Guiding a general-purpose C verifier to prove cryptographic protocols

Citation for published version:
Dupressoir, F, Gordon, A, Jürjens, J & Naumann, DA 2014, 'Guiding a general-purpose C verifier to prove cryptographic protocols' Journal of Computer Security, vol. 22, no. 5, pp. 823-866. DOI: 10.3233/JCS-140508

Digital Object Identifier (DOI):
10.3233/JCS-140508

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Early version, also known as pre-print

Published In:
Journal of Computer Security

Guiding a General-Purpose C Verifier to Prove Cryptographic Protocols

François Dupressoir
The Open University

Andrew D. Gordon
Microsoft Research

Jan Jürjens
TU Dortmund & Fraunhofer ISST

David A. Naumann
Stevens Institute of Technology

***Abstract*—We describe how to verify security properties of C code for cryptographic protocols by using a general-purpose verifier. We prove security theorems in the symbolic model of cryptography. Our techniques include: use of ghost state to attach formal algebraic terms to concrete byte arrays and to detect collisions when two distinct terms map to the same byte array; decoration of a crypto API with contracts based on symbolic terms; and expression of the attacker model in terms of C programs. We rely on the general-purpose verifier VCC; we guide VCC to prove security simply by writing suitable header files and annotations in implementation files, rather than by changing VCC itself. We formalize the symbolic model in Coq in order to justify the addition of axioms to VCC.**

I. Introduction

Economies of scale suggest that it is better, where possible, to adapt an existing general-purpose tool to a specialist problem, than to go to the expense of building a specialist tool for each niche application area.

Our particular concern is the specialist problem of verifying the implementation code of cryptographic protocols [30][29][10]. This code is mostly written in C, and is often the first—and sometimes the only—completely precise description of the message formats and invariants of cryptographic protocols. Hence, reasoning about such code offers a way to find and prevent both the design and implementation flaws that lead to expensive failures (for instance, [15][38][36]).

Background: Proving Cryptographic Protocol Code: The prior work on verifying C code of security protocols relies on special-purpose tools. Csur [30] analyzes C code for secrecy properties via a custom abstract interpretation, while ASPIER [16] relies on security-specific software model-checking techniques, obtaining good results on the main loop of OpenSSL. Both these tools use the symbolic model of cryptography introduced by Dolev and Yao [24].

Another line of work considers the problem of verifying reference implementations written as functional programs. Initial approaches rely on security-specific analyzers. The tools FS2PV [10] and FS2CV [28] translate functional programs in F# to the process calculi accepted by the specialised verifiers ProVerif [13] and CryptoVerif [14] for automatic verification in the symbolic and computational models; an implementation of TLS [11] is a substantial case study.

Instead of translating to a protocol verifier, the type-checker F7 [9] checks F# by using security-specific refinement types, types qualified with formulas, to express security properties. The theory of F7 is based on the symbolic model, although in some circumstances it can be adapted to be provably computationally sound [5][27].

A subsequent, more flexible, method of using refinement types, based on *invariants for cryptographic structures* [12], relies on axiomatizations of cryptographic predicates (such as which data is public); first implemented for F7, the method works in principle with any general-purpose refinement-type checker.

Our strategy is to port this method to a verifier for C.

Background: General-Purpose C Verifiers: By now there are several general-purpose and more-or-less automatic verification tools for C, including Frama-C [22], VeriFast [32], and VCC [18]. This paper describes our techniques for guiding one of these, VCC, to verify a range of security protocol implementations. Although we adopt VCC, we expect our method would port to other tools.

VCC verifies C code against specifications written as function contracts in the tradition of Floyd-Hoare logic. It translates C to an intermediate verifier, Boogie [7], which itself relies on an SMT solver, Z3 [23]. The translation to Boogie encodes an accurate low-level memory model for C. VCC supports concurrency, which we use to model distributed execution of protocols as well as for multithreaded code. Specifications use *ghost state*, that is, specially-marked program variables that may be mentioned in contracts, but that are not allowed to affect ordinary state or control flow. We aim to scale to verify large amounts of off-the-shelf C code such as OpenSSL; VCC has already proved itself capable of verifying large pre-existing codebases [18].

A. Outline of our Techniques

We summarize the main aspects of our adaptation to C and VCC of the method of *invariants for cryptographic structures* [12]. Later on, we describe the differences from the prior work on F# and F7.

1) Language-independent definitional theory: We develop a theory of symbolic cryptography, independent of any programming language, within the interactive proof assistant Coq. The theory is definitional in the sense that it is developed from sound definitional principles (on the basis of symbolic cryptography), with no additional assumptions.

As usual in the symbolic model [24], the core of our theory is an algebraic type with constructors corresponding to the following: the outcomes of cryptographic algorithms such as keyed hashing, encryptions, and signatures; literal byte strings (which represent messages, principal names, keys, and nonces, etc); and reversible pairing (for message formatting; implemented with a length field).

Our theory accounts for the time-dependent history of protocol execution by defining a *log*, $\mathcal{L}$, to be a set of *events*, which records progress so far in a protocol run.

Our theory also includes an inductive definition of confidentiality levels of terms, parameterized by the log of events. Terms may either be public (known to the attacker), or private (known only to the protocol participants). We need to parameterize by the log because once events such as a principal compromise are logged, more data becomes public (such as keys known to the principal).

2) Theory imported as first-order axioms: C is a low-level language and does not directly support abstractions for algebraic types. Also, VCC cannot easily perform proofs by induction, being based on first-order logic without induction principles. Still, VCC does allow ghost state and ghost commands to manipulate unbounded data—ghost data of type mathint consists of arbitrary mathematical integers—and VCC allows us to assume arbitrary first-order axioms. Hence, we can import our Coq theory into VCC by (1) using ghost data of type mathint to represent algebraic types such as symbolic terms, and (2) importing Coq theorems about our inductive definitions as first-order axioms. Results proved by VCC hold in all models of the axioms, including the intended one inductively defined in Coq.

3) A ghost table relates bytestrings and symbolic terms: Our cryptographic library manipulates byte arrays via a C struct bytes_c, which contains a length field together with a pointer to a heap-allocated chunk of memory with that length. Additionally, the struct contains a mathint ghost field, encoding, satisfying an invariant that it encodes the actual bytestring stored in memory. In global ghost state, we maintain a *representation table*, which holds a finite one-to-one correspondence between bytestrings and symbolic terms. These are the cryptographically significant bytestrings arising so far in the run. The predicate table.DefinedB$[b]$ holds just if bytestring b exists in the table. If so, table.B2T$[b]$ is the corresponding term. Conversely, if term t is in the table, table.T2B$[t]$ is the corresponding bytestring.

We rely on the table to write VCC function contracts that specify symbolic assumptions about concrete cryptographic routines. For example, the contract for hmacsha1 follows. It enforces that t_b can be MAC'ed with t_k only when the protocol-dependent precondition MACSays(t_k, t_b) holds.

```
int hmacsha1(bytes_c *k, bytes_c *b, bytes_c *res)
requires(table.DefinedB[k->encoding])
requires(table.DefinedB[b->encoding])
ensures(!result ==> table.DefinedB[res->encoding])
requires
  (MACSays(table.B2T[k->encoding],table.B2T[b->encoding]))
ensures(!result ==>
  table.B2T[res->encoding] ==
    Hmac(table.B2T[k->encoding],table.B2T[b->encoding]));
```

The contract's first three lines express the precondition (using the **requires** keyword) that the two concrete arguments are in the table, and the postcondition (using the **ensures** keyword) that the concrete result is in the table. The fourth line requires the MACSays predicate is fulfilled. The final line ensures the term associated with the result is Hmac(t_k, t_b), where terms t_k and t_b are associated with the concrete inputs. (As these lines illustrate, we include the ghost field encoding in bytes_c because it allows succinct access in specifications to the contents of memory.)

The VCC-verified concrete implementation of hmacsha1, called a *hybrid wrapper*, simply calls a routine trusted to compute the MAC of the inputs, and then in ghost code updates the table with the result, if it is new.

4) Protocol roles described as ordinary C code: Each role of a protocol is simply code in C, executed as normal, and verified for memory safety and security with VCC. We model distributed execution by multiple threads that communicate concretely by message passing via a network API, but that share a single representation table. The protocol code can itself be multithreaded and use shared memory, but that feature is not used in the simple examples presented here.

Throughout this article, we take as a running example the following simple authenticated RPC [12]. This two-party protocol uses a pre-shared secret key to authenticate requests and responses and link responses to the corresponding request, using a keyed hash as MAC. Our verified C code for this protocol is available in the full version online.

Running example: an authenticated RPC protocol

a	: Log(Request(a, b, payload))
a $\rightarrow$ b:	payload | hmac(kab, "1" | payload)
b	: **assert**(Request(a, b, payload))
b	: Log(Response(a, b, payload, payload'))
b $\rightarrow$ a:	payload' | hmac(kab, "2" | payload | payload')
a	: **assert**(Response(a, b, payload, payload'))

The narration logs events marking a's intent to send a request, and b's intent to send a response. At these points in our code, ghost commands add events to the log. The narration also includes correspondence assertions marking b's conclusion that a has sent a request, and a's conclusion that b has sent a response. In our code, these correspondences become **assert** statements, to be verified by VCC.

