Verifying implementations of security protocols by refinement

Nadia Polikarpova[1] and Michał Moskal[2]

[1] Chair of Software Engineering, ETH Zurich, Switzerland
nadia.polikapova@inf.ethz.ch
[2] Microsoft Research Redmond
michal.moskal@microsoft.com

Abstract. We propose a technique for verifying high-level security properties of cryptographic protocol implementations based on stepwise refinement. Our refinement strategy supports reasoning about abstract protocol descriptions in the symbolic model of cryptography and gradually concretizing them towards executable code. We have implemented the technique within a general-purpose program verifier VCC and applied it to an extract from a draft reference implementation of Trusted Platform Module, written in C.

1 Introduction

Vulnerabilities in security-critical code can arise either from defects in underlying cryptographic protocols, or from inconsistencies between the protocol and the implementation. A lot of research in security is devoted to verifying abstract protocol definitions against high-level *security goals*, such as secrecy and authentication. An example of such a goal could be that an honest client only accepts a message from a server, if it is related to a previously made request.

On the implementation side, techniques of static and dynamic program analysis are applicable for checking low-level properties of the code, such as absence of buffer overruns. A challenge that is addressed by very few existing approaches is bridging the gap between the two sides, that is, verifying that a program implements a certain protocol and accomplishes the same security goals.

Most protocol implementations are written manually in an imperative programming language, often C. In this work we use VCC [5], a general-purpose verifier for C programs, to prove the security of both the protocol and its implementation.

This task poses two main challenges. First, VCC is a verifier for arbitrary functional properties but does not support expressing secrecy and authentication in a straightforward manner. Second, it is desirable to decouple the security goals of the protocol from the properties of its implementation to allow for simpler specifications and more modular proofs.

VCC has been used previously for verification of cryptographic protocols [8]. Our work is an extension thereupon, with two main differences. First, our approach relies entirely on VCC and does not require external security proofs in Coq, which avoids translation between different formalisms and utilizes VCC strengths in handling mutable state. Second, we have applied the technique to a more complex, stateful protocol,

which prompted the scalability problems and thus development of our refinement-based verification approach, which is the main contribution of this paper.

Our refinement strategy consist of three levels. The initial model (L0) describes a security protocol on the level of abstraction of a conventional protocol definition language (message sequence chart, role script and similar). The purpose of the first refinement (L1) is to switch from an abstract representation of protocol messages to their concrete byte string format. Finally, the second refinement (L2) connects the protocol model to the real implementation.

We applied this approach to a key management protocol from a slightly simplified version of the code submitted by Microsoft to the Trusted Computing Group for the reference implementation of the Trusted Platform Module version 2. The simplifications we made are described in Sect. 5; however note that we did not change the data structures that store cryptographic keys or the code manipulating those keys. We therefore believe that the case study still represents a realistic protocol implementation in C; moreover it is not a standalone program, but a part of a software system of substantial size.

The next two sections introduce the VCC verifier and the Trusted Platform Module. Sect. 4 describes the proposed refinement approach in detail. Sect. 5 summarizes the results of our case study, Sect. 6 discusses some related work and Sect. 7 concludes.

2 Background: The VCC Verifier

VCC is a deductive verifier for C programs. It checks that a program adheres to a specification in the form of inline assertions, function pre- and post-conditions and *object invariants*, which are associated with compound C types (struct and unions). It works by generating verification conditions from the program and its specification, via the Boogie intermediate language [2]. These conditions are discharged by a reasoning engine of choice, usually Z3 [6].

The specifications are provided as annotations directly in the source code and are expressed in a language that includes the C expression language, and extends it with quantification, user-defined predicates and functions. Additionally, the annotations can provide ghost code manipulating ghost data, i.e., code and data removed before the program is executed. VCC supports unbounded integers, maps between arbitrary types (defined using lambda-expressions), and user-defined algebraic data-types, which all can be used in ghost code. This added expressivity is useful for specifying data structures. For example, a red-black tree can be equipped with a ghost field of a map type representing the current value of the tree; functions operating on the tree can then be fully functionally specified with respect to that field. We shall use a very similar mechanism to represent the state of a protocol execution.

