Computational Verification of C Protocol Implementations by Symbolic Execution

Mihhail Aizatulin
The Open University

Andrew D. Gordon
Microsoft Research & University of Edinburgh

Jan Jürjens
TU Dortmund & Fraunhofer ISST

ABSTRACT

We verify cryptographic protocols coded in C for correspondence properties with respect to the computational model of cryptography. Our first step uses symbolic execution to extract a process calculus model from a C implementation of the protocol. The new contribution is the second step in which we translate the extracted model to a CryptoVerif protocol description, such that successful verification with CryptoVerif implies the security of the original C implementation. We implement our method and apply it to verify several protocols out of reach of previous work in the symbolic model (using ProVerif), either due to the use of XOR and Diffie-Hellman commitments, or due to the lack of an appropriate computational soundness result. We analyse only a single execution path, so our tool is limited to code following a fixed protocol narration. This is the first security analysis of C code to target a verifier for the computational model. We successfully verify over 3000 LOC. One example (about 1000 LOC) is independently written and currently in testing phase for industrial deployment; during its analysis we uncovered a vulnerability now fixed by its author.

Categories and Subject Descriptors

C.2.2 [**Computer-Communication Networks**]: Network Protocols—*Protocol verification*; D.2.4 [**Software Engineering**]: Software/Program Verification

Keywords

security, verification, protocols, symbolic execution, CryptoVerif, computational

1. INTRODUCTION

The Problem of Verifying Cryptographic Software in C The C programming language is a popular choice for writing cryptographic software, such as protocols or devices. Still, both the design of protocols and their implementation in C are notoriously error prone—there are frequent security advisories of design and implementation flaws in cryptographic software. In response to this problem, researchers are developing a wide range of techniques for C cryptographic software to find bugs and to prove security properties. These techniques span the spectrum between testing and verification, and include specialist verifiers [36], random testing [32, 33], model checking [39, 23], general-purpose verifiers [27, 46], and symbolic execution [5, 24].

```
unsigned char * payload = malloc(PAYLOAD_LEN);
if(payload == NULL) exit(1);
RAND_bytes(payload, PAYLOAD_LEN);
ulong msg_len = PAYLOAD_LEN + 1;
unsigned char * msg = malloc(msg_len);
* msg = 0x01;                                  // add the tag
memcpy(msg + 1, payload, PAYLOAD_LEN); // add payload
unsigned char * pad = otp(msg_len);    // one-time pad
xor(msg, pad, msg_len);
send(msg, msg_len);
```

---

new *nonce1* : $\texttt{fixed}_{20}$;
out($XOR(01|nonce1, pad)$)

Figure 1: Example C fragment and extracted model.

All this prior work is based on the *symbolic model* of cryptography [26], rather than the *computational model* [34]. One reason is that automated techniques for the symbolic model were invented earlier than for the computational model, and are much better understood. Still, security in the symbolic model is generally weaker than security in the computational model, and hence is less convincing. To address this problem, [44] uses a process calculus over a term language that captures probabilistic polynomial time to prove an authentication property of a cryptographic protocol. [2] pioneered principles of *computational soundness*, which establish that certain properties in the symbolic model imply corresponding properties in the computional model. [5] reports on a tool, here referred to as csec-modex, which applies the CoSP framework [7] for computational soundness to obtain computational security for one of the ProVerif [17] models extracted from C by symbolic execution [40]; to the best of our knowledge, that one model amounts to the only published proof of computational security for a C program.

In general, computational soundness principles have significant limitations. For example, for [5], the authors could only obtain a security result in the computational model for one example out of five, which was written by the authors to use non-standard tagging to comply with the CoSP assumptions. To take another example, computational soundness for XOR is problematic. The method presented in [5] cannot prove secrecy of *nonce1* in the example in Fig. 1 because of

CCS'12, October 16–18, 2012, Raleigh, North Carolina, USA.

XOR. Obtaining computational soundness for XOR in general is not possible [50], although it is possible for a limited class of protocols [41], not including the one-time pad scenario of the example mentioned above. Hence, instead of relying on computational soundness, recent automated tools, including CryptoVerif [19], EasyCrypt [10], and F7 [11] enriched with modules for computational cryptography [30], obtain computational security directly.

This Paper: Applying a Computational Prover to C Code Our aim is to connect one of these computational provers to C code, by enriching a model extraction tool csec-modex to target its input format. We target CryptoVerif, but our approach may well be adapted to other computational tools. The foot of Fig. 1 shows the verifiable CryptoVerif model extracted from the C code, a small example of the benefits of switching to a computational verifier. In all, we obtain computational results for more than 3000 LOC in C, rather more than the 450 LOC in the prior work to which computational soundness applies. One example (about 1000 LOC) is code for smart electricity meters, written outside our group, without verification in mind, and in which we found a new bug just before its test deployment.

To put the technical contributions in context, our overall vision is to enable the verification of new and old protocol code in C by leveraging progress in automatic verifiers for protocol models. In particular, we envisage that new protocols may be developed by writing a reference implementation in C, suitable for interoperability testing and possibly use in production, while simultaneously extracting a protocol model for verification with automatic tools. We continue the vision of a line of work on verified reference implementations in functional languages [16, 35, 13, 14, 12] but target C because of its ubiquity, and its greater acceptability to practitioners (and because of its challenges).

In the rest of the section we sketch the paper's structure.

Description of the Method Section 3 introduces the main new ideas of the paper—how to adapt the results of the symbolic execution of C to the CryptoVerif calculus—by way of a detailed example. Beforehand, Section 2 informally recalls the intermediate process calculus, IML, and the target CryptoVerif calculus.

Our verification method proceeds in several steps (cf. Fig. 2). The method takes as input the C source code of the protocol participants as well as a CryptoVerif *template file*, which contains the cryptographic assumptions about the primitives used by the implementation, the environment process which spawns the participants and generates shared cryptographic material, and a query for the property that the implementation is supposed to satisfy. The template file omits the actual model of the protocol participants. That model is extracted from C code by symbolic execution and rewriting. If the verification fails, typically the user must provide more annotations in the template file. The input also includes for each cryptographic function f called by the implementation the C code of a function f_proxy that describes the correspondence of the C arguments of f to the formal arguments of the implemented primitive in the CryptoVerif model. The template file and the proxy functions form the trusted base of the verification.

In the first step, we compile the program down to a simple stack-based instruction language (CVM) using CIL [45] to parse and simplify the C input. In the next step we symbolically execute CVM programs to eliminate memory accesses

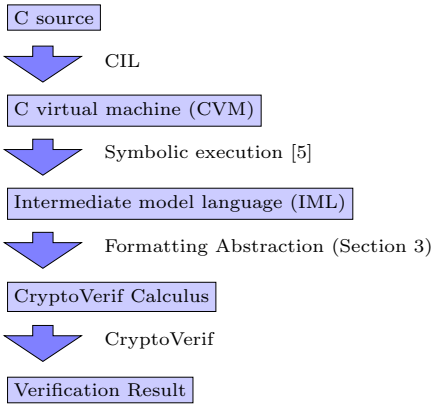

Figure 2: An outline of the method

and destructive updates, thus obtaining an equivalent program in an intermediate model language (IML)—a version of the applied pi calculus extended with bitstring manipulation primitives. Both these steps are the same as in [5]. We review them briefly in Section 4.

The new part of the method, described in Section 3, is the step that takes IML to the CryptoVerif calculus. This step abstracts away the details of message formatting that are present in IML by replacing the bitstring manipulation by application of new *formatting functions* for encoding and parsing bitstrings, so that, say, $01|\,\mathrm{len}(x_1)|x_1|x_2$ becomes $\mathit{conc}(x_1, x_2)$ and $x\{5, x\{1, 4\}\}$ becomes $\mathit{parse}(x)$, where $b|b'$ denotes the concatenation of b and b' and $b\{o, l\}$ is the substring of b starting at position o of length l.