5) Attacker model expressed using C interface: We prove protocol code secure against a network-based attacker [37], rather than against say local malware. We consider an attacker to be a C program consisting simply of a series of calls to functions in the *attacker interface*. In keeping with the symbolic model, the attacker cannot directly manipulate bytestrings using the bitwise operators of C, but only via this interface. It includes functions for cryptography, to send and receive network messages, to create new principals (but without access to their keys), to create instances of protocol roles, and to cause the compromise of principals (after which their keys are available). Since our security results hold in spite of an arbitrary attacker, we place no bound on the number of distinct principals or concurrent sessions.

6) Security theorems obtained by running a general-purpose verifier: By running VCC on the protocol implementation, we prove both correspondence properties, expressing authentication and integrity properties, and weak secrecy properties. As mentioned, correspondences amount to embedded **assert** statements. Standard symbolic cryptography assumptions are expressed using local **assume** statements. Weak secrecy properties amount to consequences of invariants, respected by all verified code. A typical secrecy property can be explained as follows: if k is a key shared between a and b and k is public, then either principal a or principal b is compromised. Proof with VCC is semi-automatic in that it relies on automatic deductive inference, but with the help of user-supplied annotations.

B. Contributions of the Paper

To the best of our knowledge, this is the first published verification methodology for C implementations of cryptographic protocols that proves both memory safety and security properties for unbounded sessions. Csur [30] proves secrecy properties, but does not show memory safety; in fact, verification succeeds despite the example code allowing accidental access to uninitialized memory. ASPIER [16] proves various security properties by software model-checking, but verification considers only a few concurrent sessions, and relies on substantial abstractions.

Our work takes the idea of invariants for cryptographic structures [12] away from strongly-typed functional programming in F# and F7, and recasts it in the setting of a weakly-typed low-level imperative language. In C we can neither rely on abstract types nor escape from the difficulties of reasoning about mutable memory and aliasing. These are probably the main new difficulties we address compared to the prior work with F7 [12]; verifying C is much harder than verifying F#. In return, we enjoy vastly wider applicability, as the bulk of production cryptographic protocols is in C. A less obvious and more technical benefit is that in F7, the log of events is implicit and its impact on inductively defined predicates requires a bespoke notion of semantic safety for F#. By making the log explicit in ghost state, we can work within a completely standard semantics of assertions on C programs.

Our method forces the developer to precisely specify memory safety and security properties. We verify them with a scalable and practically reliable tool that has clear semantics in terms of standard C, its compilers and hardware architecture. Since user interaction is by way of code annotation, the verification effort may be expected to evolve well as the code base evolves. We may also hope to reap the benefit of ongoing improvements in automation for general-purpose verifiers for C. Although we prove our security-specific theory in Coq, we do trust the VCC/Boogie/Z3 tool chain and the C compiler.

We have validated our approach on implementations we developed using our own cryptographic APIs. In future work, we intend to apply our techniques to pre-existing code.

C. Structure of the Paper

We verify the following stack of C program files, listed in dependency order, which link to form an executable.

crypto.h/c	library: crypto, malloc, etc (not verified)
RPCdefs.h	representation table, event log
RPChybrids.h/c	hybrid wrappers
RPCprot.h/c	protocol roles, setup
RPCshim.h/c	network attacker interface
RPCattack_0.c	sample attack / application

Section II introduces features of VCC used in our treatment of symbolic cryptography (the topic of Section III, file RPCdefs.h) and its connection to concrete data (the topic of Section IV, files RPChybrids.h and RPChybrids.c). Section V states our assumptions about VCC. Section VI models symbolic attacks as programs (e.g., RPCattack_0.c) using an API RPCshim.h. Section VII states and proves safety of an example protocol implementation (RPCprot.h and RPCprot.c). Section VIII summarizes our results including verification of other examples. Section IX covers related work. Section X concludes with remarks on limitations and future work.

A preliminary form of this work was presented at an unrefereed workshop [25]. A technical report [26] has additional details. All of the Coq and VCC files are available online.

II. Background on the VCC Verifier

VCC uses an automatic theorem prover to statically check correctness of C code with respect to specifications written as function contracts and other annotation comments. The tool is based on a precise model of multithreaded, shared-memory executions of C programs. In order to verify rich functional specifications without the need for interactive theorem proving and yet scaling to industrial software using idiomatic C, VCC relies on a somewhat intricate methodology for specifications. This section sketches pertinent features of the model and methodology. For details that are glossed over here, see [18], [19] and the tool documentation. We expect the reader is familiar with C syntax such as macro definitions (**#define**) and record declarations (**typedef struct**).

VCC's semantics of C is embodied in its verification condition generator (VCG). The VCG reflects a reasoning methodology that includes memory safety and locally checked invariants. The VCG models preemptive multithreading by interpreting code in terms of its atomic steps, between each of which there may be arbitrary interference on shared state, constrained only by invariants associated with data types declared in the program as explained later. Atomicity is with respect to sequentially consistent hardware and data types like integers with atomic read and write. (The methodology has been adapted to reasoning about a weaker memory model, Total Store Order, but that has not yet been implemented in VCC [20].)

Memory blocks are arrays of bytes, but a typed view is imposed in order to simplify reasoning while catering for idiomatic C and standard compilers. The verifier attempts to associate a type with each pointer dereferenced by the program, and imposes the requirement that distinct pointers reference separate parts of memory. For example two integers cannot partially overlap. Structs may nest as fields inside other structs, in accord with the declared struct types, but distinct values do not otherwise overlap. Annotations can specify, however, the re-interpretation of an int as an array of bytes, changing the typestate of a union, etc.

The declaration of a struct type can be annotated with an invariant: a formula that refers to fields of an instance $this$. (We often say "invariant" for what are properly called "type invariants".) Invariants need not hold of uninitialized objects, so there is a boolean ghost field that designates whether the object is *open* or *closed*: in each reachable state, every closed object should satisfy the invariant associated with its type.

Useful invariants often refer to more than one object, but the point of associating invariants with objects is to facilitate local reasoning: when a field is written, the verifier only needs to check the invariants of relevant objects, owing to *admissibility* conditions VCC imposes on invariants. Invariants and other specifications designate an *ownership* hierarchy: if object o_1 owns o_2 then the invariant of o_1 may refer to the state of o_2 and thus must be maintained by updates of o_2. The state of a thread is modeled by a ghost object. An object is *wrapped* if it is owned by the current thread (object) and closed. The owner of an object is recorded in a ghost field. VCC provides notations **unwrap** and **wrap** to open/close an object, with **wrap** also asserting the invariant.

Ownership makes manifest that the invariant for one object o_1 may depend on fields of another object o_2, so the VCG can check o_1's invariant when o_2 is updated. Since hierarchical ownership is inadequate for shared objects like locks, VCC provides another way to make manifest that o_1 depends on the state of o_2: it allows that o_1 maintains a *claim* on o_2—a ghost object with no concrete state but an invariant that depends on o_2. Declaring a type to be claimable introduces implicit ghost state used by the VCG to track outstanding claims. The ghost code to create a claim or store it in a field is part of the annotation provided by the programmer.

The term *invariant* encompasses *two-state* predicates for the before and after states of a state transition. In this way, invariants serve as the rely conditions in a form of rely-guarantee reasoning. Usually two-state invariants are written as ordinary formulas, using the keyword **old** to designate expressions interpreted in the before state. We say an invariant is *one-state* to mean that it does not depend on the before state.

A thread can update an object that it owns, using unwrap/wrap. However, in many cases such as locks, having a single owner is too restrictive, and another mechanism is needed to allow multiple threads to update the object concurrently. VCC interprets fields marked **volatile** as being susceptible to update by other threads, in accord with the interpretation of the **volatile** keyword by C compilers. An atomic step is allowed to update a volatile field without opening it, provided that the object is proved closed and the update maintains the object's two-state invariant (that being the interference condition on which interleaved threads rely). The standard idiom for locks is that several threads each maintain a claim that the lock is closed, so they may rely on its invariant; outstanding claims prevent even the owner from unwrapping the object. Atomic blocks are explicitly marked as such. An atomic block may make at most one concrete update, to be sound for C semantics, but may update multiple ghost fields. We do not use **assume** statements in atomic blocks.

III. Symbolic Cryptography in VCC

In this section, we introduce the inductive model of symbolic cryptography and show how it is approximated in VCC by first-order logic axioms over uninterpreted function symbols and code manipulating ghost state.

A. Term Algebra

We use a standard symbolic model of cryptography, where cryptographic primitives (and further operations such as pairing) are modelled as constructors of an inductive datatype. Some details are protocol-specific, so for clarity we focus on a simple model adequate for our running example, the RPC protocol. We use ordinary mathematical notation for the following definitions, which are formalized in our Coq development.

An algebra of cryptographic terms

$t_i, k, m, p, a, b ::=$	term
Literal bs	with bs a byte array
Pair t_1 t_2	
Hmac k m	

Automated verifiers like VCC support specifications in first-order logic without inductive definitions. So we use an over-approximation of the term algebra given by uninterpreted function symbols and first-order axioms. The reasoning performed by VCC thus holds for all models of the axioms, in particular for the intended inductive model. The following shows the first-order axioms corresponding to the algebra above.

The VCC **spec**() syntax indicates that all symbols declared within are ghost objects. In particular, it allows the use of the mathint type of mathematical integers (in $\mathbb{Z}$).