In VCC object invariants specify ownership relations: an object would usually own all the objects that comprise its internal representation. Together with ghost fields abstracting over values of complex object trees, ownership provides for data abstraction.

Our approach to verifying security properties can be applied to any verifier with a similar feature-set. While both ownership and invariants are built into VCC, we be-

lieve the technique would work equally well if they were implemented on top of, e.g., separation logic in a tool like VeriFast [10].

3 Case study: Trusted Platform Module

A Trusted Platform Module (TPM) is a secure cryptoprocessor used in most modern personal computers. A TPM device is equipped with volatile and non-volatile memory and can generate cryptographic keys, restrict their further usage, perform random number generation, as well as other cryptographic operations (hashing, encryption, signing). TPM can provide a wide range of hardware security services, most popular being cryptographic key management, ensuring platform integrity and disk encryption (for example the BitLocker Drive Encryption feature of Microsoft Windows).

Being a member of Trusted Computing Group, Microsoft has submitted for comments a draft of the new generation of the TPM standard: version 2.0. It consists of about 1000 pages of natural language specification and a reference implementation in C (approximately 30000 lines of code). TPM[3] supports a total of 102 commands, such as create a cryptographic key, load a key for further use, decrypt a ciphertext, etc.

In this case study we restrict ourselves to the key management functionality of TPM, as it is the core part needed for most applications. Keys used by the TPM (also called *objects*)[4] are organized in a hierarchy, where child keys are protected by their parents. The TPM specification poses a restriction that objects protecting other objects (called *storage keys*) are not allowed to be imported, that is, have to be created and used by the same TPM. We only consider management of such storage keys.

Internally a storage key consists of a *public* and a *sensitive* area. The former contains an asymmetric public key together with object attributes and parameters; the latter consists of an asymmetric private key as well as a *symmetric protection key*. Whenever an object resides in the TPM internal memory, its sensitive area can be stored unencrypted; however when the object is stored off TPM, its sensitive area is *protected* by the protection key of its parent. A protected sensitive area is called a *private* area.

3.1 Creating and Loading Objects

A typical usage scenario of a storage key is given in Listing 1. To create a key one invokes the Create command of the TPM, which returns the public and private areas of the new object. Before the key can be used (e.g., to decrypt data or protect other keys), it needs to be loaded into the internal memory of the TPM using the Load command.

We would like to verify the following two properties about this scenario:

1. sensitive areas of all the objects in the hierarchy remain secret (not known outside the TPM);
2. whenever a storage key is successfully loaded into a TPM under some parent, it had been created by the same TPM under the same parent.

[3] For brevity, we will use this name to refer to the draft specification.

[4] Note that the notion of object in TPM (a cryptographic key) is different from object in VCC (an instance of a compound type), but the former can be implemented using the latter.

```
<public, private> =
    Create(parent_handle);
// other code
// ...
handle = Load(parent_handle,
    public, private);
plain = Decrypt(handle, cipher);
```

Listing 1: Example usage of a storage key

```
Create(parent_handle) {
    <public, sensitive> = Random();
    integrity = Hash(<sensitive, public>);
    private = Cipher(<integrity, sensitive>,
        protection_key(parent_handle));
    return <public, private>;
}
```

Listing 2: Pseudocode of Create

Looking at an abstract description of Create (Listing 2), it is easy to reason informally that those properties are ensured by the format of the private area. Indeed, an external client cannot encrypt arbitrary messages with a secret protection key, thus to load a fake object he would have to *guess* two values pub and $priv$, such that, if $priv$ decrypts to a pair $< i, s >$, then $i = h(s, pub)$ (where h is a known hash function).

3.2 Key Management as a Security Protocol

The scenario described above can be expressed as a single-message security protocol, where an agent C (Create) sends a message to an agent L (Load):

$$C \to L : par, p, \{|h(p, s), s|\}_{k(C,L,par)}$$

Here comma is used for pairing, $\{| \cdot |\}$ is symmetric encryption, par is an identifier of the parent, p and s are the public and the sensitive area of an object and $k(C, L, par)$ is a key shared between C and L (the protection key of the parent).

