Verifying Cryptographic Code in C: Some Experience and the Csec Challenge

Mihhail Aizatulin François Dupressoir
Andrew D. Gordon Jan Jürjens

November 2011

Technical Report
MSR–TR–2011–118

Microsoft Research
Roger Needham Building
7 J.J. Thomson Avenue
Cambridge, CB3 0FB
United Kingdom

An abridged form of this report appears in the proceedings of the *8th International Workshop on Formal Aspects of Security and Trust*, September 15–16, 2011, Leuven, Belgium.

Verifying Cryptographic Code in C: Some Experience and the Csec Challenge

Mihhail Aizatulin[1], François Dupressoir[1],
Andrew D. Gordon[2], and Jan Jürjens[3]

[1] The Open University
[2] Microsoft Research and University of Edinburgh
[3] TU Dortmund and Fraunhofer ISST

Abstract. The security of much critical infrastructure depends in part on cryptographic software coded in C, and yet vulnerabilities continue to be discovered in such software. We describe recent progress on checking the security of C code implementing cryptographic software. In particular, we describe projects that combine verification-condition generation and symbolic execution techniques for C, with methods for stating and verifying security properties of abstract models of cryptographic protocols. We illustrate these techniques on C code for a simple two-message protocol.

1 Introduction

We describe our experience of verifying security properties of cryptographic software in C. This problem is far from solved, but we approach it in the context of much recent progress on theories and tools for reasoning about cryptographic protocols and their implementations.

The plan of this article (and the invited talk it accompanies) is to explain two different approaches to the problem in the setting of a simple example. Section 2 describes this example, a simple client-server protocol, introduced by Fournet et al. (2011b), which relies on authenticated encryption to achieve both authentication and secrecy, and outlines the structure of our C programs for the client and server roles of the protocol.

In Section 3 we describe a method (Aizatulin et al. 2011b) for extracting abstract models of cryptographic code by symbolic execution. The technique yields models that may be verified with ProVerif (Blanchet 2001) to obtain results in the symbolic model of cryptography. For some protocols, we may appeal to the CoSP framework (Backes et al. 2009) to obtain computational soundness.

Next, in Section 4, we describe a method (Dupressoir et al. 2011) for applying a general-purpose C verifier, specifically VCC (Cohen et al. 2009), to proving protocol properties using the method of invariants for cryptographic structures (Bhargavan et al. 2010), a method developed originally for functional code using the F7 refinement-type checker (Bengtson et al. 2008). For this second method, we obtain results only in the formal model (although we have work underway on

recasting in VCC recent techniques (Fournet et al. 2011b) for directly obtaining computational guarantees via F7).

Section 5 discusses related work and Section 6 concludes.

Our verification work assumes correctness of the code for the underlying cryptographic algorithms; others have addressed how to verify code of such algorithms (Erkök et al. 2009; Barbosa et al. 2010). Instead, our concern is to check the correct usage of cryptographic algorithms so as to ensure security properties of protocols and devices.

We have made our code, our verification tools, and our verification results available on the web. A package at `http://research.microsoft.com/csec` includes the source code for our example and the logs from running our tools, and also, to replicate our results, instructions for first downloading our tools and their dependencies, and then re-running verification.

Moreover, as described in Section 6, we have launched a companion website, the *Csec Challenge*, to curate examples of cryptographic code, including protocols and software for hardware tokens, as a basis for evaluating different verification techniques.

2 Example: Encryption-Based Authenticated RPC

We define an example protocol, together with a deliberately informal statement of security properties. In later sections we give a specific interpretation of these properties, for each of our two verification techniques.

2.1 Protocol Description and Security Properties

We consider a protocol, due to Fournet et al. (2011b), that is an encryption-based variant of the RPC protocol considered in previous papers (Bhargavan et al. 2010; Dupressoir et al. 2011; Aizatulin et al. 2011b). In the following, the curly braces $\{m\}_k$ stand for the encryption, using an authenticated encryption mechanism, of plaintext m under key k while the comma represents an injective pairing operation in infix form.

We consider a population of principals, ranged over by A and B (and later, in code, by `a` and `b`). The following protocol narration describes the process of A in client role communicating to B in server role.

Authenticated RPC: RPC-enc

A : `event` $client\_begin(A, B, req)$
$A \to B$: $A, \{req, k_S\}_{k_{AB}}$
B : `event` $server\_reply(A, B, req, resp)$
$B \to A$: $\{resp\}_{k_S}$
A : `event` $client\_accept(A, B, req, resp)$

The key k_{AB} is a unidirectional long-term key shared between A and B, for A in client role, and B in server role. (Should B wish to play the client role, they

would rely on a key k_{BA}, distinct from k_{AB}.) The key k_S is the session key freshly generated by A and the payloads req and $resp$ come from the environment.

Our attacker model is a network-based adversary, able to receive, rewrite, and send messages on the network. When considering security properties, we allow the possibility that one or more long-term keys k_{AB} is *compromised*, that is, is known to the attacker. We write $bad(A, B)$ to mean the key k_{AB} is compromised.

The top-level security properties we wish to establish are of two kinds:

(1) Authentication properties state that each principal can ensure that a received message was produced by the correct protocol participant before moving on to the next protocol steps (or that a long-term key between the principals has been compromised by the attacker). These properties are specified using event correspondences of the form:

$$server\_reply(A, B, req, resp) \Longrightarrow client\_begin(A, B, req) \vee bad(A, B)$$
$$client\_accept(A, B, req, resp) \Longrightarrow server\_reply(A, B, req, resp) \vee bad(A, B)$$

The first property states that, whenever the event *server_reply* happens, either the event *client_begin* has happened with corresponding parameters or the long-term key is compromised. Similarly, the second property states that, whenever the event *client_accept* happens, either the event *server_reply* has happened, or the long-term key is compromised.

(2) Key disclosure, or weak secrecy, properties state that keys are only disclosed to the attacker if a long-term key has been compromised. We can express this as follows:

$$attacker(k_{AB}) \vee attacker(k_S) \Rightarrow bad(A, B)$$

We do not consider secrecy of the payloads req and $resp$, because they are not generated at random and so formulating a secrecy property for them is more difficult.

2.2 Concrete Operations and Network Packet Formats

We describe our implementation of the pairing operator $(,)$ and authenticated encryption $\{m\}_k$ in practice, and give a byte-wise description of the format of network packets. The sign $|$ stands for bytestring concatenation, and len() denotes the partial function operating on bytestrings that returns the 4-byte network-order representation of the length of its argument, when possible. The length operation is undefined on bytestrings of length greater than $2^{32} - 1$, and therefore we restrict each message to be of at most that length.

Pairing The pairing operator is implemented uniformly, for simplicity, by concatenating a fixed one-byte tag `'p'`, followed by the 4-byte length of the first element of the pair, followed by the pair elements, in order. More formally, pairing is defined as follows.

$$(a, b) = \texttt{'p'} \,|\, \mathrm{len}(a) | a | b$$

Authenticated Encryption To implement authenticated encryption as a function of a key and a plaintext, we use AES in Galois/Counter Mode (GCM) (McGrew and Viega 2005), and generate a random 16-byte long initialization vector (IV) that is used by the primitive call and prepended to the ciphertext. To provide authentication, we append the 16-byte authentication tag produced by the call to the encryption primitive. In the following formal description, we represent the primitive call to AES in GCM mode using a function `AES_GCM_enc`() that takes the key, the plaintext, and the initialization vector as arguments, and returns the output IV, the ciphertext and the 16-byte authentication tag. We also assume a random function that outputs n random bytes, where n is passed in as an argument.

$$\begin{aligned}\{m\}_k = &\texttt{let } IV = \texttt{random}(16) \texttt{ in}\\ &\texttt{let } \_, ctxt, tag = \texttt{AES\_GCM\_enc}(k, m, IV) \texttt{ in}\\ &IV|ctxt|tag\end{aligned}$$

Network Packet Formats To simplify the modelling of network operations, all messages sent over the network are sent in two stages: first, the length is sent in the clear, expressed as four bytes in network order, and then the message itself, its length being known, can be retrieved from the network by iterating until the expected number of bytes is received.