By constructing suitable queries to an automatic prover (the SMT solver Yices [28]), we check that the expression underlying each formatting function matches the CryptoVerif type for the function. For instance, if x_1 and x_2 are both bitstrings of at most 20 bytes then $\mathit{conc}(x_1, x_2)$ is a bitstring of at most 45 bytes. We also establish certain properties that, if true, may help CryptoVerif to verify the protocol:

- parsing equations such as $\mathit{parse}(\mathit{conc}(x_1, x_2)) = x_1$,
- injectivity of encoding functions,
- applications of parsing functions in which we check that the parsed value is in the range of a certain encoder. In this case we can use the pattern matching construct of CryptoVerif, and this allows more models to be verified.

Section 4 describes two theorems that establish the soundness of our method and form the theoretical justification of our work. Section 5 describes a practical evaluation of our method against a range of cryptographic examples, totalling more than 3000 LOC.

To complete the paper, Section 6 discusses related work, while Section 7 summarizes and discusses future work.

Original Contributions of the Paper Our two main contributions are:

- A method for automatic verification of C code in the computational model, by re-targetting a model extractor based on symbolic execution (csec-modex) to an automatic verifier based on game-rewriting (CryptoVerif).
- Verification of a substantial codebase, including an independently-written real-world protocol (for smart metering), leading to a vulnerability, acknowledged and fixed by its author.

Thanks to CryptoVerif, our method supports a wider range of crypto-algorithms than before: the protocols that we verified make use of XOR, Diffie-Hellman commitments, as well as operations including public-key encryption, MACs, and authenticated encryption supported by the previous method. Previously, to use a new primitive one had to prove a computational soundness theorem or find an existing theorem and link it to our security definition. Now we have full access to all of the cryptography supported by CryptoVerif.

Challenges The main difficulty when transforming from IML to CryptoVerif is to handle message types explicitly—the previous work mentioned above targets an untyped version of ProVerif. The main task is to prove that formatting functions satisfy the given types and a peculiarity is that we make typechecking context-sensitive: when calling $f(b)$ we consider the information available at the call site to help establish that b is of the desired type (similar to contract-based verification).

The main challenge when justifying the method is to relate the semantics of IML and CryptoVerif. The main issue is that our underlying execution model (PTS) by means of which the semantics of IML is defined uses external attackers and CryptoVerif uses internal attackers. In other words, in IML the attacker is part of the semantics, whereas in CryptoVerif it is part of the process, running in parallel with the process representing the protocol.

An interesting theoretical challenge arises from the fact that cryptographic security definitions are given with respect to a security parameter, but a C implementation is written for a single value of the security parameter. In particular, it can only address a memory of a fixed size. This problem is also discussed in [42]. In our work we deal with this problem by showing how to generalize the extracted IML model to arbitrary security parameters such that all the relevant properties of the model are preserved. Unlike for C programs, this can be done rather easily, by constructing appropriate generalised parsing and encoding functions that can deal with messages of arbitrary sizes.

Limitations Given the complexity of the C language, analysis tools rely on front ends that translate the language into a much simpler representation. Examples are LLVM and CIL used by KLEE and Frama-C [21, 31]. We wrote a plugin for CIL that instruments a C program to output its own CVM representation when run. As there is no formal proof of correctness for CIL, our formal results apply to CVM programs, and the link to C is made only informally.

As in the initial work on csec-modex we only analyse a single execution path of the protocol. The CVM is produced by a single run R of the program P and thus represents a pruned program P' such that P' produces the same run. In the pruned program every statement of the form **if** (c){A}**else**{B} is replaced by **if** (c){A}**else**{ exit (1);} or **if** (c){ exit (1);} **else**{B}, depending on which branch was taken in R, and loops are unrolled the number of times they were executed in R. All our soundness results apply to the program P'. In the case of crypto-protocols we believe P' to be a close approximation of P, as the structure the protocols (such as those in the extensive SPORE repository [47]) is often linear. The code of libraries such as PolarSSL contains a lot of if-statements, but many of those have conditions that become constant for particular settings of configuration variables, or deal with cryptographic security checks and abort the execution immediately if those checks fail. We would like to provide a quantitative validation of this statement in future work. In any case, P' is a program that can successfully execute a session of the protocol in place of P, and so verifying P' increases confidence in the correctness of P.

$b \in MS$, $x, i \in Var$, $f \in$ **Ops**	
$e \in IExp ::=$	expression
b	concrete bitstring
x	variable
$f(e_1, \ldots, e_n)$	function application
$e \mid e'$	concatenation
$e\{e_o, e_l\}$	substring extraction
$\mathrm{len}(e)$	length
$P \in IML ::=$	process
0	nil
$P \mid P'$	parallel composition
$!^{i \leq N} P$	replication N times
$\mathbf{in}(c[e_1, \ldots, e_n], x);\ P$	input
$\mathbf{new}\ x\colon T;\ P$	random number
$\mathbf{out}(c[e_1, \ldots, e_n], e);\ P$	output
$\mathbf{let}\ x = e\ \mathbf{in}\ P$	assignment
$\mathbf{if}\ e\ \mathbf{then}\ P\ [\mathbf{else}\ P']$	conditional
$\mathbf{event}\ ev(e_1, \ldots, e_n);\ P$	event

Figure 3: The syntax of IML.

Our work is currently limited to trace properties of protocols, such as authentication or weak secrecy. We leave treatment of observational equivalence properties, such as strong secrecy, to future work.

An extended version of the paper with full details is available [6], together with our implementation (with outputs for each of our examples).

2. REVIEW—IML AND CRYPTOVERIF

We present the process calculus we use, the *intermediate model language* (IML), produced from symbolic execution of C, together with the CryptoVerif calculus to which IML models are translated. IML is slightly modified from the version in [5] so that the CryptoVerif calculus forms a syntactic subset of it. At the same time, the subset of IML produced by symbolic execution coincides with [5] so that it is easy to transfer the symbolic execution soundness result from there. This section presents an informal overview of the calculus, details are given in the full version.

The syntax of IML is shown in Fig. 3. We work with two semantics for IML. The first originates from [5] and is dictated by the semantics of C programs from which IML models are extracted. The second is the semantics used in the CryptoVerif tool which will be given for a subset of IML. In this paper we shall develop a transformation from IML to the CryptoVerif subset of IML such that the CryptoVerif semantics of the transformed program is sound with respect to the IML semantics of the original program.

The main differences between IML's and CryptoVerif's semantics are as follows. First, CryptoVerif semantics assumes an arbitrary security parameter. Since real-life cryptographic protocols are typically designed and implemented for a fixed value of the security parameter, we fix an arbitrary $k_0 \in \mathbb{N}$ and define the IML semantics with respect to k_0 only. Second, CryptoVerif uses a special value $\perp$ that is

distinct from any bitstring and can be returned by a computation to represent failure. The execution does not stop when such a value is encountered. There is no such special value in C, and the only way a function can fail is by aborting the execution of a process. Thus in IML if any computation returns $\bot$ at any time, the execution immediately stops. Third, CryptoVerif processes can operate on bitstrings of any length. C programs can only operate on bitstrings of a certain maximal length, so if any computation in IML, say, concatenation, would return a longer bitstring, the execution stops. We choose a fixed but arbitrary $\mathcal{N} \in \mathbb{N}$ to be the size of a machine pointer in bits and let $MS = \{0,1\}^{<2^{\mathcal{N}}}$ be the set of bitstrings that fit into machine memory. Finally, CryptoVerif programs assign types to bitstrings, but C programs and IML processes do not (in C one can assign a type to a variable, but not to the contents of a memory buffer in which a message is stored). Types will be denoted by T. For each security parameter k the interpretation of a type T is given by $I_k(T) \subseteq BS \cup \{\bot\}$, where BS is the set of all bitstrings. We shall use the following types in our modelling: the type `bitstring` representing all bitstrings and `bitstringbot` representing all bitstrings together with $\bot$, the type `bool` representing the set $\{\text{true}, \text{false}\}$, where true $= 1$ and false $= 0$, and for each $n < 2^{\mathcal{N}}$ the types $\texttt{fixed}_n$ and $\texttt{bounded}_n$ such that for the security parameter k_0 the type $\texttt{fixed}_n$ is the type of bitstrings with exactly n bits and $\texttt{bounded}_n$ is the type of bitstrings with up to and including n bits.

