with a chunk of the predicate `decryption_pre` where the `gar` argument is `false`. Its body then specifies that a decryption permission is required. As one can see in the definition of `decryption_post`, this permissions is simply returned in the postcondition of `aes_crypt_cfb128` if the input was not garbage and the provided key and IV were correct.

If the input is garbage, or if the provided key or IV was not correct, a chunk of the predicate `garbage` is returned. After checking that the decrypted payload has the expected structure,

[10]We omitted the concept of key classifiers here. It is discussed in [12].

one can invoke the lemma `garbage` from Listing 25 to retrieve the decryption permission and to prove that a collision has occurred. The second part of `decryption_pre` is only relevant for protocol implementations that use some form of nested encryption. This feature is further discussed in [12].

IV. Verified Protocol Example: Yahalom

In this section we briefly discuss an exemplary protocol implementation verified with our extended symbolic model in VeriFast. Other verified protocols that illustrate the full capabilities of our approach can be found in the latest VeriFast release and a summary of that verified protocol suite is given in Section V. As exemplary verified implementation, we discuss the Yahalom protocol [2] which establishes authentication and a shared key between two parties using a trusted arbitrator.

In Section II we narrowed the scope of our discussion to symmetric encryption and the generation of random keys and nonces. Since these are exactly the primitives that the Yahalom protocol requires, it is a suitable protocol to discuss here. All annotated source code of the verified protocol is listed in the appendix and the protocol implementation there was verified in the complete version of our extended symbolic model.

A protocol transcript of the Yahalom protocol is shown in Figure 6. In this transcript A and B are the two parties that want to establish a new shared key K_{AB}. S is the trusted arbitrator with whom A already shares the key K_A and with whom B already shares the key K_B. After the trusted arbitrator S has processed the second message, S is sure of non-injective agreement [9] on N_A and N_B with A, and on the identity of A. The processing of the third message guarantees A injective agreement [9] on K_{AB}, N_A and N_B. After B has processed the final message, A and B are in mutual agreement on K_{AB}, N_A and N_B. So in total, a series of four messages are exchanged after which only A, B and S know the freshly generated secret key K_{AB}. Otherwise, at least one participant must have leaked a secret nonce or key.

Looking at the protocol transcript, one can see that the nonce N_A must be public, while the nonce N_B is never sent unencrypted. To distinguish between these nonces, to record protocol progress, and to distinguish the keys K_A and K_B from K_{AB}, we use the pure function `cg_info` from Listing 11 to associate different information with all these cryptograms. Think of the information associated with a random generated value as the knowledge that is available at the point of its creation during protocol execution. The encoding in Figure 7 uses the definition of `int_pair` from Listing 27 to facilitate this step. Due to space constraints we do not go into further detail here about how to choose an encoding for associated information. We refer interested readers to [12].

1. $A \rightarrow B : A, N_A$
2. $B \rightarrow S : \{A, N_A, N_B\}_{K_B}$
3. $S \rightarrow A : \{B, K_{AB}, N_A, N_B\}_{K_A}, \{A, K_{AB}\}_{K_B}$
4. $A \rightarrow B : \{A, K_{AB}\}_{K_B}, \{N_B\}_{K_{AB}}$

Figure 6. Protocol transcript for the Yahalom protocol

```
/*@
fixpoint int int_left(int p);
fixpoint int int_right(int p);
fixpoint int int_pair(int f, int s);

lemma void int_left_int_pair(int f, int s);
  requires true;
  ensures  int_left(int_pair(f, s)) == f;

lemma void int_right_int_pair(int f, int s);
  requires true;
  ensures  int_right(int_pair(f, s)) == s;
@*/
```

Listing 27. Flexible encoding of associated information of cryptograms

```
/*@
fixpoint bool yahalom_public_nonce(int p0, int info)
{
  return bad(p0) ||
         int_left(info) == 1 ||
         int_left(info) == 2 &&
          (bad(int_left(int_right(info))) ||
           bad(int_right(int_right(info))));
}

fixpoint bool yahalom_public_key(int p0, int info)
{
  return bad(p0) ||
         int_left(info) == 3 &&
           bad(int_right(info)) ||
         int_left(info) == 4 &&
          (bad(int_left(int_right(info))) ||
           bad(int_right(int_right(info))));
}
@*/
```

Listing 28. Fixing an encoding of information for the Yahalom protocol

Once the encoding of associated information is in place, we can specify when nonces and keys are public in terms of their creator and this information using the definitions from Listing 27. This is exactly what the function `yahalom_public_nonce` and `yahalom_public_key` are defined for in Listing 28. A nonce, for example, is public if: the creator is `bad`, it is the nonce N_A, or it is the nonce N_B and one the principals that has access to it during the protocol execution is `bad`. The protocol-specific invariant can then be defined using these auxiliary functions and they also come in handy when specifying the protocol goals in the contracts of the C functions that implement a protocol role. Listing 29 shows a contract for the C function `sender` which implements the protocol role A. As one can see in its postcondition, the generated key received by A is indeed not public if none of the protocol participants is `bad`.

After completing the steps discusses, nearly everything is in place to start verifying the protocol code. The last hurdle is defining a protocol-specific invariant by completing the

N_A	`int_pair(1, B)`
N_B	`int_pair(2, int_pair(S, int_pair(A, int_pair(NA_p, NA_c))))`
K_A	`int_pair(3, S)`
K_B	`int_pair(3, S)`
K_{AB}	`int_pair(4, int_pair(A, int_pair(B, int_pair(NA_p, int_pair(NA_c, int_pair(NB_p, NB_c))))))`

Figure 7. Information associated with cryptograms

```
void sender(int server, int sender, int receiver,
            char *key, char *generated_key);
/*@ requires
      [_]public_invar(yahalom_pub) &*&
      principal(sender, ?s_id) &*&
      [?f]cryptogram(key, KEY_SIZE, ?key_cs, ?key_cg) &*&
      key_cg == cg_symmetric_key(sender, _) &*&
      cg_info(key_cg) == int_pair(3, server) &*&
      chars(generated_key, KEY_SIZE, _); @*/
/*@ ensures
      principal(sender, _) &*&
      [f]cryptogram(key, KEY_SIZE, key_cs, key_cg) &*&
      cryptogram(generated_key, KEY_SIZE, _, ?g_key_cg) &*&
      col || bad(server) || bad(sender) || bad(receiver) ?
        true
      :
        g_key_cg == cg_symmetric_key(server, ?id) &*&
        !yahalom_public_key(server, cg_info(g_key_cg)); @*/
```

Listing 29. Contract for the initiator of the Yahalom protocol

```
/*@
predicate yahalom_pub(cryptogram cg) =
  switch (cg)
  {
    case cg_nonce(p0, c0):
      return true == yahalom_public_nonce(p0, cg_info(cg));
    case cg_symmetric_key(p0, c0):
      return true == yahalom_public_key(p0, cg_info(cg));
    case cg_encrypted(p0, c0, cs0, ent0):
      return yahalom_public_key(p0, c0) ?
        [_]public_generated(yahalom_pub)(cs0)
      : true /* ... */;
  };
@*/
```

Listing 30. Extract from the invariant for the verified Yahalom protocol

skeleton from Listing 30 for when an encrypted message is public. For authentication, the invariant must also record protocol progress using events and authentication itself is expressed through event correspondences. A complete and working invariant for the Yahalom implementation is given in appendix. The only thing left to do then, is interactively verifying all protocol code with VeriFast.

V. Results

Using the approach described in this paper we were able to verify a significant number of cryptographic protocol implementations. For now we only verified custom-written protocol implementations and we leave tackling preexisting implementations for future work. Figure 8 shows some statistics about the protocol implementations we verified. The annotation-to-source-code ratios indicate a high annotation effort and this is as expected since we are dealing with intrinsically difficult problems. However, the complexity of the annotations to verify protocol implementations is relatively low (this is a subjective assessment and difficult to quantify). The motivation for this judgment is that most of the complexity is contained in the cryptographic API. Although we have not applied our method to preexisting protocol implementations at the moment, these initial verification efforts give some hope that the approach described in this text is suited to verify functional correctness of preexisting implementations.

We also implemented a classical symbolic API on top of the extended symbolic API. This API is very similar to the one in [1] or [6] and applies the message concept from Listing 5 instead of the concept of cryptograms used

Protocol	SLOC	ALOC	Ratio	VTime
dummy	125	90	0.72	1.46
hmac	246	132	0.54	1.43
rpc	416	175	0.42	1.43
enc_and_hmac	371	177	0.48	1.54
enc_then_hmac	398	176	0.44	1.58
hmac_then_enc	401	187	0.47	1.56
hmac_then_enc_tagged	398	189	0.47	1.61
hmac_then_enc_nested	508	229	0.45	2.10
auth_enc	282	157	0.56	1.45
sign	330	181	0.55	1.49
nsl	1010	308	0.30	1.80
yahalom	1109	386	0.35	34.82

ALOC = Annotation Lines of Code
SLOC = Source Lines of Code
Ratio = ALOC/SLOC
VTime = Verification time (in seconds)

Figure 8. Results on protocols verified with the described approach

Protocol	SLOC	ALOC	Ratio	VTime
high-level API	4631	1590	2.91	43.63
dummy_protocol	127	94	1.35	1.49
secure_storage	214	123	1.74	1.46
secure_storage_asym	215	121	1.78	1.54
rpc	353	189	1.87	1.80
recursive_otway_rees	663	423	1.57	1.86

Figure 9. Results on high-level API and protocols

throughout this text. It provides C function for composing and decomposing the in-memory representations of messages and the actual C buffers that contain the character representation of these messages are hidden from the clients of the API. We implemented this classical API to get an estimate of the consistency and usability of our extended symbolic API. Some protocol implementations were also written on top of this classical symbolic API and the results of this effort are shown in Figure 9. We do not further discuss the classical API here. Its fully verified implementation is described in [11] and can be found in the latest VeriFast release.

VI. Related Work

C is a low-level programming language that provides few abstractions from the underlying machine. This makes proving properties of C code hard and verifying protocol implementations in C is definitely a challenging task. Still, tools like ASPIER [3] and Csur [7] take on the challenge of automatically verifying protocols written in C. While they show impressive results, they have their limitations: Csur does not allow to prove memory safety and ASPIER only supports a bounded number of simultaneous protocol runs as stated in [10].

Other invariant-based methods are more closely related to the approach described in this text. The TAPS verifier [4] allows to specify protocols in an abstract specification language. The tool then automatically tries to extract an invariant from that specification in order to prove the stated security goals. In [1] protocols are written in the F# programming language. Invariants and security goals are stated with refinement types. The F7 type checker is used to prove that the implementation indeed satisfies the properties encoded in the types. The approach in [6], finally, works for C implementations. It is very

close to our approach and uses VCC [5] instead of VeriFast. However, the structured message concept makes it unsuitable for preexisting cryptographic APIs.

VII. Conclusions

In this paper we discussed our extended symbolic model of cryptography. The approach presented is a further development of the method described in [11] and is explained in full in [12]. As in [1] and [6] we verify cryptographic protocol implementations, but we target preexisting implementations written in the C programming language. While our approach was developed with this goal in mind, we only verified protocol implementations we wrote ourselves against a preexisting cryptographic API. Applying our approach to complete preexisting implementations is left for future work.

Before discussing our approach, we motivated why the traditional symbolic model is insufficient when verifying preexisting cryptographic protocol implementations. Instead, we propose the extended symbolic model of cryptography for verifying preexisting implementations. After a detailed discussion on the definition of this model in VeriFast, we showed some extracts of verified protocols. Then we presented an overview of successful verification efforts within our extended symbolic model. These efforts seem to indicate that our approach is well-suited for the verification of preexisting cryptographic protocol implementations. Currently, a formalization of the entire approach is being developed to prove the soundness of the extended symbolic model of cryptography.

Acknowledgements

The research leading to these results has received funding from the European Union Seventh Framework Programme [FP7/2007-2013] under grant agreement n317753, and more precisely from the EU FP7 project STANCE (a Source code analysis Toolbox for software security AssuraNCE).

This research is also partially funded by project G.0058.13 of the Flemish Research Fund, by the Research Fund KU Leuven, and by the EU FP7 project NESSoS. With the financial support from the Prevention of and Fight against Crime Programme of the European Union (B-CCENTRE).

References

[1] K. Bhargavan, C. Fournet, and A. D. Gordon. Modular verification of security protocol code by typing. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 445–456, New York, NY, USA, 2010. ACM.