The following table gives a byte-by-byte description of the first message, without the preceding total length:

First Byte[length]	1[1]	2[4]	6[len(A)]	$6 + \text{len}(A)$ [len(req, k_S) + 32]
Contents	'p'	len(A)	A	$\{req, k_S\}_{k_{AB}}$

The second message is simply $\{resp\}_{k_S}$, preceded by its four byte length.

2.3 Implementation

We implement the protocol in C in about 700 lines of code calling to a GCM library (McGrew and Viega 2005) for encryption and to PolarSSL (PolarSSL) for random number generation and network communication. The implementation is executable, and consists of a client and a server that communicate through a TCP/IP connection. We list a trace of a protocol run below. A sample of the code appears in Fig. 3.

```
Server: Now listening on localhost, port 4433.
Server: Accepted client connection.
Client: Preparing to send request: What is the weather like?
and session key: 26427b9510a0285246030e957e25cea3
Client: Sending message: p | 9 | localhost | 6c509cb95d1e0628920006709d...
Server: Authenticated request: What is the weather like?
Server: Authenticated session key: 26427b9510a0285246030e957e25cea3
Server: Preparing response: Look out the window.
Server: Sending encrypted message: ab826de07c761dee8b...
Client: Received and authenticated response: Look out the window.
```

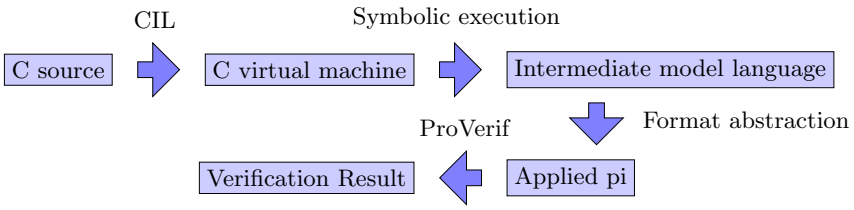

Fig. 1. An outline of model extraction

3 Verification by Model Extraction

In this section we describe a verification approach in which a high-level model is extracted from the code and verified using an existing tool, ProVerif (Blanchet 2001). The full details are described elsewhere (Aizatulin et al. 2011b).

Our starting point is that our implementation code typically contains three sorts of action: (1) configuration, (2) creating and parsing messages by direct memory manipulation, and (3) applying cryptographic primitives. This observation applies to other implementations, such as OpenSSL or PolarSSL. The memory manipulation code is not encapsulated and is intermingled with the application of cryptography.

Our intent is to extract the cryptographic core of the protocol, by eliminating the memory operations via symbolic execution of the C code (King 1976). We simplify configuration code as well, because we perform the verification for specific constant values of configuration parameters. The extracted model contains the cryptographic core of the protocol in the ProVerif modelling language, a form of the applied pi-calculus (Abadi and Fournet 2001), suitable for verification with ProVerif.

The method takes as input:

- The C implementations of the protocol participants, containing calls to a special function event. For instance, before creating the request the client in our example calls

```
event3("client_begin", clState.self, clState.self_len, clState.other,
    clState.other_len, clState.request, clState.request_len);
```

  This call executes the event $client\_begin(A, B, req)$ where A, B, and req are the contents of the buffers clState.self, clState.other, and clState.request. Security properties are stated in terms of correspondences of these events, as described in section 2.1.
- An environment process (in the modelling language) which spawns the participants, distributes keys, etc.
- Symbolic models of cryptographic functions used by the implementation. These models are themselves expressed in C via what we call *proxy functions*, explained in more detail in Section 3.2.
- An intended correspondence or secrecy property to be proved by ProVerif.

The verification steps are outlined in Fig. 1 and are explained by example in the following sections. The main limitation of the current method is that it deals

only with a single execution path (as determined by a concrete test run of the code). This limitation is mitigated by the observation that a great majority of protocols (such as those in the extensive SPORE repository (Project EVA 2007)) follow a fixed narration of messages between participants, where any deviation from the expected message leads to termination.

3.1 C Virtual Machine (CVM)

We start by compiling the program to a simple stack-based instruction language with random memory access (CVM, from "C Virtual Machine"). The language contains primitive operations that are necessary for implementing security protocols: reading values from the network or the execution environment, choosing random values, writing values to the network and signalling events.

Our implementation performs the conversion from C to CVM at runtime—the C program is instrumented using CIL (Necula et al. 2002) so that it outputs its own CVM representation when run. For example, the following are the CVM instructions corresponding to a call to malloc in the client:

```
// client.c:39
LoadStackPtr client.i:m1_len[3638]; LoadInt 8; SetPtrStep; LoadMem; Call malloc_proxy
```

3.2 Extracting an IML Model by Symbolic Execution

Next, we symbolically execute CVM programs to eliminate memory accesses and destructive updates, to obtain an equivalent program in an *intermediate model language* (IML). IML is the applied pi-calculus of ProVerif augmented with arithmetic operations and bytestring manipulation primitives: $b|b'$ is the concatenation of bytestrings b and b'; $b\{b_o, b_l\}$ is the substring of b starting at offset b_o of length b_l; and $\mathrm{len}(b)$ is the length of b (in bytes). A slightly simplified IML model of both the client and the server is shown in Fig. 2.

The key idea behind the model extraction algorithm is to execute the program in a symbolic semantics, in which memory locations are associated with symbolic expressions that describe how the contents of these locations were computed.

Memory locations are of two kinds: either (1) a stack location, $\mathrm{stack}\, v$, associated to a program variable v, or (2) a heap location, $\mathrm{heap}\, i$, for $i \in \mathbb{N}$ returned by a call to malloc. A pointer is represented symbolically as $\mathrm{ptr}(loc, e)$, that is, a location together with the symbolic offset relative to the beginning of the location. All pointer arithmetic is performed on the offset while the location remains fixed.

The variables in symbolic expressions represent unknown data obtained from the network, the program environment, or the random number generator. An application expression $op(e_1, \ldots, e_n)$ models computation. Operations can either be the basic operations of the language or cryptographic primitives. The language of symbolic expressions additionally contains the bytestring operations of the IML language.

The symbolic memory is a map from symbolic memory locations to symbolic expressions. We also maintain a set of known logical facts and an allocation

```
let A =
  event client_begin(clientID, serverID, request);
  new kS1;
  let msg1 = 'p'|len(request)|request|kS1 in
  let cipher1 = E(key(clientID, serverID), msg1) in
  out(c, 5 + len(clientID) + len(cipher1));
  out(c, 'p'|len(clientID)|clientID|cipher1);
  in(c, msg4);
  event client_accept(clientID, serverID, request, D(kS1, msg4));
let B =
  in(c, msg1);
  if 'p' = msg1{0, 1} then
  if len(msg1) ≤ 5 + msg1{1, 4} then
  let client1 = msg1{5, msg1{1, 4}} in
  let cipher1 = msg1{5 + msg1{1, 4}, len(msg1) − (5 + msg1{1, 4})} in
  let msg2 = D(key(client1, serverID), cipher1) in
  if 'p' = msg2{0, 1} then
  if len(msg2) ≤ 5 + msg2{1, 4} then
  let var2 = msg2{5, msg2{1, 4}} in
  event server_reply(client1, serverID, var2, response);
  let key1 = msg2{5 + msg2{1, 4}, len(msg2) − (5 + msg2{1, 4})} in
  let cipher2 = E(key1, response) in
  out(c, len(cipher2));
  out(c, cipher2);
```

Fig. 2. The IML model extracted from the C code.