The security goals listed above can be expressed as follows:

1. (*secrecy*): s, $k(C, L, par)$ are secret, shared between agents C and L;
2. (*authentication*): agent L agrees with C on par, p, s; that is, whenever L receives a message of the above format, he can be sure that is has once been sent by C with the same parameters par, p and s.

Symbolic formalisms for protocol verification usually consider agents communicating through a network controlled by a *Dolev-Yao attacker* [7]. Such an attacker can eavesdrop, decompose and recompose messages, but never relies on luck, e.g., he cannot guess a secret key or accidentally obtain a required hash value due to a collision. We adopt the same attacker model for our approach.

4 Refinement approach

To deal with the complexity of specification and verification of real-world security code we propose an approach based on stepwise refinement in the style of Event-B [1]. The main idea is to first reason about a high-level protocol description (like the one given in the previous section) and then gradually concretize it towards the real implementation. Refinement enables us to separate different aspects of the problem, in particular the security of the protocol and whether the protocol is correctly implemented by the code.

```
_(ghost typedef struct {
  // Unbounded integer:
  \integer a;
  // Invariant on the state space:
  _(invariant a > 0)
} State_0;
State_0 s0;

void event_0()
  _(updates &s0)
  _(ensures s0.a == \old(s0.a) + 1)
{
  _(unwrapping &s0) {
    s0.a = s0.a + 1;
  }
})
```

Listing 3: An example initial model in VCC

```
_(ghost typedef struct {
  \integer b, c;
  _(invariant \mine(&s0)) // Ownership
  // Gluing invariant:
  _(invariant s0.a == b + c)
} State_1;
State_1 s1;

void event_1()
  _(updates &s1)
  _(ensures s1.b == \old(s1.b) + 1)
{
  _(unwrapping &s1) {
    event_0();
    s1.b = s1.b + 1;
  }
})
```

Listing 4: An example refinement in VCC

In Event-B the *initial model* of a system consists of an abstract state space constrained by invariants and a set of events that update the abstract state. The purpose of the invariants is to express high-level system requirements (in our case, the security goals) in a formal but straightforward way; all events have to be proved to maintain the invariants.

Each *refinement* introduces new (concrete) variables that are linked to the old (abstract) variables by a *gluing invariant*. An essential feature of stepwise refinement is that verifying a concrete event (that manipulates concrete variables) does not require re-verifying preservation of the abstract invariants, but only of the gluing invariants. This enables decomposition of proofs in addition to specifications.

4.1 Refinement in VCC

Unlike Event-B we do not use a special-purpose notation and tool support, but rather encode refinement within a general-purpose program verifier. One of the benefits is a seamless transition from the model to the actual implementation, without relying on a code generator.

In VCC we represent the state space of the initial model as a ghost struct and often make use of unbounded integers and infinite maps to achieve a higher level of abstraction (see Listing 3 for an example). An event is encoded as a ghost function with a contract _(**updates** &state). This contract allows the function to open up the state object (using a statement _(**unwrapping** &state)) and modify it, as long as its invariant is preserved, which is exactly the proof obligation we would like to impose on an event.

At least one of the events in a model has to ensure the invariant without requiring it (it is called the *initialization event*), in order to make sure that the invariant is consistent.

On each refinement level concrete state is encoded as a struct (physical if it is the final refinement, and ghost otherwise), equipped with a gluing invariant (Listing 4). The ownership system of VCC makes it possible to express the refinement relation in an

elegant way. VCC does not allow mentioning abstract variables in the gluing invariant, unless the abstract state is *owned* by the concrete state. The system then guarantees that whenever the concrete state is known to be consistent (i.e., in between executing the events) no code could modify the abstract state without updating the concrete variables accordingly.

Concrete events are encoded as functions with an _(**updates** ...) contract for concrete state. A body of such a function calls its abstract counterpart and then updates the concrete state to reestablish the gluing invariant. Because the abstract event is already known to maintain the invariant of the abstract state, only the preservation of the gluing invariant has to be proved at this point.