The calculus uses a finite set **Ops** of functions symbols such that each symbol $f \in \mathbf{Ops}$ has an associated arity n and a type declaration $f\colon T_1 \times \ldots \times T_n \to T$. The set **Ops** is meant to contain both the primitive operations of the language (such as the arithmetic or comparison operators of C) and the cryptographic primitives that are used by the implementation. IML and CryptoVerif use two different interpretations of function symbols. In IML the interpretation of a function symbol f with arity n is a function $I(f)$ from MS^n to $MS \cup \{\bot\}$ (a partial function). In CryptoVerif the interpretation of f for a security parameter k is given by a function $\tilde{I}_k(f)$ from $I_k(T_1) \times \ldots \times I_k(T_n)$ to $I_k(T)$. In both cases the functions are required to be computable in polynomial time in the length of the arguments and the security parameter. The equality function is overloaded: for each type T there is the equality function $=_T\colon T \times T \to \texttt{bool}$, but we shall omit the type index to lighten notation. We require that for each function symbol f the IML interpretation $I(f)$ and the CryptoVerif interpretation $\tilde{I}_{k_0}(f)$ coincide on arguments for which both are defined.

The calculus uses parameters, denoted by N, to bound the number of process replications. The value of a parameter N depends on the security parameter k and is denoted by $I_k(N) \in \mathbb{N}$. As IML is defined with respect to a fixed security parameter k_0, its semantics uses the value $I_{k_0}(N)$. The calculus assumes a countable set of channel names. We denote channels by c.

Expressions are evaluated with respect to an *environment* $\eta\colon Var \rightharpoonup BS$ that assigns variables to bitstrings. The expression $f(e_1, \ldots, e_n)$ applies a function to its arguments, the expression $e|e'$ concatenates two bitstrings, the expression $e\{e_o, e_l\}$ extracts a substring of e starting at position e_o of length e_l, and the expression $\text{len}(e)$ returns an $\mathcal{N}$-bit value containing the length of e. In IML if the result of any subexpression evaluation is not well-defined (for instance, substring extraction is out of bounds), the overall result is $\bot$. Also if the result of any subexpression evaluation is longer than $2^{\mathcal{N}} - 1$ bits, the overall result is $\bot$. This models the fact that a machine would crash if an intermediate result does not fit into memory. In CryptoVerif the bitstring-manipulation expressions are not available.

$x, i \in Var$, $f \in \mathbf{Ops}$

$e ::=$		expression
	x	variable
	$f(e_1, \ldots, e_n)$	function application
$Q \in CV ::=$		input process
	0	nil
	$Q \mid Q'$	parallel composition
	$!^{i \leq N} Q$	replication N times
	$\mathbf{in}(c[e_1, \ldots, e_n], x);\ P$	input
$P ::=$		output process
	$\mathbf{out}(c[e_1, \ldots, e_n], e);\ Q$	output
	$\mathbf{new}\ x\colon T;\ P$	random number
	$\mathbf{let}\ x = e\ \mathbf{in}\ P$	assignment
	$\mathbf{if}\ e\ \mathbf{then}\ P\ [\mathbf{else}\ P']$	conditional
	$\mathbf{event}\ ev(e_1, \ldots, e_n);\ P$	event

Figure 4: The syntax of the CryptoVerif calculus.

The process structure of IML is standard and is designed to mimic the CryptoVerif structure. The process 0 does nothing, the process $P|P'$ executes P and P' in parallel, the process $!^{i \leq N} P$ executes N copies of P in parallel, where N is a parameter that depends on k, as defined above. In the nth copy of P the replication index i is set to n. Communication is performed using the **in** and **out** constructs in which channel names are used with parameters, typically replication indices. The communication between two endpoints is performed only when the parameters match. This gives the attacker the power to precisely specify the recipient of a message. The construct $\mathbf{new}\ x\colon T$ generates a value of type T uniformly at random. For this T must be a fixed-length type. The construct $\mathbf{event}\ ev(e_1, \ldots, e_n)$ with an event label ev is used to flag an event during the execution. The security definitions will refer to probabilities of certain event traces in an execution. The rest of the language is standard.

The CryptoVerif calculus shown in Fig. 4 is a subset of IML in which the bitstring manipulation primitives are no longer available and the processes are separated into alternating input and output processes. We require CryptoVerif processes to be well-typed with respect to function types introduced above. The differences from the original CryptoVerif calculus in [19] are as follows:

- The **find** form of the full CryptoVerif calculus is omitted as we do not require it for modelling.
- In [19], each variable is an array accessed using replication indices, that is, variable access has the form $x[i_1, \ldots, i_n]$. This is used in the **find** construct to access variables of other processes by bringing into scope a replication index of another process. Given that we omit the **find** construct, such access is no longer possible, so in our version array access is not explicit, and the semantics uses local environments for each process instead of a single global environment. The replication indices are only used as channel parameters.

A : **event** *client_begin*$(A, B, request)$
$A \to B$: $A, \{request, k_S\}_{k_{AB}}$
B : **event** *server_reply*$(A, B, request, response)$
$B \to A$: $\{response\}_{k_S}$
A : **event** *client_accept*$(A, B, request, response)$

Figure 5: Authenticated RPC: RPC-enc

```
let A =
  if clientID = xClient then
  event client_begin(clientID, serverID, request);
  new kS_seed1: keyseed;
  let k_S = kgen(kS_seed1) in
  let msg1 = 'p'|len(request)|request|k_S in
  new nonce1: seed;
  let cipher1 = E(msg1, lookup(clientID, serverID, db), nonce1) in
  let msg2 = 'p'|len(clientID)|clientID|cipher1 in
  out(c, msg2);
  in(c, msg3);
  event client_accept(clientID, serverID, request,
          injbot^-1(D(msg3, k_S))); 0 .
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
  if client1 = xClient then
  let msg2 = injbot^-1(D(cipher1, lookup(client1, serverID, db))) 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 k_S = msg2{5 + msg2{1, 4},
              len(msg2) − (5 + msg2{1, 4})} in
  if len(msg2) − (5 + msg2{1, 4}) = 16 then
  new nonce1: seed;
  let cipher2 = E(response, k_S, nonce1) in
  out(c, cipher2); 0 .
```

Figure 6: The IML model of RPC-enc extracted from the C code.

3. TRANSLATING IML TO CRYPTOVERIF

We show the transformations that turn an IML process into a CryptoVerif process on an example. The extended version formalizes these transformations and includes the full CryptoVerif input of the example.

The Example Protocol and its Security Goals Our example protocol, due to [30], is an encryption-based variant of the RPC protocol, already considered in the papers [14, 27, 5, 4]. In the following, $\{m\}_k$ stands for the encryption, using an authenticated encryption mechanism, of plaintext m under key k while the comma represents an injective pairing operation. The protocol narration in Fig. 5 describes the process of A in client role communicating to B in server role. The key k_{AB} is a unidirectional long-term key shared between A and B (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 *request* and *response* are freshly generated. Like in [4] we assume that the server is always honest, but the client may be compromised. We write $bad(A)$ to mean that the client A is compromised.

We aim to prove authentication and secrecy properties:

1. Authentication properties state that each principal can ensure that a received message was produced by the correct protocol participant. These properties are specified using event correspondences of the form:

```
let A =
  in(c_in, ());
  if clientID = xClient then
  event client_begin(clientID, serverID, request);
  new kS_seed1: keyseed;
  let k_S = kgen(kS_seed1) in
  let msg1 = conc1(request, cast_fixed16→bitstring(k_S)) in
  new nonce1: seed;
  let cipher1 = E(msg1, lookup(clientID, serverID, db), nonce1) in
  let msg2 = conc1(clientID, cipher1) in
  out(c_out, msg2);
  in(c_in, msg3: bitstring);
  let injbot(var1) = D(msg3, k_S) in
  event client_accept(clientID, serverID, request, var1);
  out(c_out, ()); 0 .

let B =
  in(c_in, msg1: bitstring);
  let conc1(client1, cipher1) = msg1 in
  if client1 = xClient then
  let injbot(msg2) = D(cipher1, lookup(client1, serverID, db)) in
  let conc1(var2, k_S) = msg2 in
  event server_reply(client1, serverID, var2, response);
  new nonce1: seed;
  let cipher2 = E(response, cast_bitstring→fixed16(k_S), nonce1) in
  out(c_out, cipher2); 0 .
```

Figure 7: The CryptoVerif translation of the IML model of RPC-enc.