table (a map from memory locations to length expressions) used for checking memory safety. As an example, the symbolic memory when entering the function send_request of the client looks as follows:

$$
\begin{aligned}
&\text{stack } ctx \Rightarrow \text{ptr}(\text{heap } 1,\ 0)\\
&\text{heap } 1 \Rightarrow \{\ \mathsf{request} = \text{ptr}(\text{heap } 2,\ 0),\ \mathsf{request\_len} = \text{len}(\mathit{request}),\\
&\qquad\qquad \mathsf{self} = \text{ptr}(\text{heap } 3,\ 0),\ \mathsf{self\_len} = \text{len}(\mathit{clientID}),\\
&\qquad\qquad \mathsf{other} = \text{ptr}(\text{heap } 4,\ 0),\ \mathsf{other\_len} = \text{len}(\mathit{serverID}),\\
&\qquad\qquad \mathsf{k\_s} = \text{ptr}(\text{heap } 5,\ 0),\ \mathsf{k\_s\_len} = \text{len}(k_S),\\
&\qquad\qquad \mathsf{k\_ab} = \text{ptr}(\text{heap } 6,\ 0),\ \mathsf{k\_ab\_len} = \text{len}(\mathit{key}(\mathit{clientID}, \mathit{serverID}))\}\\
&\text{heap } 2 \Rightarrow \mathit{request},\ \text{heap } 3 \Rightarrow \mathit{clientID},\ \text{heap } 4 \Rightarrow \mathit{serverID},\\
&\text{heap } 5 \Rightarrow k_S,\ \text{heap } 6 \Rightarrow \mathit{key}(\mathit{clientID}, \mathit{serverID})
\end{aligned}
$$

The parameter of the function is the structure ctx that holds pointers to values relevant to the protocol execution: the identities of both the client and the server (fields self and other), the value of the request (field request), as well as fields k_ab and k_s pointing to a long-term key and the session key. The values *request*, *clientID*, *serverID*, and k_S are symbolic variables that have been created during symbolic execution of the preceding code, by a call to a random number generator in case of k_S or by reading values from the environment in case of other variables. The value $\mathit{key}(\mathit{clientID}, \mathit{serverID})$ is a symbolic expression representing a long-term key. It is generated during symbolic execution of the call to get_shared_key.

We examine the symbolic execution of the function send_request line by line in Fig. 3, assuming that we start with the symbolic memory shown above. The left column shows the source code and the right column shows the corresponding updates to the symbolic memory as well as the generated IML expressions (in the last two lines).

In Line 1, we compute the length of the encrypted part of the request message and store it in m1_len. The values $\text{len}(k_S)$ and $\text{len}(\mathit{request})$ are extracted from the fields of the ctx structure, the value 4 is the result of the sizeof operation.

C line	symbolic execution steps
`int send_request(RPCstate * ctx){`	

1. `uint32_t m1_len, m1_e_len, full_len;`
`unsigned char * m1, * p, * m1_e;`
`m1_len = 1 + ctx→k_s_len`
`+ sizeof(ctx→request_len)`
`+ ctx→request_len;`
	stack $m1\_len \Rightarrow 1 + \text{len}(k_S) + 4 + \text{len}(request)$

2. `p = m1 = malloc(m1_len);`	stack $p \Rightarrow \text{ptr}(\text{heap}\,6,\, 0)$ stack $m1 \Rightarrow \text{ptr}(\text{heap}\,6,\, 0)$
3. `memcpy(p, "p", 1);`	heap $6 \Rightarrow$ 'p'
4. `p += 1;`	stack $p \Rightarrow \text{ptr}(\text{heap}\,6,\, 1)$
5. `* (uint32_t *) p = ctx→request_len;`	heap $6 \Rightarrow$ 'p'$\vert\,\text{len}(request)$
6. `p += sizeof(ctx→request_len);`	stack $p \Rightarrow \text{ptr}(\text{heap}\,6,\, 5)$
7. `memcpy(p, ctx→request, ctx→request_len);`	heap $6 \Rightarrow$ 'p'$\vert\,\text{len}(request)\vert request$
8. `p += ctx→request_len;`	stack $p \Rightarrow \text{ptr}(\text{heap}\,6,\, 5 + \text{len}(request))$
9. `memcpy(p, ctx→k_s, ctx→k_s_len);`	heap $6 \Rightarrow$ 'p'$\vert\,\text{len}(request)\vert request\vert k_S$

10. `full_len = 1 + sizeof(ctx→self_len)`
`+ ctx→self_len`
`+ encrypt_len(ctx→k_ab, ctx→k_ab_len,`
`m1, m1_len);`
	stack $full\_len \Rightarrow 5 + \text{len}(clientID) + encrypt\_len(msg1)$
	where $msg1 =$ 'p'$\vert\,\text{len}(request)\vert request\vert k_S$

11. `p = m1_e = malloc(full_len);`	stack $p \Rightarrow \text{heap}\,7$ stack $m1\_e \Rightarrow \text{heap}\,7$
12. `memcpy(p, "p", 1);`	heap $7 \Rightarrow$ 'p'
13. `p += 1;`	stack $p \Rightarrow \text{ptr}(\text{heap}\,7,\, 1)$
14. `* (uint32_t *) p = ctx→self_len;`	heap $7 \Rightarrow$ 'p'$\vert\,\text{len}(clientID)$
15. `p += sizeof(ctx→self_len);`	stack $p \Rightarrow \text{ptr}(\text{heap}\,7,\, 5)$
16. `memcpy(p, ctx→self, ctx→self_len);`	heap $7 \Rightarrow$ 'p'$\vert\,\text{len}(clientID)\vert clientID$
17. `p += ctx→self_len;`	stack $p \Rightarrow \text{ptr}(\text{heap}\,7,\, 5 + \text{len}(clientID))$

18. `m1_e_len`
`= encrypt(ctx→k_ab, ctx→k_ab_len,`
`m1, m1_len, p);`
	heap $7 \Rightarrow$ 'p'$\vert\,\text{len}(clientID)\vert clientID\vert cipher1$
	stack $m1\_e\_len \Rightarrow \text{len}(cipher1)$
	new fact: $\text{len}(cipher1) \leq encrypt\_len(msg1)$
	$cipher1 = E(key(clientID, serverID), msg1)$

19. `full_len = 1 + sizeof(ctx→self_len)` `+ ctx→self_len + m1_e_len;`	stack $full\_len \Rightarrow 5 + \text{len}(clientID) + \text{len}(cipher1)$
20. `send(&(ctx→bio),` `&full_len, sizeof(full_len));`	*generate IML:* **out**$(c,\, 5 + len(clientID) + len(cipher1))$;
21. `send(&(ctx→bio), m1_e, full_len);}`	*generate IML:* **out**$(c,$ 'p'$\vert\,\text{len}(clientID)\vert clientID\vert cipher1)$;

Fig. 3. Symbolic execution of the send_request function.

The call to the malloc function is treated specially by the symbolic execution. Line 2 creates a new memory location heap 6 and stores the pointer to the beginning of the memory location (with offset 0) in p and m1. Initially there is no mapping for heap 6 in the symbolic memory which means that the contents of the memory location is undefined. The allocation table associates the value of m1_len with heap 6 and all future writes into heap 6 are checked to be within allocated bounds. We have removed the NULL checks after malloc for simplicity and because the symbolic interpretation only considers a single execution path and thus effectively assumes that any failing function immediately aborts execution.

The function memcpy is treated specially too. In line 3 it copies the literal bytestring 'p' into the memory location heap 6 stored in p. Line 4 increments p by 1 so that the value stored in p becomes $\text{ptr}(\text{heap}\,6, 1)$.

Line 5 stores the value len(*request*) from ctx→ request_len into p. Now that p points to the offset 1 from the beginning of heap 6 the value len(*request*) is written just past the value 'p' that is already in heap 6, so that the memory location now contains a concatenation. Line 6 increments p to point just past the end of the concatenation.

