Verifying Protocol Implementations by Augmenting Existing Cryptographic Libraries with Specifications

Gijs Vanspauwen and Bart Jacobs

iMinds-DistriNet, KU Leuven, 3001 Leuven, Belgium
{gijs.vanspauwen,bart.jacobs}@cs.kuleuven.be

Abstract. Specifying correct cryptographic protocols has proven to be a difficult task. The implementation of such a protocol in a lower-level programming language introduces additional room for errors. While a lot of work has been done for proving the correctness of high-level (often non-executable) protocol specifications, methodologies to prove properties of protocol implementations in a lower-level language are less well-studied. Such languages however, like the C programming language, are still frequently used to write cryptographic software. We propose a static verification approach for cryptographic protocol implementations written in the C programming language. This approach employs our own extended symbolic model of cryptography which we formalized in VeriFast, a separation logic-based verifier for C programs. By giving formal contracts to the primitives of an existing cryptographic library (i.e. PolarSSL), we were able to prove, besides memory safety, interesting security properties of a small protocol suite that demonstrates the usage of those primitives.

Keywords: static verification, verification of C programs, cryptographic protocols, symbolic model of cryptography, cryptographic libraries

1 Introduction

Cryptographic protocols form the backbone of today's Internet security. They provide confidentiality, authentication and data integrity during remote communication sessions. Specifying correct cryptographic protocols has proven to be a difficult task. There are numerous examples of proposed protocols which turned out to be incorrect (see [8] or [16] for example). An infamous example amongst these is the flaw in the original formulation of the public-key Needham-Schroeder protocol [15], which was noticed and corrected by G. Lowe [14].

Formal verification is a common means to convince oneself that such flaws are absent from protocol descriptions. The successful verification of a protocol does not only give the guarantee that the protocol achieves its goals and that the description does not contain any flaws, it also forces one to formalize what properties the protocol tries to establish and how exactly it achieves these. The strength of the proven properties depends on the chosen model for defining the semantics of cryptographic computations.

There exist mainly two kinds of models that have been successfully applied: symbolic models [4, 5, 10] and computational models [1, 2, 6, 11, 13]. In a symbolic model messages are terms of some abstract algebra. These terms can be constructed through the cryptographic primitive operators or via a pairing operator, and messages are exchanged over a completely untrusted network in the presence of an adversary that can construct any message using the same operators. Dolev and Yao [9] were the first to formalize such a model, so symbolic models are also referred to as Dolev-Yao models. In a computational model cryptographic primitives are probabilistic algorithms that produce actual bit strings and an adversary has polynomially bounded resources to perform his attack. These features allow one to specify and prove properties with a higher security assurance, but it makes reasoning about protocols more complex.

Whichever model a tool or methodology applies, it must provide a language to specify protocols. Some tools provide an abstract specification language (e.g. EasyCrypt [2], ProVerif [5] or CryptoVerif [6]). This abstract language facilitates reasoning about protocol properties, but the resulting protocol description is not immediately executable. In other approaches, protocols are implemented in a programming language and a general-purpose verifier is used to prove properties about this executable code. Bhargavan et al. [4] for example, write their protocol implementations in the F# programming language, an ML variant for the .NET platform. Their invariant-based method uses the F7 type checker, an SMT-based type checker for refinement types, to prove cryptographic properties. Another example is the work by Dupressoir et al. [10]. They write protocols in the C programming language and apply the same invariant-based method in VCC [7], a general-purpose verifier for C source code. Hybrid approaches have also been proposed: synthesize code from verified abstract protocol specifications [1] or extract a model from protocol implementations and verify the model [3].

In the end, all of these approaches have as an objective the development of cryptographic software with high security assurances. However, existing cryptographic libraries are often written in a lower-level programming language (e.g. the C programming language) and routinely require security patches due to newly discovered bugs. Even very recently, severe flaws were discovered in different cryptographic libraries (e.g. OpenSSL, GnuTLS, SChannel and SecureTransport) that completely broke their security goals. To increase our trust in these big chunks of security critical software, verification is essential.

In this paper we propose a static verification approach for existing cryptographic protocol implementations written in the C programming language. Our approach was developed in VeriFast [12], a separation logic-based verifier for C programs, but can naturally be ported to other similar tools. Initially, VeriFast did not directly support the verification of protocols. As a first step to overcome this limitation, we chose to verify protocols within a symbolic model and leave verifying within a computational model for future work. Since we target existing implementations in which the bytes calculated by the cryptographic primitives are visible to the protocol participants, we could not apply a classical symbolic

model. Instead, we started from a classical Dolev-Yao model and extended it in order to give sensible contracts to the primitives.