A first-order model of cryptographic terms

```
spec(
  typedef mathint bytes;
  typedef mathint term;

  ispure mathint tag_term(term);

  // Constructors (declared as uninterpreted functions)
  ispure term Literal(bytes bs);
  ispure term Pair(term t1, term t2);
  ispure term Hmac(term k, term m);

  // Theorems
  theorem(Literal_Injective,
    forall(bytes bs1,bs2; Literal(bs1) == Literal(bs2)
          ==> bs1 == bs2));
  theorem(Pair_Injective,
    forall(term a1,a2,b1,b2; Pair(a1,b1) == Pair(a2,b2)
          ==> a1 == a2 && b1 == b2));
  theorem(Hmac_Injective,
    forall(term k1,m1,k2,m2; Hmac(k1,m1) == Hmac(k2,m2)
          ==> k1 == k2 && m1 == m2));

  theorem(Literal_Disjoint,
    forall(bytes bs; tag_term(Literal(bs)) == 0));
  theorem(Pair_Disjoint,
    forall(term t1,t2; tag_term(Pair(t1,t2)) == 1));
  theorem(Hmac_Disjoint,
    forall(term k,m; tag_term(Hmac(k,m)) == 2));
)
```

We use the bytes type to manipulate whole byte arrays as values, and assume a bijection between finite byte arrays and mathematical integers. (One such bijection interprets a byte array as an integer, pairs that with its length to account for leading zeroes, and injects the pair into $\mathbb{Z}$.) We will designate the injection from byte arrays to type bytes as: Encode(**unsigned char**∗, **unsigned long**).

The **ispure** keyword is used to specify that a given function is to be interpreted as a total function whose return value depends only on the value of its arguments and memory locations listed in its **reads**() clauses. Only pure functions can be used in function contracts and assertions, and purity needs to be explicitly specified even for **spec** functions, as they may update ghost state.

We use axioms to state properties of the declared function symbols that cannot easily be expressed using pre and postconditions (injectivity and disjointness, in this case).[1] Those axioms are separately proved in Coq to hold about the intended, inductive model of cryptography. The **theorem** notation is a simple macro that generates a VCC **axiom**; its first argument, ignored by VCC, is the name of the corresponding Coq theorem.

B. Events and Log

Much as in prior work [12], we use a global log of events to express the wanted correspondence properties. Events themselves are defined using a protocol-specific algebra (see below).

The hashkey usage and some top-level events are specific to the RPC protocol, but some constructors are of general use and are needed for most protocols. The top-level of events always includes a **New** event, logging the intended usage of freshly generated bytestrings. In particular, the **AttackerGuess** models that the corresponding term represents a bytestring known to the attacker, because it was provided as a starting parameter, or because it represents a fixed bytestring appearing in the protocol specification (e.g. a tag, or a fixed format string).

Algebra of events for RPC

hu ::=	hashkey usage
KeyAB a b	
us ::=	usage
AttackerGuess	
HashKey hu	
ev ::=	event
New t us | **Bad** a | **Request** a b t | **Response** a b t_1 t_2	

As with the cryptographic terms, a first-order approximation of this algebra is given to VCC using uninterpreted functions and axioms. The log itself, intended as the set of events that have occurred so far, is defined next as a structure containing one set for each kind of event. We use boolean maps to model sets. The VCC notation is similar to that of arrays, e.g., **bool** B[mathint] declares B to be a boolean-valued total function on the integers.

Encoding of the log in VCC (RPCdefs.h)

```
#define stable(_F)      old(_F) ==> _F

spec(
  typedef struct log_s {
    volatile bool New[term][usage];
    volatile bool Bad[term];
    volatile bool Request[term][term][term];
    volatile bool Response[term][term][term][term];

    // Misc. conditions
    invariant(forall(term t; usage u1,u2;
      New[t][u1] && New[t][u2] ==> u1 == u2))
    invariant(forall(term t; usage u;
      New[t][u] ==> exists(bytes b; t == Literal(b))))

    // Stability
    invariant(forall(term t; usage u; stable(New[t][u])))
    invariant(forall(term t; stable(Bad[t])))
    invariant(forall(term a,b,s;
      stable(Request[a][b][s])))
    invariant(forall(term a,b,s,t;
      stable(Response[a][b][s][t])))
  } Log;)

spec(Log log;)

#define valid_log\
  forall(term T; usage U1,U2;\
    log.New[T][U1] && log.New[T][U2] ==> U1 == U2) &&\
  forall(term T; usage U;\
    log.New[T][U] ==> exists(bytes B; T == Literal(B)))

#define stable_log\
  forall(term T; usage U; stable(log.New[T][U])) &&\
  forall(term T; stable(log.Bad[T])) &&\
  forall(term A,B,S; stable(log.Request[A][B][S])) &&\
  forall(term A,B,S,T; stable(log.Response[A][B][S][T]))
```

Logical formulas have type **bool**, e.g., the expression Request[a][b][t] can be used as an assertion saying that this event has occurred. We use two-state invariants to express

[1]Injectivity could, in general, be expressed as a postcondition, but VCC imposes syntactic restrictions on the postconditions of pure functions to ensure that they are total and computable.

that the log can only grow (see the “Stability” group of invariants in the code displayed above). We use a one-state invariant to express that bytestrings, and in particular keys, should only be given one usage. Other protocol-specific one-state invariants could be added. We call them “Miscellaneous conditions”, and will say that a log is *good*, or *valid*, when its miscellaneous conditions hold. We also introduce macros valid_log and stable_log, which expand to both invariant blocks, to simplify the expression of certain properties; valid_log is used in the inversion principle for the *Pub()* predicate.

C. Inductive Predicates for Cryptography

We use inductive predicates to express the correct usage of cryptographic primitives, as specified by a given protocol. In particular, we define a predicate *Pub()* that holds on all terms that can be published without compromising the protocol's goals. We also define a *Bytes()* predicate that holds on byte arrays an honest protocol participant is allowed to build. We ensure by definition that *Bytes()* holds for all terms on which *Pub()* holds. Both of these predicates, and all intermediate predicates used in their definition, are actually functions of the log. We write $\mathcal{L} \vdash P$ to say P holds in log $\mathcal{L}$. The following shows an excerpt of the inductive rules defining the *Pub()* predicate.

Some inductive predicates for cryptography

(MACSays KeyAB Request)

$$\frac{\begin{array}{c}\textbf{New}\ k\ (\textbf{HashKey}\ (\textbf{KeyAB}\ a\ b))\ \in\ \mathcal{L}\\ \textbf{Request}\ req\ a\ b\ \in\ \mathcal{L}\\ m\ =\ \textbf{Pair}\ (\textbf{Literal}\ \mathit{TagRequest})\ req\end{array}}{\mathcal{L}\ \vdash\ \mathit{MACSays}\ k\ m}$$

(MACSays KeyAB Response)

$$\frac{\begin{array}{c}\textbf{New}\ k\ (\textbf{HashKey}\ (\textbf{KeyAB}\ a\ b))\ \in\ \mathcal{L}\\ \textbf{Response}\ req\ resp\ a\ b\ \in\ \mathcal{L}\\ m\ =\ \textbf{Pair}\ (\textbf{Literal}\ \mathit{TagResponse})\ (\textbf{Pair}\ req\ resp)\end{array}}{\mathcal{L}\ \vdash\ \mathit{MACSays}\ k\ m}$$

(Pub AttackerGuess)

$$\frac{\textbf{New}\ (\textbf{Literal}\ b)\ \textbf{AttackerGuess}\ \in\ \mathcal{L}}{\mathcal{L}\ \vdash\ \mathit{Pub}\ (\textbf{Literal}\ b)}$$

(Pub Hmac)

$$\frac{\begin{array}{c}\mathcal{L}\ \vdash\ \mathit{MACSays}\ k\ m\\ \mathcal{L}\ \vdash\ \mathit{Bytes}\ k\\ \mathcal{L}\ \vdash\ \mathit{Pub}\ m\end{array}}{\mathcal{L}\ \vdash\ \mathit{Pub}\ (\textbf{Hmac}\ k\ m)}$$

(Pub Hmac Pub)

$$\frac{\mathcal{L}\ \vdash\ \mathit{Pub}\ k\quad \mathcal{L}\ \vdash\ \mathit{Pub}\ m}{\mathcal{L}\ \vdash\ \mathit{Pub}\ (\textbf{Hmac}\ k\ m)}$$

In order to simplify the notations in VCC, we do not make the log an explicit argument in the predicates' VCC declaration. Instead, we express that the functions intended to model the predicates depend on the state, by marking them with the **reads**(set_universe()) contract, stating that the value returned by the function may depend on the set of pointers that contains all addresses in the state. This makes the program state an implicit parameter to the function; later we use axioms to frame the function's dependency on the state more precisely. We can then use axioms to give the intended meaning of these symbols. In particular, we prove in Coq that our predicate symbols are monotonic functions of the log and import this fact as an axiom in VCC (see the theorem Pub Monotonic below). To that end we give the following definitions. The notation in_state(S, e) denotes the value of expression e in the state S.

Log stability between explicit states

```
#define s_stable(S1,S2,_F)\
  in_state(S1,_F) ==> in_state(S2,_F)

#define s_stable_log(S1,S2)\
  forall(term T; usage U;\
    s_stable(S1,S2,log.New[T][U])) &&\
  forall(term T;\
    s_stable(S1,S2,log.Bad[T])) &&\
  forall(term A,B,S;\
    s_stable(S1,S2,log.Request[A][B][S])) &&\
  forall(term A,B,S,T;\
    s_stable(S1,S2,log.Response[A][B][S][T]))
```

Here are the VCC axioms for the Pub() predicate and some example theorems.