Note that this approach uses refinement only as a methodology for structuring specifications and proofs; the validity of verification results does not depend on maintaining a simulation relation between the models on different levels of abstraction. Using the example above, if the postcondition of event_1 faithfully describes its intended effect, and the invariants of State_0 and State_1 capture all the system requirements, then it does not matter if event_1 refines event_0; in fact, the latter can be eliminated from the system, letting the former update s0 directly (and thus take up the burden of proving the preservation of s0's invariant).

In the next three sections we describe in detail the three models (the initial model and its two refinements) that we propose for verifying protocol implementations. We explain the approach in terms of the create-load protocol from our case study, however we believe that the description extends naturally to other similar protocols and security goals.

4.2 High-level protocol definition (L0)

Our initial model describes the protocol in the formalism of symbolic cryptography. In this formalism the set of messages is modelled as a *term algebra*, which in VCC can be defined using the **datatype** construct (Listing 6).[5]

In our example the algebra consists of byte string literals, terms denoting cryptographic operations (encryption and hashing) as well as three different kinds of *compound terms*: Sensitive, Object and Private. These three constructors essentially all represent pairs; we distinguish them, as their concrete representations in the actual code have different formats. To justify the distinction, we have to prove that no two compound terms of different kinds can correspond to the same concrete representation, which is discussed in Sect. 4.3.

The variables of the initial model keep track of the state of a protocol execution, namely:

- internal TPM state: the object hierarchy and loaded objects;
- all the terms created by honest participants (i.e., the TPM) as part of the protocol messages;
- all the terms known to the attacker, either eavesdropped or constructed.

[5] The listings in this section are somewhat simplified for clarity; the full VCC source code of our case study is available under http://se.inf.ethz.ch/people/polikarpova/tpm.zip.

```
_(ghost
typedef struct {
  // Objects in the hierarchy
  bool objects[Term];
  // Mapping from an object to its parent
  Term parent[Term];
  // Loaded objects
  bool loaded[Term];
  // Terms generated by honest agents
  bool protocol[Term];
  // Terms known to the attacker
  bool attacker[Term];

  _(invariant \forall Term pub, sen; objects[Object(pub, sen)] ==> !attacker[sen])
  _(invariant \forall Term pub, sen, k; is_object_sym_key(\this, k) ==>
    attacker[Cipher(Private(Hash(Integrity(sen, pub)), sen), k)] ==>
      objects[Object(pub, sen)] && symkey(parent[Object(pub, sen)]) == k)
  // ... more invariants ...
} Log;
Log log;)
```

Listing 5: Protocol log

```
_(datatype Term {
  case Literal(ByteString s);
  case Sensitive(Term skey, Term symkey);
  case Object(Term pub, Term sen);
  case Private(Term int_hash, Term sen);
  case Cipher(Term t, Term k);
  case Hash(Term t);
})
```

Listing 6: Term algebra

We use infinite maps (denoted **bool** ...[Term]) to represent sets of terms through their characteristic functions. The state space is encoded in a ghost struct called Log (Listing 5). We also define a function **bool** used(Log, Term) that represents the union of protocol and attacker sets of a log.

Each event of the model encodes a protocol step of one of three kinds: either an honest agent *sending* a message (revealing it to the attacker), an honest agent *receiving* a message, or the attacker constructing a new message from his current knowledge.

Secrecy goals are encoded as invariants of the log, restricting the attacker set. The first of the object invariants in Listing 5 is an example of such a *security invariant*: it states that for all objects in the hierarchy their sensitive area is never known to the attacker. The second one is an *auxiliary invariant*, which is not part of the requirements and is only used to assist verification.

An authentication goal is always associated with an honest agent receiving a message: upon receipt the agent wants to be sure that the message came from a certain principal. Thus authentication goals are encoded as *security postconditions* of receive events (see Load for an example).