The extended symbolic model we propose demands a manual code review. For this reason we implemented a verified library with a high-level Dolev-Yao style API on top of the low-level annotated cryptographic primitives. When writing protocols against this high-level API the manual code review is not required anymore, since the library implementation contains the code that needs to be reviewed. In rest of this paper, we explain our extended symbolic model of cryptography and discuss a methodology to:

- give meaningful and useful formal contracts to the cryptographic primitives of an existing cryptographic library
- prove security properties of implementations that use those primitives

Concretely, in Section 2 we describe our extended symbolic model and discuss which contracts we propose for the primitives of an existing cryptographic library (i.e. PolarSSL). Then in Section 3 we illustrate how to prove, besides memory safety, interesting security properties of protocols that are implemented with these primitives. Section 4 describes our library with a simple and high-level Dolev-Yao style API that illustrates the usability of the proposed contracts, and finally, we give our conclusions and discuss future work in Section 5.

2 Extended symbolic model of cryptography

Before we address our extended symbolic model of cryptography, we need to discuss the language in which we formalized it. For our cryptographic proofs we rely on VeriFast [12], a general-purpose verifier for C programs. To check memory safety and functional correctness with symbolic execution, VeriFast requires these programs to be annotated with preconditions and postconditions written in a separation logic-based specification language (isolated from normal code by special comments `//@...` or `/*@...*@/` and using keyword `&*&` for the separating conjunction). This specification language allows to define inductive datatypes, primitive recursive pure functions over those datatypes, abstract separation logic predicates, and lemma functions (i.e. pure functions that serve as proofs).

While other general-purpose verifiers could as well be used to employ our extended symbolic model of cryptography, extracts of definitions and examples shown are in VeriFast syntax. In the remainder of this text we focus on the generation of keyed hashes. Complete definitions and all examples (including symmetric encryption, asymmetric encryption and the generation of hashes or random values) can be found in the `examples/crypto` folder of the latest VeriFast release downloadable on the website http://distrinet.cs.kuleuven.be/software/VeriFast.

2.1 Cryptographic Terms

We start the discussion of our extended symbolic model of cryptography by defining the terms of our cryptographic algebra. These terms, which we interchange-

List. 1. Definition of the inductive datatype `cryptogram_t`

```
/*@ inductive cryptogram_t =
    | key_cg (int principal, int count)
    | hmac_cg(int principal, int count, list<char> pay)
    | ...;                                                    @*/
```

ably call cryptograms[1], are instances of the inductive datatype `cryptogram_t` defined in Listing 1. The first constructor `key_cg` introduces terms that represent keys for symmetric encryption and keyed hash generation. Parameters `principal` and `count` together serve as unique identifiers for keys (e.g. the 5th key generated by the 3rd principal). The second constructor introduces terms that represent keyed hashes. While the first two parameters identify which key was used, the third parameter represents the payload. The type of this parameter `pay` suggests a deviation from standard symbolic models: it has the type `list<char>` instead of `cryptogram_t`. So the definition of `cryptogram_t` is not recursive and the actual bits that were provided to the keyed hash primitive will be recorded in the corresponding constructor as a character list. Another deviation from standard symbolic models is that there is no constructor for pairing two cryptograms. In Subsection 2.4 we discuss how to combine cryptograms.

2.2 Linking Memory Regions to Terms

Since we want to verify executable protocol implementations, we need a way to link in-memory cryptographic results to their corresponding idealized abstract terms. To achieve this we introduce the definitions from Listing 2. For each instance of `cryptogram_t` the pure function `chars_for_cg` returns the exact character representation. Although this declaration has no body, its function values will be determined by the results of the cryptographic primitives during symbolic execution. Linking a memory region to its corresponding term is then established by an assertion of the form `cryptogram_p(buffer, len, cs, cg)`. The body of the predicate `cryptogram_p` states that `buffer` points to a correctly allocated memory region of size `len` and its contents is `cs`, the character representation of the cryptogram `cg`. Axiom `chars_for_hmac_inj` finally, ensures that `chars_for_cg` is injective for signed hashes up to collisions (for each kind of cryptogram such an axiom is added). A collision occurs (i.e. `collision` equals `true`) when different cryptograms of the same kind have an identical character representation in a specific symbolic execution branch.