VCC axiomatic definition for Pub() (excerpt)

```
spec(ispure bool Pub(term t)
     reads(set_universe());)

theorem(Pub_Monotonic,
  forall(state_t s1,s2; term x;
    s_stable_log(s1,s2) ==> s_stable(s1,s2,Pub(x))));

rule(Pub_AttackerGuess,
  forall(bytes b;
    log.New[Literal(b)][AttackerGuess()] ==>
    Pub(Literal(b))));
```

Inversion theorems (excerpt)

```
theorem(KeyAB_WeakSecrecy,
  forall(term k,a,b;
    valid_log &&
    log.New[k][HmacKey(KeyAB(a,b))] &&
    Pub(k) ==>
      log.Bad[a] || log.Bad[b]));

theorem(MACSays_RPC,
  forall(term a,b,m,k;
    valid_log && log.New[k][HmacKey(KeyAB(a,b))] &&
    MACSays(k,m) ==>
      (exists(term s; Requested(m,s) &&
                      log.Request[a][b][s]) ||
       exists(term s,t; Responded(m,s,t) &&
                        log.Response[a][b][s][t]))));

theorem(Pub_MACSays_Pub,
  forall(term k,m; Pub(Hmac(k,m)) ==>
    MACSays(k,m) || Pub(k)));
```

The axioms introduced using **rule** correspond to the inductive definition rules shown earlier. Again, the axioms introduced using **theorem** correspond to results that are proved to hold in the model inductively defined in Coq. Similar axioms are used to define Bytes() and MACSays(), as well as theorems similar to those used in F7 [12].

IV. Representation Table and Hybrid Wrappers

A. The Representation Table

Symbolic models of cryptography generally assume that two distinct symbolic terms yield two distinct byte strings, and that fresh literals cannot be guessed by an attacker. The intent is to use such a model with cryptographic operations that, in the computational model, have a negligible probability of collision. Verification in the symbolic model is a way of ruling out a well-defined class of attacks, which in practice do not depend on collisions or lucky guesses.

Prior work on cryptographic software in F#, for example [10], [9], relies on type abstraction to verify protocol code when running with purely symbolic libraries, which satisfy these assumptions, instead of concrete libraries, which do not. In the absence of type abstraction in C, we must verify protocol code when running with concrete cryptographic algorithms on byte strings. Our aim remains to verify against an attacker in the symbolic model. To do so, we instrument the program with specification code that maintains a *representation table*, which tracks the correspondence between concrete byte strings and symbolic terms. We intercept all calls to cryptographic functions with ghost code to update the representation table. We say a *collision* occurs when the table associates a single byte string with two distinct symbolic terms.

For example, suppose x and y are two distinct bytestrings that have the same HMAC, h, under a key k. After the first call to hmac() the table looks like this:

Bytestring	Term
k	**Literal** k
x	**Literal** x
y	**Literal** y
h	**Hmac** k x

When computing the second HMAC, our instrumented hmac() function tries to insert the freshly computed h and the corresponding term **Hmac** k y in the table, but detects that h is already associated with a distinct term **Hmac** k x.

We make the absence of such collisions an explicit hypothesis in our specification by assuming, via an **assume** statement in the ghost code updating the table, that a collision has not occurred. This removes from consideration any computation following a collision, as is made precise in Section V. We treat the event of the attacker guessing a non-public value in a similar way; we assume it does not happen, using an **assume** statement. In this way we prove symbolic security properties of the C code. A separate argument may be made that such collisions only happen with low probability.

Like the log, the representation table, given next, is a structure containing maps.

The representation table in VCC

```
spec(
  typedef vcc(claimable) struct table_s {
    volatile bool DefinedB[bytes];
    volatile term B2T[bytes];

    volatile bool DefinedT[term];
    volatile bytes T2B[term];

    // Bijectivity
    invariant(forall(bytes b;
      DefinedB[b] ==> T2B[B2T[b]] == b))
    invariant(forall(term t;
      DefinedT[t] ==> B2T[T2B[t]] == t))
    invariant(forall(bytes b;
      DefinedB[b] ==> DefinedT[B2T[b]]))
    invariant(forall(term t;
      DefinedT[t] ==> DefinedB[T2B[t]]))

    // Stability
    invariant(forall(bytes b;
      stable(DefinedB[b])))
    invariant(forall(bytes b;
      old(DefinedB[b]) ==> unchanged(B2T[b])))
    invariant(forall(term t;
      stable(DefinedT[t])))
    invariant(forall(term t;
      old(DefinedT[t]) ==> unchanged(T2B[t])))

    // Validity
    invariant(forall(bytes b;
      DefinedT[Literal(b)] ==> T2B[Literal(b)] == b))
    invariant(forall(term t;
      DefinedT[t] ==> Bytes(t)))

    // Ownership
    invariant(keeps(&log))
  } Rep;)

spec(Rep table;)
```

We use two maps to store the bijection between bytes, which are byte string values (not pointers), and terms (e.g., the byte string b corresponds to the algebraic term B2T[b]). Two-state invariants express that the table can only grow. There is an ownership invariant: the representation table always owns the log. This means that whenever &rep is closed, &log has to be closed itself, so the invariants for the representation table can depend on values in the log in accord with the VCC ownership methodology.

B. The Hybrid Wrappers

We want to ensure that all cryptographic operations are used in ways that preserve the representation table invariants. We provide hybrid wrappers around the concrete library functions; wrappers are not only verified to maintain the table's invariants but also serve to give symbolic contracts to a cryptographic interface working with concrete bytes.

For simplicity in this paper, the hybrid wrappers manipulate a structure type bytes_c containing all information pertaining to a byte array.

A type for byte strings

```
typedef struct {
        unsigned char *ptr;
        unsigned long len;

        spec(bytes encoding;)
        invariant(keeps(as_array(ptr,len)))
        invariant(encoding == Encode(ptr,len))
} bytes_c;
```

In particular, we keep not only a pointer to the concrete byte array considered and its length, but we also add a ghost field of type bytes, representing—as a mathematical integer—the byte string value contained by the len bytes at memory location ptr. For the invariants of bytes_c to be admissible, we also make sure, using the keeps keyword, that the heap-allocated byte array of length len pointed to by ptr is always owned by the structure, ensuring in particular that it is never modified while the structure is kept closed.

As an example, here is the contract of our hybrid wrapper for the hmac_sha1() cryptographic function.

Hybrid interface for hmacsha1()

```
int hmacsha1(bytes_c *k, bytes_c *b, bytes_c *res
           claimp(c))
  // Stability of log and table
always(c, closed(&table) &&
         c_stable_log && c_stable_table)
ensures(stable_log && stable_table)
  // Properties of input byte strings
maintains(wrapped(k))
maintains(wrapped(b))
  // Properties of out parameter
writes(span(res))
ensures(unchanged(emb(res)))
ensures(!result ==> wrapped_dom(res))
  // Cryptographic contract
requires(table.DefinedB[k->encoding])
requires(table.DefinedB[b->encoding])
ensures(!result ==> table.DefinedB[res->encoding])
    // Cryptographic properties on input terms
requires(
  MACSays(table.B2T[k->encoding],table.B2T[b->encoding])
  || (Pub(table.B2T[k->encoding]) &&
      Pub(table.B2T[b->encoding])))
    // Cryptographic properties on output term
ensures(!result ==>
  table.B2T[res->encoding] ==
    Hmac(table.B2T[k->encoding],table.B2T[b->encoding]));
```

For log and table stability, as well as concurrent access, there is a ghost parameter containing a claim c (represented by the claimp macro). The function's contract says, using the **always**(c,...) construction, that the claim parameter should ensure that the table object is closed, and that both table and log have only grown since the claim was created (which is expressed using the c_stable variant of the s_stable macro). Additionally, the function should ensure that the log and table only grow during its execution. The next lines of the contract concern memory-safety, e.g., the arguments used to pass values in are **wrapped** both at call-site and at return-site (**maintains**(F) expands to **requires**(F) **ensures**(F)), and the third argument is used as an output parameter, to return the function's result. The latter is specified by stating that the function is allowed to write the set of memory locations contained within the **span** of the out-parameter. Additionally, we also specify that the embedding (**emb**) of the out-parameter is left **unchanged** by a call to the function, meaning that the pointer refers to the same typed memory location. It is then specified that a call to this function may write all fields of the structure passed in the res argument, and that this memory update has to **wrap** the structure when the function call is successful.

The first three lines under "Cryptographic contract" deal only with the table, stating that the input byte strings should appear in the table, and that, upon successful return from the function, the output byte string appears in the table. Then comes an important cryptographic precondition: that either *MACSays()* holds on the terms associated with the input byte strings (modelling an honest participant's calling conditions), or they are both in *Pub()* (modelling a call by the attacker or a compromised principal). The postcondition states that, upon successful return, the output byte string is associated with the term obtained by applying the **Hmac** constructor to the terms associated with the input byte strings.

An honest client, when calling a function with this contract, needs to establish the first disjunct of the precondition: that *MACSays()* holds on the terms associated with the input byte arrays. For this, to hold, the term associated with the key k must be given, in the log, a certain usage **HashKey** hu for some hashkey usage hu, and the term associated with the message to authenticate b must be formatted correctly, as specified by the corresponding inference rule (here (MACSays KeyAB Request) or (MACSays KeyAB Response)).