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

The first property states that, whenever *server_reply* happens, either *client_begin* has happened with corresponding parameters or the client is compromised. Similarly, the second property states that, whenever the event *client_accept* happens, the event *server_reply* has happened before. There is no compromise in this case, as we assume the server to be honest.

2. Secrecy properties state that the attacker cannot learn values of payloads, where the secrecy of response is conditional on the client being honest.

Transforming IML to CryptoVerif

The main goal of our transformations is to eliminate the string-manipulating expressions that contain concatenation and substring extraction operations. A crucial observation is that these primitives are used to implement tupling and projection operations: we replace them by application of newly introduced function symbols for tupling and projection.

We call an expression e with variables $x_1, \ldots, x_n$ an *encoding expression* when $e = e_1|\ldots|e_m$ and each e_i is either a constant, a variable x_j for some $j \leq n$, or has the form $\mathrm{len}(x_j)$ for some $j \leq n$. We call an expression e with a single variable x a *parsing expression* when $e = x\{e_o, e_l\}$ and e_o and e_l are arithmetic expressions that can themselves contain $\mathrm{len}(x)$ or other parsing expressions with the variable x. An *encoder* is a function f_c such that $f_c(b_1, \ldots, b_n) = e_c[b_1/x_1, \ldots, b_n/x_n]$ for some encoding expression e_c and a *parser* is a function f_p such that $f_p(b) = e_p[b/x]$ for some parsing expression e_p. Encoders and parsers are collectively called *formatting functions*.

To describe our transformation steps, we use RPC-enc as a running example: Fig. 6 shows the initial IML code, while Fig. 7 shows the resulting CryptoVerif model. For

$$\frac{e\colon T \quad \mathcal{E}, \mathcal{F}_P, T \vdash e \rightsquigarrow e' \quad \mathcal{E}\{x \mapsto T\} \vdash P \rightsquigarrow P'}{\mathcal{E} \vdash \textbf{let } x = e \textbf{ in } P \rightsquigarrow \textbf{let } x = e' \textbf{ in } P'}$$

$$\frac{f\colon T_1 \times \ldots \times T_n \to T \quad \mathcal{E}, \mathcal{F}_P, T_i \vdash e_i \rightsquigarrow e_i' \quad \text{prove}(\mathcal{F} \Rightarrow f(e_1, \ldots, e_n)\colon T) \quad \text{prove}(\mathcal{F} \Rightarrow f(e_1, \ldots, e_n)\colon T')}{\mathcal{E}, \mathcal{F}, T' \vdash f(e_1, \ldots, e_n) \rightsquigarrow \text{cast}_{T \to T'}(f(e_1', \ldots, e_n'))}$$

Figure 8: Selected typechecking rules.

the details of how IML is extracted from C by symbolic execution see our prior work [5, 4]. The intended meaning of the variables is as follows: *clientID* and *serverID* are global constants containing the names of an honest client and an honest server, *xClient* is the attacker-chosen name of the client that the server should communicate with, *request* and *response* are attacker-chosen bitstrings, and *db* is the key database used to look up shared keys (some of which may be compromised).

Next, we consider each transformation step in turn.

Collecting facts We record the facts that hold at each point in the process. For each subprocess P, let the *context* $\mathcal{F}_P$ be the set of conditions that are checked in the if-statements above P. During the transformations, we implicitly propagate these fact sets, that is, whenever we substitute a subprocess P by P', we also set $\mathcal{F}_{P'} = \mathcal{F}_P$.

Extracting Encoders We replace each subexpression of the form $e[e_1/x_1, \ldots, e_m/x_m]$, where e is an encoding expression, by an application of a fresh function symbol c. In our example, the definitions of *msg1* and *msg2* in A become

let *msg1* = *conc1*(*request*, k_S) **in** ...
let *msg2* = *conc1*(*clientID*, *cipher1*) **in** ...

where $\mathit{conc1}(x_1, x_2) = \text{'p'}|\,\text{len}(x_1)|x_1|x_2$.

Extracting Parsers We replace each subexpression of the form $e[e'/x]$, where e is a parsing expression, by an application of a fresh function symbol p. In our example, the definitions of *client1* and *cipher1* in B are changed to

let *client1* = *parse1*(*msg1*) **in** ...
let *cipher1* = *parse2*(*msg1*) **in** ...

where

$$\mathit{parse1}(x) = x\{5, x\{1,4\}\},$$
$$\mathit{parse2}(x) = x\{5 + x\{1,4\}, \text{len}(x) - (5 + x\{1,4\})\}.$$

Parsing Equations We record the result of applying each parser to each encoder. For each parser f_p and encoder f_c we simplify the expression $e_p[e_c/x]$, where e_p and e_c are the parsing and the encoding expression that define f_p and f_c. If the expression simplifies to a variable x_i, we record the equation $f_p(f_c(x_1, \ldots, x_n)) = x_i$. In our example, we obtain the equations:

$$\mathit{parse1}(\mathit{conc1}(x_1, x_2)) = x_1$$
$$\mathit{parse2}(\mathit{conc1}(x_1, x_2)) = x_2$$

A detail that we omit here is that the equation will not hold in IML if the arguments x_1 and x_2 are too long and *conc1* would create a result longer than $2^{\mathcal{N}}$. In the full version we treat this problem and show that the definitions of encoders and parsers can be generalised in the CryptoVerif interpretation such that the equation holds unconditionally.

Typechecking We typecheck the process, proving types of newly introduced formatting functions, and introduce typecasts where necessary. For simplicity we assume that the types for the formatting functions are given by the user along with the types of cryptographic operations. In the full version we infer most of these types during typechecking.

The fact that an expression e has type T will be denoted by a predicate intype(e, T), in particular, for each e

$$\text{intype}(e, \texttt{fixed}_n) = (\text{getLen}(e) = n),$$
$$\text{intype}(e, \texttt{bounded}_n) = (\text{getLen}(e) \leq n),$$
$$\text{intype}(e, \texttt{bitstring}) = \text{true}\,.$$

where the function getLen returns for each symbolic expression an expression representing its length in bits:

$$\text{getLen}(\text{len}(\ldots)) = \mathcal{N},$$
$$\text{getLen}(b) = |b|\,, \text{ for } b \in \mathit{MS},$$
$$\text{getLen}(x) = \text{len}(x), \text{ for } x \in \mathit{Var},$$
$$\text{getLen}(\mathit{op}(e_1, \ldots, e_n)) = \text{len}(\mathit{op}(e_1, \ldots, e_n)),$$
$$\text{getLen}(e_1|e_2) = \text{getLen}(e_1) + \text{getLen}(e_2),$$
$$\text{getLen}(e\{e_o, e_l\}) = e_l.$$

Let $f\colon T_1 \times \ldots \times T_n \to T$ be a formatting function and let e_f be the defining expression for f with variables $x_1, \ldots, x_n$. Given a context $\mathcal{F}$ and expressions $e_1, \ldots, e_n$, we denote by prove$(\mathcal{F} \Rightarrow f(e_1, \ldots, e_n)\colon T')$ the fact that the application of f satisfies the type T' (which may be different from T) *in context* $\mathcal{F}$: assuming the arguments e_i have types T_i, the expression $f(e_1, \ldots, e_n)$ must belong to type T'. Formally

$$\text{intype}(e_1, T_1) \wedge \ldots \wedge \text{intype}(e_n, T_n) \wedge \mathcal{F} \Rightarrow \text{intype}(e_f[e_1/x_1, \ldots, e_n/x_n],\, T').$$

We use two typing judgements. The judgement $\mathcal{E} \vdash P \rightsquigarrow P'$ means that in the *typing environment* $\mathcal{E}$ (a mapping from variables to types) the process P can be rewritten to a type-correct process P'. The judgement $\mathcal{E}, \mathcal{F}, T \vdash e \rightsquigarrow e'$ means that in the typing environment $\mathcal{E}$, given a context $\mathcal{F}$, the expression e can be rewritten to a type-correct expression e' of type T. We also use the judgement $e\colon T$ to mean that the declared type of e is T. In particular, $f(\ldots)\colon T$ when the return type of f is T.