2.3 Cryptographic Primitives

One of our goals was to come up with sensible contracts for the primitives of an existing cryptographic library. We chose the cryptographic primitives of PolarSSL (recently rebranded to mbed TLS), a simple library that implements a minimal but complete TLS stack. Instances of the inductive datatype `cryptogram_t` from Listing 1 represent the cryptographic values that are generated by the primitives of PolarSSL. The complete annotated declaration of the

[1] We use this term not only for cyphertexts, but for any value generated by some cryptographic primitive.

List. 2. Definitions to link cryptographic terms to allocated memory

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

    predicate cryptogram_p(char* buffer, int len,
                           list<char> cs, cryptogram_t cg) =
      chars(buffer, len, cs) &*& cs == chars_for_cg(cg) &*& ... ;

    lemma void chars_for_hmac_inj(cryptogram_t cg1, cryptogram_t cg2);
      requires cg1 == hmac_cg(_, _, _) &*& cg2 == hmac_cg(_, _, _) &*&
               chars_for_cg(cg1) == chars_for_cg(cg2);
      ensures  collision || (cg1 == cg2);                              @*/
```

List. 3. Annotated declaration of `sha512_hmac`

```
void sha512_hmac(const char *key, size_t keylen, const char *input,
                 size_t ilen, char *output, int is384);
/*@ requires [?f1]chars(key, keylen, ?cs_key) &*&
             [?f2]chars(input, ilen, ?cs_pay) &*&
             chars(output, ?olen, _) &*&
               key_id(?p, ?c) &*& ilen >= MIN_INPUT &*&
               is384 == 0 ? olen == 64 : is384 == 1 && olen == 48; @*/
/*@ ensures  [f1]chars(key, keylen, cs_key) &*&
             [f2]chars(input, ilen, cs_pay) &*&
             cs_key == chars_for_cg(key_cg(p, c)) ?
               cryptogram_p(output, olen, ?cs, ?h_cg) &*&
               h_cg == hmac_cg(p, c, cs_pay)
             :
               chars(output, olen, _); @*/
```

keyed hash primitive `sha512_hmac` is shown in Listing 3. It illustrates the usage of our earlier definitions to specify a contract for a cryptographic primitive. The contracts for other primitives, although not discussed here, are analogous.

Assertions of the form `chars(_,_,_)` in the contract of `sha512_hmac` record the required read access for input parameters and write access for output parameters. The `key_id(?p, ?c)` assertion[2] merely serves to bind the input variables `p` and `c`, and the switch `is348` determines if the function calculates keyed hashes with a size of 48 bytes or 64 bytes. If the provided input buffer `key` of size `keylen` contains the character representation of the term `key_cg(p,c)`, the function `sha512_hmac` returns the corresponding keyed hash in the output buffer. The content of this output buffer is determined by the assertion `cryptogram_p(output,olen,?cs,?h_cg)` and the postcondition then links the content of the output buffer to the cryptogram `hmac_cg(p, c, cs_pay)`, where `cs_pay` is the content of the input buffer.

2.4 Constructing Messages

A distinguishing feature of our extended symbolic model is the way that cryptographic terms, generated by the primitives, can be composed into messages. Because we are augmenting an existing cryptographic library that works with character buffers as inputs and outputs to primitives, we cannot hide the character representation of a term behind the annotated trusted API as in other

[2] Note: $p(?x)$ is VeriFast syntax for $\exists x.p(x)$.

List. 4. Definitions to combine cryptographic terms

```
/*@ fixpoint list<cryptogram_t> cgs_exposed(list<char> cs);

    lemma void cg_exposed(cryptogram_t cg);
      requires true;
      ensures  cgs_exposed(chars_for_cg(cg)) == {cg};

    fixpoint bool cgs_exposed_bound(list<char> cs,
                                    list<cryptogram_t> cgs);

    lemma void cgs_exposed_to_bound(list<char> cs);
      requires true;
      ensures  true == cgs_exposed_bound(cs, cgs_exposed(cs));

    lemma void cgs_exposed_bound_split(list<char> cs,
                                       list<cryptogram_t> cgs, int i);
      requires 0 <= i && i <= length(cs) &*&
               true == cgs_exposed_bound(cs, cgs);
      ensures  true == cgs_exposed_bound(take(i, cs), cgs) &*&
               true == cgs_exposed_bound(drop(i, cs), cgs);