[2] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *ACM TRANSACTIONS ON COMPUTER SYSTEMS*, 8:18–36, 1990.

[3] S. Chaki and A. Datta. Aspier: An automated framework for verifying security protocol implementations. In *CSF*, pages 172–185. IEEE Computer Society, 2009.

[4] E. Cohen. TAPS: A first-order verifier for cryptographic protocols. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop, CSFW '00, Cambridge, England, UK, July 3-5, 2000*, pages 144–158, 2000.

[5] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. Vcc: A practical system for verifying concurrent c. In *Proceedings of the 22Nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, pages 23–42, Berlin, Heidelberg, 2009. Springer-Verlag.

[6] F. Dupressoir, A. D. Gordon, J. Jurjens, and D. A. Naumann. Guiding a general-purpose C verifier to prove cryptographic protocols. In *Proceedings of the 2011 IEEE 24th Computer Security Foundations Symposium*, CSF '11, pages 3–17, Washington, DC, USA, 2011. IEEE Computer Society.

[7] J. Goubault-larrecq and F. Parrennes. Cryptographic protocol analysis on real c code. In *In 6th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI05), volume 3385 of LNCS*, pages 363–379. Springer, 2005.

[8] B. Jacobs, J. Smans, and F. Piessens. The VeriFast program verifier: a tutorial. 2014.

[9] G. Lowe. A hierarchy of authentication specification. In *10th Computer Security Foundations Workshop (CSFW '97), June 10-12, 1997, Rockport, Massachusetts, USA*, pages 31–44. IEEE Computer Society, 1997.

[10] M. Moskal N. Polikarpova. *Verified Software: Theories, Tools, Experiments: 4th International Conference, VSTTE 2012, Philadelphia, PA, USA, January 28-29, 2012. Proceedings*, chapter Verifying Implementations of Security Protocols by Refinement, pages 50–65. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[11] G. Vanspauwen and B. Jacobs. Verifying protocol implementations by augmenting existing cryptographic libraries with specifications. In *Software Engineering and Formal Methods, 13th International Conference, SEFM 2015, York, UK, September 7-11, 2015. Proceedings*, pages 53–68. Springer Berlin Heidelberg, September 2015.

[12] G. Vanspauwen and B. Jacobs. Verifying cryptographic protocols: Applying the symbolic model to cryptographic APIs for C. CW Reports CW694, Department of Computer Science, KU Leuven, March 2016.

APPENDIX

YAHALOM PROTOCOL IMPLEMENTATION

This appendix lists all the source files of the verified implementation of the Yahalom protocol [2].

A. *Header file of the verified Yahalom protocol implementation*

```
#ifndef YAHALOM_H
#define YAHALOM_H

#include "../../annotated_api/polarssl_definitions/polarssl_definitions.h"

#define KEY_SIZE 32
#define NONCE_SIZE 10

/*@

// 1. A -> B. A, NA
// 2. B -> S. B, ENC(KB, {A, NA, NB})
// 3. S -> A. ENC(KA, {B, KAB, NA, NB}), ENC(KB, {A, KAB})
// 4. A -> B. ENC(KB, {A, KAB}), ENC(KAB, NB)

// 1. sender -> receiver : NAME(sender), NONCE(sender)
// 2. receiver -> server : NAME(receiver),
//                         ENC(K(receiver),
//
{NAME(sender), NONCE(sender), NONCE(receiver)}
//                         )
// 3. server -> sender   : ENC(K(sender),
//
{NAME(receiver), KEY(server),
//
NONCE(sender), NONCE(receiver)}
//                         )
//                         ENC(K(receiver),
//
{NAME(sender), KEY(server)}
//                          )
// 4. sender -> receiver : ENC(K(receiver),
//
{NAME(sender), KEY(server)}
//                         )
//
ENC(K(server), NONCE(receiver))

////////////////////////////////////////////////////////////////////////////
// Encodings for this protocol /////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////

#define IP(F, S) int_pair(F, S)
#define IF(I) int_left(I)
#define IS(I) int_right(I)

// info:
//  IP(0, _):
no information
//  IP(1, receiver):
sender nonce
//  IP(2, IP(server, IP(sender, IP(s, s_id)))):
receiver nonce
//  IP(3, server):
server key
//  IP(4, IP(sender, IP(receiver
//        IP(s, IP(s_id, IP(r, r_id))))))):
generated key

////////////////////////////////////////////////////////////////////////////
```

```
// Definition of pub for this protocol ////////////////////////////////////
////////////////////////////////////////////////////////////////////////////

fixpoint bool yahalom_public_nonce(int principal, int info)
{
  return
    bad(principal) ||
    IF(info) == 0 ||
    IF(info) == 1 ||
   (IF(info) == 2 &&
      (bad(IF(IS(info))) ||
       bad(IF(IS(IS(info))))));
}

fixpoint bool yahalom_public_key_(int principal, int info)
{
  return
    bad(principal) ||
   (IF(info) == 3 && bad(IS(info))) ||
   (IF(info) == 4 &&
     (bad(IF(IS(info))) || bad(IF(IS(IS(info))))));
}

fixpoint bool yahalom_public_key(int principal, int count, bool sym)
{
  return sym ?
           yahalom_public_key_(principal,
                               cg_info(cg_symmetric_key(principal, count)))
         :
           bad(principal);
}

predicate yahalom_proof_pred() = true;

predicate yahalom_pub_msg1(int server, int sender,
                           cryptogram NA, cryptogram NB) = true;
predicate yahalom_pub_msg2(int sender, int receiver, cryptogram NA,
                           cryptogram NB, cryptogram KAB) = true;
predicate yahalom_pub_msg3(int server, int sender, cryptogram KAB,
                           int sendr2, int a_id, int receivr2, int b_id) = true;
predicate yahalom_pub_msg4(int server, int sender, int receiver,
                           int a_id, cryptogram NB) = true;

predicate yahalom_pub(cryptogram cg) =
  switch (cg)
  {
    case cg_nonce(p0, c0):
      return true == yahalom_public_nonce(p0, cg_info(cg));
    case cg_symmetric_key(p0, c0):
      return true == yahalom_public_key(p0, c0, true);
    case cg_public_key(p0, c0):
      return true;
    case cg_private_key(p0, c0):
      return true == yahalom_public_key(p0, c0, false);
    case cg_hash(cs0):
      return true;
    case cg_hmac(p0, c0, cs0):
      return true;
    case cg_encrypted(p0, c0, cs0, ent0):
      return yahalom_public_key(p0, c0, true) ?
        [_]public_generated(yahalom_pub)(cs0)
      :
        length(cs0) == ID_SIZE + NONCE_SIZE + NONCE_SIZE ?
        (
          // ENC(KB, {A, NA, NB})
          yahalom_pub_msg1(?server, ?sender, ?NA, ?NB) &*&
          cg_info(cg_symmetric_key(p0, c0)) == IP(3, server) &*&
```

```
  NA == cg_nonce(?sender2, ?a_id) &*&
  true == cg_is_generated(NA) &*& [_]yahalom_pub(NA) &*&
  NB == cg_nonce(p0, _) &*& true == cg_is_generated(NB) &*&
  cg_info(NB) == IP(2, IP(server, IP(sender, IP(sender2, a_id)))) &*&
  length(identifier(sender)) == ID_SIZE &*&
  length(chars_for_cg(NA)) == NONCE_SIZE &*&
  length(chars_for_cg(NB)) == NONCE_SIZE &*&
  cs0 == append(identifier(sender), append(chars_for_cg(NA),
                                           chars_for_cg(NB))) &*&
  [_]public_generated(yahalom_pub)(identifier(sender))
) :
length(cs0) == ID_SIZE + KEY_SIZE + NONCE_SIZE + NONCE_SIZE ?
(
  // ENC(KA, {B, KAB, NA, NB})
  yahalom_pub_msg2(?server, ?receiver, ?NA, ?NB, ?KAB) &*&
  cg_info(cg_symmetric_key(p0, c0)) == IP(3, server) &*&
  NA == cg_nonce(?sender2, ?a_id) &*&
  true == cg_is_generated(NA) &*& [_]yahalom_pub(NA) &*&
  NB == cg_nonce(?receiver2, ?b_id) &*&
  (
  bad(server) || bad(receiver) ?
    [_]yahalom_pub(NB)
  :
    (cg_info(NB) == IP(2, IP(server, IP(p0, IP(sender2, a_id)))) &&
     receiver == receiver2)
  ) &*&
  true == cg_is_generated(NB) &*&
  KAB == cg_symmetric_key(server, _) &*&
  true == cg_is_generated(KAB) &*&
  cg_info(KAB) == IP(4, IP(p0, IP(receiver,
                    IP(sender2, IP(a_id, IP(receiver2, b_id)))))) &*&
  length(identifier(receiver)) == ID_SIZE &*&
  length(chars_for_cg(KAB)) == KEY_SIZE &*&
  length(chars_for_cg(NA)) == NONCE_SIZE &*&
  length(chars_for_cg(NB)) == NONCE_SIZE &*&
  cs0 == append(identifier(receiver), append(chars_for_cg(KAB),
         append(chars_for_cg(NA), chars_for_cg(NB)))) &*&
  [_]public_generated(yahalom_pub)(identifier(receiver))
) :
length(cs0) == ID_SIZE + KEY_SIZE ?
(
  // ENC(KB, {A, KAB})
  yahalom_pub_msg3(?server, ?sender, ?KAB, ?sender2, ?a_id,
                                           ?receiver2, ?b_id) &*&
  cg_info(cg_symmetric_key(p0, c0)) == IP(3, server) &*&
  KAB == cg_symmetric_key(server, _) &*&
  cg_info(KAB) == IP(4, IP(sender, IP(p0,
                    IP(sender2, IP(a_id, IP(receiver2, b_id)))))) &*&
  true == cg_is_generated(KAB) &*&
  length(identifier(sender)) == ID_SIZE &*&
  length(chars_for_cg(KAB)) == KEY_SIZE &*&
  cs0 == append(identifier(sender), chars_for_cg(KAB)) &*&
  [_]public_generated(yahalom_pub)(identifier(sender))
) :
length(cs0) == NONCE_SIZE ?
(
  // ENC(KAB, NB)
  yahalom_pub_msg4(?server, ?sender, ?receiver, ?a_id, ?NB) &*&
  NB == cg_nonce(?receiver2, ?b_id) &*&
  cs0 == chars_for_cg(NB) &*&
  true == cg_is_generated(NB) &*&
  (
     bad(server) || bad(sender) || bad(receiver) ?
      [_]yahalom_pub(NB)
    :
      receiver == receiver2 &*&
      p0 == server &*&
```

```
              cg_info(NB) ==
                IP(2, IP(p0, IP(sender, IP(sender, a_id)))) &*&
              cg_info(cg_symmetric_key(p0, c0)) ==
                IP(4, IP(sender, IP(receiver, IP(sender, IP(a_id,
                                                 IP(receiver2, b_id))))))
          )
        ) :
        false;
    case cg_auth_encrypted(p0, c0, cs0, ent0):
      return true == yahalom_public_key(p0, c0, true) &*&
             [_]public_generated(yahalom_pub)(cs0);
    case cg_asym_encrypted(p0, c0, cs0, ent0):
      return [_]public_generated(yahalom_pub)(cs0);
    case cg_asym_signature(p0, c0, cs0, ent0):
      return true == yahalom_public_key(p0, c0, false);
  }
;

@*/

////////////////////////////////////////////////////////////////////////////
// Implementation prototypes for this protocol ////////////////////////////
////////////////////////////////////////////////////////////////////////////

void server(int server, int sender, int receiver,
            char *s_key, char *r_key);
/*@ requires [_]public_invar(yahalom_pub) &*&
             [_]decryption_key_classifier(yahalom_public_key) &*&
             principal(server, _) &*&
             [?f1]cryptogram(s_key, KEY_SIZE, ?s_key_cs, ?s_key_cg) &*&
               s_key_cg == cg_symmetric_key(sender, _) &*&
               cg_info(s_key_cg) == IP(3, server) &*&
             [?f2]cryptogram(r_key, KEY_SIZE, ?r_key_cs, ?r_key_cg) &*&
               r_key_cg == cg_symmetric_key(receiver, _) &*&
               cg_info(r_key_cg) == IP(3, server); @*/
/*@ ensures  principal(server, _) &*&
             [f1]cryptogram(s_key, KEY_SIZE, s_key_cs, s_key_cg) &*&
             [f2]cryptogram(r_key, KEY_SIZE, r_key_cs, r_key_cg); @*/

void sender(int server, int sender, int receiver,
            char *key, char *generated_key);
/*@ requires [_]public_invar(yahalom_pub) &*&
             [_]decryption_key_classifier(yahalom_public_key) &*&
             principal(sender, ?s_id) &*&
             [?f]cryptogram(key, KEY_SIZE, ?key_cs, ?key_cg) &*&
               key_cg == cg_symmetric_key(sender, _) &*&
               cg_info(key_cg) == IP(3, server) &*&
             chars(generated_key, KEY_SIZE, _); @*/
/*@ ensures  principal(sender, _) &*&
             [f]cryptogram(key, KEY_SIZE, key_cs, key_cg) &*&
             cryptogram(generated_key, KEY_SIZE, _, ?g_key_cg) &*&
             col || bad(server) || bad(sender) ?
               true
             :
               g_key_cg == cg_symmetric_key(server, ?id) &*&
               IF(cg_info(g_key_cg)) == 4 &*&
               IF(IS(cg_info(g_key_cg))) == sender &*&
               IF(IS(IS(cg_info(g_key_cg)))) == receiver &*&
               IF(IS(IS(IS(cg_info(g_key_cg))))) == sender &*&
               IF(IS(IS(IS(IS(cg_info(g_key_cg)))))) ==
s_id + 1 &*&
               bad(receiver) ||
                 IF(IS(IS(IS(IS(IS(cg_info(g_key_cg)))))) == receiver &&
                 !yahalom_public_key(server, id, true); @*/

void receiver(int server, int sender, int receiver,
              char *key, char *generated_key);
```

```
/*@ requires [_]public_invar(yahalom_pub) &*&
             [_]decryption_key_classifier(yahalom_public_key) &*&
             principal(receiver, ?r_id) &*&
             [?f]cryptogram(key, KEY_SIZE, ?key_cs, ?key_cg) &*&
               key_cg == cg_symmetric_key(receiver, _) &*&
               cg_info(key_cg) == IP(3, server) &*&
             chars(generated_key, KEY_SIZE, _); @*/
/*@ ensures  principal(receiver, _) &*&
             [f]cryptogram(key, KEY_SIZE, key_cs, key_cg) &*&
             cryptogram(generated_key, KEY_SIZE, _, ?g_key_cg) &*&
             col || bad(server) || bad(sender) || bad(receiver) ?
               true
             :
               g_key_cg == cg_symmetric_key(server, ?id) &*&
               IF(cg_info(g_key_cg)) == 4 &*&
               IF(IS(cg_info(g_key_cg))) == sender &*&
               IF(IS(IS(cg_info(g_key_cg)))) == receiver &*&
               IF(IS(IS(IS(cg_info(g_key_cg))))) == sender &*&
               IF(IS(IS(IS(IS(IS(cg_info(g_key_cg)))))) == receiver &*&
               IS(IS(IS(IS(IS(IS(cg_info(g_key_cg)))))) == r_id + 1; @*/

////////////////////////////////////////////////////////////////////////////
// Attacker proof obligations for this protocol ////////////////////////////
////////////////////////////////////////////////////////////////////////////

//@ PUBLIC_INVARIANT_PROOFS(yahalom)
//@ DECRYPTION_PROOFS(yahalom)

#endif
```