Lines 7–9 proceed similarly and append *request* and k_S to the contents of the memory location. For each memory read and write we must find where the pointer offset points to within the contents of the memory location. This is done with the help of the SMT solver *Yices* (Dutertre and de Moura 2006).

Line 10 computes an estimate of the length of the request message. The function encrypt_len gives a prediction of the length of an encryption based on the key and the plaintext. In this case the plaintext is the concatenation in heap 6. We are not verifying correctness of cryptographic implementations and so the call to encrypt_len is replaced by a call to encrypt_len_proxy that instructs the symbolic execution about how to represent the action of encrypt_len symbolically. The proxies for cryptographic primitives are written by hand and form the trusted base of the verification, similar to the annotations on cryptographic primitives in the VCC approach. The proxy for encrypt_len simply returns the expression *encrypt_len*(*msg1*), where *msg1* is the plaintext.

Lines 11–17 work similarly to lines 2–9 and build up the request message in the memory location heap 7. In line 18 we add the ciphertext to the message by calling encrypt. The call is redirected to a proxy function which generates a new symbolic expression $cipher1 = E(key(clientID, serverID), msg1)$ that represents the ciphertext and writes that expression through the pointer passed as argument. The function returns the expression len(*cipher1*). It additionally checks that len(*cipher1*) $\leq$ *encrypt_len*(*msg1*). This fact is added to the set of known facts and is used to prove that the length of the contents of heap 7 is still less than or equal to the allocated size of heap 7, that is, the encryption function does not write past the end of the allocated buffer.

Line 19 computes the full length of the request message. Lines 20 and 21 send this length followed by the actual message. The function send is treated specially and generates the IML expression **out**(*c*, *e*), where *e* is the symbolic expression contained in the buffer passed to send.

The soundness of the C to IML translation is established in Aizatulin et al. (2011b). There we define a language-agnostic notion of embedding that allows to include, say, a C program as a subprocess of a pi-calculus process. Formally, an environment P_E is a process containing a hole [] and the embedding $P_E[P]$ of a process P into P_E is defined by replacing all instances of [] with P. Both CVM and IML (and thus also C and the pi-calculus) are given computational semantics and the security is defined in terms of the probability that a trace property is violated. The soundness theorem states that if a single-path C program P yields an IML model $\tilde{P}$ then for any IML environment P_E the process $P_E[P]$ is no less secure (up to a fixed polynomial) than $P_E[\tilde{P}]$. An example of an environment process P_E is shown in Section 3.3. The ability to specify an environment of a

C program in the pi-calculus allows us to do threat modelling without having to add concurrency to the C language itself.

3.3 Translating to Pi-Calculus by Message Format Abstraction

The extracted IML model is much simpler than the original C code—it uses no mutable memory. Unfortunately it is still too low-level to be given to ProVerif because of the bytestring manipulation primitives. The key observation for the next step is that the symbolic expressions that use concatenation are used to construct tuples and the symbolic expressions that use substring extraction are used to extract fields from tuples. The strategy will thus be to introduce new operation symbols, replacing the bytestring-manipulating expressions. Of course we shall need to prove that the substituted expressions satisfy the algebraic laws that are expected of tuples.

As an example, consider the client request message

$$\text{'p'}|\operatorname{len}(\mathit{clientID})|\mathit{clientID}|\\ E(\mathit{key}(\mathit{clientID},\mathit{serverID}),\text{'p'}|\operatorname{len}(\mathit{request})|\mathit{request}|k_S).$$

By introducing $\mathit{conc}_1(b_1,b_2) = \text{'p'}|\operatorname{len}(b_1)|b_1|b_2$ the request message becomes $\mathit{conc}_1(\mathit{clientID}, E(\mathit{key}(\mathit{clientID},\mathit{serverID}), \mathit{conc}_1(\mathit{request}, k_S))$. Similarly the part of the IML process

$$\textbf{if } \mathit{msg}_1\{0,1\} = \text{'p'} \textbf{ then if } \operatorname{len}(\mathit{msg}_1) \le 5 + \mathit{msg}_1\{1,4\} \textbf{ then}\\ \textbf{let } \mathit{client}_1 = \mathit{msg}_1\{5, \mathit{msg}_1\{1,4\}\} \textbf{ in } \ldots$$

can be rewritten to **let** $\mathit{client}_1 = \mathit{parse}_1(\mathit{msg}_1)$ **in** ... by defining

$$\mathit{parse}_1(b) = \text{if } \neg((b\{0,1\} = \text{'p'}) \wedge (\operatorname{len}(b) \le 5 + b\{1,4\})) \text{ then } \bot \text{ else } b\{5, b\{1,4\}\}$$

and adding the rewrite rule $\mathit{parse}_1(\mathit{conc}_1(x,y)) = x$. Similarly we extract a function parse_2 with the property $\mathit{parse}_2(\mathit{conc}_1(x,y)) = y$. This yields the following pi-calculus processes for the client and the server:

```
let A =
 event client_begin(clientID, serverID, request);
 new kS1;
 let msg1 = conc1(clientID, E(key(clientID, serverID), conc1(request, kS1))) in
 out(c, msg1);
 in(c, msg1);
 event client_accept(clientID, serverID, request, D(kS1, msg1)); 0.
let B =
 in(c, msg1);
 event server_reply(parse1(msg1), serverID, parse1(D(key(parse1(msg1),serverID), parse2(msg1))),response);
 let msg2 = E(parse2(D(key(parse1(msg1), serverID), parse2(msg1))), response) in
 out(c, msg2); 0.
```

In addition to the models for the client and server the ProVerif input contains a handwritten environment process that describes the interaction of clients and servers and binds their free variables. (The client A has free variables clientID and serverID, while the server B has free variable serverID.) Our environment includes dynamic key compromise and models dynamic key lookup using a private function key:

```
free request, response.
process
    ! (in(c, clientID); in(c, serverID); !A)
  | ! (in(c, serverID); !B)
  | ! (in(c, (clientID, serverID)); event bad(clientID, serverID); out(c, key(clientID, serverID)))
```

Finally the ProVerif input contains user-supplied equations for cryptographic operations and the required security properties:

```
fun E/2.   private fun key/2.   reduc D(k, E(k, x)) = x.

query ev:client_accept(client, server, req, resp) ==> ev:server_reply(client, server, req, resp) | ev:bad(client,
      server).
query ev:server_reply(client, server, req, resp) ==> ev:client_begin(client, server, req) | ev:bad(client, server).
query attacker:key(client, server) ==> ev:bad(client, server).
query attacker:kS1[clientID = client; serverID = server] ==> ev:bad(client, server).
query ev:client_accept(client, server, req, resp) ==> ev:bad(client, server).
```

These properties correspond to Section 2.1. The first two properties are the authentication correspondences. The next three properties are the secrecy of the long-term key, and the session key, and of the payloads. The values in the square brackets bind the keys and the payloads to the client and server identities under which they are created. For instance, the value kS1[clientID = client; serverID = server] is the session key created after having received client as clientID and server as serverID. The last property is used to check the sanity of the model by checking the reachability of the final state; unlike the other properties, we intend that it be false. If it is false, it means there is an execution of the protocol that reaches the end, the client_accept event, without compromise of the long-term key.

The soundness result proved in Aizatulin et al. (2011b) says that an IML process P is no less secure (up to a fixed polynomial) than the pi process $\tilde{P}$ that it translates to. The result requires that the substituted expressions actually do behave like their symbolic counterparts. This relies on the following properties, all of which are proved automatically in our implementation:

- The ranges of all constructor operations should be disjoint. This is assumed for cryptographic primitives and proved for the newly introduced concatenation operations by enforcing the use of distinct tags for each concatenation, such as the tag 'p' in $conc_1$.
- The rewrite rules like $parse_1(conc_1(x, y)) = x$ above should be satisfied. This is proved by substituting the definitions of $parse_1$ and $conc_1$ and simplifying the resulting expression to x.
- The parser should fail (that is, return $\perp$) for any bytestring that is not in the range of the corresponding concatenation function. As proved in (Aizatulin et al. 2011b), this is satisfied whenever the parser checks all the tag fields and checks the consistency of the length fields with the actual length of the message.