Two crucial typechecking rules are shown in Fig. 8. The first rule rewrites let-statements in straightforward manner. The second rule rewrites formatting function applications. It checks that the function satisfies both its declared type T and its required type T' and inserts a typecast if necessary. A typecast from a type T to a type T' is performed by a function $\text{cast}_{T \to T'}\colon T \to T'$ which is identity on the intersection of T and T' and arbitrary elsewhere.

In our RPC-enc example we declare

$$\mathit{conc1}\colon \texttt{bitstring} \times \texttt{bitstring} \to \texttt{bitstring}$$
$$\mathit{parse1}, \mathit{parse2}\colon \texttt{bitstring} \to \texttt{bitstring}$$

We cannot give a more precise type to *conc1* because it is used to carry data of different type in the first and the second message. In particular, the session key k_S has the type **bitstring** on the server side, as it is the result of parsing the incoming message. At the same time the encryption function E has type $\texttt{bitstring} \times \texttt{fixed}_{16} \times \texttt{seed} \to \texttt{bitstring}$. Thus k_S must be guaranteed to be 16 bytes long before being passed to the encryption function. This is indeed checked

by the IML process before applying the function, and the typechecking algorithm makes use of this context information before replacing k_S by $\text{cast}_{\texttt{bitstring}\to\texttt{fixed}_{16}}(k_S)$ in the encryption.

Injectivity of Encoders We check which encoders f_c are injective, meaning that $f_c(x_1, \ldots, x_n) = f_c(x'_1, \ldots, x'_n)$ implies $x_i = x'_i$ for all i. Our check takes into account the type of the encoder. A sufficient condition for the encoder to be injective is when there is at most one argument which is not preceded by its length in the encoding and is not of a fixed-length type. Formally, an encoder $f_c\colon T_1 \times \ldots \times T_n \to T$ is injective whenever in its defining expression $e_c = e_1|\ldots|e_m$ there is at most one i such that $e_i = x_j$ for some $j \leq n$, there is no expression $\text{len}(x_j)$ preceding e_i in e, and T_j is not a fixed-length type. Applying this criterion, we see that *conc1* is injective: there is only one argument variable (x_2) that is not preceded by its length.

Treatment of Inverse Functions Some functions in the IML input may be marked by the user as inverses of other functions (this information forms part of the trusted cryptographic model). In our example these are the functions *injbot* and $injbot^{-1}$. The definition of the encryption scheme in CryptoVerif provides the natural injection function $injbot\colon T \to \texttt{bitstringbot}$ from the plaintext type T associated with the encryption scheme into the type of all bitstrings. The process uses the inverse function $injbot^{-1}\colon \texttt{bitstringbot} \to T$ to check that the result of the decryption is not $\perp$ and belongs to the type T (in the case of RPC-enc the type T is simply `bitstring`). The IML implementation $A_{injbot^{-1}}$ is assumed to return $\perp$ whenever its argument is not in the range of *injbot* (the CryptoVerif implementation $\tilde{I}_{k_0}(injbot^{-1})$ is allowed to return anything in this case).

If a function f^{-1} is marked as an inverse of an injective function f, whenever an expression $f^{-1}(e)$ occurs at the top of a subprocess P, we make explicit the check that e is in the range of f:

let $x = f^{-1}(e)$ **in if** $f(x) = e$ **then** P

In CryptoVerif, this can be abbreviated using pattern matching as **let** $f(x) = e$ **in** P, and CryptoVerif succeeds more often when given the pattern-matching form. In our example we replace the applications of $injbot^{-1}$ by pattern matching on *injbot*:

let $injbot(var1) = D(msg3, k_S)$ **in**
event $client\_accept(clientID, serverID, request, var1)$;
...
let $injbot(msg2)$
$\quad = D(cipher1, lookup(client1, serverID, db))$ **in** ...

Parsing Safety In some cases before an application of a parser $f_p(e)$ at the top of a subprocess P, the facts in the context $\mathcal{F}_P$ are sufficient to establish that the parsing is *safe*, that is, e is actually in the range of some encoder f_c. The exact method is the same as in [5] and is described in the extended version of the paper. In summary, given a bitstring b, to check that b is in the range of f_c, it is sufficient to check all the constant (tag) fields and to check that the sum of the length fields is consistent with the actual length of b. For instance, the following facts (checked in lines 2 and 3 of B) establish that *msg1* belongs to the range of *conc1*:

$$\text{'p'} = msg1\{0,1\}; \quad len(msg1) \leq 5 + msg1\{1,4\}.$$

The first fact says that *msg1* has the correct tag for *conc1* and the second says that the sum of all field lengths is consistent with the overall length of *msg1*. Any parsing of *msg1* that occurs after these facts have been checked is safe.

If f_c is an injective encoder of arity n and the parsing equation $f_p(f_c(x_1, \ldots, x_n)) = x_i$ holds for some $i \leq n$, the subprocess above is replaced by **let** $f_c(x_1, \ldots, x_n) = e$ **in** $P[x_i/f_p(e)]$, which is an abbreviation for

let $x_1 = f^{-1}_{c,1}(e)$ **in** ... **let** $x_n = f^{-1}_{c,n}(e)$ **in**
if $f_c(x_1, \ldots, x_n) = e$ **then** $P[x_i/f_p(e)]$

where $f^{-1}_{c,i}$ are the partial inverses of f_c provided by injectivity. In our example the messages *msg1* and *msg2* in B get decomposed as

let $conc1(client1, cipher1) = msg1$ **in** ...
let $conc1(var2, key) = msg2$ **in** ...

Input-Output Normalisation We insert dummy inputs and outputs to make sure that the input-output alternation is correct. In our example, we add the dummy input at the beginning of A and the dummy output at the end of A.

Replication Indices We add explicit replication indices to inputs and outputs to make sure the attacker can specifically pinpoint the process he wants to send a message to. We replace each subprocess of the form $\textbf{in}(c[], x);\ P$ by $\textbf{in}(c_{in}[b, i_1, \ldots, i_n], x);\ P$, where b is a fresh constant and $i_1, \ldots, i_n$ are the replication indices in replications above P. Similarly we replace each subprocess of the form $\textbf{out}(c[], x);\ P$ by $\textbf{out}(c_{out}[b, i_1, \ldots, i_n], x);\ P$.

In the implementation we only need to give distinct names to all the channels, as replication indices are already implicit in the CryptoVerif input.

Removing Auxiliary Tests We remove if-statements in which the condition is unlikely to help CryptoVerif, in particular those that contain inequalities or arithmetic expressions. Removing if-statements may only add new-behaviours, so if the process after removal satisfies a trace property then so does the original process.

Generating Additional Facts for CryptoVerif

We describe the additional information that we supply to CryptoVerif, beyond the process generated from IML, to help it verify the model. This information is automatically generated during the translation.

Disjointness of Encoders If two encoders $f\colon T_1 \times \ldots \times T_n \to T$ and $f'\colon T'_1 \times \ldots \times T'_m \to T'$ are known to have disjoint ranges, that is, $f(b_1, \ldots, b_n) \neq f'(b'_1, \ldots, b'_m)$ for any $b_1, \ldots, b_n$ and $b'_1, \ldots, b'_m$ of suitable types then in case $T = T'$ we add the following fact to the CryptoVerif input:

forall $x_1\colon T_1, \ldots, x_n\colon T_n, x'_1 : T'_1, \ldots, x'_m\colon T'_m;$
$\quad f(x_1, \ldots, x_n) \neq f'(x'_1, \ldots, x'_m).$

Currently we make use of constant tags in the encoding expressions to establish disjointness: if two encoding expressions have different constant fields at the same position, then the encoders that they define are disjoint. We give details in the extended version.

Parsing Equations The parsing equations that we compute during the translation are made available to CryptoVerif.