B. Source file of the verified Yahalom protocol implementation

```
#include "yahalom.h"

#include <stdlib.h>
#include <string.h>
#include <stdio.h>

//@ #include "quantifiers.gh"

#define SENDER_PORT 121212
#define RECVER_PORT 232323
#define SERVER_PORT 343434

#define INCLUDE_SERVER
#define INCLUDE_SENDER
#define INCLUDE_RECEIVER

void encrypt(havege_state *state, char *key, char *msg,
             unsigned int msg_len, char* output)
/*@ requires [_]public_invar(yahalom_pub) &*&
             havege_state_initialized(state) &*&
             principal(?principal1, _) &*&
             [?f1]cryptogram(key, KEY_SIZE, ?key_cs, ?key_cg) &*&
               key_cg == cg_symmetric_key(?principal2, ?id) &*&
             [?f2]crypto_chars(?kind, msg, msg_len, ?msg_cs) &*&
               1024 >= msg_len &*&
               msg_len >= MINIMAL_STRING_SIZE &*&
             chars(output, 16 + msg_len, _); @*/
/*@ ensures  havege_state_initialized(state) &*&
             principal(principal1, _) &*&
             [f1]cryptogram(key, KEY_SIZE, key_cs, key_cg) &*&
             [f2]crypto_chars(kind, msg, msg_len, msg_cs) &*&
             chars(output, 16, ?iv_cs) &*&
             cryptogram(output + 16, msg_len, _, ?enc_cg) &*&
             enc_cg == cg_encrypted(principal2, id, msg_cs, iv_cs); @*/
{
  //@ open principal(principal1, _);
  char iv[16];
  aes_context aes_context;
  unsigned int iv_off = 0;
  //@ chars_limits(output);

  //@ close random_request(principal1, IP(0, 0), false);
  if (havege_random(state, iv, 16) != 0) abort();
  //@ open cryptogram(iv, 16, ?iv_cs, ?iv_cg);
  //@ close yahalom_pub(iv_cg);
  //@ leak yahalom_pub(iv_cg);
  memcpy(output, iv, 16);
  //@ close cryptogram(output, 16, iv_cs, iv_cg);
  //@ public_cryptogram(output, iv_cg);

  //@ close aes_context(&aes_context);
  if (aes_setkey_enc(&aes_context, key, (unsigned int) KEY_SIZE * 8) != 0)
    abort();
  if (aes_crypt_cfb128(&aes_context, AES_ENCRYPT,
                       (unsigned int) msg_len,
                       &iv_off, iv, msg, output + 16) != 0)
    abort();
  zeroize(iv, 16);
  aes_free(&aes_context);
  //@ open aes_context(&aes_context);
  //@ assert cryptogram(output + 16, msg_len, ?enc_cs, ?enc_cg);
  //@ assert enc_cg == cg_encrypted(principal2, id, msg_cs, iv_cs);
  //@ close principal(principal1, _);
}
```

```
void decrypt(char *key, char *msg, unsigned int msg_len, char* output)
/*@ requires [_]public_invar(yahalom_pub) &*&
             decryption_pre(true, false, ?principal1, ?s, ?msg_cs) &*&
             random_permission(principal1, _) &*&
             [?f1]cryptogram(key, KEY_SIZE, ?key_cs, ?key_cg) &*&
               key_cg == cg_symmetric_key(?principal2, ?id) &*&
             [?f2]chars(msg, 16, ?iv_cs) &*&
             [f2]chars(msg + 16, msg_len, msg_cs) &*&
               msg_len >= MINIMAL_STRING_SIZE &*& msg_len < 1024 &*&
             chars(output, msg_len, _); @*/
/*@ ensures  decryption_post(true, ?garbage, principal1,
                           s, principal2, id, ?dec_cs) &*&
             random_permission(principal1, _) &*&
             [f1]cryptogram(key, KEY_SIZE, key_cs, key_cg) &*&
             [f2]chars(msg, 16 + msg_len, append(iv_cs, msg_cs)) &*&
             crypto_chars(?kind, output, msg_len, dec_cs) &*&
             exists(?enc_cg) &*& [_]yahalom_pub(enc_cg) &*&
             msg_cs == chars_for_cg(enc_cg) &*&
             enc_cg == cg_encrypted(?p, ?c, ?dec_cs2, ?iv_cs2) &*&
             garbage ?
               kind == normal
             :
               kind == secret &*&
               p == principal2 &*& c == id &*&
               dec_cs == dec_cs2 && iv_cs == iv_cs2; @*/
{
  char iv[16];
  aes_context aes_context;
  unsigned int iv_off = 0;
  //@ chars_limits(msg);

  //@ cryptogram iv_cg = chars_for_cg_sur(iv_cs, tag_nonce);
  //@ public_chars(msg, 16);
  //@ public_chars_extract(msg, iv_cg);
  //@ chars_to_secret_crypto_chars(msg, 16);
  memcpy(iv, msg, 16);
  //@ public_crypto_chars(msg, 16);

  //@ close aes_context(&aes_context);
  if (aes_setkey_enc(&aes_context, key, (unsigned int) KEY_SIZE * 8) != 0)
    abort();
  //@ interpret_encrypted(msg + 16, msg_len);
  //@ open [f2]cryptogram(msg + 16, msg_len, msg_cs, ?enc_cg);
  //@ close [f2]cryptogram(msg + 16, msg_len, msg_cs, enc_cg);
  //@ assert enc_cg == cg_encrypted(?p, ?c, ?pay_cs2, ?iv_cs2);
  //@ open [_]yahalom_pub(enc_cg);
  //@ close exists(enc_cg);
  if (aes_crypt_cfb128(&aes_context, AES_DECRYPT,
                       (unsigned int) msg_len,
                       &iv_off, iv, msg + 16, output) != 0)
    abort();
  //@ public_cryptogram(msg + 16, enc_cg);
  zeroize(iv, 16);
  aes_free(&aes_context);
  //@ open aes_context(&aes_context);
  //@ chars_join(msg);
}

#ifdef INCLUDE_SERVER

void server(int server, int sender, int receiver,
            char *s_key, char *r_key)
/*@ requires [_]public_invar(yahalom_pub) &*&
             [_]decryption_key_classifier(yahalom_public_key) &*&
             principal(server, _) &*&
             [?f1]cryptogram(s_key, KEY_SIZE, ?s_key_cs, ?s_key_cg) &*&
               s_key_cg == cg_symmetric_key(sender, ?s_id) &*&
```

```
              cg_info(s_key_cg) == int_pair(3, server) &*&
            [?f2]cryptogram(r_key, KEY_SIZE, ?r_key_cs, ?r_key_cg) &*&
              r_key_cg == cg_symmetric_key(receiver, ?r_id) &*&
              cg_info(r_key_cg) == int_pair(3, server); @*/
/*@ ensures  [_]public_invar(yahalom_pub) &*&
             principal(server, _) &*&
             [f1]cryptogram(s_key, KEY_SIZE, s_key_cs, s_key_cg) &*&
             [f2]cryptogram(r_key, KEY_SIZE, r_key_cs, r_key_cg); @*/
{
  //@ open principal(server, _);
  int socket;
  int socket_in;
  int socket_out;
  havege_state havege_state;
  char NA[NONCE_SIZE];
  //@ cryptogram NB_cg;
  char NB[NONCE_SIZE];
  //@ cryptogram NA_cg;
  char KAB[KEY_SIZE];
  //@ bool condition;

  if(net_bind(&socket, NULL, SERVER_PORT) != 0)
    abort();
  if(net_accept(socket, &socket_in, NULL) != 0)
    abort();
  if(net_set_block(socket_in) != 0)
    abort();
  net_usleep(60000);
  if(net_connect(&socket_out, NULL, SENDER_PORT) != 0)
    abort();
  if(net_set_block(socket_out) != 0)
    abort();

  //@ close havege_state(&havege_state);
  havege_init(&havege_state);

  {
    // 2. B -> S. B, ENC(KB, {A, NA, NB})
    int prefix_size = ID_SIZE + NONCE_SIZE;
    int d_size = prefix_size + NONCE_SIZE;
    char *decrypted = malloc(d_size); if (decrypted == 0) abort();
    int m_size = ID_SIZE + 16 + d_size;
    char *message = malloc(m_size); if (message == 0) abort();

    // Receive the message
    net_recv(&socket_in, message, (unsigned int) m_size);
    //@ chars_split(message, ID_SIZE);
    //@ chars_split(message + ID_SIZE, 16);
    //@ assert chars(message, ID_SIZE, ?r_id_cs);
    //@ public_chars(message, ID_SIZE);
    //@ assert chars(message + ID_SIZE, 16, ?iv_cs);
    //@ assert chars(message + ID_SIZE + 16, d_size, ?enc_cs);
    //@ chars_to_crypto_chars(message, ID_SIZE);
    //@ close check_identifier_ghost_args(true, false, 0, 0, append(iv_cs, enc_cs));
    check_identifier(message, receiver);
    //@ public_crypto_chars(message, ID_SIZE);
    //@ assert r_id_cs == identifier(receiver);
    //@ structure s = known_value(0, identifier(sender));
    //@ close decryption_pre(true, false, server, s, enc_cs);
    decrypt(r_key, (void*) message + ID_SIZE, (unsigned int) d_size, decrypted);
    //@ open [_]yahalom_pub(?enc_cg);
    /*@ open decryption_post(true, ?garbage, server, s,
                             receiver, r_id, ?dec_cs); @*/
    //@ crypto_chars_split(decrypted, ID_SIZE);
    //@ crypto_chars_split((void*) decrypted + ID_SIZE, NONCE_SIZE);
    memcpy(NA, (void*) decrypted + ID_SIZE, NONCE_SIZE);
    memcpy(NB, (void*) decrypted + prefix_size, NONCE_SIZE);
```

```
//@ assert crypto_chars(?kind, decrypted, ID_SIZE, ?id_cs);
//@ assert crypto_chars(kind, decrypted + ID_SIZE, NONCE_SIZE, ?NA_cs);
/*@ assert crypto_chars(kind, (void*) decrypted + ID_SIZE + NONCE_SIZE,
                        NONCE_SIZE, ?NB_cs); @*/
//@ NA_cg = chars_for_cg_sur(NA_cs, tag_nonce);
//@ NB_cg = chars_for_cg_sur(NB_cs, tag_nonce);
//@ condition = col || yahalom_public_key(receiver, r_id, true);
/*@ if (col || garbage || condition)
    {
      if (col || garbage)
      {
        chars_to_crypto_chars(decrypted, ID_SIZE);
        public_chars(decrypted + ID_SIZE, NONCE_SIZE);
        chars_to_crypto_chars(decrypted + ID_SIZE, NONCE_SIZE);
        public_chars(decrypted + ID_SIZE + NONCE_SIZE, NONCE_SIZE);
        chars_to_crypto_chars(decrypted + ID_SIZE + NONCE_SIZE, NONCE_SIZE);
      }
      else
      {
        public_generated_split(yahalom_pub, dec_cs, ID_SIZE);
        public_generated_split(yahalom_pub, append(NA_cs, NB_cs), NONCE_SIZE);
        public_crypto_chars(decrypted, ID_SIZE);
      }
      public_crypto_chars_extract(NA, NA_cg);
      public_crypto_chars_extract(NB, NB_cg);
      chars_to_crypto_chars(NB, NONCE_SIZE);
    }
    else
    {