3.4 Verification with ProVerif

Running ProVerif with the above input successfully verifies the properties:

```
> proverif −in pi pvmodel.out | grep RESULT
RESULT ev:client_accept(...) ==>ev:server_reply(...) | ev:bad(...) is true.
RESULT ev:server_reply(...) ==>ev:client_begin(...) | ev:bad(...) is true.
RESULT attacker:key(client_534,server_535) ==>ev:bad(client_534,server_535) is true.
RESULT attacker:kS1_28[...] ==>ev:bad(clientID_26[...],server_362) is true.
RESULT ev:client_accept(client_32,server_33,req_34,resp_35) ==>ev:bad(client_32,server_33) is false.
```

The ProVerif result may be interpreted in two ways. The first interpretation would establish the security in the computational model as developed in Aizatulin et al. (2011b) by appealing to a computational soundness result like Backes et al. (2009). In such a model the attacker is an arbitrary machine that exchanges bytestrings with the C program or the executing pi process. Unfortunately, such results often place substantial restrictions on the cryptographic operations used by the protocol as well as the structure of the protocol itself. In particular, keys travelling over the network (like k_S in our protocol) and key compromise are difficult. We are not aware of any computational soundness result that applies to the protocol analysed in this paper.

Instead, in this case we rely on a second interpretation with respect to a symbolic model of cryptography, as in Dupressoir et al. (2011). In this interpretation the attacker is weaker—in our case it is restricted to be a pi-calculus process that interacts with our protocol process. Furthermore the properties are guaranteed to hold only for those traces in which there are no collisions, that is, where syntactically distinct symbolic expressions evaluate to different bytestrings. Due to the limitations of such an interpretation and the limitations of computational soundness results, we are working on verification of the models directly in the computational setting using CryptoVerif.

To summarize, our first approach automatically extracts a verifiable model in pi calculus from protocol code in C. We assume that the protocol follows a single path, with any deviation leading to immediate termination. Given this assumption, which holds of our example protocol, the extracted model captures all runs of the protocol code, and we prove correspondence and secrecy properties of the model.

4 Verification using a General-Purpose Verifier

Our second approach to C protocol verification relies on stating and proving invariants of program data structures using the general-purpose verifier VCC (Cohen et al. 2009). We adapt to C the method of *invariants on cryptographic structures* first developed in the setting of F7 (Bhargavan et al. 2010; Fournet et al. 2011a).

Our formulation of trace-based security goals is superficially different but in fact equivalent to that of the previous section. For secrecy, we prove properties of the cryptographic invariants. For authentication, instead of relying on global correspondence assertions, we prove the correction of assertions embedded in our code according to the following variation of our protocol narration.

Authenticated RPC: RPC-enc

A : `event` $Request(A, B, req)$
$A \rightarrow B$: $A, \{req, k_S\}_{k_{AB}}$
B : `assert` $Request(A, B, req) \vee Bad(A, B)$
B : `event` $Response(A, B, req, resp)$
$B \rightarrow A$: $\{resp\}_{k_S}$
A : `assert` $Response(A, B, req, res) \vee Bad(A, B)$

The original work on cryptographic invariants in F7 introduces inductive definitions simply by listing Horn clauses. In our work with VCC, we express the symbolic algebra and its cryptographic invariants as explicit Coq definitions. For the sake of brevity these definitions are omitted from our previous publication (Dupressoir et al. 2011). In this paper, we take the opportunity to explain the Coq definitions in detail in the following section, before describing how to embed the development into VCC, so as to prove a security theorem about the C code.

4.1 Coq Development: Symbolic Algebra and Level Predicate

This section describes a type term of symbolic cryptographic expressions, a type log of sets of events during runs of a protocol, and a type level, either Low or High. Given these types, we make an inductive definition of a predicate Level l t L, meaning that the term t may arise at level l after the events in log L have happened. The set of terms at level Low is an upper bound on any attacker's knowledge, while the set of terms at level High is an upper bound on any principal's knowledge. (The set of High terms is a strict superset of the Low terms.) We make these definitions in the Coq proof assistant, and use it to check security theorems. Subsequently, we import the definitions and theorems into VCC, confident in their soundness.

First, we define the term type, with constructors to build literal terms from bytestrings, to injectively pair two terms (the $(\cdot, \cdot)$ operation), and to perform symmetric authenticated encryption ($\{\cdot\}_\cdot$). (To accommodate other protocols, we may extend the type with constructors for other standard cryptographic primitives, such as asymmetric encryption and signature, and HMAC computations.)

We define an auxiliary type usage, whose values describe the purposes of freshly generated bytestrings of the protocol. These may be guesses generated by the attacker, or protocol keys, or nonces sent as messages to help us specify secrecy properties. There are two kinds of key usage, for long-term keys and session keys, and there are two kinds of nonces, for request and response messages that are meant to remain secret.

```
Inductive term :=
  | Literal: (bs: bytes)
  | Pair: (t1 t2: term)
  | SEnc: (k p: term).
```

```
Inductive nonceUsage :=
  | U_Request: (a b: term)
  | U_Response: (a b req: term).
```

```
Inductive sencKeyUsage :=
  | U_RPCKeyAB: (a b: term)
  | U_RPCSessionKey: (a b req: term).
```

```
Inductive usage :=
  | AttackerGuess
  | Nonce: nonceUsage
  | SEncKey: sencKeyUsage.
```

Next, we introduce the log type as being a set of events, where there are four constructors of the event type: (1) an event New (Literal bs) u means that the fresh bytestring bs has one of the key or nonce usages u; (2) an event Request a b req means that client a intends to send server b the request req; (3) an event Response a b req resp means that server b has accepted the request req from client a and intends to reply with response resp; (4) an event Bad a b means that any long-term keys between client a and server b are compromised. We also define a predicate Good L to mean that the New events in L ascribe a unique usage to each nonce or key, and apply only to bytestring literals.

```
Inductive ev :=
  | New: (t: term) (u: usage)
  | Request: (a b req: term)
  | Response: (a b req resp: term)
  | Bad: (a b: term).

Definition log := ListSet.set event.
Definition Logged (e: ev) (L: log) :=
  ListSet.set_In e L.
```

```
Definition log_leq (L L': log) :=
  ∀x, Logged x L →Logged x L'.

Definition Good (L: log) :=
  (∀ t u, Logged (New t u) L →
      ∃bs, t = Literal bs) /\
  (∀ t u1 u2, Logged (New t u1) L →
      Logged (New t u2) L →u1 = u2).
```

A central idea of cryptographic invariants is that each key usage has an associated *payload property*, which relates keys and payloads to which honest principals can apply the corresponding cryptographic primitive. The payload property RPCKeyABPayload a b m L says that a long-term key shared between a and b may encrypt a payload m when m is a pair composed of a request from a to b on which the Request event has been logged in L, together with a session key for a and b generated specifically for that request. The payload property RPCSessionKeyPayload a b req m L says that a session key key may encrypt a payload m if it has been logged as a response to req. We combine these two payload properties in the definition below of canSEnc, which serves as a precondition, in code, to the encryption function when called by honest participants.

```
Definition RPCKeyABPayload (a b m: term) (L: log) :=
  ∃req, ∃k,
    m = Pair req k /\
    Logged (Request a b req) L /\
    Logged (New k (SEncKey(U_RPCSessionKey a b req))) L.

Definition RPCSessionKeyPayload (a b req m: term) (L: log) :=
  Logged (Response a b req m) L.

Definition canSEnc (k p: term) (L: log) :=
  (∃ a, ∃b, ∃req,
      Logged (New k (SEncKey(U_RPCSessionKey a b req))) L /\
      RPCSessionKeyPayload a b req p L) \/
  (∃ a, ∃b,
      Logged (New k (SEncKey(U_RPCKeyAB a b))) L /\
      RPCKeyABPayload a b p L).
```