    lemma void cgs_exposed_bound_join(
                              list<char> cs1, list<cryptogram_t> cgs1,
                              list<char> cs2, list<cryptogram_t> cgs2);
      requires true == cgs_exposed_bound(cs1, cgs1) &*&
               true == cgs_exposed_bound(cs2, cgs2);
      ensures  true == cgs_exposed_bound(append(cs1, cs2),
                                         union(cgs1, cgs2));         @*/
```

approaches [4, 10]. Therefore, messages are simply character buffers and we introduce the notion of the set of cryptograms exposed by a list of characters, embodied in the pure function `cgs_exposed` from Listing 4. Axiom `cg_exposed` then allows one to show this set is the expected singleton for the character representation of a cryptogram. We use a finite `list` representation for this set, which is reasonable as we only consider finite prefixes of protocol runs and only a finite number of cryptograms can be generated in a finite run.

Since the character representation of cryptograms is exposed by the PolarSSL API, we want to allow an adversary to split and join these character buffers at will and perform any other operation on them (giving an adversary more capabilities than in other symbolic models). For this reason, we introduce the notion of an upper bound on the set of cryptograms exposed by a list of characters as expressed by `cgs_exposed_bound`. Axiom `cgs_exposed_to_bound` then enables to show the set of cryptograms exposed by a list of characters indeed forms an upper bound. Finally, the axioms `cgs_exposed_bound_split` and `cgs_exposed_bound_join` allow to split and join lists of characters while tracking an upper bound on the cryptograms exposed. Axioms can easily be added for other operations. In general, for any operation f, an axiom can be provided that states $\texttt{cgs\_exposed}(f(cs)) \subseteq \texttt{cgs\_exposed}(cs)$.

2.5 Invariants for Public Messages

To show the correctness of a protocol implementation, one needs to specify which messages are confidential and which can safely be published on the untrusted

List. 5. Definitions for specifying invariants on public messages

```
/*@ predicate network(predicate(cryptogram_t) pub);

    lemma void network_init(predicate(cryptogram_t) pub);
      requires true;
      ensures  [_]network(pub) &*& ...;

    predicate public_message(predicate(cryptogram_t) pub,
                             char* chars, int len, list<char> cs) =
      chars(chars, len, cs) &*&
      [_]foreach(cgs_exposed(cs), pub) &*& ...;                    @*/

int net_send(void *ctx, const char *buf, size_t len);
  /*@ requires [_]network(?pub) &*&
               public_message(pub, buf, len, ?cs) &*& ...; @*/
  /*@ ensures  public_message(pub, buf, len, cs) &*& ...; @*/

int net_recv(void *ctx, char *buf, size_t len);
  /*@ requires [_]network(?pub) &*& ... &*&
               chars(buf, len, _); @*/
  /*@ ensures  result <= 0 ?
                 chars(buf, len, _)
               :
                 result <= len &*&
                 chars(buf + result, len - result, _) &*&
                 public_message(pub, buf, result, _); @*/
```

network without interfering with the protocol. Those that are safe to publish are messages constructed and transmitted by honest participants during a valid protocol run, or messages that are produced by the attacker. Determining these messages takes the form of an invariant for messages on the network: they can only expose *public* cryptograms. We allow one to indirectly define this invariant for a specific protocol through a publicness predicate with only one parameter of type `cryptogram_t`. A cryptogram is then public if and only if it satisfies the assertion in the body of the publicness predicate. One can make use of events [4, 5, 16] [3] to record protocol progress while defining this publicness predicate. Note that publicness is a crude measure since a cryptogram should be non-public even if a single bit is confidential in the protocol at hand.

Listing 5 shows the networking API. We assume that messages are published on the network only through this API or other similarly annotated APIs. Before the networking API can be used, it must be initialized with the publicness predicate definition for the protocol at hand. This is done through an invocation of the function `network_init` where the publicness predicate is passed as an argument. An assertion of the from `[_]network(pub)` then binds that predicate value in the postcondition of the initialization function. For a specific publicness predicate, public messages are determined by the predicate `public_message`. A public message is a memory buffer that contains a list of characters exposing a set of cryptograms for which each member is public according to the given publicness predicate. The contracts for the functions from the networking API

[3] Events are called event predicates in other approaches, but we call them events here to avoid confusion with separation logic predicates.

List. 6. Extract from attacker model

```
/*@ typedef lemma void public_hmac(predicate(cryptogram_t) pub)
                                   (cryptogram_t hmac);
      requires  hmac == hmac_cg(?p, ?c, ?pay) &*&
                  length(pay) <= INT_MAX &*&
                [_]pub(key_cg(p, c)) &*&
                [_]foreach(cgs_exposed(pay), pub) &*& ...;
      ensures   [_]pub(hmac) &*& ...;                                  @*/