      assert [_]yahalom_pub_msg1(_, ?s2, ?NA2, ?NB2);
      take_append(ID_SIZE, identifier(s2),
                  append(chars_for_cg(NA2), chars_for_cg(NB2)));
      public_crypto_chars(decrypted, ID_SIZE);
    }
@*/
/*@ close check_identifier_ghost_args(true, garbage, receiver,
                                      r_id, append(NA_cs, NB_cs)); @*/
check_identifier(decrypted, sender);
//@ crypto_chars_join((void*) decrypted + ID_SIZE);
//@ if (!col && !garbage) chars_to_secret_crypto_chars(decrypted, ID_SIZE);
//@ crypto_chars_join(decrypted);
/*@ if (!col && !garbage && !condition)
    {
      assert [_]yahalom_pub_msg1(?srv2, ?s2, ?NA2, ?NB2);
      take_append(ID_SIZE, identifier(sender), append(NA_cs, NB_cs));
      take_append(ID_SIZE, identifier(s2),
                           append(chars_for_cg(NA2), chars_for_cg(NB2)));
      drop_append(ID_SIZE, identifier(sender), append(NA_cs, NB_cs));
      drop_append(ID_SIZE, identifier(s2),
                           append(chars_for_cg(NA2), chars_for_cg(NB2)));
      take_append(NONCE_SIZE, NA_cs, NB_cs);
      take_append(NONCE_SIZE, chars_for_cg(NA2), chars_for_cg(NB2));
      drop_append(NONCE_SIZE, NA_cs, NB_cs);
      drop_append(NONCE_SIZE, chars_for_cg(NA2), chars_for_cg(NB2));
      assert srv2 == server;
      equal_identifiers(s2, sender);
      assert s2 == sender;
      assert chars_for_cg(NA2) == NA_cs;
      NA_cg = NA2;
      close cryptogram(NA, NONCE_SIZE, NA_cs, NA_cg);
      public_cryptogram(NA, NA_cg);
      assert chars_for_cg(NB2) == NB_cs;
      NB_cg = NB2;
      assert NB_cg == cg_nonce(receiver, _);
    }
@*/
zeroize(decrypted, d_size);
```

```
  free(decrypted);
  //@ chars_join(message);
  free(message);
}

//@ assert chars(NA, NONCE_SIZE, ?cs_NA);
//@ public_chars(NA, NONCE_SIZE);
//@ public_chars_extract(NA, NA_cg);
//@ assert cs_NA == chars_for_cg(NA_cg);
//@ assert NA_cg == cg_nonce(?s2, ?a_id);
//@ assert true == cg_is_generated(NA_cg);
//@ assert crypto_chars(_, NB, NONCE_SIZE, ?cs_NB);
//@ assert cs_NB == chars_for_cg(NB_cg);
//@ assert NB_cg == cg_nonce(?r2, ?b_id);
/*@ close random_request(server, IP(4, IP(sender, IP(receiver,
                                 IP(s2, IP(a_id, IP(r2, b_id)))))), true); @*/

if (havege_random(&havege_state, KAB, KEY_SIZE) != 0) abort();
//@ open cryptogram(KAB, KEY_SIZE, ?cs_KAB, ?cg_KAB);
//@ assert cg_KAB == cg_symmetric_key(server, ?id_KAB);
/*@ if (yahalom_public_key(server, id_KAB, true))
    {
      close cryptogram(KAB, KEY_SIZE, cs_KAB, cg_KAB);
      close yahalom_pub(cg_KAB);
      leak yahalom_pub(cg_KAB);
      public_cryptogram(KAB, cg_KAB);
      public_chars(KAB, KEY_SIZE);
      chars_to_secret_crypto_chars(KAB, KEY_SIZE);
      assert [_]public_generated(yahalom_pub)(cs_KAB);
    }
@*/

{
  // 3. S -> A. ENC(KA, {B, KAB, NA, NB}), ENC(KB, {A, KAB})
  int size1 = 16 + ID_SIZE + KEY_SIZE + NONCE_SIZE + NONCE_SIZE;
  int size2 = 16 + ID_SIZE + KEY_SIZE;
  char *enc1 = malloc(size1); if (enc1 == 0) abort();
  char *enc2 = malloc(size2); if (enc2 == 0) abort();
  {
    //ENC(KA, {B, KAB, NA, NB})
    int s = ID_SIZE + KEY_SIZE + NONCE_SIZE + NONCE_SIZE;
    char* m = malloc(s); if (m == 0) abort();
    write_identifier(m, receiver);
    //@ chars_to_secret_crypto_chars(m, ID_SIZE);
    //@ assert crypto_chars(secret, m, ID_SIZE, identifier(receiver));
    memcpy(m + ID_SIZE, KAB, KEY_SIZE);
    //@ chars_to_secret_crypto_chars(NA, NONCE_SIZE);
    memcpy(m + ID_SIZE + KEY_SIZE, NA, NONCE_SIZE);
    //@ public_crypto_chars(NA, NONCE_SIZE);
    /*@ if (col || yahalom_public_key(receiver, r_id, true))
          chars_to_secret_crypto_chars(NB, NONCE_SIZE); @*/
    memcpy(m + ID_SIZE + KEY_SIZE + NONCE_SIZE, NB, NONCE_SIZE);
    //@ crypto_chars_join(m + ID_SIZE + KEY_SIZE);
    //@ crypto_chars_join(m + ID_SIZE);
    //@ crypto_chars_join(m);
    //@ close principal(server, _);
    encrypt(&havege_state, s_key, m,
            (unsigned int) (ID_SIZE + KEY_SIZE + NONCE_SIZE + NONCE_SIZE), enc1);
    //@ open cryptogram(enc1 + 16, s, ?enc_cs, ?enc_cg);
    /*@ if (!col)
        {
          list <char> dec_cs = append(identifier(receiver),
                               append(cs_KAB, append(cs_NA, cs_NB)));
          assert enc_cg == cg_encrypted(sender, s_id, dec_cs, ?ent);

          if (yahalom_public_key(sender, s_id, true))
          {
```

```
          assert true == yahalom_public_key(server, id_KAB, true);
          take_append(ID_SIZE, identifier(receiver),
                      append(cs_KAB, append(cs_NA, cs_NB)));
          drop_append(ID_SIZE, identifier(receiver),
                      append(cs_KAB, append(cs_NA, cs_NB)));
          take_append(KEY_SIZE, cs_KAB, append(cs_NA, cs_NB));
          drop_append(KEY_SIZE, cs_KAB, append(cs_NA, cs_NB));
          take_append(NONCE_SIZE, cs_NA, cs_NB);
          crypto_chars_split(m, ID_SIZE);
          crypto_chars_split(m + ID_SIZE, KEY_SIZE);
          crypto_chars_split(m + ID_SIZE + KEY_SIZE, NONCE_SIZE);
          public_crypto_chars(m, ID_SIZE);
          close yahalom_pub(cg_KAB);
          leak yahalom_pub(cg_KAB);
          close cryptogram(m + ID_SIZE, KEY_SIZE, cs_KAB, cg_KAB);
          public_cryptogram(m + ID_SIZE, cg_KAB);
          public_crypto_chars(m + ID_SIZE + KEY_SIZE, NONCE_SIZE);
          if (condition)
          {
            public_crypto_chars(m + ID_SIZE + KEY_SIZE + NONCE_SIZE,
                                NONCE_SIZE);
          }
          else
          {
            close yahalom_pub(NB_cg);
            leak yahalom_pub(NB_cg);
            close cryptogram(m + ID_SIZE + KEY_SIZE + NONCE_SIZE,
                             NONCE_SIZE, cs_NB, NB_cg);
            public_cryptogram(m + ID_SIZE + KEY_SIZE + NONCE_SIZE, NB_cg);
          }
          chars_join(m + ID_SIZE + KEY_SIZE);
          chars_join(m + ID_SIZE);
          chars_join(m);
          public_chars(m, s);
          chars_to_crypto_chars(m, s);
        }
        else
        {
          close yahalom_pub_msg2(server, receiver, NA_cg,
                                 NB_cg, cg_KAB);
        }
        close yahalom_pub(enc_cg);
        leak yahalom_pub(enc_cg);
        close cryptogram(enc1 + 16, ID_SIZE + KEY_SIZE + NONCE_SIZE + NONCE_SIZE,
                         enc_cs, enc_cg);
        public_cryptogram(enc1 + 16, enc_cg);
        chars_join(enc1);
        public_chars(enc1, s + 16);
        chars_to_crypto_chars(enc1, s + 16);
      }
      else
      {
        crypto_chars_to_chars(enc1 + 16, s);
        chars_join(enc1);
        public_chars(enc1, s + 16);
        chars_to_crypto_chars(enc1, s + 16);
      }
  @*/
  zeroize(m, ID_SIZE + KEY_SIZE + NONCE_SIZE + NONCE_SIZE);
  free(m);
}

{
  //ENC(KB, {A, KAB})
  int s = ID_SIZE + KEY_SIZE;
  char* m = malloc(s); if (m == 0) abort();
  write_identifier(m, sender);
```

```
    //@ chars_to_secret_crypto_chars(m, ID_SIZE);
    //@ assert crypto_chars(secret, m, ID_SIZE, identifier(sender));
    memcpy(m + ID_SIZE, KAB, KEY_SIZE);
    //@ crypto_chars_join(m);

    encrypt(&havege_state, r_key, m, (unsigned int) (ID_SIZE + KEY_SIZE), enc2);
    //@ open cryptogram(enc2 + 16, ID_SIZE + KEY_SIZE, ?enc_cs, ?enc_cg);
    /*@ if (!col)
        {
          assert enc_cg == cg_encrypted(receiver, r_id,
                              append(identifier(sender), cs_KAB), ?ent);
          if (yahalom_public_key(receiver, r_id, true))
          {
            assert true == yahalom_public_key(server, id_KAB, true);
            take_append(ID_SIZE, identifier(sender), cs_KAB);
            drop_append(ID_SIZE, identifier(sender), cs_KAB);
            crypto_chars_split(m, ID_SIZE);
            public_crypto_chars(m, ID_SIZE);
            close cryptogram(m + ID_SIZE, KEY_SIZE, cs_KAB, cg_KAB);
            close yahalom_pub(cg_KAB);
            leak yahalom_pub(cg_KAB);
            public_cryptogram(m + ID_SIZE, cg_KAB);
            chars_join(m);
            public_chars(m, s);
            chars_to_crypto_chars(m, s);
          }
          else
          {
            close yahalom_pub_msg3(server, sender, cg_KAB,
                                   s2, a_id, r2, b_id);
          }
          close yahalom_pub(enc_cg);
          leak yahalom_pub(enc_cg);
          close cryptogram(enc2 + 16, ID_SIZE + KEY_SIZE, enc_cs, enc_cg);
          public_cryptogram(enc2 + 16, enc_cg);
          chars_join(enc2);
          public_chars(enc2, s + 16);
          chars_to_crypto_chars(enc2, s + 16);
        }
        else
        {
          crypto_chars_to_chars(enc2 + 16, s);
          chars_join(enc2);
          public_chars(enc2, s + 16);
          chars_to_crypto_chars(enc2, s + 16);
        }
    @*/
    zeroize(m, ID_SIZE + KEY_SIZE);
    free(m);
  }

  {
    int size = size1 + size2;
    char *message = malloc(size); if (message == 0) abort();
    memcpy(message, enc1, (unsigned int) size1);
    memcpy(message + size1, enc2, (unsigned int) size2);
    //@ crypto_chars_join(message);
    //@ crypto_chars_to_chars(message, size);