A typical hybrid wrapper implementation first performs the concrete operation on byte strings (e.g., by calling a cryptographic library) before performing updates on the ghost state to ensure the cryptographic postconditions, whilst maintaining the log and table invariants. To do so, it first computes the expected cryptographic term by looking up, in the table, the terms associated with the input byte strings and applying the suitable constructor. Once both the concrete byte string and the corresponding terms are computed, the implementation can check for collisions, and in case there are none, update the table (and the log) as expected. In case a collision happens, an **assume** statement expresses that our symbolic cryptography assumptions have been violated.

A hybrid wrapper for hmacsha1()

```
int hmacsha1(bytes_c *k, bytes_c *b, bytes_c *res
           claimp(c))
{ spec(term tb,tk,th;)
  spec(bool collision = false;)

  res->len = 20;
  res->ptr = malloc(res->len);
  if (res->ptr == NULL)
    return 1;
  sha1_hmac(k->ptr, unchecked((int) k->len), b->ptr,
       unchecked((int) b->len), res->ptr);

  spec(
    res->encoding = Encode(res->ptr,res->len);
    wrap(as_array(res->ptr,res->len));
    wrap(res);)
  spec(
    atomic(c, &table)
    {
      tb = table.B2T[b->encoding];
      tk = table.B2T[k->encoding];
      th = Hmac(tk,tb);  // Compute the symbolic term

      if ((table.DefinedB[res->encoding] &&
           table.B2T[res->encoding] != th) ||
          (table.DefinedT[th] &&
           table.T2B[th] != res->encoding))
        collision = true;
      else
      {
        table.DefinedT[th] = true;
        table.T2B[th] = res->encoding;
        table.DefinedB[res->encoding] = true;
        table.B2T[res->encoding] = th;
      }
    })
  assume(!collision);  // Our symbolic crypto assumption
  return 0; }
```

Our implementation of an HMAC SHA1 wrapper, shown above, uses the PolarSSL project's sha1_hmac() function ([39]).

The **unchecked** keyword is used to let VCC ignore the potential arithmetic overflow due to the type casts.

Since the table is shared and its fields marked volatile, all reads and writes from and to it need to occur in an atomic block guarded by a claim c ensuring, among other things, that the global table object is closed.

We also provide a function toString converting an ordinary string pointer to a bytes_c, the input type for functions like hmacsha1. It logs a New event with usage AttackerGuess and assumes the guessed literal does not collide with any other term already in the table. We also provide a function bytescmp, that compares two bytes_c objects.

That completes the groundwork needed to specify and verify the RPC protocol code (RPCprot.c). The following shows a slightly simplified version of the annotated code for the client role, where the **Request** event is logged by the atomic assignment and the final correspondence is asserted as a disjunction of events taking into account the potential compromise of one of the principals involved. Each of the function calls is verified to happen in a state where the function's precondition holds. In particular, the call to the channel_write() function yields a proof obligation that *Pub()* holds on the term corresponding to the second argument. The **return** statements are for various kinds of failure.

Annotated RPC client code

```
void client(bytes_c *alice, bytes_c *bob, bytes_c *kab,
      bytes_c *req, channel* chan claimp(c))
maintains(wrapped(alice) && wrapped(bob) &&
          wrapped(kab) && wrapped(req))
always(c,closed(&table) && c_stable_log && c_stable_table)
writes(c)
requires(table.DefinedB[alice->encoding] &&
         table.DefinedB[bob->encoding] &&
         table.DefinedB[kab->encoding] &&
         table.DefinedB[req->encoding])
requires(Pub(table.B2T[alice->encoding]) &&
         Pub(table.B2T[bob->encoding]) &&
         Pub(table.B2T[req->encoding]) &&
         Bytes(table.B2T[kab->encoding]))
requires(table.B2T[kab->encoding] == Literal(kab->encoding
      ))
requires(
  log.New[table.B2T[kab->encoding]]
         [HmacKey(KeyAB(table.B2T[alice->encoding],
                        table.B2T[bob->encoding]))]);
{
  spec(claim_t tmp;)
  bytes_c *toMAC1, *mac1, *msg1;
  bytes_c *msg2, *resp, *toMAC2, *mac2;
  // Event
  spec(atomic(c,&table,&log) {
         log.Request[table.B2T[a->encoding]]
                    [table.B2T[b->encoding]]
                    [table.B2T[req->encoding]] = true;})
  // Build and send request message
  if ((toMAC1 = malloc(sizeof(*toMAC1))) == NULL) return;
  if (request(req, toMAC1 spec(c))) return;

  if ((mac1 = malloc(sizeof(*mac1))) == NULL) return;
  if (hmacsha1(kab, toMAC1, mac1 spec(c))) return;

  if ((msg1 = malloc(sizeof(*msg1))) == NULL) return;
  if (pair(req, mac1, msg1 spec(c))) return;
  if (channel_write(chan, msg1 spec(c))) return;

  // Receive and check response message
  if ((msg2 = malloc(sizeof(*msg2))) == NULL) return;
  if (channel_read(chan, msg2 spec(c))) return;

  if ((resp = malloc(sizeof(*resp))) == NULL) return;
  if ((mac2 = malloc(sizeof(*mac2))) == NULL) return;
  if (destruct(msg2, resp, mac2 spec(c))) return;

  if ((toMAC2 = malloc(sizeof(*toMAC2))) == NULL) return;
  if (response(req, resp, toMAC2 spec(c))) return;

  if (!hmacsha1Verify(kab, toMAC2, mac2 spec(c))) return;

  // Correspondence assertion
  assert(log.Response[table.B2T[alice->encoding]]
                     [table.B2T[bob->encoding]]
                     [table.B2T[req->encoding]]
                     [table.B2T[resp->encoding]]
      || log.Bad[table.B2T[a->encoding]]
      || log.Bad[table.B2T[b->encoding]]);
}
```

To prove that the correspondence assertion holds, VCC will use the postconditions of hmacsha1Verify() stating that a zero return value implies that the third argument is indeed a valid hmac, the fact that the byte array toMAC2 is known to have a correct response format as it is the result of a succesful call to the response() function, and the fact that *Pub()* holds on the response message, as it was read from the network. Using these facts, VCC can use the inversion theorems shown in Section III-C and prove the correspondence assertion.

V. Assumptions Concerning the C Verifier

Several research papers [18], [19] document the VCC system but there is no formal model of its semantics of programs and specifications aside from the VCG itself. To be able to formulate a precise specification of the program properties (in particular security properties) verified by VCC, we sketch a conventional operational semantics, in terms of which we specify what we assume about the verifier. The model sketched here has been formalized as part of our Coq development. The model idealizes from low level features of C, using instead a simple Java-like heap model (following [19]), but please keep in mind that VCC reasons soundly about the gory details of low level C code.

An *execution environment* consists of a self-contained collection of type and function declarations. For a given execution environment, a runtime *configuration* takes the form (h, ts, qs) where h is the heap, ts is the thread pool, and qs is a map from channel names to message queues. A *thread state* consists of a command (its current continuation) and a local *store* (i.e., a mapping of locals and parameters to their current values); a *thread pool* is a finite list of thread states. Thus threads share the heap and the message queues (which hold messages sent but not yet received). A *run* is a series of configurations that are successors in the transition relation. The transition relation allows nondeterministic selection of any thread that is not blocked waiting to receive on an empty channel. A single step (transition) may be an assignment, the test of a branch condition, creation of a new thread, etc. Nondeterministic scheduling models all interleavings including ones that may be preferred by an attacker.

A *state predicate* is a predicate on a heap together with a store. The store is used for function parameters and results, which are thread local. The precondition of a function contract is a state predicate; its postcondition is a two-state predicate that refers to the initial and final state of the function's invocation. An invariant is a predicate on a pointer together with a pair of heaps, as described earlier.

The only unusual feature of the semantics is our treatment of assumptions, which are usually only given an axiomatic semantics. If there is any thread poised to execute the command **assume** p, and the condition p does not hold in the current configuration, then there is no transition—we say there is an *assumption failure*. If all current assumptions hold, then some thread takes a step. Thus some runs end with a "stuck" configuration from which there are no successors. The only other stuck configurations are those where every thread is blocked waiting on an empty channel. Execution of **assume** p takes a single step with no effect on state. Execution of **assert** p also has no effect on the state—nor does it have an enabling condition. An assertion is effectively a labelled skip, in terms of which we formulate correctness.

Definition 1 (safe command): An *assertion failure* is a run in which there is a configuration where some thread's active command is **assert** p for some p that does not hold in that configuration, or some object's invariant fails to hold, and there is no assumption failure at that point. A configuration is *safe* if none of its runs are assertion failures. A command c is *safe* under precondition p if for states satisfying p, the configuration with that initial state and the single thread c is a safe configuration.

Given our treatment of assumptions, safety means that there is no assertion failure unless and until there is an assumption failure.

VCC works in a procedure-modular way: it verifies that each function implementation satisfies its contract, under the assumption of specified contracts for all functions directly called in the body.