void attacker(...);
  /*@ requires [_]network(?pub) &*&
               is_public_hmac(?proof, pub) &*& ...; @*/
  /*@ ensures  is_public_hmac(proof, pub) &*& ...; @*/
```

then enforce the invariant that only public messages can be sent over the network: the `net_send` function requires a public message and a successful call of the `net_recv` function returns a public message.

2.6 Attacker Model

Another element of our extended symbolic model is the attacker. As mentioned earlier, the attacker is able to look inside a public message and see the individual bytes. This means he can perform any operation on public messages, such as splitting a public message into two separate messages and joining two messages at will using the definitions from Listing 4 and Listing 5. This renders our attacker model more powerful than most other symbolic models [10, 4] (e.g. he can split a nonce in two parts and publish both on the network). Besides these capabilities, the attacker can call any cryptographic primitive using some bytes he finds on the network or comes up with himself.

All capabilities of the attacker are encoded as theorems about the publicness predicate and specifically, without going into details of the VeriFast annotation language, as lemma function type definitions. Definition `public_hmac` in Listing 6 is an example of such a theorem. The contract of `public_hmac` loosely states that if the attacker pulled a key and a public message from the network, the calculated keyed hash of that message must also be public. To verify the attacker implementation `attacker`, one needs to write, for each capability theorem about the publicness predicate, a lemma function serving as proof. Once a lemma function is written, VeriFast allows for a witness of its existence to be generated. For the theorem `public_hmac` and the publicness predicate `pub`, this witness is encoded as the assertion `is_public_hmac(?proof, pub)`. All capability theorems must be proven before invoking the attacker implementation. Using these proofs for a specific publicness predicate, VeriFast can verify the annotated attacker implementation for the corresponding protocol.

2.7 Handling Data Values

As described in Subsection 2.5, exclusively public messages (i.e. character buffers that expose only public cryptograms) are allowed on the network. But raw data buffers containing for example cleartexts, message tags or protocol version numbers, are not computed by some cryptographic primitive. So there is no assurance

List. 7. Axiom for assuming that a data value is public

```
/*@ lemma void assume_public_chars(predicate(cryptogram_t) pub,
                                    list<char> cs);
      requires true;
      ensures  [_]foreach(cgs_exposed(cs), pub) &*& ... ;            @*/
```

List. 8. Axiom to interpret a received message as a keyed hash

```
/*@ lemma cryptogram_t chars_for_hmac_cg_surj(list<char> cs);
      requires true;
      ensures  result == hmac_cg(_, _) &*&
               cs == chars_for_cg(result);                           @*/
```

about which cryptograms are exposed by their contents. Still, protocol participants should be able to transmit them over the network. For this reason, we introduce the axiom `assume_public_chars` shown in Listing 7. It allows one to introduce the assumption that a specific list of characters only exposes public cryptograms, which is the weakest requirement for data values to send them on the network. This axiom has to be treated with uttermost care. It must be invoked only on literal data hardcoded into the program text, user input or other data that is independent from any secret coin tosses. Unfortunately VeriFast has no support for the kind of taint analysis that is required for checking this constraint, so it must be checked by hand (which should be straightforward for well-written code). Additionally, a manual audit must be performed to ensure that there are no implicit flows from secret bits into the program's control flow; i.e., the audit must check that the condition of each if statement, while statement, or similar statements exposes only public cryptograms.

2.8 Interpreting a Received Public Message

Each protocol defines its own message format and during a protocol run each received message is parsed according to this format. After parsing a received message, certain parts need be interpreted as the character representation of some cryptogram. Therefore we add, for each type of cryptogram, an axiom analogous to `chars_for_hmac_cg_surj` as shown in Listing 8. Such an axiom allows to show that a specific list of characters is the character representation of some cryptogram, without knowing the exact cryptogram. Then, by the contract of `net_recv` from Listing 5 we know that whichever cryptogram it is, it must be public. In this way knowledge about the received message can be extracted.

Finally, proofs by induction on cryptograms allow one to prove interesting properties of received messages and protocols. Since `cryptogram_t` is not defined recursively, proofs by induction have to be supported axiomatically. For this, we assign a level to each cryptogram and cryptograms without payload have zero as level. For a cryptogram `cg` with a payload, we add an axiom stating that the level of a cryptogram exposed by the payload `cg` is lower than the level of `cg` itself. We ensure that this axiom (which is not shown), can only be invoked on cryptograms that are actually generated (to break cryptographic cycles).

3 Memory safety and security properties

The previous section discussed all the elements of our extended symbolic model of cryptography and how we encoded it in VeriFast. Here we show how to apply that model to prove memory safety and interesting security properties of a specific protocol implementation. To verify a protocol implementation the following steps (not necessarily in this order) have to be performed:

- Define events to record protocol progress
- Define the publicness predicate for the protocol
- Prove theorems about the publicness predicate required to verify the attacker
- Give a contract to all participants of the protocol
- Verify the participant implementations

The rest of this section illustrates these steps with a simple example: an authenticated remote procedure call (RPC) protocol. A protocol transcript for RPC is given in Figure 1 where `key` is assumed to be a secret shared between participants `A` and `B`. This transcript specifies that `A` first sends an authenticated request to `B` and then `B` sends his authenticated response (prefixed with the request) back. The actual data exchanged (i.e. `request` and `response`) is integrity protected, but not encrypted.

Fig. 1. Protocol transcript of RPC

Message 1	$A \rightarrow B$:	$\{request\}_{HMAC(key)}$
Message 2	$B \rightarrow A$:	$\{request, response\}_{HMAC(key)}$

Listing 9 shows the definitions for events and the definition of the publicness predicate of RPC. For convenience, messages are tagged with their sequence number, and the request and the response messages have a fixed size of `LEN` bytes. The two events `request` and `response` are defined as bodyless pure functions with a boolean return value. Event `request` represents the fact that principal `A` wants to send `req_pay` to `B` and `response` is the event in which `B` wants to send `resp_pay` in response to `A`'s request. The definition of the publicness predicate `rpc_pub` uses two more pure functions that have a clear meaning. Function `bad` is true for dishonest principals that have compromised keys (e.g. the attacker) and for a specific key, the function `shared_with` returns the principal with whom the creator shared the key. Since the exchanged data is solely integrity protected through a keyed hash, the only constructors of type `cryptogram_t` that are of interest in the definition of `rpc_pub`, are `key_cg` and `hmac_cg`. The boolean formula for the constructor `key_cg` states that a key is considered public in RPC if the creator of the key or the participant with whom the creator shared the key is dishonest. According to the constructor `hmac_cg` there are three cases in which a keyed hash is public in RPC:

1. The key that was used to generate the keyed hash is public
2. The payload `cs` of the keyed hash is a tagged message that `principal` wanted to send to `shared_with(principal, count)`
3. The payload `cs` is a tagged message that `shared_with(principal, count)` wanted to send in response to a request from `principal`

List. 9. Definitions for Verification of RPC Protocol