    //@ open principal(server, _);
    net_send(&socket_out, message, (unsigned int) size);

    free(message);
  }

  free(enc1);
  free(enc2);
}
```

```
  havege_free(&havege_state);
  //@ open havege_state(&havege_state);

  zeroize(KAB, KEY_SIZE);
  zeroize(NB, NONCE_SIZE);

  net_close(socket);
  net_close(socket_in);
  net_close(socket_out);
  //@ close principal(server, _);
}

#endif
#ifdef INCLUDE_SENDER

void sender(int server, int sender, int receiver,
            char *key, char *generated_key)
/*@ requires [_]public_invar(yahalom_pub) &*&
             [_]decryption_key_classifier(yahalom_public_key) &*&
             principal(sender, ?s_id) &*&
             [?f]cryptogram(key, KEY_SIZE, ?key_cs, ?key_cg) &*&
               key_cg == cg_symmetric_key(sender, ?s_id2) &*&
               cg_info(key_cg) == IP(3, server) &*&
             chars(generated_key, KEY_SIZE, _); @*/
/*@ ensures  principal(sender, _) &*&
             [f]cryptogram(key, KEY_SIZE, key_cs, key_cg) &*&
             cryptogram(generated_key, KEY_SIZE, _, ?g_key_cg) &*&
             col || bad(server) || bad(sender) ?
               true
             :
               g_key_cg == cg_symmetric_key(server, ?id) &*&
               IF(cg_info(g_key_cg)) == 4 &*&
               IF(IS(cg_info(g_key_cg))) == sender &*&
               IF(IS(IS(cg_info(g_key_cg)))) == receiver &*&
               IF(IS(IS(IS(cg_info(g_key_cg))))) == sender &*&
               IF(IS(IS(IS(IS(cg_info(g_key_cg)))))) ==
s_id + 1 &*&
               bad(receiver) ||
                 IF(IS(IS(IS(IS(IS(cg_info(g_key_cg)))))) == receiver &&
                 !yahalom_public_key(server, id, true); @*/
{
  //@ open principal(sender, s_id);
  int socket;
  int socket_in;
  int socket_out;
  havege_state havege_state;
  char NA[NONCE_SIZE];
  char NB[NONCE_SIZE];
  char *MB;
  //@ cryptogram cg_NA2;
  //@ cryptogram cg_NB;
  //@ cryptogram cg_KAB;

  net_usleep(40000);
  if(net_connect(&socket_out, NULL, RECVER_PORT) != 0)
    abort();
  if(net_set_block(socket_out) != 0)
    abort();
  if(net_bind(&socket, NULL, SENDER_PORT) != 0)
    abort();
  if(net_accept(socket, &socket_in, NULL) != 0)
    abort();
  if(net_set_block(socket_in) != 0)
    abort();

  //@ close havege_state(&havege_state);
```

```
havege_init(&havege_state);
//@ close random_request(sender, int_pair(1, int_pair(sender, receiver)), false);
if (havege_random(&havege_state, NA, NONCE_SIZE) != 0) abort();
//@ open cryptogram(NA, NONCE_SIZE, ?cs_NA, ?cg_NA);
//@ close cryptogram(NA, NONCE_SIZE, cs_NA, cg_NA);
//@ assert cg_NA == cg_nonce(sender, s_id + 1);
//@ close yahalom_pub(cg_NA);
//@ leak yahalom_pub(cg_NA);
//@ public_cryptogram(NA, cg_NA);
//@ public_chars(NA, NONCE_SIZE);
//@ chars_to_crypto_chars(NA, NONCE_SIZE);

{
  // 1. A -> B. A, NA
  char* message = malloc(ID_SIZE + NONCE_SIZE); if (message == 0) abort();

  write_identifier(message, sender);
  //@ crypto_chars_split(message, ID_SIZE);
  memcpy((void*) message + ID_SIZE, NA, NONCE_SIZE);
  //@ crypto_chars_join(message);
  //@ crypto_chars_join(message);
  //@ crypto_chars_to_chars(message, ID_SIZE + NONCE_SIZE);
  net_send(&socket_out, message, (unsigned int) ID_SIZE + NONCE_SIZE);
  free(message);
}

{
  // 3. S -> A. ENC(KA, {B, KAB, NA, NB}), ENC(KB, {A, KAB})
  int size1 = ID_SIZE + KEY_SIZE + NONCE_SIZE + NONCE_SIZE;
  int size2 = 16 + ID_SIZE + KEY_SIZE;
  int size = 16 + size1 + size2;
  char *msg = malloc(size); if (msg == 0) abort();
  char *dec = malloc(size1); if (dec == 0) abort();
  MB = malloc(size2); if (MB == 0) abort();

  net_recv(&socket_in, msg, (unsigned int) size);
  //@ chars_split(msg, 16 + size1);
  //@ chars_to_crypto_chars(msg + 16 + size1, size2);
  memcpy(MB, msg + 16 + size1, (unsigned int) size2);
  //@ chars_split(msg, 16);
  //@ assert chars(msg, 16, ?iv_cs);
  //@ assert chars(msg + 16, size1, ?enc_cs);
  //@ structure s = known_value(0, identifier(receiver));
  //@ close decryption_pre(true, false, sender, s, enc_cs);
  decrypt(key, (void*) msg, (unsigned int) size1, dec);
  //@ open [_]yahalom_pub(?enc_cg);
  /*@ open decryption_post(true, ?garbage, sender, s,
                           sender, s_id2, ?dec_cs); @*/
  //@ crypto_chars_to_chars(msg + 16 + size1, size2);
  //@ chars_join(msg);
  free(msg);
  //@ assert crypto_chars(?kind, dec, size1, dec_cs);
  //@ assert enc_cg == cg_encrypted(?p2, ?c2, ?dec_cs2, ?iv_cs2);
  //@ crypto_chars_split(dec, ID_SIZE);
  //@ crypto_chars_split(dec + ID_SIZE, KEY_SIZE);
  //@ crypto_chars_split(dec + ID_SIZE + KEY_SIZE, NONCE_SIZE);
  //@ assert crypto_chars(kind, dec, ID_SIZE, ?id_cs);
  //@ assert crypto_chars(kind, dec + ID_SIZE, KEY_SIZE, ?cs_KAB);
  //@ assert crypto_chars(kind, dec + ID_SIZE + KEY_SIZE, NONCE_SIZE, ?cs_NA2);
  /*@ assert crypto_chars(kind, dec + ID_SIZE + KEY_SIZE + NONCE_SIZE,
                          NONCE_SIZE, ?cs_NB); @*/
  //@ list<char> dec_cs0 = append(cs_KAB, append(cs_NA2, cs_NB));
  /*@ take_append(ID_SIZE, identifier(receiver), append(cs_KAB,
                  append(cs_NA, cs_NB))); @*/
  /*@ drop_append(ID_SIZE, identifier(receiver), append(cs_KAB,
                  append(cs_NA, cs_NB))); @*/
  //@ take_append(KEY_SIZE, cs_KAB, append(cs_NA, cs_NB));
```

```
//@ drop_append(KEY_SIZE, cs_KAB, append(cs_NA, cs_NB));
//@ take_append(NONCE_SIZE, cs_NA, cs_NB);
//@ drop_append(NONCE_SIZE, cs_NA, cs_NB);
//@ take_append(NONCE_SIZE, cs_NA, cs_NB);
//@ drop_append(NONCE_SIZE, cs_NA, cs_NB);
//@ assert dec_cs == append(id_cs, dec_cs0);

//@ bool condition = col || yahalom_public_key(sender, s_id2, true);
//@ cg_NA2 = chars_for_cg_sur(cs_NA2, tag_nonce);
//@ cg_NB = chars_for_cg_sur(cs_NB, tag_nonce);
//@ cg_KAB = chars_for_cg_sur(cs_KAB, tag_symmetric_key);
memcpy(generated_key, (void*) dec + ID_SIZE, KEY_SIZE);
memcpy(NB, (void*) dec + ID_SIZE + KEY_SIZE + NONCE_SIZE, NONCE_SIZE);
/*@ if (col || garbage || condition)
    {
      if (col || garbage)
      {
        crypto_chars_to_chars(dec, ID_SIZE);
        crypto_chars_to_chars(dec + ID_SIZE, KEY_SIZE);
        crypto_chars_to_chars(dec + ID_SIZE + KEY_SIZE, NONCE_SIZE);
        crypto_chars_to_chars(dec + ID_SIZE + KEY_SIZE + NONCE_SIZE, NONCE_SIZE);
      }
      else
      {
        public_generated_split(yahalom_pub, dec_cs, ID_SIZE);
        public_generated_split(yahalom_pub, dec_cs0, KEY_SIZE);
        public_generated_split(yahalom_pub, append(cs_NA2, cs_NB), NONCE_SIZE);
        public_crypto_chars(dec, ID_SIZE);
        public_crypto_chars(dec + ID_SIZE, KEY_SIZE);
        public_crypto_chars(dec + ID_SIZE + KEY_SIZE, NONCE_SIZE);
        public_crypto_chars(dec + ID_SIZE + KEY_SIZE + NONCE_SIZE, NONCE_SIZE);

      }
      public_chars_extract(dec + ID_SIZE + KEY_SIZE + NONCE_SIZE, cg_NB);
      public_chars_extract(dec + ID_SIZE, cg_KAB);
      chars_to_crypto_chars(dec, ID_SIZE);
      chars_to_crypto_chars(dec + ID_SIZE, KEY_SIZE);
      chars_to_crypto_chars(dec + ID_SIZE + KEY_SIZE, NONCE_SIZE);
      public_chars(dec + ID_SIZE + KEY_SIZE + NONCE_SIZE, NONCE_SIZE);
      chars_to_crypto_chars(dec + ID_SIZE + KEY_SIZE + NONCE_SIZE, NONCE_SIZE);
    }
    else
    {
      assert [_]yahalom_pub_msg2(?s2, ?r2, ?NA2, ?NB2, ?KAB);
      take_append(ID_SIZE, identifier(r2),append(chars_for_cg(KAB),
                  append(chars_for_cg(NA2), chars_for_cg(NB2))));
      drop_append(ID_SIZE, identifier(r2),append(chars_for_cg(KAB),
                  append(chars_for_cg(NA2), chars_for_cg(NB2))));
      take_append(KEY_SIZE, chars_for_cg(KAB),
                  append(chars_for_cg(NA2), chars_for_cg(NB2)));
      drop_append(KEY_SIZE, chars_for_cg(KAB),
                  append(chars_for_cg(NA2), chars_for_cg(NB2)));
      take_append(NONCE_SIZE, chars_for_cg(NA2), chars_for_cg(NB2));
      drop_append(NONCE_SIZE, chars_for_cg(NA2), chars_for_cg(NB2));
      public_crypto_chars(dec, ID_SIZE);
      close cryptogram(dec + ID_SIZE + KEY_SIZE, NONCE_SIZE, chars_for_cg(NA2), NA2);
      public_cryptogram(dec + ID_SIZE + KEY_SIZE, NA2);
      chars_to_crypto_chars(dec + ID_SIZE + KEY_SIZE, NONCE_SIZE);
    }
@*/
//@ close check_identifier_ghost_args(true, garbage, sender, s_id2, dec_cs0);
check_identifier(dec, receiver);
//@ assert id_cs == identifier(receiver);
if (memcmp(NA, dec + ID_SIZE + KEY_SIZE, NONCE_SIZE) != 0) abort();
//@ assert cs_NA == cs_NA2;
//@ assert crypto_chars(kind, NB, NONCE_SIZE, cs_NB);
/*@ if (!col && !garbage && !condition)
```

```
      {
        assert [_]yahalom_pub_msg2(?server2, ?receiver2, ?NA2, ?NB2,
                                   ?KAB2);
        assert length(identifier(receiver2)) == ID_SIZE;
        assert identifier(receiver2) == identifier(receiver);
        assert cs_KAB == chars_for_cg(KAB2);
        assert cs_NA == chars_for_cg(NA2);
        equal_identifiers(receiver2, receiver);
        assert receiver2 == receiver;
        cg_KAB = KAB2;
        assert cg_KAB == cg_symmetric_key(server, _);
        chars_for_cg_inj(cg_NA, NA2);