Definition 2 (verifiable): We write $api.h \vdash p.c \rightsquigarrow q.h$ to mean there exists $p'.c$ that instruments $p.c$ with additional ghost code (but no assumptions, and no other changes), and $q'.h$ that may extend $q.h$ with contracts for additional functions (but not alter those in $q.h$) and type invariants, such that VCC successfully verifies the implementation of each function f in $p'.c$ against the contract for f in $q'.h$, under hypotheses $api.h$ and $q'.h$; moreover admissibility holds for all the type invariants.

We use names ending in .c or .h for code or interface texts, as mnemonic for usual file names, but these may be catenations of multiple files.

An immediate consequence of Definition 2 is the following, where the $+$ operator stands for catenation.

Lemma 1 (VCC Modularity): If $p.h \vdash q.c \rightsquigarrow q.h$ and $p.h + q.h \vdash r.c \rightsquigarrow r.h$ then $p.h \vdash q.c + r.c \rightsquigarrow q.h + r.h$.

The VCC methodology supports verification conditions for sound modular reasoning, but it is not easy to give a VCG-independent semantics for the verifiability judgement $p.h \vdash q.c \rightsquigarrow q.h$. Fortunately, for our purposes it is enough to consider soundness for complete programs. A complete program is verified as $\emptyset \vdash m.c \rightsquigarrow$ main.h.

main.h
```
void main()
requires(program_entry_point())
writes(set_universe());
```

The program_entry_point() precondition means that all global objects exist and are owned by the current thread at the beginning of this function, as it is the first function that is called when the process is started.

Assumption 1 (VCC Soundness): If $\emptyset \vdash m.c \rightsquigarrow$ main.h then the body of function main in $m.c$ is safe for the precondition in main.h.

VCC checks that ghost state is used in ways that are sound for reasoning about actual observations; i.e., it has no influence on non-ghost state except for introducing additional steps that do not change non-ghost state. We formalize this in the long version of the paper.

VI. Attack Programs

An attacker in the symbolic model can intercept messages on unprotected communication links (such as the Internet) and send messages constructed from parts of intercepted messages, as specified by a term algebra. We model the set of all possible attacks, each attack being represented by an *attack program* (or just "attack"). In this section we sketch the formal definition of attack program, relative to a suitable interface, and give an example. Attack programs are what enable us, in Section VII, to use an ordinary program verifier to reason about active attackers.

An attack program is a straight-line C program that compiles against an *attacker interface*. Such an interface provides some "opaque" type declarations together with some function signatures; these include message send/receive, standard cryptographic operations, and protocol-specific actions like creating sessions and initiating roles.

Attacker interfaces

T ::=	type
bool | **unsigned char**$*$ | $X*$	
μ ::=	entry in an interface
typedef X;	type declaration
$T\ f(T_1\ x_1, \ldots, T_n\ x_n)$	function prototype ($n \geq 0$)
void $f(T_1\ x_1, \ldots, T_n\ x_n)$	procedure prototype ($n \geq 0$)
$\mathcal{I}$ ::= $\mu_1 \ldots \mu_n$	interface ($n \geq 0$)

For an annotated interface $p.h$, we let $erase(p.h)$ be the attacker interface obtained by deleting annotations and the bodies of type declarations. Recall the software stack shown in Section I-C; the file RPCshim.h provides a network attacker interface including generic cryptography and network operations as well as protocol specific functions.

An attacker interface: *erase*(RPCshim.h)

```
typedef bytespub;

bytespub* att_toBytespub(unsigned char* ptr,
                         unsigned long len);
bytespub* att_pair(bytespub* b1, bytespub* b2);
bytespub* att_fst(bytespub* b);
bytespub* att_snd(bytespub* b);

bytespub* att_hmacsha1(bytespub* k, bytespub* b);
bool att_hmacsha1Verify(bytespub* k,
                        bytespub* b,
                        bytespub* m);

void att_channel_write(channel* chan, bytespub* b);
bytespub* att_channel_read(channel* chan);

typedef session;

session* att_setup(bytespub* cl, bytespub* se);

void att_run_client(session* s, bytespub* request);
void att_run_server(session* s);

bytespub* att_compromise_client(session*s);
bytespub* att_compromise_server(session*s);

channel* att_getChannel_client(session* s);
channel* att_getChannel_server(session* s);
```

Type bytespub is critical: its invariant constrains its values to be concrete byte arrays that correspond to terms that satisfy the *Pub()* predicate. Verifying the implementation of this attacker interface therefore provides a proof that *Pub()* is closed under attacker actions. The function contracts in RPCshim.h and code in RPCshim.c are similar to the hybrid wrappers in Section IV-B but oriented to Pub data. They are more complicated, due to memory safety annotations dealing with thread fork and messaging, though that is mostly protocol-independent. An example contract appears in Section VII.

Attack program for given interface $\mathcal{I}$

An *attack program* for a given interface $\mathcal{I}$ has the form:
`void main() { D C }`
where D is a sequence of local variable declarations
and C a sequence of commands, such that:
1) Each of the declarations in D has the form T x;, where T is either **bool**, **unsigned char***, or T* where T is declared in $\mathcal{I}$.
2) Each command in the sequence C is either (a) a function call assignment with variables as arguments, x = f(y ...) ;
(b) a procedure call with variables as arguments f(y ...) ;
or (c) an assignment x = s; where s is a string literal.
3) A variable is assigned at most once and every variable mentioned is declared in D.
4) For each function or procedure call, each argument variable is assigned earlier in the sequence of commands.
5) In each call to a function or procedure f, there is a declaration of f in $\mathcal{I}$ and each argument variable in the call has declared type identical to that of the corresponding parameter of f.
6) In a function call assignment x = f(y ...) ;, the declared type of x is the result type of f. In a string assignment x = s; the declared type of x is **char***.

Owing to item 2, an attack program does not directly assign any object field, nor any global variable. Nor does it directly invoke any operations except functions and procedures in $\mathcal{I}$ (item 5).

An attack program for RPCshim.h (from RPCattack_0.c)

```
void main()
{ unsigned char *a,*b,*r;
  bytespub *alice,*bob,*arg,*req,*resp;
  channel *clientC,*serverC;
  session *s;

  a = "Alice"; alice = att_toBytespub(a,5);
  b = "Bob"; bob = att_toBytespub(b,3);
  r = "Request"; arg = att_toBytespub(r,7);
  s = att_setup(alice,bob);
  clientC = att_getChannel_client(s);
  serverC = att_getChannel_server(s);
  att_run_server(s);
  att_run_client(s,arg);
  req = att_channel_read(clientC);
  att_channel_write(serverC,req);
  resp = att_channel_read(serverC);
  att_channel_write(clientC,resp);}
```

VII. An Example Security Theorem

An attack program for the RPC protocol is a program that relies only on RPCshim.h. To form an executable, it needs to be combined with System which we define to be the catenation crypto.c + RPChybrids.c + RPCprot.c. Here crypto.c is the library of cryptographic algorithms (and we let it subsume OS libraries, e.g., for memory allocation and sockets), which is used in RPChybrids.c and RPCprot.c.

Before providing the formal results, we informally describe a key property on which soundness of our approach rests. Consider any attack program M.c and any run of the program System + RPCshim.c + M.c. It is an invariant that at every step of the run, the representation table holds every term that has arisen by cryptographic computation or by invocation of the toBytesPub function which an attack must use to convert guessed bytestrings to type bytespub as needed to invoke the other functions of RPCshim. This is not an invariant that we state in the program annotations; its only role is to justify our use of assumptions. The only assumptions used are in RPCshim.c and RPChybrids.c where collisions are detected. In light of the key invariant, this means that in any run that reaches an assumption failure, the sequence of terms computed includes a hash collision or an attacker guess of a term that is not public according to the symbolic model of cryptography.

The contracts in RPCshim.h all follow a similar pattern; we give one for reference in the following proof.

Example contract from RPCshim.h

```
bytespub* att_hmacsha1(bytespub* k, bytespub* b claimp(c))
maintains(wrapped(k))
maintains(wrapped(b))
writes(k,b)
always(c, closed(&table)&&c_stable_log&&c_stable_table)
writes(c)
ensures(wrapped(result));
```

Attack programs were defined in order to show that their behaviours are among those of interfering threads encompassed by the verification conditions VCC imposes on protocol code. This is formalized by way of the following.

Lemma 2: If $M.c$ is an attack program for $erase($RPCshim.h$)$, then RPCshim.h $\vdash M.c \rightsquigarrow$ main.h.

Proof: (Sketch) According to Definition 2 we have to show admissibility of the type invariants in RPCshim.h, which we have checked using VCC. It remains to prove verifiability of the attack program against main.h. A general argument is needed since there are infinitely many attacks. (Here we idealize: VCC's resources can be taxed in many ways, e.g., $M.c$ could declare a vast number of variables.)

Because the contract in main.h does not impose a postcondition and its write specification is vacuous, we just need to show that invariants are established and maintained. Let $M.c$ be **void** main(){D C}. In accord with Definition 2 we will show verifiability of code C′ which augments the statements of C with two sorts of instrumentation. The first is simply to prefix C with ghost code that initializes the representation table and log. This code is defined as macro init (), shown in the code sample below, where maps are defined using VCC's **lambda** notation, and the constants tagRequest and tagResponse are separately defined to be the integer encoding of the 1-byte-long strings "1" and "2", respectively.