```
/*@ fixpoint bool  request(int A, int B, list<char> req_pay);
    fixpoint bool response(int A, int B, list<char> req_pay,
                           list<char> resp_pay);

    fixpoint bool bad(int principal);
    fixpoint int shared_with(int principal, int count);

    predicate rpc_pub(cryptogram_t cg) =
      switch (cg)
      {
        case key_cg(principal, count): return
          bad(principal) || bad(shared_with(principal, count));
        case hmac_cg(principal, count, cs): return
          bad(principal) || bad(shared_with(principal, count)) ||
          switch (cs)
          {
            case cons(c0, cs0): return
              c0 == '1' ?
                request(principal, shared_with(principal, count), cs0)
            : c0 == '2' ?
                response(principal, shared_with(principal, count),
                                    take(LEN, cs0), drop(LEN, cs0))
            : false;
            case nil: return false;
          };
        case ...
      }; @*/
```

The only nontrivial attacker capability for RPC is the one concerning keyed hashing from Listing 6 and the theorem representing that capability is proven straightforwardly for the definition of `rpc_pub`. A contract for the implementation of participant `A` is then given in Listing 10. This contract expresses that participant `A` needs its key, some public request and a buffer to store the response together with the event that the application wants to send the request. After successful completion of the protocol, the postcondition ensures that participant `A` received a public response and that if neither participant `A` nor the receiver of the request are `bad`, then the event of the receiver wanting to respond with the received message occurred. We successfully verified the annotated implementation of both protocol participants (which are not shown).

We implemented and verified several additional protocols against the annotated PolarSSL API. Figure 2 gives an overview of the small verified protocol suite. The source lines of code (SLOC), annotation lines of code (ALOC), the ratio of ALOC to SLOC and the verification times (VTime) are given there. Considering that we are verifying software with complex properties, these ratios are quite low compared to other software verified with VeriFast (because many complex definitions and proofs are incorporated in the annotated API). Since VeriFast sets out to be an interactive verification tool, verification times are predictable and pretty low in general. However, the verification times for the protocol implementations shown in Figure 2 are quite long for VeriFast standards. The reason for this is that a lot of branching occurs during symbolic execution due to nested case analyses.

List. 10. Implementation of principal A for RPC