        assert cg_NA == cg_nonce(?s2, ?a_id);
        assert NB2 == cg_nonce(?r2, ?b_id);
        cg_NB = NB2;
        chars_for_cg_inj(cg_NB, NB2);
        assert chars_for_cg(cg_NB) == cs_NB;
        assert server2 == server;
        assert cg_info(cg_KAB) == IP(4, IP(sender, IP(receiver,
                                    IP(s2, IP(a_id, IP(r2, b_id))))));
        chars_to_secret_crypto_chars(dec, ID_SIZE);
        chars_to_secret_crypto_chars(dec + ID_SIZE + KEY_SIZE, NONCE_SIZE);
      }
  @*/
  //@ assert crypto_chars(?k, generated_key, KEY_SIZE, cs_KAB);
  //@ if (k == normal) chars_to_secret_crypto_chars(generated_key, KEY_SIZE);
  //@ close cryptogram(generated_key, KEY_SIZE, cs_KAB, cg_KAB);
  //@ crypto_chars_join(dec + ID_SIZE + KEY_SIZE);
  //@ crypto_chars_join(dec + ID_SIZE);
  //@ crypto_chars_join(dec);
  zeroize(dec, size1);
  free(dec);
}

//@ assert crypto_chars(?k, NB, NONCE_SIZE, ?cs_NB);
//@ if (k == normal) chars_to_secret_crypto_chars(NB, NONCE_SIZE);
//@ assert crypto_chars(secret, NB, NONCE_SIZE, cs_NB);
//@ assert cs_NB == chars_for_cg(cg_NB);
//@ open cryptogram(generated_key, KEY_SIZE, ?cs_KAB, cg_KAB);
//@ close cryptogram(generated_key, KEY_SIZE, cs_KAB, cg_KAB);

//@ assert cg_KAB == cg_symmetric_key(?p0, ?c0);
//@ assert col || bad(sender) || bad(server) || p0 == server;
//@ assert cg_NA == cg_nonce(?sender2, ?a_id);
//@ assert col || (sender == sender2 && a_id == s_id + 1);

//@ assert cg_NB == cg_nonce(?receiver2, ?b_id);
/*@ assert col || bad(server) || bad(sender) ||
             bad(receiver) || receiver == receiver2; @*/
{
  // 4. A -> B. ENC(KB, {A, KAB}), ENC(KAB, NB)
  int size1 = 16 + ID_SIZE + KEY_SIZE;
  int size2 = 16 + NONCE_SIZE;
  char *enc2 = malloc(size2); if (enc2 == 0) abort();
  {
    //@ close principal(sender, _);
    encrypt(&havege_state, generated_key, NB, NONCE_SIZE, enc2);
    //@ open cryptogram(enc2 + 16, NONCE_SIZE, ?enc_cs, ?enc_cg);
    /*@ if (!col)
        {
          if (yahalom_public_key(p0, c0, true))
          {
            if (bad(receiver))
            {
              assert [_]yahalom_pub(cg_NB);
              close cryptogram(NB, NONCE_SIZE, cs_NB, cg_NB);
              public_cryptogram(NB, cg_NB);
```

```
                public_chars(NB, NONCE_SIZE);
                chars_to_crypto_chars(NB, NONCE_SIZE);
              }
            }
            else
            {
              if (bad(server) || bad(sender))
              {
                public_crypto_chars_extract(NB, cg_NB);
                chars_to_crypto_chars(NB, NONCE_SIZE);
              }
              close yahalom_pub_msg4(server, sender, receiver, a_id, cg_NB);
            }
            close yahalom_pub(enc_cg);
            leak yahalom_pub(enc_cg);
            close cryptogram(enc2 + 16, NONCE_SIZE, enc_cs, enc_cg);
            public_cryptogram(enc2 + 16, enc_cg);
          }
          else
          {
            crypto_chars_to_chars(enc2 + 16, NONCE_SIZE);
          }
      @*/
      //@ chars_join(enc2);
      //@ chars_to_crypto_chars(enc2, size2);
    }

    {
      int size = size1 + size2;
      char *message = malloc(size); if (message == 0) abort();
      memcpy(message, MB, (unsigned int) size1);
      memcpy(message + size1, enc2, (unsigned int) size2);
      //@ crypto_chars_join(message);
      //@ crypto_chars_to_chars(message, size);
      //@ open principal(sender, _);
      net_send(&socket_out, message, (unsigned int) size);
      free(message);
    }

    free(enc2);
  }

  zeroize(NB, NONCE_SIZE);
  free(MB);

  havege_free(&havege_state);
  //@ open havege_state(&havege_state);

  net_close(socket);
  net_close(socket_in);
  net_close(socket_out);
  //@ close principal(sender, _);
}

#endif
#ifdef INCLUDE_RECEIVER

void receiver_(int socket_in, int sender, int receiver, int server,
               char *key, char *generated_key, char* NA, char* NB)
  /*@ requires [_]public_invar(yahalom_pub) &*&
               [_]decryption_key_classifier(yahalom_public_key) &*&
               principal(receiver, _) &*&
               net_status(socket_in, ?ip, ?port, connected) &*&
               [?f1]cryptogram(key, KEY_SIZE, ?key_cs, ?key_cg) &*&
                 key_cg == cg_symmetric_key(receiver, ?r_id1) &*&
                 cg_info(key_cg) == int_pair(3, server) &*&
               chars(generated_key, KEY_SIZE, _) &*&
```

```
                crypto_chars(?kna, NA, NONCE_SIZE, ?cs_NA) &*&
                  exists(?cg_NA) &*& cs_NA == chars_for_cg(cg_NA) &*&
                  cg_NA == cg_nonce(?s, ?s_id) &*&
                cryptogram(NB, NONCE_SIZE, ?cs_NB, ?cg_NB) &*&
                  cg_NB == cg_nonce(receiver, ?r_id0) &*&
                  cg_info(cg_NB) ==
                    IP(2, IP(server, IP(sender, IP(s, s_id)))); @*/
  /*@ ensures   principal(receiver, _) &*&
                net_status(socket_in, ip, port, connected) &*&
                [f1]cryptogram(key, KEY_SIZE, key_cs, key_cg) &*&
                cryptogram(generated_key, KEY_SIZE, ?cs_KAB, ?cg_KAB) &*&
                crypto_chars(kna, NA, NONCE_SIZE, cs_NA) &*&
                cryptogram(NB, NONCE_SIZE, cs_NB, cg_NB) &*&
                col || bad(server) || bad(sender) || bad(receiver) ?
                  true
                :
                  cg_KAB == cg_symmetric_key(server, ?id) &*&
                  IF(cg_info(cg_KAB)) == 4 &*&
                  IF(IS(cg_info(cg_KAB))) == sender &*&
                  IF(IS(IS(cg_info(cg_KAB)))) == receiver &*&
                  IF(IS(IS(IS(cg_info(cg_KAB))))) == sender &*&
                  IF(IS(IS(IS(IS(IS(cg_info(cg_KAB)))))) == receiver &*&
                  IS(IS(IS(IS(IS(IS(cg_info(cg_KAB)))))) == r_id0; @*/
{
  //@ open principal(receiver, _);
  // 4. A -> B. ENC(KB, {A, KAB}), ENC(KAB, NB)
  int size1 = ID_SIZE + KEY_SIZE;
  int size2 = NONCE_SIZE;
  int size = 16 + size1 + 16 + size2;
  char *msg = malloc(size); if (msg == 0) abort();
  char *dec1 = malloc(size1); if (dec1 == 0) abort();
  char *dec2 = malloc(size2); if (dec2 == 0) abort();
  net_recv(&socket_in, msg, (unsigned int) size);
  //@ chars_split(msg, 16);
  //@ chars_split(msg + 16, size1);
  //@ assert chars(msg, 16, ?iv_cs1);
  //@ assert chars(msg + 16, size1, ?enc_cs1);
  //@ structure s1 = known_value(0, identifier(sender));
  //@ close decryption_pre(true, false, receiver, s1, enc_cs1);
  decrypt(key, (void*) msg, (unsigned int) size1, dec1);
  //@ open [_]yahalom_pub(?enc_cg);
  //@ assert enc_cg == cg_encrypted(?p, ?c, ?dec_cs2, ?iv_cs2);
  /*@ open decryption_post(true, ?garbage, receiver, s1,
                           receiver, r_id1, ?dec_cs); @*/
  //@ assert crypto_chars(?kind, dec1, size1, ?cs_dec1);
  //@ crypto_chars_split(dec1, ID_SIZE);
  //@ assert crypto_chars(kind, dec1, ID_SIZE, ?cs_id1);
  //@ assert crypto_chars(kind, dec1 + ID_SIZE, KEY_SIZE, ?cs_KAB);
  //@ take_append(ID_SIZE, cs_id1, cs_KAB);
  //@ drop_append(ID_SIZE, cs_id1, cs_KAB);
  //@ cryptogram cg_KAB = chars_for_cg_sur(cs_KAB, tag_symmetric_key);
  /*@ if (col || garbage || yahalom_public_key(receiver, r_id1, true))
      {
        if (col || garbage)
        {
          public_chars(dec1, ID_SIZE);
          public_chars(dec1 + ID_SIZE, KEY_SIZE);
        }
        else
        {
          public_generated_split(yahalom_pub, dec_cs, ID_SIZE);
          public_crypto_chars(dec1, ID_SIZE);
          public_crypto_chars(dec1 + ID_SIZE, KEY_SIZE);
        }
        public_chars_extract(dec1 + ID_SIZE, cg_KAB);
        chars_to_crypto_chars(dec1, ID_SIZE);
        chars_to_crypto_chars(dec1 + ID_SIZE, KEY_SIZE);
```

```
    }
    else
    {
      assert [_]yahalom_pub_msg3(?server2, ?sender2, ?KAB2, ?s2,
                                 ?a_id2, ?r2, ?b_id2);
      take_append(ID_SIZE, identifier(sender2), chars_for_cg(KAB2));
      drop_append(ID_SIZE, identifier(sender2), chars_for_cg(KAB2));
      public_crypto_chars(dec1, ID_SIZE);
    }
@*/
memcpy(generated_key, dec1 + ID_SIZE, KEY_SIZE);
/*@ close check_identifier_ghost_args(true, garbage, receiver,
                                      r_id1, cs_KAB); @*/
check_identifier(dec1, sender);
/*@ if(!col && !yahalom_public_key(receiver, r_id1, true))
    {
      chars_to_crypto_chars(dec1, ID_SIZE);
      assert [_]yahalom_pub_msg3(?server2, ?sender2, ?KAB2, ?s2,
                                 ?a_id2, ?r2, ?b_id2);
      assert server2 == server;
      equal_identifiers(sender2, sender);
      assert sender2 == sender;
      cg_KAB = KAB2;
      assert cg_KAB == cg_symmetric_key(server, _);
    }
@*/
//@ assert crypto_chars(?k, generated_key, KEY_SIZE, cs_KAB);
//@ if (k == normal) chars_to_secret_crypto_chars(generated_key, KEY_SIZE);
//@ close cryptogram(generated_key, KEY_SIZE, cs_KAB, cg_KAB);
//@ assert cg_KAB == cg_symmetric_key(?p4, ?c4);
//@ structure st = known_value(0, cs_NB);
//@ chars_split(msg + 16 + size1, 16);
//@ assert chars(msg + 2 * 16 + size1, size2, ?msg_cs);
//@ close decryption_pre(true, false, receiver, st, msg_cs);