Verifying the events against the security invariants and postconditions, guarantees that after an arbitrary sequence of protocol steps all secrecy properties will hold, and if a receive event is then invoked, its associated authentication properties will also hold. For verification to be meaningful we need to make sure that all the security requirements of interest are formalized as either invariants of the log or postconditions of receive events, and the set of attacker capabilities is not too restrictive. In our approach those specifications are expressed explicitly and straightforwardly, which reduces the probability of a modeling error. Auxiliary invariants, on the other hand, can be added and corrected gradually: VCC will check if they are erroneous or insufficient.

```
void create_0(Term parent, Term obj)
  _(requires log.loaded[parent])
  _(requires \forall Term t;
    subterms(obj)[t] ==> !used(t))
  _(updates &log)
  _(ensures log.attacker[pub(obj)] &&
    log.attacker[Cipher(private_term(obj),
        symkey(parent))])
{
  _(unwrapping &log) {
    log.objects[obj] = \true;
    log.parent[obj] = parent;
    log.protocol = set_union(
        log.protocol, subterms(obj));

    Term enc_private = Cipher(
        private_term(obj),
        symkey(parent));
    log.protocol = set_union(
        log.protocol, subterms(enc_private));

    log.attacker[pub(obj)] = \true;
    log.attacker[enc_private] = \true;
  }
}
```

Listing 7: Create event on L0

```
void load_0(Term parent, Term obj)
  _(requires log.loaded[parent])
  _(requires log.attacker[pub(obj)] &&
    log.attacker[Cipher(private_term(obj),
        symkey(parent))])
  _(updates &log)
  // Authentication postcondition:
  _(ensures \old(log.objects)[obj] &&
    \old(log.parent)[obj] == parent)
{
  _(unwrapping &log)
    log.loaded[obj] = \true;
}

void att_decrypt_0(Term t, Term k)
  _(requires log.attacker[Cipher(t, k)])
  _(requires log.attacker[k])
  _(updates &log)
  _(ensures log.attacker[t])
{
  _(unwrapping &log)
    log.attacker[t] = \true;
}
```

Listing 8: Load event and attacker's decryption capability on L0

A send event (Create in our example) is modelled by a ghost function that adds terms to the protocol set, publishes some of them in the attacker set and possibly updates the internal state (Listing 7). It might seem counterintuitive that create_0 receives the new object as an argument. In fact, the actual generation of a fresh storage key happens in the physical code; this key is then passed to create_0, whose only purpose is to register the key the log.

A receive event, such as Load, does not change the protocol and attacker sets, but might update the internal state. The event's precondition states that the message must be a part of the attacker knowledge, which models the fact that all communications in the system go through the attacker. In our example (Listing 8) the event is equipped with an authentication postcondition, which states that the loaded object belongs to the TPM hierarchy (and thus was sent by the Create event, as no other event can modify the hierarchy), and it is loaded under its initial parent.

A Dolev-Yao attacker is usually modelled as a set of deduction rules that transitively define the set of messages he can construct from some initial set. We encode those rules as events that add terms to the attacker set, with the premise of the rule corresponding to the event's precondition and the conclusion of the rule corresponding to the postcondition. For example, Listing 8 shows an event that models attacker's capability to decrypt a ciphertext once he knows the key.

For our case study we used the standard Dolev-Yao rules: generating a fresh literal, construction and destruction of compound terms, encryption and decryption, hash-

ing. We also encoded several non-standard attacker capabilities in order to relax overly strong assumptions of symbolic cryptography. One of them is encrypting a fresh literal with a key not known to the attacker; it models the situation when the attacker provides an honest agent with a random string, when an encryption is expected. The other one is decomposing an encryption of a compound term with an unknown key into two encryptions and vice versa; this event models the distributivity of stream and block ciphers over concatenation.

Verifying correctness of the events requires adding auxiliary invariants to the log. While these invariants are checked by VCC and thus do not need to be trusted (they cannot accidentally weaken the security invariants or the attacker model), getting them right is a non-trivial task. Based on our experience, we can suggest the following invariant patterns:

1. *Dolev-Yao rules* describing how the attacker could have learnt a particular term. For instance an invariant `\forall Term t, k; attacker[Cipher(t, k)] ==> protocol[Cipher(t, k)] || (attacker[t] && attacker[k])` says that he can learn a ciphertext by either eavesdropping it or constructing it from a plaintext and a key he knows. Those invariants do not depend on the protocol, but only on the attacker model and the term algebra (and thus reusable).
2. Invariants stating that the protocol set and the used set are closed under addition of subterms (also protocol independent).
3. *Message format* invariants, describing the shape of the messages generated by honest agents. For example an invariant `\forall Term t, k; protocol[Cipher(t, k)] ==> is_object_sym_key(\this, k)` says that honest agents only produce encryptions with secret symmetric keys. These invariants are in general not reusable, but can be derived straightforwardly from the protocol. Note that there is no harm in making them stronger than required by adding all the knowledge about the protocol messages.
4. *Internal data* invariants, for example saying that a public key of an object in the hierarchy never coincides with a private key of the same or another object. These invariants are protocol-specific and most of the time have to be deduced from verification errors.
5. Additional *attacker restrictions* that do not correspond directly to the security goals. These are protocol-dependent and usually tricky to figure out. For example, we had to state that honest agents never publish a plaintext private area: `\forall Term t1, t2; protocol[Private(t1, t2)] ==> !attacker[Private(t1, t2)]`, because the protocol security relies on the fact that private areas are only sent encrypted.

4.3 From term algebra to byte strings (L1)

The initial model represents protocol messages as symbolic terms, while in the physical code messages are plain byte strings. In order to connect the two descriptions, we need a means to match a term to its string representation and vice versa. To this end, we have developed a VCC library containing a ghost type ByteString that encodes finite sequences of bytes of arbitrary size, together with a number of specification functions manipulating those sequences. For example, the function `from_array(BYTE *data, \integer size)` returns a byte string stored in a physical byte array.

```
_(ghost typedef struct {
  // Set of used strings:
  bool strings[ByteString];
  // Mapping from strings to terms:
  Term term[ByteString];

  // Ownership:
  _(invariant \mine(&log))
  // Gluing invariants:
  _(invariant \forall ByteString s;
    strings[s] ==> used(log, term[s])
      && string(term[s]) == s)
  _(invariant \forall Term t;
    used(log, t) ==> strings[string(t)]
      && term[string(t)] == t)
} Table)
_(ghost Table table)
```

Listing 9: Representation table

```
void att_compute_hash_1(ByteString s)
  _(requires attacker_string(s))
  _(updates &table)
  _(ensures attacker_string(lib_hash(s)))
{
  // Symbolic assumption:
  _(assume table.strings[lib_hash(s)] ==>
    is_hash(table.term[lib_hash(s)]) &&
    string(hash_arg(
        table.term[lib_hash(s)])) == s)

  _(unwrapping &table) {
    att_compute_hash_0(table.term[s]);
    add(Hash(table.term[s]));
  }
}
```

Listing 10: Attacker's hashing capability on L1

Matching terms to strings is straightforward, as each term has a unique string representation. We introduce a specification function ByteString string(Term t) that defines the string format for every term constructor in such a way that it corresponds to the physical code. As for cryptographic terms (Cipher and Hash), we assume that the implementation uses a trusted cryptographic library to compute the corresponding byte strings, and its specification and verification is outside of the scope of our problem. Thus we model these operations with uninterpreted functions lib_encrypt and lib_hash that operate on values of type ByteString.

The other direction — mapping strings to terms — cannot be expressed as a function for cardinality reasons (e.g., hashing is generally not injective, and a ciphertext can in principle coincide with a hash value). To be able to apply the symbolic model of cryptography to byte string messages, following [8], we make *symbolic assumptions* on string-manipulating operations:

- an operation that corresponds to constructing a new term, cannot produce a string that had been used before as a representation of a different term;
- if an operation requires the corresponding term to be of a particular type, it cannot be performed on a string that represents a term of a different type (for example, a string that is obtained as a hash cannot be decrypted).

In the code instances of these assumptions appear in inline **assume** statements of L1 events (see examples below).

Symbolic assumptions guarantee that string has no collisions within the set of terms used in a protocol execution, and thus there exists a mapping from used string to used terms. This mapping, together with the set of used strings and the obvious gluing invariant is stored in a data structure called *representation table* (Listing 9).