```
void A(char *key, int key_len, char *request, char *response)
  /*@ requires [_]network(rpc_pub) &*&
               [?f1]cryptogram_p(key, key_len, ?k_cs, ?k_cg) &*&
                 k_cg == key_cg(?A, ?c) &*&
               [?f2]public_message(rpc_pub, request, LEN, ?req_cs) &*&
               request(A, shared_with(A, c), req_cs) == true &*&
               chars(response, LEN, _); @*/
  /*@ ensures  [f1]cryptogram_p(key, key_len, k_cs, k_cg) &*&
               [f2]public_message(rpc_pub, request, LEN, req_cs) &*&
               public_message(rpc_pub, response, LEN, ?resp_cs) &*&
               bad(A) || bad(shared_with(A, c)) ||
               response(A, shared_with(A, c), req_cs, resp_cs); @*/
{ ... }
```

Fig. 2. Metrics of verified protocols using the Polarssl API

Protocol	**SLOC**	**ALOC**	**ALOC/SLOC**	**VTime**
Secure Storage	300	153	2.0	3.3s
Authenticated RPC	486	173	2.8	2.9s
Encrypt and HMAC	391	203	2.1	7.0s
Authenticated Encryption	358	183	2.0	5.7s

4 Dolev-Yao style API

Using the low-level cryptographic primitives of the PolarSSL API which we augmented with contracts, demands a manual code review as discussed in Subsection 2.7. Although for well-written protocols this code review should be modest, we propose here an alternative approach that does not require manual review. For this reason and to illustrate the usability of our extended symbolic model, we implemented a verified library with a simple Dolev-Yao style API on top of the annotated PolarSSL API. When writing protocols against this high-level API the manual code review is not required, since the verified library implementation contains the code that needs to be reviewed.

The Dolev-Yao style API provides the same cryptographic functionality and comes with a Dolev-Yao style attacker implementation that uses only the cryptographic primitives from this high-level API to inspect and construct messages. To retain our stronger attacker model however, this high-level attacker also invokes the lower-level PolarSSL attacker. The actual characters of the values calculated by the cryptographic primitives are not exposed to the principals (including the high-level attacker) of a protocol. They are hidden behind the declared, but not defined, C structure `item` from Listing 11. The terms of the cryptographic algebra in our Dolev-Yao style API are items, i.e. members of the type `item_t`. Notice that this definition is recursive, in contrast to the definition of `cryptogram_t` from Listing 1. Besides the constructors `key_item` and `hmac_item`, the constructors `data_item` and `pair_item` are required here to, respectively, introduce a term from raw data or to combine two terms into one with a reversible encoding. Indeed, the concatenation of two character buffers is again a character buffer, but there is no trivial way to combine two instances of an undefined C structure into one instance.

List. 11. Extract from Dolev-Yao style API

```
struct item;

/*@ inductive item_t =
    | data_item(list<char> data)
    | pair_item(item_t first, item_t second)
    | key_item (int principal, int count)
    | hmac_item(int principal, int count, option<item_t> payload)
    | ... ;

predicate item_p(struct item *item, item_t i);

predicate world(predicate(item_t) pub);                                @*/

struct item *create_data_item(char* data, int length);
  /*@ requires [?f]world(?pub) &*&
               chars(data, length, ?cs) &*& length > 0; @*/
  /*@ ensures  [f]world(pub) &*&
               chars(data, length, cs) &*&
               item_p(result, data_item(cs)); @*/