decrypt(generated_key, msg + 16 + size1, (unsigned int) size2, dec2);
/*@ open decryption_post(true, ?garbage2, receiver,
                         st, p4, c4, _); @*/
//@ open exists(?enc_cg2);
//@ assert enc_cg2 == cg_encrypted(?p3, ?c3, ?dec_cs3, ?iv_cs3);
//@ open [_]yahalom_pub(enc_cg2);
/*@ if (col || garbage2 || yahalom_public_key(p4, c4, true))
    {
      if (col || garbage2)
        public_chars(dec2, NONCE_SIZE);
      else
        public_crypto_chars(dec2, NONCE_SIZE);
      chars_to_crypto_chars(dec2, NONCE_SIZE);
    }
    else
    {
      assert [_]yahalom_pub_msg4(?server2, ?sender2,
                                 ?receiver2, ?a_id2, ?NB2);
      close memcmp_secret(dec2, NONCE_SIZE, dec_cs3, NB2);
    }
@*/
//@ open cryptogram(NB, NONCE_SIZE, cs_NB, cg_NB);
//@ close memcmp_secret(NB, NONCE_SIZE, cs_NB, cg_NB);
if (memcmp(dec2, NB, NONCE_SIZE) != 0) abort();
//@ close cryptogram(NB, NONCE_SIZE, cs_NB, cg_NB);
//@ assert crypto_chars(_, dec2, NONCE_SIZE, cs_NB);
/*@ if (garbage2)
    {
      close exists(pair(nil, nil));
      close has_structure(cs_NB, st);
      leak has_structure(cs_NB, st);
      decryption_garbage(dec2, NONCE_SIZE, st);
      chars_to_secret_crypto_chars(dec2, NONCE_SIZE);
```

```
      }
  @*/
  /*@ if (!col && !bad(server) && !bad(sender) && !bad(receiver))
      {
        assert [_]yahalom_pub_msg4(?server2, ?sender2,
                                   ?receiver2, ?a_id2, ?NB2);
        chars_for_cg_inj(cg_NB, NB2);
        if (bad(server2) || bad(sender2) || bad(receiver2))
        {
          open [_]yahalom_pub(cg_NB);
          assert false;
        }
      }
  @*/
  //@ assert crypto_chars(?k1, dec2, NONCE_SIZE, cs_NB);
  //@ if (k1 == normal) crypto_chars_to_chars(dec2, NONCE_SIZE);
  //@ if (k1 == normal) chars_to_secret_crypto_chars(dec2, NONCE_SIZE);

  free(msg);
  //@ assert crypto_chars(?k2, dec1 + ID_SIZE, KEY_SIZE, _);
  //@ if (k2 == secret) chars_to_secret_crypto_chars(dec1, ID_SIZE);
  //@ crypto_chars_join(dec1);
  zeroize(dec1, size1);
  free(dec1);
  zeroize(dec2, size2);
  free(dec2);
  //@ close principal(receiver, _);
}

void receiver(int server, int sender, int receiver,
              char *key, char *generated_key)
/*@ requires [_]public_invar(yahalom_pub) &*&
             [_]decryption_key_classifier(yahalom_public_key) &*&
             principal(receiver, ?r_id1) &*&
             [?f]cryptogram(key, KEY_SIZE, ?key_cs, ?key_cg) &*&
               key_cg == cg_symmetric_key(receiver, ?r_id2) &*&
               cg_info(key_cg) == int_pair(3, server) &*&
             chars(generated_key, KEY_SIZE, _); @*/
/*@ ensures  [_]public_invar(yahalom_pub) &*&
             principal(receiver, _) &*&
             [f]cryptogram(key, KEY_SIZE, key_cs, key_cg) &*&
             cryptogram(generated_key, KEY_SIZE, _, ?g_key_cg) &*&
             col || bad(server) || bad(sender) || bad(receiver) ?
               true
             :
               g_key_cg == cg_symmetric_key(server, ?id) &*&
               IF(cg_info(g_key_cg)) == 4 &*&
               IF(IS(cg_info(g_key_cg))) == sender &*&
               IF(IS(IS(cg_info(g_key_cg)))) == receiver &*&
               IF(IS(IS(IS(cg_info(g_key_cg))))) == sender &*&
               IF(IS(IS(IS(IS(IS(cg_info(g_key_cg))))))) == receiver &*&
               IS(IS(IS(IS(IS(IS(cg_info(g_key_cg))))))) == r_id1 + 1; @*/
{
  //@ open principal(receiver, _);
  int socket;
  int socket_in;
  int socket_out;
  havege_state havege_state;
  char NA[NONCE_SIZE];
  char NB[NONCE_SIZE];
  //@ cryptogram cg_KAB;

  net_usleep(20000);
  if(net_connect(&socket_out, NULL, SERVER_PORT) != 0)
    abort();
  if(net_set_block(socket_out) != 0)
    abort();
```

```
if(net_bind(&socket, NULL, RECVER_PORT) != 0)
  abort();
if(net_accept(socket, &socket_in, NULL) != 0)
  abort();
if(net_set_block(socket_in) != 0)
  abort();

{
  // 1. A -> B. A, NA
  int size = ID_SIZE + NONCE_SIZE;
  char* message = malloc(size); if (message == 0) abort();
  net_recv(&socket_in, message, (unsigned int) size);
  //@ chars_split(message, ID_SIZE);
  //@ public_chars(message, ID_SIZE);
  //@ close check_identifier_ghost_args(true, false, 0, 0, nil);
  check_identifier(message, sender);
  //@ public_crypto_chars(message, ID_SIZE);
  //@ assert chars(message, ID_SIZE, identifier(sender));
  //@ assert chars(message + ID_SIZE, NONCE_SIZE, ?cs_NA);
  //@ chars_to_crypto_chars(message + ID_SIZE, NONCE_SIZE);
  memcpy(NA, (void*) message + ID_SIZE, NONCE_SIZE);
  //@ public_chars(message + ID_SIZE, NONCE_SIZE);
  //@ crypto_chars_to_chars(NA, NONCE_SIZE);
  //@ chars_join(message);
  free(message);
}

//@ assert chars(NA, NONCE_SIZE, ?cs_NA);
//@ cryptogram cg_NA = chars_for_cg_sur(cs_NA, tag_nonce);
//@ assert cg_NA == cg_nonce(?s, ?s_id);
//@ public_chars_extract(NA, cg_NA);
//@ chars_to_secret_crypto_chars(NA, NONCE_SIZE);
//@ close havege_state(&havege_state);
havege_init(&havege_state);
/*@ close random_request(receiver, IP(2, IP(server,
                                     IP(sender, IP(s, s_id)))), false); @*/
if (havege_random(&havege_state, NB, NONCE_SIZE) != 0) abort();
//@ open cryptogram(NB, NONCE_SIZE, ?cs_NB, ?cg_NB);

/*@ if (bad(server) || bad(sender) || bad(receiver))
    {
      close cryptogram(NB, NONCE_SIZE, cs_NB, cg_NB);
      close yahalom_pub(cg_NB);
      leak yahalom_pub(cg_NB);
      public_cryptogram(NB, cg_NB);
      public_chars(NB, NONCE_SIZE);
      assert [_]public_generated(yahalom_pub)(cs_NB);
      chars_to_secret_crypto_chars(NB, NONCE_SIZE);
    }
@*/

{
  // 2. B -> S. B, ENC(KB, {A, NA, NB})
  int prefix_size = ID_SIZE + NONCE_SIZE;
  int p_size = prefix_size + NONCE_SIZE;
  char *plaintext = malloc(p_size); if (plaintext == 0) abort();
  int m_size = ID_SIZE + 16 + p_size;
  char *message = malloc(m_size); if (message == 0) abort();

  write_identifier(message, receiver);
  //@ crypto_chars_split(message, ID_SIZE);
  //@ crypto_chars_to_chars(message, ID_SIZE);
  //@ assert chars(message, ID_SIZE, identifier(receiver));

  write_identifier(plaintext, sender);
  //@ crypto_chars_split(plaintext, ID_SIZE);
  //@ crypto_chars_to_chars(plaintext, ID_SIZE);
```

```
  //@ chars_to_secret_crypto_chars(plaintext, ID_SIZE);
  //@ assert crypto_chars(secret, plaintext, ID_SIZE, identifier(sender));
  memcpy((void*) plaintext + ID_SIZE, NA, NONCE_SIZE);
  memcpy((void*) plaintext + ID_SIZE + NONCE_SIZE, NB, NONCE_SIZE);
  //@ crypto_chars_join(plaintext);
  //@ crypto_chars_join(plaintext);
  //@ assert crypto_chars(secret, plaintext, p_size, ?p_cs);
  //@ append_assoc(identifier(sender), cs_NA, cs_NB);
  //@ assert p_cs == append(identifier(sender), append(cs_NA, cs_NB));
  //@ close principal(receiver, _);
  encrypt(&havege_state, key, plaintext,
          (unsigned int) p_size, (void*) message + ID_SIZE);
  //@ open cryptogram((void*) message + ID_SIZE + 16, p_size, ?enc_cs, ?enc_cg);
  /*@ if (!col)
      {
        if (yahalom_public_key(receiver, r_id2, true))
        {
          take_append(ID_SIZE, identifier(sender), append(cs_NA, cs_NB));
          drop_append(ID_SIZE, identifier(sender), append(cs_NA, cs_NB));
          take_append(NONCE_SIZE, cs_NA, cs_NB);
          drop_append(NONCE_SIZE, cs_NA, cs_NB);
          crypto_chars_split(plaintext, ID_SIZE);
          crypto_chars_split(plaintext + ID_SIZE, NONCE_SIZE);
          public_crypto_chars(plaintext, ID_SIZE);
          public_crypto_chars((void*) plaintext + ID_SIZE, NONCE_SIZE);
          public_crypto_chars((void*) plaintext + ID_SIZE + NONCE_SIZE, NONCE_SIZE);
          chars_join(plaintext + ID_SIZE);
          chars_join(plaintext);
          public_chars(plaintext, p_size);
          chars_to_crypto_chars(plaintext, p_size);
        }
        else
        {
          close yahalom_pub_msg1(server, sender, cg_NA, cg_NB);
        }
        close yahalom_pub(enc_cg);
        leak yahalom_pub(enc_cg);
        close cryptogram((void*) message + ID_SIZE + 16, p_size, enc_cs, enc_cg);
        public_cryptogram((void*) message + ID_SIZE + 16, enc_cg);
      }
      else
      {
        crypto_chars_to_chars(message + ID_SIZE + 16, p_size);
      }
  @*/
  //@ chars_join(message);
  //@ chars_join(message);
  //@ open principal(receiver, _);
  net_send(&socket_out, message, (unsigned int) m_size);
  zeroize(plaintext, p_size);
  free(message);
  free(plaintext);
  //@ crypto_chars_to_chars(message + ID_SIZE, 0);
  //@ crypto_chars_to_chars(plaintext + ID_SIZE, 0);
}
//@ close principal(receiver, _);
//@ close exists(cg_NA);
//@ close cryptogram(NB, NONCE_SIZE, cs_NB, cg_NB);
receiver_(socket_in, sender, receiver, server, key,
          generated_key, NA, NB);
//@ open cryptogram(NB, NONCE_SIZE, cs_NB, cg_NB);
//@ close principal(receiver, _);

//@ public_crypto_chars(NA, NONCE_SIZE);
zeroize(NB, NONCE_SIZE);

havege_free(&havege_state);
```

```
  //@ open havege_state(&havege_state);

  net_close(socket);
  net_close(socket_in);
  net_close(socket_out);
  //@ close principal(receiver, _);
}

#endif
```