Another central idea is that each nonce or key has a *compromise condition*, which needs to be fulfilled before a literal given that usage can be released to the attacker. Implicitly, bytestrings with usage AttackerGuess are always known to the attacker. Our next two predicates define the compromise conditions for other sorts of nonce and key.

```
Definition nonceComp (n: term) (L: log) :=
  (∃ a, ∃b, Logged (New n (U_Request a b)) L /\ Logged (Bad a b) L) \/
  (∃ a, ∃b, ∃req, Logged (New n (Nonce(U_Response a b req))) L /\ Logged (Bad a b) L).

Definition sencComp (k: term) (L: log) :=
  (∃ a, ∃b, Logged (New k (SEncKey(U_RPCKeyAB a b))) L /\ Logged (Bad a b) L) \/
  (∃ a, ∃b, ∃req, Logged (New k (SEncKey(U_RPCSessionKey a b req))) L /\ Logged (Bad a b) L).
```

Given these auxiliary predicates, we now define the Level predicate. We intend that given a log L, any term t sent or received on the network satisfies Level Low t L, while if t is data manipulated internally by principals, we must have Level High t L. (The Level predicate consolidates both the Pub and Bytes predicates from Dupressoir et al. (2011); specifically, Level Low is a predicate equivalent to Pub, and Level High is a predicate equivalent to Bytes.) It easily follows from the definition that any term satisfying Level Low also satisfies Level High (but not the converse, because for example uncompromised keys and nonces satisfy Level High but not Level Low). We also prove that Level l is a monotonic function of its log argument for all l, which will help greatly when using the definitions in VCC. Additionally, we also prove slight variants of some of the rules that will be used to help VCC and Z3 efficiently instantiate quantified variables.

```
Inductive level := Low | High.
Inductive Level: level →term →log →Prop :=
  | Level_AttackerGuess: ∀l bs L, (* AttackerGuesses are always Low *)
    Logged (New (Literal bs) AttackerGuess) L →
    Level l (Literal bs) L

  | Level_Nonce: ∀l bs L nu, (* Nonces are Low when compromised *)
    Logged (New (Literal bs) (Nonce nu)) L →
    (l = Low →nonceComp (Literal bs) L) →
    Level l (Literal bs) L

  | Level_SEncKey: ∀l bs L su, (* SEncKeys are Low when compromised *)
    Logged (New (Literal bs) (SEncKey su)) L →
    (l = Low →sencComp (Literal bs) L) →
    Level l (Literal bs) L

  | Level_Pair: ∀l t1 t2 L, (* Pairs have same level as their components *)
    Level l t1 L →
    Level l t2 L →
    Level l (Pair t1 t2) L

  | Level_SEnc: ∀l l' k p L, (* SEnc with plaintext matching payload property *)
    canSEnc k p L →
    Level l' p L →
    Level l (SEnc k p) L

  | Level_SEnc_Low: ∀l k p L, (* SEnc with compromised or Low key *)
    Level l k L →
    Level l p L →
    Level l (SEnc k p) L

Theorem Low_High: ∀t L, Level Low t L →Level High t L.
Theorem Level_Positive: ∀l t L L', log_leq L L' →Level l t L →Level l t L'.
```

As mentioned previously, we state secrecy properties of the protocol as consequences of the invariants respected by the code. We prove in the following two theorems that fresh nonces used as requests and responses are kept secret unless keys are compromised. We actually state the contrapositive: that if Level Low holds on the nonce (intuitively, if the nonce is not secret), then the long-term key is compromised. The proof is an almost direct application of the inversion principle for the Level_Nonce inductive rule above: the only way for a nonce to be Low is for its compromise condition to hold.

```
Theorem SecrecyRequest: ∀a b req L,
  Good L →
  Logged (New req (Nonce(U_Request a b))) L →
  Level Low req L →
  Logged (Bad a b) L.

Theorem SecrecyResponse: ∀a b req resp L,
  Good L →
  Logged (New resp (Nonce (U_Response a b req))) L →
  Level Low resp L →
  Logged (Bad a b) L.
```

These secrecy properties state the absence of a direct flow of a nonce to the opponent, unless the associated key is compromised. We do not address here how to show noninterference properties, the absence of indirect flows.

Finally, we state our correspondence properties for the request and response methods. We embed the assertions from the narration at the start of this section within the code at the points that the request and response messages have been validated; to verify these assertions we rely on the following theorems about our cryptographic invariants. The first states that if there is a public message encrypted with the long-term key, then either the plaintext is a well-formed request or the key is compromised. The second states that if there is a public message encrypted under a session key, then either the plaintext is a well-formed response or the corresponding long-term key is compromised.

```
Theorem AuthenticationRequest: ∀a b req kAB k L,
  Good L →
  Logged (New kAB (SEncKey(U_RPCKeyAB a b))) L →
  Level Low (SEnc kAB (Pair req k)) L →
  Logged (Request a b req) L /\
  Logged (New k (SEncKey(U_RPCSessionKey a b req))) L) \/
  Logged (Bad a b) L.

Theorem AuthenticationResponse: ∀a b req resp k L,
  Good L →
  Logged (New k (SEncKey(U_RPCSessionKey a b req))) L →
  Level Low (SEnc k resp) L →
  Logged (Response a b req resp) L) \/
  Logged (Bad a b) L.
```

In these authentication theorems, we do not expect request and response messages to be freshly generated nonces. Therefore, the authentication results can be used in all possible applications of the protocol, even those applications that do not make use of the secrecy property. In this report, we verify the C implementation without any assumptions on the request and response, apart from the fact that they are at least High. Intuitively, this means that only cryptographic constructors can be applied by the protocol roles to payloads, as noth-

ing is known about the latter. In practice, an application that makes use of this protocol to exchange specific payloads would treat the Request and Response events as interpreted predicates instead of uninterpreted events (for example, for a direct application of our security result, Request a b r could be defined as Logged r (U_Request a b)). We do not consider the problem of composition in this paper, but we expect to obtain the same level of modularity as (Bhargavan et al. 2010).

4.2 VCC Theory of Symbolic Cryptography

As in Dupressoir et al. (2011), we import the definitions and theorems into VCC as first order program constructs. We refer readers to the previous work for more details on this translation. The VCC language has evolved since the status reported in Dupressoir et al. (2011) and now includes support for datatype declarations, which we exploit to provide cleaner, simpler, and more efficient declarations for the inductive datatypes. In addition, we now equip the VCC axiomatic approximation of the Coq model with triggers that help Z3 through the proof, therefore limiting the amount of additional user annotations required, and sometime greatly reducing the verification time.

For example, this small code snippet defines the term algebra using VCC's inductive datatype syntax.

```
_(datatype term {
  case Literal(ByteString s);
  case Pair(term t1, term t2);
  case SEnc(term k, term p); })
```

An alternative to importing the inductively defined predicates such as Level into VCC would be to develop the security proof directly in VCC. We prefer to use Coq as it has better developed support for inductive reasoning, and because by doing the proof in Coq, we prove security theorems once and may use them to prove several implementations, indeed even implementations written in different languages (for example, both F# and C).

We rely on ghost state to represent the event log; the Coq predicate Good is an invariant on the log. We also rely on ghost state to associate concrete bytestrings in the C program with the terms developed in Coq. As discussed in our prior paper, inconsistencies may arise if two distinct terms correspond to the same concrete bytestring. We assume an implementation of cryptography that keeps track at run-time of all operations performed, linking symbolic terms to the concrete results obtained, and aborts the execution whenever it happens that two distinct terms are represented by the same bytestring.