struct item *create_hmac(struct item *key, struct item *payload);
  /*@ requires [?f]world(?pub) &*&
               item_p(payload, ?pay) &*& item_p(key, ?k) &*&
               k == key_item(?principal, ?count); @*/
  /*@ ensures  [f]world(pub) &*&
               item_p(payload, pay) &*& item_p(key, k) &*&
               item_p(result, ?hmac) &*&
               collision() ?
                 true
               :
                 hmac == hmac_item(principal, count, some(pay)); @*/
```

Like before a predicate (i.e. `item_p`) is used to link an in memory representation of a cryptographic term (i.e. an instance of the C structure `item`) to the term itself (i.e. an instance of the inductive datatype `item_t`). These items are simply the messages that can be sent over the network and public messages are determined by defining a publicness predicate with a single parameter of type `item_t`. Once the Dolev-Yao style API is initialized, the predicate `world` is used to bind the provided publicness predicate. The function `create_data_item` allows to create a data item from raw bytes. Its contract specifies that if the input buffer has content `cs` the resulting item is represented by the term `data_item(cs)`. The function `create_hmac_item` is a primitive to generate a keyed hash. As one can see, the recursive definition of `item_t` greatly simplifies the specification of a contract for cryptographic primitives. There is no notion of the character representation of a cryptographic value. The postcondition of `create_hmac_item` simply states that (if no collision occurs), given a key item and some other item, the returned item is the signed hash of that other item (i.e. `hmac_item(principal,count,some(pay))`).

Figure 3 gives an overview of the small verified protocol suite we created using the Dolev-Yao style API. The most interesting element here is the high-level API implementation. As one can see the ALOC to SLOC ratio is quite high compared to the other protocol implementations. This is the case because a lot

Fig. 3. Metrics of verified protocols using the Doley-Yao style API

Program	SLOC	ALOC	ALOC/SLOC	VTime
High-Level API implementation	1522	5233	3.4	43.4s
Secure Storage protocol	115	225	2.0	2.3s
Authenticated RPC protocol	184	368	2.0	3.8s
Yahalom protocol	303	932	3.1	2.9s
Needham-Schroeder-Lowe protocol	305	805	2.6	4.4s

of complexity (including the verified code that needs to be manually reviewed) is hidden behind the API. The metrics for the other protocols are comparable with the ones from Figure 2 and although also the ALOC to SLOC ratios are comparable, the annotations themselves are significantly more straightforward.

5 Conclusions and Future Work

In this paper, we described an extended symbolic model of cryptography. This extended model was the result of our efforts to give sensible and meaningful contracts to the cryptographic primitives of an existing cryptographic library (i.e. PolarSSL). We showed the immediate usability of these contract by writing verified protocols using the annotated cryptographic primitives.

Since the attacker of our extended symbolic model can look at the individual bytes of cryptographic values generated by the primitives, we argued that our attacker is more powerful than in a standard symbolic model. However, our embedding of this extended symbolic model in the VeriFast annotation language, requires a manual code review for verified protocols. For this reason, we created a verified Dolev-Yao style library on top of the low-level cryptographic primitives. This library does not only illustrate the usability of our contracts for the cryptographic primitives of PolarSSL, it also removes the burden of performing a manual code review from the user since the library itself contains the code that needs to be reviewed. At this time, we did not create a formal model of our extended symbolic model of cryptography and prove its soundness. This is left for future work.

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 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. J. B. Almeida, M. Barbosa, G. Barthe, and F. Dupressoir. Certified computer-aided cryptography: efficient provably secure machine code from high-level imple-

mentations. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & communications security*, CCS '13, pages 1217–1230, New York, NY, USA, 2013. ACM.

2. G. Barthe, B. Grégoire, S. Heraud, and S. Z. Béguelin. Computer-aided security proofs for the working cryptographer. In *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings*, pages 71–90, 2011.
3. K. Bhargavan, C. Fournet, R. Corin, and E. Zălinescu. Verified cryptographic implementations for tls. *ACM Trans. Inf. Syst. Secur.*, 15(1):3:1–3:32, March 2012.
4. K. Bhargavan, C. Fournet, and A. D. Gordon. Modular verification of security protocol code by typing. In *37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'10)*, pages 445–456, 2010.
5. B. Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *In 14th IEEE Computer Security Foundations Workshop (CSFW-14*, pages 82–96. IEEE Computer Society Press, 2001.
6. B. Blanchet. A computationally sound mechanized prover for security protocols. In *IEEE Symposium on Security and Privacy*, pages 140–154, 2006.
7. 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.
8. D. E. Denning and G. M. Sacco. Timestamps in key distribution protocols. *Commun. ACM*, 24(8):533–536, August 1981.
9. D. Dolev and A. C. Yao. On the security of public key protocols. Technical report, Stanford, CA, USA, 1981.
10. 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.
11. C. Fournet, M. Kohlweiss, and P.-Y. Strub. Modular code-based cryptographic verification. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 341–350, New York, NY, USA, 2011. ACM.
12. B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. VeriFast: a powerful, sound, predictable, fast verifier for c and java. In *Proceedings of the Third international Conference on NASA Formal methods*, NFM '11, pages 41–55, Berlin, Heidelberg, 2011. Springer-Verlag.
13. R. Küsters, T. Truderung, and J. Graf. A Framework for the Cryptographic Verification of Java-like Programs. In *IEEE Computer Security Foundations Symposium, CSF 2012*, pages 198–212. IEEE Computer Society, 2012.
14. G. Lowe. An attack on the needham-schroeder public-key authentication protocol. *Inf. Process. Lett.*, 56(3):131–133, November 1995.
15. R. M. Needham and M. D. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, December 1978.
16. L. C. Paulson. The inductive approach to verifying cryptographic protocols. *J. Comput. Secur.*, 6(1-2):85–128, January 1998.