C. Main function that executes the verified Yahalom protocol

```
#include "yahalom.h"

#include "../general.h"

//@ ATTACKER_PRE(yahalom)

void *attacker_t(void* data) //@ : pthread_run_joinable
  //@ requires pthread_run_pre(attacker_t)(data, ?info);
  //@ ensures  false;
{
  while(true)
    //@ invariant pthread_run_pre(attacker_t)(data, info);
  {
    //@ open pthread_run_pre(attacker_t)(data, info);
    //@ close yahalom_proof_pred();
    attacker();
    //@ open yahalom_proof_pred();
    //@ close pthread_run_pre(attacker_t)(data, info);
  }

  return 0;
}

struct yahalom_args
{
  int server;
  int sender;
  int receiver;

  char* s_key;
  char* r_key;
  char* key;
};

/*@

predicate_family_instance pthread_run_pre(server_t)(void *data, any info) =
  [_]public_invar(yahalom_pub) &*&
  [_]decryption_key_classifier(yahalom_public_key) &*&
  yahalom_args_server(data, ?server) &*&
  yahalom_args_sender(data, ?sender) &*&
  yahalom_args_receiver(data, ?receiver) &*&
  yahalom_args_s_key(data, ?s_key) &*&
  yahalom_args_r_key(data, ?r_key) &*&
  principal(server, _) &*&
  [1/2]cryptogram(s_key, KEY_SIZE, ?s_key_cs, ?s_key_cg) &*&
    s_key_cg == cg_symmetric_key(sender, ?s_id) &*&
    cg_info(s_key_cg) == int_pair(3, server) &*&
  [1/2]cryptogram(r_key, KEY_SIZE, ?r_key_cs, ?r_key_cg) &*&
    r_key_cg == cg_symmetric_key(receiver, ?r_id) &*&
    cg_info(r_key_cg) == int_pair(3, server) &*&
  info == IV(server, IV(sender, IV(receiver, PV(s_key, CL(s_key_cs, IV(s_id,
              PV(r_key, CL(r_key_cs, IV(r_id, nil)))))))));

predicate_family_instance pthread_run_post(server_t)(void *data, any info) =
  yahalom_args_server(data, ?server) &*&
  yahalom_args_sender(data, ?sender) &*&
  yahalom_args_receiver(data, ?receiver) &*&
  yahalom_args_s_key(data, ?s_key) &*&
  yahalom_args_r_key(data, ?r_key) &*&
  principal(server, _) &*&
  [1/2]cryptogram(s_key, KEY_SIZE, ?s_key_cs, ?s_key_cg) &*&
    s_key_cg == cg_symmetric_key(sender, ?s_id) &*&
  [1/2]cryptogram(r_key, KEY_SIZE, ?r_key_cs, ?r_key_cg) &*&
    r_key_cg == cg_symmetric_key(receiver, ?r_id) &*&
```

```
  info == IV(server, IV(sender, IV(receiver, PV(s_key, CL(s_key_cs, IV(s_id,
            PV(r_key, CL(r_key_cs, IV(r_id, nil)))))))));
@*/

void *server_t(void* data) //@ : pthread_run_joinable
  //@ requires pthread_run_pre(server_t)(data, ?x);
  //@ ensures pthread_run_post(server_t)(data, x) &*& result == 0;
{
  struct yahalom_args *args = data;
  //@ open pthread_run_pre(server_t)(data, _);
  server(args->server, args->sender, args->receiver,
         args->s_key, args->r_key);
  //@ close pthread_run_post(server_t)(data, _);
  return 0;
}

/*@
predicate_family_instance pthread_run_pre(sender_t)(void *data, any info) =
  [_]public_invar(yahalom_pub) &*&
  [_]decryption_key_classifier(yahalom_public_key) &*&
  yahalom_args_server(data, ?server) &*&
  yahalom_args_sender(data, ?sender) &*&
  yahalom_args_receiver(data, ?receiver) &*&
  yahalom_args_s_key(data, ?s_key) &*&
  yahalom_args_key(data, ?key) &*&
  principal(sender, _) &*&
  [1/2]cryptogram(s_key, KEY_SIZE, ?s_key_cs, ?s_key_cg) &*&
    s_key_cg == cg_symmetric_key(sender, ?s_id) &*&
    cg_info(s_key_cg) == int_pair(3, server) &*&
  chars(key, KEY_SIZE, _) &*&
  info == IV(server, IV(sender, IV(receiver, PV(s_key, CL(s_key_cs, IV(s_id,
            PV(key, nil)))))));

predicate_family_instance pthread_run_post(sender_t)(void *data, any info) =
  yahalom_args_server(data, ?server) &*&
  yahalom_args_sender(data, ?sender) &*&
  yahalom_args_receiver(data, ?receiver) &*&
  yahalom_args_s_key(data, ?s_key) &*&
  yahalom_args_key(data, ?key) &*&
  principal(sender, _) &*&
  [1/2]cryptogram(s_key, KEY_SIZE, ?s_key_cs, ?s_key_cg) &*&
    s_key_cg == cg_symmetric_key(sender, ?s_id) &*&
  cryptogram(key, KEY_SIZE, _, ?key_cg) &*&
    col || bad(server) || bad(sender) ||
    int_left(cg_info(key_cg)) == 4 &*&
  info == IV(server, IV(sender, IV(receiver, PV(s_key, CL(s_key_cs, IV(s_id,
            PV(key, nil)))))));
@*/

void *sender_t(void* data) //@ : pthread_run_joinable
  //@ requires pthread_run_pre(sender_t)(data, ?x);
  //@ ensures pthread_run_post(sender_t)(data, x) &*& result == 0;
{
  struct yahalom_args *args = data;
  //@ open pthread_run_pre(sender_t)(data, _);
  sender(args->server, args->sender, args->receiver,
         args->s_key, args->key);
  //@ close pthread_run_post(sender_t)(data, _);
  return 0;
}

/*@
predicate_family_instance pthread_run_pre(receiver_t)(void *data, any info) =
  [_]public_invar(yahalom_pub) &*&
  [_]decryption_key_classifier(yahalom_public_key) &*&
  yahalom_args_server(data, ?server) &*&
  yahalom_args_sender(data, ?sender) &*&
```

```
  yahalom_args_receiver(data, ?receiver) &*&
  yahalom_args_r_key(data, ?r_key) &*&
  yahalom_args_key(data, ?key) &*&
  principal(receiver, _) &*&
  [1/2]cryptogram(r_key, KEY_SIZE, ?r_key_cs, ?r_key_cg) &*&
    r_key_cg == cg_symmetric_key(receiver, ?r_id) &*&
    cg_info(r_key_cg) == int_pair(3, server) &*&
  chars(key, KEY_SIZE, _) &*&
  info == IV(server, IV(sender, IV(receiver, PV(r_key, CL(r_key_cs, IV(r_id,
            PV(key, nil)))))));

predicate_family_instance pthread_run_post(receiver_t)(void *data, any info) =
  yahalom_args_server(data, ?server) &*&
  yahalom_args_sender(data, ?sender) &*&
  yahalom_args_receiver(data, ?receiver) &*&
  yahalom_args_r_key(data, ?r_key) &*&
  yahalom_args_key(data, ?key) &*&
  principal(receiver, _) &*&
  [1/2]cryptogram(r_key, KEY_SIZE, ?r_key_cs, ?r_key_cg) &*&
    r_key_cg == cg_symmetric_key(receiver, ?r_id) &*&
  cryptogram(key, KEY_SIZE, _, ?key_cg) &*&
    col || bad(server) || bad(sender) || bad(receiver) ||
    int_left(cg_info(key_cg)) == 4 &*&
  info == IV(server, IV(sender, IV(receiver, PV(r_key, CL(r_key_cs, IV(r_id,
            PV(key, nil)))))));
@*/

void *receiver_t(void* data) //@ : pthread_run_joinable
  //@ requires pthread_run_pre(receiver_t)(data, ?x);
  //@ ensures pthread_run_post(receiver_t)(data, x) &*& result == 0;
{
  struct yahalom_args *args = data;
  //@ open pthread_run_pre(receiver_t)(data, _);
  receiver(args->server, args->sender, args->receiver,
           args->r_key, args->key);
  //@ close pthread_run_post(receiver_t)(data, _);
  return 0;
}

int main(int argc, char **argv) //@ : main_full(main_app)
    //@ requires module(main_app, true);
    //@ ensures true;
{
  pthread_t a_thread;
  havege_state havege_state;

  printf("\n\tExecuting \"");
  printf("yahalom protocol");
  printf("\" ... \n\n");

  //@ PROTOCOL_INIT(yahalom)
  //@ int server = principal_create();
  //@ assert server == 1;
  //@ int sender = principal_create();
  //@ assert sender == 2;
  //@ int receiver = principal_create();
  //@ assert receiver == 3;

  //@ int attacker = principal_create();
  //@ assume (bad(attacker));
  //@ close havege_state(&havege_state);
  havege_init(&havege_state);
  //@ assume (bad(attacker));
  //@ close pthread_run_pre(attacker_t)(NULL, some(attacker));
  pthread_create(&a_thread, NULL, &attacker_t, NULL);

  int i = 0;
```

```
#ifdef EXECUTE
  while (i++ < 10)
#else
  while (true)
#endif
    /*@ invariant [_]public_invar(yahalom_pub) &*&
                  [_]decryption_key_classifier(yahalom_public_key) &*&
                  havege_state_initialized(&havege_state) &*&
                  principal(server, ?serv_count) &*&
                  principal(sender, ?send_count) &*&
                  principal(receiver, ?rcvr_count);
    @*/
  {
    char s_key[KEY_SIZE];
    char r_key[KEY_SIZE];
    char key1[KEY_SIZE];
    char key2[KEY_SIZE];

    //@ open principal(sender, _);
    //@ close random_request(sender, int_pair(3, server), true);
    if (havege_random(&havege_state, s_key, KEY_SIZE) != 0) abort();
    //@ close principal(sender, _);
    //@ assert cryptogram(s_key, KEY_SIZE, ?cs_s_key, ?cg_s_key);
    //@ assert cg_s_key == cg_symmetric_key(sender, send_count + 1);

    //@ open principal(receiver, _);
    //@ close random_request(receiver, int_pair(3, server), true);
    if (havege_random(&havege_state, r_key, KEY_SIZE) != 0) abort();
    //@ close principal(receiver, _);
    //@ assert cryptogram(r_key, KEY_SIZE, ?cs_r_key, ?cg_r_key);
    //@ assert cg_r_key == cg_symmetric_key(receiver, rcvr_count + 1);

    {
      pthread_t serv_thread, s_thread, r_thread;
      struct yahalom_args serv_args, s_args, r_args;

      serv_args.server = 1;
      serv_args.sender = 2;
      serv_args.receiver = 3;
      serv_args.s_key = s_key;
      serv_args.r_key = r_key;

      s_args.server = 1;
      s_args.sender = 2;
      s_args.receiver = 3;
      s_args.s_key = s_key;
      s_args.key = key1;

      r_args.server = 1;
      r_args.sender = 2;
      r_args.receiver = 3;
      r_args.r_key = r_key;
      r_args.key = key2;

      //@ close pthread_run_pre(server_t)(&serv_args, ?serv_data);
      pthread_create(&serv_thread, NULL, &server_t, &serv_args);
      //@ close pthread_run_pre(sender_t)(&s_args, ?s_data);
      pthread_create(&s_thread, NULL, &sender_t, &s_args);
      //@ close pthread_run_pre(receiver_t)(&r_args, ?r_data);
      pthread_create(&r_thread, NULL, &receiver_t, &r_args);

      pthread_join(serv_thread, NULL);
      //@ open pthread_run_post(server_t)(&serv_args, serv_data);
      pthread_join(s_thread, NULL);
      //@ open pthread_run_post(sender_t)(&s_args, s_data);
      pthread_join(r_thread, NULL);
      //@ open pthread_run_post(receiver_t)(&r_args, r_data);
```

```
    }

    //@ open cryptogram(key1, KEY_SIZE, ?cs_key1, _);
    zeroize(key1, KEY_SIZE);
    //@ open cryptogram(key2, KEY_SIZE, ?cs_key2, _);
    zeroize(key2, KEY_SIZE);
    //@ open [1/2]cryptogram(s_key, KEY_SIZE, _, _);
    //@ open [1/2]cryptogram(s_key, KEY_SIZE, _, _);
    zeroize(s_key, KEY_SIZE);
    //@ open [1/2]cryptogram(r_key, KEY_SIZE, _, _);
    //@ open [1/2]cryptogram(r_key, KEY_SIZE, _, _);
    zeroize(r_key, KEY_SIZE);
    printf(" |%i| ", i);
  }

  printf("\n\n\t\tDone\n");
  return 0;
}
```