However, in this particular implementation, we inline the pairing operations to allow for various performance optimizations, for example when it is advantageous to fill in the result buffer from the end forward rather than the other way around. As a result, the hybrid wrapper approach described by Dupressoir et al. (2011) cannot be directly applied to the code discussed here. Instead, we write and verify MakePair and DestructPair ghost functions that will be called once the concrete pairing is complete. Provided that the byte string passed as argument

has the correct format for a pair (as described in Section 2.2), it will update the table (or detect a collision) accordingly. A simplified contract for MakePair is shown below.

```
_(ghost void MakePair(ByteString b1, ByteString b2, ByteString b, \claim c)
  _(always c, (&table)↦\closed && table_claim_stable())
  _(requires b == cons('p',
                       concat(int_bytes(b1.length,4), concat(b1,b2))))
  _(ensures table.B2T[b] == Pair(table.B2T[b1],table.B2T[b2]));
```

4.3 Attacker Model and Security Results

Attacker model The attacker is given complete control over the network (all messages are exchanged through the attacker, who controls scheduling and can eavesdrop and modify messages as symbolic terms), can setup and run new instances of the protocol roles, either with Low requests or freshly generated High requests, can compromise long-term keys, and has complete control over the scheduling of instructions. We formalize the attacker, as in Dupressoir et al. (2011), by providing an attacker interface, called the *shim*, that is verified to maintain the cryptographic invariants whilst providing the attacker with the intended capabilities.

```
void toLiteral(BYTE* buffer, uint32_t length);
void pair(BYTE* b1, uint32_t b1_len, BYTE* b2, uint32_t b2_len, BYTE* b);
void destruct(BYTE* buffer, uint32_t length, BYTE** b1, BYTE** b2);
void sEncrypt(BYTE* key, uint32_t key_len, BYTE* plain, uint32_t plain_len,
              BYTE** cipher, uint32_t* cipher_len);
void sDecrypt(BYTE* key, uint32_t key_len, BYTE* cipher, uint32_t cipher_len,
              BYTE** plain, uint32_t* plain_len);

session* setup_secrets(BYTE* alice, uint32_t alice_len,
                       BYTE* bob, uint32_t bob_len);
session* setup_public(BYTE* alice, uint32_t alice_len,
                      BYTE* bob, uint32_t bob_len,
                      BYTE* request, uint32_t request_len);

channel getClientChannel(session* s);
channel getServerChannel(session* s);
void write(channel c, BYTE* buffer, uint32_t length);
void read(channel c, BYTE** buffer, uint32_t* length);
```

Security of authenticated encryption-based RPC The final security result, once the protocol code and the shim have been verified, can be stated as follows.

Theorem 1. *For all attack programs P written as a well-formed sequence of calls to functions in the shim, the correspondence assertions and secrecy invariants hold in all states until two distinct terms are associated with the same bytestring.*

The table below summarises the current verification results on the implementation. The verification times shown are general ideas, obtained on a mid-performance laptop.

Function Name	LoC	LoA	Verification Time
`send_request`	50	50	10 min
`recv_response`	30	30	1 s
`recv_request`	70	70	Incomplete
`send_response`	20	20	1 s
`MakePair`	–	30	1 s
`DestructPair`	–	20	1 s
`encrypt` (hybrid)	3	20	5 s
`decrypt` (hybrid)	3	20	5 s

The amount of additional annotation effort required to get a security result on top of memory safety was very variable on this problem: the short functions such as send_response and recv_response, as well as the hybrids, took less than an hour each in annotation and debugging time. However, longer functions are much harder to work on, as feedback cannot be obtained as easily: some verification runs will loop or run out of memory instead of failing to prove an assertion, and the traces produced by failed runs are often too big for the more advanced debugging tools.

Most of the time, the prover spends a lot of time discharging, in the context of the security verification, the memory-safety proof obligations that have already been proved separately. As mentioned in prior work, it may be beneficial to split the verification runs in two, the first proving memory safety only, and the second proving security whilst assuming memory safety (by turning assertions proved in the first run into assumptions).

We also prove more than simple memory-safety and cryptographic properties, establishing a well-formedness result on the protocol context that is passed around as the unique argument, and recording all cryptographic properties of each field on successful return. This property could then be used to prove security and functional properties at the level of an application using the protocol, or to compose protocols together.

4.4 Verification of an Example Function

The code below is extracted from the client-side implementation. It represents the entire recv_response() function, with simplified contracts and annotations, so that only the event assertion and top-level results appear. The fully-annotated version carries approximately one line of annotation per line of code, most of which help to establish the memory-safety of the shared log, and that its invariants are preserved.

```
int recv_response(RPCstate * ctx)
  _(requires Good_Client(*ctx))
  _(ensures \result == 0 ==> Good_Client(*ctx))
{
  unsigned char * m2_e;
  uint32_t m2_e_len;

  recv_uint32(&(ctx->bio),&m2_e_len, sizeof(m2_e_len));
  m2_e = malloc(m2_e_len);
  if (m2_e == NULL) return -1;
```

```
  recv(&(ctx→bio), m2_e, m2_e_len);

  ctx→response_len = decrypt_len(ctx→k, ctx→k_len, m2_e, m2_e_len);
  ctx→response = malloc(ctx→response_len);
  if (ctx→response == NULL)
  {
    free(m2_e);
    return -1;
  }
  ctx→response_len =
      decrypt(ctx→k, ctx→k_len, m2_e, m2_e_len, ctx→response);
  if (ctx→response_len == 0)
  {
    free(m2_e);
    return -1;
  }
  _(assert log.Response[getTerm(ctx→self,ctx→self_len)]
                       [getTerm(ctx→other,ctx→other_len)]
                       [getTerm(ctx→request,ctx→request_len)]
                       [getTerm(ctx→response,ctx→response_len)] ||
          log.Bad[getTerm(ctx→self,ctx→self_len)]
                 [getTerm(ctx→other,ctx→other→len)])

  return 0;
}
```

The Good_Client() predicate on contexts expresses that a context is well-formed from a security point of view. For example, we state that if the request field is initialized, then the event Request(a,b,r) is in the log, where r is the term associated with the request field and a and b are the terms associated with the self and other field, respectively. It could also be used to express non-security properties of the context, and we indeed do so by expressing that the end field of a Good_Client context has to be CLIENT. In all generality, such a predicate could be used to express arbitrarily complex properties, for example, that an OpenSSL context represents an abstract state machine in a given state, allowing to verify highly complex functional properties along with the desired security properties.

In this case, however, we are only interested in the final event assertion. To establish it, VCC will make use of the postcondition of the decryption primitive, whose contract (shown below with all memory safety contracts erased) simply states that, on successful return, the ciphertext that was passed in is guaranteed to have been built as the encryption of the decryption's output under the decryption key.

```
extern uint32_t decrypt(unsigned char * key, uint32_t keylen, unsigned char *
    in, uint32_t inlen, unsigned char * outbuf)
  _(writes \array_range(outbuf,decrypt_len(key,keylen,in,inlen)))
  _(ensures \result != 0 ==> getTerm(in,inlen) == SEnc(getTerm(key,keylen),
      getTerm(outbuf,\result)));
```

Using this contract, given that the ciphertext was Low (as it was read from the network), and the imported Coq theorem AuthenticationResponse, we can deduce that the Response event has been logged on the term associated with the contents of ctx→ response, unless the long-term key used has been compromised.

4.5 Discussion

Our approach with VCC allows us to prove memory safety and symbolic security of C code, that is, safety of the protocol code against a network-based adversary

in the symbolic model of cryptography. It does not prevent attacks outside this model, such as computational or physical attacks.

We presented our Coq definitions for a particular protocol. It would be useful future work to generalize our definitions to form a domain-specific language for protocols, in which message formats, events, payload conditions, compromise conditions, and other protocol-specific parameters could be expressed, and from which proofs and perhaps some protocol code, such as routines to marshal and unmarshal messages, could be extracted.

5 Related Work

We describe the main prior work on C. For recent surveys of related work in higher-level languages, see Fournet et al. (2011a) and Hriţcu (2011).