The init () macro

```
#define init(...)\
  spec(\
    /* Log */\
    log.New = lambda(term t;
                lambda(usage u; false));\
    log.Request = lambda(term a;
                    lambda(term b;
                      lambda(term s; false)));\
    log.Response = lambda(term a;
                     lambda(term b;
                       lambda(term s;
                         lambda(term t; false))));\
    wrap(&log);\
    /* Representation table. It initially contains
         tagRequest and tagResponse. */\
    table.DefinedB =
      lambda(bytes b; b == tagRequest
                || b == tagResponse);\
    table.B2T[tagRequest] = Literal(tagRequest);\
    table.B2T[tagResponse] = Literal(tagResponse);\
    table.DefinedT =
      lambda(term t; t == Literal(tagRequest)
                || t == Literal(tagResponse));\
    table.T2B[Literal(tagRequest)] = tagRequest;\
    table.T2B[Literal(tagResponse)] = tagResponse;\
    wrap(&table);\
    c = claim(&table, closed(&table) &&
                c_stable_log && c_stable_table);)
```

We verified a sample attack using init (), which serves to prove that init () establishes the log and table invariants, as the invariants are proved to hold when the objects are **wrapped**. Furthermore, init () creates a claim c on the table (which owns the log) that says they remain wrapped and stable. Owing to the contracts in RPCshim.h, this claim will be maintained, which ensures from that point on that the log and table can never be opened so their invariants are maintained even in the presence of interference from interleaved threads. Thus the second sort of instrumentation in C′ passes the claim c as ghost parameter to each function and procedure call in C, in accord with their contracts in RPCshim.h. We also add an assertion before each function and procedure call. (Though in fact these assertions are not needed for VCC to verify the example attack.) Let us say "pointer variable" for the variables declared in D with pointer type. Preceding each procedure call $f(\overline{y})$ and function call $x = f(\overline{y})$ in C′ we can assert a conjunction of the form **wrapped**(x_0)&& . . . &&**wrapped**(x_j) where $x_0, \ldots, x_j$ are the pointer variables that have been assigned up to this point. (We gloss over memory safety assertions needed for strings.) By induction on the length of C, we argue that each of these assertions holds, and moreover the type invariants are maintained. An assignment, say $x = f(y, z, w)$;, satisfies the preconditions of f owing to the added claim, the requirement that y, z, w were previously assigned, and the assertion that y, z, w are all wrapped. By inspection of the contracts for each f in RPCshim.h (e.g., att_hmacsha1() given above), that is all that is needed. The postcondition of f ensures that results are wrapped, so in particular x is wrapped at the next assertion (and the claim maintained). ■

Running VCC on RPCattack_0.c not only served to check the init () code used in the proof but also as a sanity check on this Lemma.

Theorem 1: Assume $\emptyset \vdash$ crypto.c $\rightsquigarrow$ crypto.h. For any attack program $M.c$ against the interface $erase($RPCshim.h$)$, the program System + RPCshim.c + $M.c$ is safe.

Proof: We have verified with VCC that:
crypto.h $\vdash$ (RPChybrids.c+RPCprot.c+RPCshim.c) $\rightsquigarrow$ RPCshim.h
By assumption $\emptyset \vdash$ crypto.c $\rightsquigarrow$ crypto.h and Lemma 1 we get
$\vdash$ (crypto.c+RPChybrids.c+RPCprot.c+RPCshim.c) $\rightsquigarrow$ RPCshim.h
i.e., we have $\emptyset \vdash$ (System + RPCshim.c) $\rightsquigarrow$ RPCshim.h by definition of System. By Lemma 2, since $M.c$ is an attack program for $erase($RPCshim.h$)$, we get RPCshim.h $\vdash M.c \rightsquigarrow$ main.h and thus by Lemma 1 we get: $\emptyset \vdash$ (System + RPCshim.c + $M.c$) $\rightsquigarrow$ main.h. So by Assumption 1 the program System + RPCshim.c + $M.c$ is safe. ■

Informal corollary: For all applications A verified against RPCprot.h and the rest of the API (excluding RPCshim.h), A + RPCprot.c + RPChybrids.c + crypto.c is safe in the presence of any active network attacker (under the symbolic model of cryptography). The software stack shown in Section I-C is executable but its real purpose is to show security for a different software stack, without RPCshim.c and RPCattack_0.c but with additional application code that is verified to be memory safe and conform to the protocol API RPCprot.h.

VIII. Summary of Empirical Results

In this section, we summarize our experimental results on implementations of RPC and the variant of the Otway-Rees protocol presented by Abadi and Needham [1].

A. *Results*

We prove authentication properties of the implementations using non-injective correspondences, expressed as assertions on a log of events, by relying on weak secrecy properties, which we prove formally as invariants of the log. The attacker controls the network, can instantiate an unbounded number of principals, and can run unbounded instances of each protocol role —but can never cause a correspondence assertion to fail and can never break the secrecy invariants, unless the Dolev-Yao assumption (no collisions or lucky guesses) has already been violated In particular, we prove the following properties about our sample protocol implementations.

1) RPC: Our implementation of RPC does not let the server reply to unwanted requests, and does not let the client accept a reply that is not related to a previously sent request. Moreover, their shared key remains secret unless either the client or the server is compromised by the attacker.

2) Otway-Rees: The initiator and responder only accept replies from the trusted server that contain a freshly generated key for their specific usage, and this key remains secret unless either the initiator or the responder is compromised.

As both a side-effect and a requirement to use a general purpose verifier, we also prove memory safety properties of our implementations. This can significantly slow verification, especially in parts of the code that handle the building of messages by catenation, and is a large part of the annotation burden.

B. *Performance*

The following table shows verification times, as well as lines of code (LoC) and lines of annotation (LoA) estimations for various implementation files. Times are given as over-approximations of the verification time in minutes (on a mid-end laptop). The number of lines of annotation includes the function contracts, but not earlier definitions. For example, when verifying a function in hybrids.c, all definitions from symcrypt.h can be used but are not counted towards the total. The shim and sample attack programs are verified, as part of the proof of the Theorem, but they are not part of the protocol verification and so are omitted here.

File/Function	LoC	LoA	Time (mins)
symcrypt.h	-	50	≤ 1
table.h	-	50	≤ 1
RPCdefs.h	-	250	≤ 1
ORdefs.h	-	250	≤ 1
hybrids.c	150	300	≤ 5
destruct()	20	**40**	≤ 5
hmacsha1()	20	20	≤ 1
RPCprot.c	130	80	≤ 15
client()	40	20	≤ 5
server()	40	10	≤ 10
ORprot.c	300	100	~ 100
initiator()	40	15	≤ 5
responder()	100	**100**	~ 60
server()	40	15	~ 30

These two case studies confirm previous observations on the annotation burden that comes with general purpose verifiers (and VCC in particular), in "order of one line of annotation per line of code" [19], which was also similar in the F7 implementation of the RPC protocol [12], where the trusted libraries correspond to our hybrid wrappers. For the RPC protocol, our verification times are a lot higher than those of F7 (which were all under 30 seconds), because VCC verifies the program's memory safety simultaneously, whereas F7 relies on F#'s underlying type system and memory structure to do so.

In order to focus on verifying security properties, we simplified several aspects of the implementation that were not relevant to symbolic security and usually require extensive annotations: details of network operations are ignored by the verifier (in particular, each principal is only given one single channel to the attacker), and memory is not freed after use.

IX. Related Work on Protocol Code

We discussed the closely related tools Csur and ASPIER for C and some tools for F# in Section I. We discuss other work on verifying executable code of security protocols.

Pistachio [41] verifies compliance of a C code with a rule-based specification of the communication steps of a protocol. It proves conformance rather than specific security properties. DYC [33] is a C API for symbolic cryptographic protocol messages which can be used to generate executable protocol implementations, and also to generate constraints which can be fed to a constraint solver to search for attacks. Code is checked by model-checking a finite state space rather than being fully verified.

In this paper, we present how a high-level security model can be expressed as part of a C program. Conversely, one can extract a high-level model of the implemented protocol. Symbolic execution of C code is a promising technique for this purpose. Corin and Manzano [21] extend the KLEE symbolic execution engine to represent the outcome of cryptographic algorithms symbolically, but do not consider protocol code. Other recent work [2] extracts verifiable ProVerif models by symbolic execution of C protocol code, on code similar to that of this paper.

There are approaches for verifying implementations of security protocols in other languages. Jürjens [34] describes a specialist tool to transform a Java program's control-flow graph to a Dolev-Yao formalization in FOL which is verified for security properties with automated theorem provers such as SPASS. O'Shea [40] translates Java implementations into formal models to the LySa process calculus so as to perform a security verification. The VerifiCard project uses the ESC/Java2 static verifier to check conformance of JavaCard applications to protocol models (e.g., [31]).

Work on RCF, the concurrent lambda calculus underpinning F7, is directly related. [5] provides conditions under which symbolic security of programs in RCF using cryptographic idealizations implies computational security using cryptographic algorithms. [4] enhances RCF with union and intersection types for the verification of the source code of cryptographic-protocol implementations in F#.

X. Conclusion

We describe a method for guiding a general-purpose C verifier to prove both memory safety and authentication and weak secrecy properties of security protocols and their implementations. Still, our use of VCC leaves clear room for improvement in terms of reducing verification times and numbers of user-supplied annotations. Our strategy of building on a general-purpose C verifier aims to benefit from economies of scale, and in particular to benefit from future improvements in C verification in general. This paper establishes a workable method and a baseline. We encourage verification specialists to take up the challenge.