Length-Regularity of Encoders By construction all the encoders that we define are *length-regular*, that is, the length of the encoding only depends on the length of the arguments. This property is important to establish the security of RPC-enc, where the client sends the encryption of $msg1 = conc1(request1, k_S)$ and we need to prove that this encryption does not leak any information about the key k_S. By security assumption the only information that the encryption leaks is the length of the plaintext, therefore it is important that the length of $msg1$ does not depend on the value of k_S.

When CryptoVerif applies the security definition of the symmetric encryption, it replaces the plaintext $msg1$ by $Z_{\texttt{bitstring}}(msg1)$ where the intended meaning of $Z_{\texttt{bitstring}}$ is to set all bits of the input to 0. Thus we need to give enough facts to CryptoVerif so that it can rewrite $Z_{\texttt{bitstring}}(msg1)$ to a value that does not depend on k_S. This is done as follows: for each encoder $f_c\colon T_1 \times \ldots \times T_n \to T$ we consider a function $Z_T\colon T \to T$ such that $Z_T(b) = 0^{|b|}$ for every $b \in I_{k_0}(T)$ and a function $Z_{f_c} = Z_T \circ f_c$. By length-regularity of f_c these functions satisfy the following equation, which is given to CryptoVerif:

forall $x_1\colon T_1, \ldots, x_n\colon T_n$;
$Z_T(f_c(x_1, \ldots, x_n)) = Z_{f_c}(Z_{T_1}(x_1), \ldots, Z_{T_n}(x_n))$.

In the case of RPC-enc we add the equation

forall $x_1\colon$ `bitstring`, $x_2\colon$ `bitstring`;
$Z_{\texttt{bitstring}}(conc1(x_1, x_2)) = Z_{conc1}(Z_{\texttt{bitstring}}(x_1), Z_{\texttt{bitstring}}(x_2))$.

To verify RPC-enc we also require that all honestly generated keys have the same length. This assumption must be added manually to the model:

const $Z_{key}\colon$ `bitstring`.
forall $x\colon$ `keyseed`; $Z_{\texttt{bitstring}}(kgen(x))) = Z_{key}$.

Injectivity The encoders that have been determined to be injective are marked as such by adding `[compos]` to their CryptoVerif definition.

4. SOUNDNESS RESULTS

This section presents our results regarding the soundness of our approach. Theorem 1 states the main theoretical result of the present work: the translation from IML to CryptoVerif described in Section 3 preserves insecurity. Theorem 2 reviews the result from our previous work [5] stating that our method for extracting IML models from C also preserves insecurity. Together these theorems allow us to use CryptoVerif to provide security guarantees for C code as described in Section 1. Formal statements and proofs for these theorems are contained in the extended version of the paper.

IML to CryptoVerif Translation Soundness In our work we concentrate on proving *trace properties* of protocols. A trace property is a prefix-closed set of sequences of events $ev(b_1, \ldots, b_n)$, where ev is an event label and $b_1, \ldots, b_n$ are bitstrings. A typical trace property would be the authentication property described in Section 3. Given a trace property ρ we are interested in the probability that the sequence of events raised during a protocol execution does not belong to ρ. We shall call this probability the *insecurity* of a protocol.

The IML semantics and the CryptoVerif semantics naturally give rise to two different insecurity definitions. The purpose of the soundness theorem below is to relate these two definitions. Both semantics describe a probabilistic labelled transition system generated by a process, with events as labels. An important difference between the two semantics, in addition to the differences outlined in Section 2 is that in IML semantics the attacker is an external entity (a probabilistic machine) that interacts with the protocol transition system, and in CryptoVerif semantics the attacker is a part of the process. In IML we define insecurity as follows: given an IML process Q, an attacker E and a trace property ρ we let $\mathrm{insec}(Q, E, \rho)$ be the probability that the sequence of events generated by Q interacting with E does not belong to ρ.

CryptoVerif uses an *evaluation context* (a process with a hole) to model the attacker, so that for a given protocol process Q and a context C we study the transition system generated by $C[Q]$. In addition to the protocol process Q CryptoVerif takes as input a set of facts Φ that describe the interpretation $\tilde{I}$ of the function symbols used by the process. For a CryptoVerif process Q, a set of facts Φ, a trace property ρ, and a security parameter k let

$$\mathrm{cvinsec}(Q, \Phi, \rho, k) = \sup \left\{ \mathrm{cvinsec}'(Q, \tilde{I}_k, \rho, k) \;\middle|\; \tilde{I}_k \models \Phi \right\},$$

where $\mathrm{cvinsec}'(Q, \tilde{I}_k, \rho, k)$ is the probability that the sequence of events generated by Q with respect to CryptoVerif semantics for security parameter k and with interpretation of function symbols given by $\tilde{I}_k$ does not belong to ρ. Given a process Q and a set of facts Φ CryptoVerif checks that Q and Φ *satisfy the property* ρ, that is, $\mathrm{cvinsec}(C[Q], \Phi, \rho, k)$ is negligible in k for any evaluation context C.

Given an IML process Q the translation described in Section 3 generates a CryptoVerif process $\tilde{Q}$ and a set of facts Φ. The soundness of this translation is captured by the following theorem:

Theorem 1 (CryptoVerif Translation is Sound) *Let Q be an IML process that successfully translates to a CryptoVerif process $\tilde{Q}$ and a set of facts Φ. Then for any attacker E there exists an evaluation context C whose runtime is polynomial in the runtime of E such that for any correspondence property ρ*

$$\mathrm{insec}(Q, E, \rho) \leq \mathrm{cvinsec}(C[\tilde{Q}], \Phi, \rho, k_0). \qquad \square$$

CryptoVerif checks that $\tilde{Q}$ satisfies the correspondence ρ, that is, $\mathrm{cvinsec}(C[\tilde{Q}], \rho, k)$ is negligible in k for any evaluation context C. Theorem 1 implies that Q is a secure realisation of $\tilde{Q}$ with respect to the property ρ for a particular security parameter k_0. In fact, CryptoVerif can provide explicit bounds on $\mathrm{cvinsec}(C[\tilde{Q}], \rho, k)$ with respect to k, which allows us to give an explicit bound on $\mathrm{insec}(Q, E, \rho, t)$.

Symbolic Execution Soundness In [5] we introduced a simple low-level instruction language CVM to which we compile C using the CIL framework [45]. The semantics of CVM is given using the same formalism as for IML, which allows us to combine both languages in the same execution. For instance, CVM processes $P_1, \ldots, P_n$ can be subprocesses of an IML process P_E with n holes, forming a mixed process $P_E[P_1, \ldots, P_n]$. This allows us to write protocol participants in C, and then specify their intended execution environments, including long-term shared key infrastructure and corruption models, in IML. The semantics and the insecurity measure are defined in such a way as to include mixed processes.

In [5], we developed a symbolic execution algorithm that, if successful, extracts an IML model from a CVM process. Even though IML defined in this paper is slightly different, it is straightforward to adapt the soundness theorem from [5] to our new setting:

Theorem 2 (Symbolic Execution is Sound) *If $P_1, \ldots, P_n$ are CVM processes and $\tilde{P}_1, \ldots, \tilde{P}_n$ are IML processes resulting from their symbolic execution then for any IML process P_E, attacker E, and trace property ρ the following holds:*

$$\mathrm{insec}(P_E[P_1, \ldots, P_n], E, \rho) \leq \mathrm{insec}(P_E[\tilde{P}_1, \ldots, \tilde{P}_n], E, \rho).\square$$

Unlike [5] we do not use the parameter that bounds the number of execution steps. The reason is that the execution time of both IML and CVM processes is polynomially bounded—CVM processes are linear and do not include any looping, and IML processes have polynomial bounds on the number of replications. In [5] replication was unbounded, and so we needed to force a bound on the runtime in the security definition.

In a nutshell, the significance of our soundness theorems is as follows: let $P_1, \ldots, P_n$ be implementations of protocol participants in CVM and let P_E be an IML process that describes an execution environment. Assume that $P_1, \ldots, P_n$ are successfully symbolically executed with resulting models $\tilde{P}_1, \ldots, \tilde{P}_n$, and the IML process $P_E[\tilde{P}_1, \ldots, \tilde{P}_n]$ is successfully translated to a CryptoVerif process Q that satisfies a trace property ρ, as checked by CryptoVerif. Then we know that Q is a protocol model that is (asymptotically) secure with respect to ρ. By Theorems 1 and 2 we know that $P_1, \ldots, P_n$ form a secure implementation of the protocol described by Q for the security parameter k_0.