Csur (Goubault-Larrecq and Parrennes 2005) pioneered the extraction of a verifiable model from cryptographic code in C. Csur extracts a set of Horn clauses from a C program for the Needham-Schroeder protocol, which are then solved using a theorem prover. We improve upon CSur in two ways in particular. First, in the approach presented in Section 3, we have an explicit attacker model with a standard computational attacker. The attacker in CSur is essentially symbolic—it is allowed to apply cryptographic operations, but cannot perform any arithmetic computations. Second, we handle authentication, as well as secrecy properties. Adding authentication to CSur would be hard, as it relies on a coarse over-approximation of C code.

Software model checking techniques have been applied to cryptographic code in C. Godefroid and Khurshid (2002) use genetic algorithms to explore the state spaces of concurrent systems, with an implementation of the Needham-Schroeder protocol as an example; neither systematic nor randomg testing could find Lowe's attack, but it was found by random search guided by applications of application-independent heuristics. Godefroid et al. (2005) apply DART, directed automated random testing, to the same implementation code, and are able to find Lowe's attack via a systematic search. ASPIER (Chaki and Datta 2009) uses software model checking to verify bounded numbers of sessions of the main loop of OpenSSL. The model checking operates on a protocol description language, which is rather more abstract than C; for instance, it contains no pointers and does not deal with messages of variable length.

(Udrea et al. 2006) reports on the Pistachio approach which verifies the conformance of an implementation with a set of rules manually extracted from a specification of the communication protocol. It does not directly support the verification of security properties.

The frameworks SAGE (Godefroid et al. 2008) and KLEE (Cadar et al. 2008) use symbolic execution to generate test cases with high path coverage. After obtaining a symbolic summary of the program these frameworks apply an SMT solver to find inputs leading to a bad state. In contrast, our approach generates models for input to a high-level cryptographic verification tool to prove absence of attacks. In contrast to SAGE or KLEE, our symbolic execution covers all

possible concrete executions along a single path. We rely on symbolic pointers to manipulate data, the length of which is not known in advance, as is typical in network protocols.

Corin and Manzano (2011) report an extension of KLEE that allows KLEE to be applied to cryptographic protocol implementations. Similarly to the approach presented in Section 3, KLEE is based on symbolic execution; the main difference is that Corin and Manzano treat every byte in a memory buffer separately and thus only supports buffers of fixed length.

Finally, in recent work, Polikarpova and Moskal (2012) develop a stepwise refinement approach to verifying invariants of security code using VCC.

6 Conclusions, and the Csec Challenge

This paper summarizes the positive results of two recent papers on verifying security properties of cryptographic software in C.

One particular surprise, in our experience, was that although there are many large C codebases (tens of thousands of lines of code) that implement protocols such as TLS or IPsec, there are very few small and readily-available benchmark problems on which to evaluate new verification techniques. Hence, we have launched the *Csec Challenge*, a collection of challenge problems, including source code, intended security properties, and the results obtained by various verification tools. We aim to collect both small benchmark problems and larger widely-deployed codebases. Our collection is available at `http://research.microsoft.com/csec-challenge/`. We hope to create a community resource to help evaluate the next generation of verification tools for cryptographic code in C.

Acknowledgements David A. Naumann contributed to the work reported here using VCC. We are grateful to Cas Cremers for helpful discussions about the RPC-enc protocol and for commenting on a draft of this paper. We also thank Patrice Godefroid for discussions and comments on a draft.

Dennis Ritchie, rest in peace.

References

M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *ACM POPL*, pages 104–115, 2001.

M. Aizatulin, A. Gordon, and J. Jürjens. Extracting and verifying cryptographic models from C protocol code by symbolic execution. In *18th ACM Conference on Computer and Communications Security (CCS 2011)*, 2011. Full version available at `http://arxiv.org/abs/1107.1017`.

M. Backes, D. Hofheinz, and D. Unruh. CoSP: A general framework for computational soundness proofs. In *ACM CCS 2009*, pages 66–78, November 2009. Preprint on IACR ePrint 2009/080.

M. Barbosa, J. Pinto, J. Filliâtre, and B. Vieira. A deductive verification platform for cryptographic software. In *Proceedings of the Fourth International Workshop on Foundations and Techniques for Open Source Software Certification (OpenCert 2010)*, volume 33 of *Electronic Communications of the EASST*. EASST, 2010.

J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement types for secure implementations. In *21st IEEE Computer Security Foundations Symposium (CSF'08)*, pages 17–32, 2008.

K. Bhargavan, C. Fournet, and A. D. Gordon. Modular verification of security protocol code by typing. In *ACM Symposium on Principles of Programming Languages (POPL'10)*, pages 445–456, 2010.

B. Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *IEEE Computer Security Foundations Workshop (CSFW'01)*, pages 82–96, 2001.

C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008)*, San Diego, CA, Dec. 2008.

S. Chaki and A. Datta. ASPIER: An automated framework for verifying security protocol implementations. In *Computer Security Foundations Workshop*, pages 172–185, 2009. doi: 10.1109/CSF.2009.20.

E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *TPHOLs*, volume 5674 of *LNCS*, pages 23–42, 2009.

R. Corin and F. A. Manzano. Efficient symbolic execution for analysing cryptographic protocol implementations. In *International Symposium on Engineering Secure Software and Systems (ESSOS'11)*, LNCS. Springer, 2011.

F. Dupressoir, A. Gordon, J. Jürjens, and D. Naumann. Guiding a general-purpose C verifier to prove cryptographic protocols. In *24th IEEE Computer Security Foundations Symposium*, pages 3–17, 2011.

B. Dutertre and L. de Moura. The Yices SMT Solver. Technical report, 2006.

L. Erkök, M. Carlsson, and A. Wick. Hardware/software co-verification of cryptographic algorithms using Cryptol. In *FMCAD*, 2009.

C. Fournet, K. Bhargavan, and A. D. Gordon. Cryptographic verification by typing for a sample protocol implementation. In *Foundations of Security and Design VI (FOSAD 2010)*, LNCS. Springer, 2011a. To appear.

C. Fournet, M. Kohlweiss, and P.-Y. Strub. Modular code-based cryptographic verification. In *18th ACM Conference on Computer and Communications Security (CCS 2011)*, 2011b. Technical report, sample code, and formal proofs available from http://research.microsoft.com/~fournet/comp-f7/.

P. Godefroid and S. Khurshid. Exploring very large state spaces using genetic algorithms. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS'02)*, volume 2280, pages 266–280. Springer, 2002.

P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *Programming Language Design and Implementation (PLDI'05)*, pages 213–223. ACM, 2005.

P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008*. The Internet Society, 2008.

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

C. Hriţcu. *Union, Intersection, and Refinement Types and Reasoning About Type Disjointness for Security Protocol Analysis*. PhD thesis, Department of Computer Science, Saarland University, 2011.

J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7): 385–394, 1976.

D. A. McGrew and J. Viega. Flexible and efficient message authentication in hardware and software. Manuscript and software available at http://www.zork.org/gcm, 2005.

G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 213–228, London, UK, 2002. Springer-Verlag. ISBN 3-540-43369-4. URL http://portal.acm.org/citation.cfm?id=647478.727796.

PolarSSL. PolarSSL. http://polarssl.org.

N. Polikarpova and M. Moskal. Verifying implementations of security protocols by refinement. In *Verified Software: Theories, Tools and Experiments (VSTTE 2012)*, 2012. To appear.

Project EVA. Security protocols open repository, 2007. http://www.lsv.ens-cachan.fr/spore/.

O. Udrea, C. Lumezanu, and J. S. Foster. Rule-Based static analysis of network protocol implementations. *IN PROCEEDINGS OF THE 15TH USENIX SECURITY SYMPOSIUM*, pages 193–208, 2006. doi: 10.1.1.111.8168. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.111.8168.