We plan, for example, to investigate whether we can improve performance by adopting cryptographic invariants in the style of TAPS [17]. Our use of separately-proved secrecy invariants resembles the first-order approach followed in TAPS. We are aware of unpublished work by Cohen on adapting this approach to use with VCC. The TAPS style relies less on axioms, which may or may not place more strain on the first-order prover.

Some of our security annotations can be re-used. In particular, the hybrid wrappers and their contracts need only be written once per cryptographic library, and can be used to verify multiple protocol implementations, as we have done for RPC and Otway-Rees. The representation table is also entirely re-usable. Moreover, we believe that some of the annotations (for example, the log and inductive predicate definitions) may be automatically generated from a high-level description of the protocol.

In future work we intend to adapt our foundations to obtain provably computationally sound results with VCC. We have designed our contracts to correspond to cryptographic assumptions; for example, encryptions give only confidentiality and MACs give only integrity. The table structure and hybrid wrappers introduced in Section IV resemble some standard methods for computational soundness, such as the the dual interpretation of the interface in BPW [6], or the hybrid wrappers used in [27]. We may also try more direct methods to obtain computational security results, for example by using the idealized interface from [27] and assuming a computationally sound implementation (or linking the C code against the F# implementation), or by extending the verifier with probabilistic semantics for C, similar to the probabilistic semantics given to the pWhile language in CertiCrypt [8].

Verification of symbolic security properties remains relevant, even without a computational soundness result, as recent attacks on prominent protocols and implementations could have been found by symbolic protocol verification. Moreover, some standard features of security protocols (such as sending encrypted keys over the network) are hard to prove secure in the computational model, but may be studied in symbolic models.

For highest assurance, the underlying libraries (`crypto.c`) would be verified, as would any application code using the protocol library. Moreover, the C verifier would be proved sound with respect to a semantics for which the compiler is proved to be correct (as in the Verified Software Toolchain [3] built on the C semantics of CompCert [35]).

Acknowledgements: Discussions with Misha Aizatulin, Karthik Bhargavan, Josh Berdine, Ernie Cohen, Cédric Fournet, Bashar Nuseibeh, and Thomas Santen were useful. Mark Hillebrand and Michał Moskal helped with VCC methodology, and Stephan Tobies helped with understanding VCC internals. Naumann acknowledges partial support from Microsoft and NSF awards CRI-0708330, CCF-0915611.

References

[1] M. Abadi and R. M. Needham, “Prudent engineering practice for cryptographic protocols,” *IEEE Trans. Software Eng.*, vol. 22, no. 1, pp. 6–15, 1996.

[2] M. Aizatulin, A. D. Gordon, and J. Jürjens, “Extracting and verifying cryptographic models from C protocol code by symbolic execution,” 2011, unpublished draft. [Online]. Available: http://users.mct.open.ac.uk/ma4962/files/paper-full.pdf

[3] A. W. Appel, “Verified software toolchain,” in *ESOP*, ser. LNCS, vol. 6602, 2011, pp. 1–17.

[4] M. Backes, C. Hriţcu, and M. Maffei, “Union and intersection types for secure protocol implementations,” in *TOSCA*, 2011.

[5] M. Backes, M. Maffei, and D. Unruh, “Computationally sound verification of source code,” in *ACM CCS*, 2010, pp. 387–398.

[6] M. Backes, B. Pfitzmann, and M. Waidner, “A composable cryptographic library with nested operations,” in *ACM CCS*, 2003, pp. 220–230.

[7] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, “Boogie: A modular reusable verifier for object-oriented programs,” in *FMCO*, ser. LNCS, vol. 4111, 2005, pp. 364–387.

[8] G. Barthe, B. Grégoire, and S. Z. Béguelin, “Formal certification of code-based cryptographic proofs,” in *POPL*, 2009, pp. 90–101.

[9] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis, “Refinement types for secure implementations,” *ACM TOPLAS*, vol. 33, no. 2, p. 8, 2011.

[10] K. Bhargavan, C. Fournet, A. D. Gordon, and S. Tse, “Verified interoperable implementations of security protocols,” *ACM TOPLAS*, vol. 31, pp. 5:1–5:61, December 2008.

[11] K. Bhargavan, C. Fournet, R. Corin, and E. Zalinescu, "Cryptographically verified implementations for tls," in *ACM CCS*, 2008, pp. 459–468.

[12] K. Bhargavan, C. Fournet, and A. D. Gordon, "Modular verification of security protocol code by typing," in *POPL*, 2010, pp. 445–456.

[13] B. Blanchet, "An efficient cryptographic protocol verifier based on prolog rules," in *CSFW*, 2001, pp. 82–96.

[14] ——, "A computationally sound mechanized prover for security protocols," in *IEEE Symposium on Security and Privacy*, 2006, pp. 140–154.

[15] I. Cervesato, A. D. Jaggard, A. Scedrov, J.-K. Tsay, and C. Walstad, "Breaking and fixing public-key Kerberos," in *ASIAN*, ser. LNCS, vol. 4435, 2006, pp. 167–181.

[16] S. Chaki and A. Datta, "ASPIER: an automated framework for verifying security protocol implementations," CyLab, Carnegie Mellon University, Technical CMU-CyLab-08-012, 2008.

[17] E. Cohen, "First-order verification of cryptographic protocols," *Journal of Computer Security*, vol. 11, no. 2, pp. 189–216, 2003.

[18] 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*, ser. LNCS, vol. 5674, 2009, pp. 23–42.

[19] E. Cohen, M. Moskal, W. Schulte, and S. Tobies, "Local verification of global invariants in concurrent programs," in *CAV*, ser. LNCS, vol. 6174, 2010, pp. 480–494.

[20] E. Cohen and B. Schirmer, "From total store order to sequential consistency: A practical reduction theorem," in *Interactive Theorem Proving*, ser. LNCS, vol. 6172, 2010, pp. 403–418.

[21] R. Corin and F. A. Manzano, "Efficient symbolic execution for analysing cryptographic protocol implementations," in *ESSoS*, ser. LNCS, vol. 6542, 2011, pp. 58–72.

[22] L. Correnson, P. Cuoq, A. Puccetti, and J. Signoles, *Frama-C User Manual*. [Online]. Available: http://frama-c.com/download/frama-c-user-manual.pdf

[23] L. M. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *TACAS*, ser. LNCS, vol. 4963, 2008, pp. 337–340.

[24] D. Dolev and A. C.-C. Yao, "On the security of public key protocols," *IEEE Transactions on Information Theory*, vol. 29, no. 2, pp. 198–207, 1983.

[25] F. Dupressoir, A. Gordon, and J. Jürjens, "Verifying authentication properties of C security protocol code using general verifiers," in *Workshop on Analysis of Security APIs (ASA-4 - FLOC 2010)*, 2010, presentations only.

[26] F. Dupressoir, A. Gordon, J. Jürjens, and D. Naumann, "Guiding a general-purpose C verifier to prove cryptographic protocols," Tech. Rep. MSR–TR–2011–50, 2011.

[27] C. Fournet, "Cryptographic soundness for program verification by typing," 2011, unpublished draft.

[28] "Crypto-verifying protocol implementations in ML," project website at http://msr-inria.inria.fr/projects/sec/fs2cv/.

[29] A. D. Gordon, "Provable implementations of security protocols," in *21th IEEE Symposium on Logic in Computer Science (LICS 2006)*, 2006, pp. 345–346.

[30] J. Goubault-Larrecq and F. Parrennes, "Cryptographic protocol analysis on real c code," in *VMCAI*, ser. LNCS, vol. 3385, 2005, pp. 363–379.

[31] E. Hubbers, M. Oostdijk, and E. Poll, "Implementing a formally verifiable security protocol in Java Card," in *Security in Pervasive Computing*, ser. LNCS, vol. 2802, 2004, pp. 213–226.

[32] B. Jacobs and F. Piessens, "The VeriFast program verifier," Katholieke Universiteit Leuven, Report CS 520, Aug. 2008.

[33] A. S. A. Jeffrey and R. Ley-Wild, "Dynamic model checking of C cryptographic protocol implementations," in *FCS-ARSPA*, 2006.

[34] J. Jürjens, "Security analysis of crypto-based java programs using automated theorem provers," in *ASE*, 2006, pp. 167–176.

[35] X. Leroy, "Formal certification of a compiler back-end or: programming a compiler with a proof assistant," in *POPL*, 2006, pp. 42–54.

[36] S. J. Murdoch, S. Drimer, R. J. Anderson, and M. Bond, "Chip and PIN is broken," in *IEEE Symposium on Security and Privacy*, 2010, pp. 433–446.

[37] R. M. Needham and M. D. Schroeder, "Using encryption for authentication in large networks of computers," *Commun. ACM*, vol. 21, no. 12, pp. 993–999, 1978.

[38] oCERT, "oCERT advisory #2008-16 multiple OpenSSL signature verification API misuse," 2009. [Online]. Available: http://www.ocert.org/advisories/ocert-2008-016.html

[39] Offspark, "Polarssl," 2008. [Online]. Available: http://polarssl.org

[40] N. O'Shea, "Using Elyjah to analyse Java implementations of cryptographic protocols," in *FCS-ARSPA-WITS*, 2008, pp. 211–223.

[41] O. Udrea, C. Lumezanu, and J. S. Foster, "Rule-based static analysis of network protocol implementations," *USENIX Security Symposium*, pp. 193–208, 2006.