Furthermore, even though [18] only mentions asymptotic security, CryptoVerif provides explicit bounds on the insecurity of the protocol. This can be used to give concrete bounds on insecurity of the original C protocol implementation.

5. IMPLEMENTATION & EXPERIMENTS

We have implemented our approach by reusing the symbolic execution stage from the implementation in [5] (4600 lines of OCaml code) and adding to it an implementation of the translation described in this paper (about 700 lines of OCaml code).

Fig. 9 shows a list of protocol implementations from the Csec Challenge repository (presented in [4]) on which we tested our method. We list some statistics regarding the sizes of the different stages of the translation: the size of the original C code, the number of generated CVM instructions, the number of lines in the extracted IML model, the number of lines in the CryptoVerif template to be provided by the user (containing the environment process and cryptographic assumptions), and the size of the CryptoVerif input, which consists of the user template and the automatically generated model. We also show execution times (with runtimes of the symbolic execution and CryptoVerif combined) and the list of used cryptographic primitives/assumptions for each protocol.

Simple MAC is an implementation of the example protocol from [5], in which a single payload is concatenated with its MAC. We successfully verify the authenticity of the payload.

Simple XOR is the one-time pad example from Fig. 1. CryptoVerif proves strong secrecy of the payload. As our soundness results apply to trace properties only, we can claim weak secrecy of the payload in the C implementation.

The NSL example is our implementation of the Needham-Schroeder-Lowe protocol. The work reported in [5] obtained a computational verification result using the soundness theorem from [7]. The theorem assumes that all cryptographic material is tagged, and in particular that nonces should be tagged. In the work we present here, we removed this assumption which led to the discovery of a potential flaw: if the 3rd message (B's nonce) is sent out without any tagging, it can be decrypted and parsed as the first message of the protocol (an encoding of the concatenation of A's nonce and A's identity). The failure of the parsing may reveal information about the nonce to the attacker. Once we fixed the problem by explicitly tagging the 3rd message, CryptoVerif successfully verified the protocol. This highlights that computational soundness results are not suitable for verifying existing protocol implementations because they assume conditions that are rarely enforced in practice.

RPC is an implementation due to François Dupressoir of the MAC-based remote procedure call protocol described in [11]. We verified the authenticity of the client request and the server response, this time with computational guarantees.

RPC-enc is the version of the remote procedure call protocol that uses authenticated encryption, described in detail in Section 3. We verified the authenticity of the request and the response, and the secrecy of the payloads (which is not protected by the MAC-based RPC). The extended version of the paper contains our full CryptoVerif model of this protocol.

The metering example is an implementation of a privacy-friendly protocol for smart electricity meters [48] developed at Microsoft Research. In the work reported in [5], an IML model of the protocol was extracted which revealed several bugs, but there was no verification result because the protocol uses XOR and Diffie-Hellman commitments. In this work, we close the gap and provide the verification result for the metering protocol, showing both authentication (whenever the consumer accepts a reading, it does indeed originate from the meter) and the strong secrecy of the readings, which implies at least weak secrecy for the C implementation. The protocol implementation works with an arbitrary number of messages, that are all batched and signed together. Neither our symbolic execution nor CryptoVerif can deal with loops, so we unroll the main loop of the protocol and only analyse it for a fixed number of messages. The table of results contains numbers for the verification with one and three messages.

When trying to verify the protocol for more than one message, we uncovered a potential security flaw that led to a fix in the code. Given two readings r_1 and r_2, the protocol would generate commitments C_1 and C_2 by using calls to bignum operations of the OpenSSL crypto library. It would then concatenate the commitments without using their lengths as "`tag`"$|C_1|C_2$ (with some extra constant fields which we omit for simplicity). As bignums in OpenSSL can have arbitrary length, this could lead to a collision where two different bignums C_1' and C_2' form the same string and lead to the same signature.

In summary, we were able to verify six protocols in the

	C LOC	CVM Instructions	IML LOC	Template LOC	CV LOC	Outcome	Time	Primitives
Simple MAC	~ 250	7K	27	54	107	verified	4s	UF-CMA MAC
Simple XOR	~ 100	3K	6	51	90	verified	4s	XOR
NSL	~ 450	25K	83	156	241	flaw, verified	26s	IND-CCA2 PKE
RPC	~ 600	44K	48	58	126	verified	5s	UF-CMA MAC
RPC-enc	~ 700	19K	60	135	200	verified	6s	IND-CPA INT-CTXT AE
Metering(1)	~ 1000	69K	58	181	266	flaw, verified	21s	UF-CMA sig, CR/PRF hash
Metering(3)	—	96K	98	181	280	as above	41s	XOR, DH

Figure 9: Summary of analysed implementations.

computational model (as opposed to only one specially designed implementation in previous work). The verification has also become much easier, as one does not need to look for a computational soundness result each time one deals with a new set of primitives, and to prove that the new soundness result is compatible with our security definitions.

6. RELATED WORK

We describe the most closely related work. A more detailed discussion of further related work (including [32, 33, 49]) can be found in [4]; for recent surveys of related work in higher-level languages, see [29] and [38].

The most closely related works include [36, 23, 24, 27, 22]. Csur [36] is one of the first attempts at cryptographic verification of C code. Here a C program is used to generate a set of Horn clauses, which are then solved using a theorem prover. ASPIER [23] applies software model checking to verify properties of bounded instances of the handshake of OpenSSL, within the symbolic model. ASPIER handled a codebase of 2400 LOC consisting of the client side and the server side of the handshake; to the best of our knowledge, this was previously the largest codebase of cryptographic software in C to be verified with a single tool. [24] presents an approach based on the KLEE test-generation tool [21] to verify cryptographic protocol implementations. Testing for trace properties is not supported (as documented in the paper). KLEE is based on symbolic execution similar to our work, however, [24] only supports buffers of fixed length. [27] uses a general-purpose verifier for security verification of C code. In comparison to the work presented here, this approach requires the code to be annotated (about one line of annotation per line of code) and performs verification only in the symbolic model. [22] nicely complements our work, by showing how to start from a verified CryptoVerif model and to transform to executable protocol code in OCaml. The transformation is illustrated on the SSH Transport Layer Protocol.

We successfully verify over 3000 LOC, more than any prior work on cryptographic code in C. On the other hand, the tools mentioned above (CSur, ASPIER, KLEE, VCC, and the CryptoVerif to OCaml compiler) have the advantage that they are not limited to single execution paths.

In recent work, [46] develop a stepwise refinement approach to verifying invariants of security code using VCC.

There is also related work on verification of crypto-protocol implementations in functional languages such as F# wrt. a symbolic or computational model. [15] presents implementations of cryptographic protocols verified against a symbolic security model using the tool fs2pv. [13] presents a verification of a small functional implementation of the Transport Layer Security protocol (TLS 1.0) using automated computational cryptographic verification with the fs2cv tool that translates F# to CryptoVerif. [8] presents an approach for security protocol verification on the source code level in a computationally sound way at the hand of F# implementations. The difference to our work lies in the complexity of C code compared to code in functional programming languages, e.g. memory and pointers. We manage this complexity in so far as our extracted models are much shorter and simpler than the original C.

This work also has to be seen within the context of other applications of the fs2cv tool, including work on computationally sound mechanized proofs for basic and public-key Kerberos [20], as well as other work on computational soundness for formal security verification of crypto-protocols, such as [2, 37, 1, 43, 25]. General-purpose verifiers have also been applied to the related problem of verifying the code of cryptographic algorithms [9]. [42] considers the problem of establishing computational indistinguishability for Java-like programs that use cryptography. It proposes an approach that combines techniques from program analysis and simulation-based security. The approach is orthogonal to ours in that it considers observation equivalence, but not trace properties. Since the paper does not report on large examples, we cannot compare scalability aspects with our approach.

7. CONCLUSION

To summarize, we adapted our existing symbolic execution tool to generate models for CryptoVerif. Qualitatively, we obtain much stronger security results, in the computational model, than in existing work—the security theorems of [4] applied only to one example. Quantitatively, we verify a larger codebase of cryptographic code in C than any prior method. In a sense, we have repeated the experiment of using symbolic execution to extract verifiable models from C code, and for a second time we find that it scales effectively.

In future, it would be valuable to further validate the scalability and usability of the method. Our tools are available online, and we are interested to work with protocol designers who are developing new protocols, towards the vision of a toolset to allow reference implementations to be developed concurrently with protocol design and verification.

The aim of our work is to verify a broad set of realistic protocol implementations while preserving soundness. We did not provide any formal completeness results, partly because no such results are known for CryptoVerif that we use as part of our toolchain. Nevertheless, an informal outline of the applicability of our method is useful, and we provide such an outline in the extended version.

Acknowledgments. We are grateful to Bruno Blanchet for his help with running CryptoVerif. George Danezis provided access and advice regarding his metering code. François Dupressoir commented on a draft. Anupam Datta provided statistics on the codebase analysed by ASPIER. The work is supported by the Microsoft Research PhD Scholarship. We thank the anonymous referees for their helpful comments.

8. REFERENCES

[1] M. Abadi, B. Blanchet, and H. Comon-Lundh. Models and proofs of protocol security: A progress report. In A. Bouajjani and O. Maler, editors, *CAV*, volume 5643 of *Lecture Notes in Computer Science*, pages 35–49. Springer, 2009.

[2] M. Abadi and P. Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). *J. Cryptology*, 15(2):103–127, 2002.

[3] M. Abe and V. D. Gligor, editors. *Proceedings of the 2008 ACM Symposium on Information, Computer and Communications Security, ASIACCS 2008, Tokyo, Japan, March 18-20, 2008*. ACM, 2008.

[4] M. Aizatulin, F. Dupressoir, A. D. Gordon, and J. Jürjens. Verifying cryptographic code in C: Some experience and the Csec challenge. In *Formal Aspects of Security and Trust (FAST 2011)*, Lecture Notes in Computer Science. Springer, 2011.

[5] M. Aizatulin, A. D. 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`.

[6] M. Aizatulin, A. D. Gordon, and J. Jürjens. Computational verification of C protocol implementations by symbolic execution. Technical Report MSR–TR–2012–80, Microsoft Research, 2012.

[7] M. Backes, D. Hofheinz, and D. Unruh. CoSP: A general framework for computational soundness proofs. In *ACM Conference on Computer and Communications Security*, pages 66–78, 2009. Preprint on IACR ePrint 2009/080.

[8] M. Backes, M. Maffei, and D. Unruh. Computationally sound verification of source code. In *CCS*, 2010.

[9] 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.

[10] G. Barthe, B. Grégoire, S. Heraud, and S. Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In *Advances in Cryptology – CRYPTO 2011*, Lecture Notes in Computer Science. Springer, 2011.

[11] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement types for secure implementations. In *CSF '08: Proceedings of the 2008 21st IEEE Computer Security Foundations Symposium*, pages 17–32. IEEE Computer Society, 2008.

[12] K. Bhargavan, R. Corin, C. Fournet, and E. Zalinescu. Automated computational verification for cryptographic protocol implementations. Unpublished draft, Oct. 2009.

[13] K. Bhargavan, C. Fournet, R. Corin, and E. Zălinescu. Cryptographically verified implementations for TLS. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 459–468, Alexandria, VA, Oct. 2008. ACM.

[14] 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.

[15] K. Bhargavan, C. Fournet, A. D. Gordon, and N. Swamy. Verified implementations of the information card federated identity-management protocol. In Abe and Gligor [3], pages 123–135.

[16] K. Bhargavan, C. Fournet, A. D. Gordon, and S. Tse. Verified interoperable implementations of security protocols. In *CSFW '06: Proceedings of the 19th IEEE workshop on Computer Security Foundations*, pages 139–152. IEEE Computer Society, 2006.

[17] B. Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *CSFW*, pages 82–96. IEEE Computer Society, 2001.

[18] B. Blanchet. Computationally sound mechanized proofs of correspondence assertions. In *20th IEEE Computer Security Foundations Symposium (CSF'07)*, pages 97–111, Venice, Italy, July 2007. IEEE.

[19] B. Blanchet. A computationally sound mechanized prover for security protocols. *IEEE Transactions on Dependable and Secure Computing*, 5(4):193–207, Oct.–Dec. 2008.

[20] B. Blanchet, A. D. Jaggard, A. Scedrov, and J.-K. Tsay. Computationally sound mechanized proofs for basic and public-key kerberos. In Abe and Gligor [3], pages 87–99.

[21] 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.

[22] D. Cadé and B. Blanchet. From computationally-proved protocol specifications to implementations. In *International Conference on Availability, Reliability and Security (ARES 2012)*, 2012.

[23] S. Chaki and A. Datta. ASPIER: An automated framework for verifying security protocol implementations. In *Computer Security Foundations Workshop*, pages 172–185, 2009.

[24] 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.

[25] V. Cortier, S. Kremer, and B. Warinschi. A survey of symbolic methods in computational analysis of cryptographic systems. *J. Autom. Reasoning*, 46(3-4):225–259, 2011.

[26] D. Dolev and A. Yao. On the Security of Public-Key Protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

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

[28] B. Dutertre and L. D. Moura. The Yices SMT Solver.

Technical report, Computer Science Laboratory, SRI International, 2006.

[29] 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)*, Lecture Notes in Computer Science. Springer, 2011. To appear.

[30] C. Fournet, M. Kohlweiss, and P.-Y. Strub. Modular code-based cryptographic verification. In *18th ACM Conference on Computer and Communications Security (CCS 2011)*, 2011.

[31] `http://frama-c.cea.fr/`, 2009.

[32] 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.

[33] 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.

[34] S. Goldwasser and S. Micali. Probabilistic encryption. *Journal of Computer and System Science*, 28(2):270–299, 1984.

[35] A. D. Gordon. Provable implementations of security protocols. In *LICS*, pages 345–346, 2006.

[36] 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.

[37] P. Gupta and V. Shmatikov. Towards computationally sound symbolic analysis of key exchange protocols. In V. Atluri, P. Samarati, R. Küsters, and J. C. Mitchell, editors, *FMSE*, pages 23–32. ACM, 2005.

[38] 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.

[39] A. Jeffrey and R. Ley-Wild. Dynamic model checking of C cryptographic protocol implementations. In *Proceedings of Workshop on Foundations of Computer Security and Automated Reasoning for Security Protocol Analysis*, 2006.

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

[41] R. Küsters and T. Truderung. Reducing protocol analysis with XOR to the XOR-free case in the horn theory based approach. *Journal of Automated Reasoning*, 46(3):325–352, 2011.

[42] 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*. IEEE Computer Society, 2012.

[43] R. Küsters and M. Tuengerthal. Computational soundness for key exchange protocols with symmetric encryption. In E. Al-Shaer, S. Jha, and A. D. Keromytis, editors, *ACM Conference on Computer and Communications Security*, pages 91–100. ACM, 2009.

[44] P. Lincoln, J. C. Mitchell, M. Mitchell, and A. Scedrov. Probabilistic polynomial-time equivalence and security analysis. In J. M. Wing, J. Woodcock, and J. Davies, editors, *World Congress on Formal Methods*, volume 1708 of *Lecture Notes in Computer Science*, pages 776–793. Springer, 1999.

[45] 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.

[46] N. Polikarpova and M. Moskal. Verifying implementations of security protocols by refinement. In *Verified Software: Theories, Tools and Experiments (VSTTE 2012)*, volume 7152 of *Lecture Notes in Computer Science*, pages 50–65. Springer, 2012.

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

[48] A. Rial and G. Danezis. Privacy-friendly smart metering. Technical Report MSR–TR–2010–150, Microsoft Research, 2010.

[49] 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.

[50] D. Unruh. The impossibility of computationally sound XOR, July 2010. Preprint on IACR ePrint 2010/389.