The set of events of L1 closely corresponds to that of L0, except that they are specified using byte strings rather than terms. Following the general approach of Sect. 4.1, each refined event calls its L0 counterpart to modify the log and then updates the rep-

```
_(dynamic_owns) typedef struct {
  OBJECT_SLOT slots[MAX_LOADED];

  _(invariant \forall \integer i;
    0 <= i && i < MAX_LOADED
      ==> \mine(&slots[i]))
  _(invariant \mine(&table))
  // Gluing invariant:
  _(invariant \forall \integer i;
    0 <= i && i < MAX_LOADED &&
    slots[i].occupied ==>
    log.loaded[term(&slots[i].object)])
} TPM_STORAGE;
TPM_STORAGE storage;
```

Listing 11: Physical state of the TPM

```
typedef struct {
  UINT16 keySize;
  BYTE key[MAX_SYM_DATA];
  _(ghost ByteString content)
  _(invariant
      keySize <= MAX_SYM_DATA)
  _(invariant
      content ==
      from_array(key, keySize))
} SYM_KEY;
```

Listing 12: Physical representation of a symmetric key

resentation table accordingly. As an example let us consider the attacker's hashing capability (Listing 10). The attacker_string predicate states that a string represents a term known to the attacker. The add(Term) function adds a term and the corresponding string to the representation table, provided the no-collision condition holds: the string is not yet associated with another term. To satisfy this condition we have to add a symbolic assumption to the body of att_compute_hash_1. It states that if lib_hash(s) already occurs in the representation table, its corresponding term is a Hash, and moreover the argument of the hash can only map to s (i.e., we did not encounter a collision of lib_hash).

Symbolic assumptions are weaker for terms whose string representation is indeed injective. For example, when adding an encryption lib_encrypt(s, k) it is sufficient to assume that the corresponding term, if in the table, is a Cipher and its key maps to k; we can then prove that its plaintext also maps to s.

Sound handling of compound terms requires their string representation to be injective not only in both parts of the compound, but also in its type. Essentially, one has to verify for the message format used in the physical code that for any byte string there is at most one way to parse it into a compound term. With this property, the only symbolic assumption that is needed to add a compound term to the table is that its representation does not coincide with the representation of any literal, encryption or hash, which is reasonable. Without relying on injectivity it is hard to justify absence of collisions among compound terms.

In general, one has to be careful with symbolic assumptions, as they are a part of the trusted specification. Similarly to [8], our first refinement makes those assumptions *explicit*, which simplifies their informal validation.

4.4 Physical code (L2)

The concrete variables of the second refinement are the global variables of the physical code. In our case, TPM stores a list of loaded objects (as an array of object slots that can be either occupied or empty). We add a gluing invariant connecting this array to the loaded set of the log (Listing 11).

```
TPM_RC Create(UINT32 parentHandle, PUBLIC *public, PRIVATE *private)
  _(requires object_exists(parentHandle))
  _(updates &storage)
  _(ensures attacker_string(public->content)) // "Gluing" postcondition
  _(ensures attacker_string(private->content)) // "Gluing" postcondition

TPM_RC Load(UINT32 parentHandle, PRIVATE *private, PUBLIC *public,
  UINT32 *objectHandle)
  _(requires object_exists(parentHandle))
  _(requires attacker_string(public->content)) // "Gluing" precondition
  _(requires attacker_string(private->content)) // "Gluing" precondition
  _(updates &storage)
  // Authentication postcondition:
  _(ensures \result == SUCCESS ==>
    \old(log.objects)[term(storage.slots[*objectHandle].object)] &&
    \old(log.parent)[term(storage.slots[*objectHandle].object)] ==
      term(storage.slots[parentHandle].object))

void SymmetricEncrypt(UINT16 keyBits, BYTE *key, UINT16 dataSize, BYTE* data)
    _(ensures from_array(data, dataSize) ==
        lib_encrypt(\old(from_array(data, dataSize)), from_array(key, keyBits / 8)))
```

Listing 13: Some contracts of Create and Load and an extract from the cryptographic library

A TPM object is represented in the code by an instance of the OBJECT struct, which contains instances of PUBLIC and SENSITIVE. As expected, PUBLIC stores the public asymmetric key, while SENSITIVE stores the secret asymmetric key and an instance of SYM_KEY, which in turn stores the symmetric protection key. All keys are stored in statically allocated buffers with a length field (see Listing 12 for an example). To access the byte string stored in a buffer we add a ghost field content to all structs that contain such buffers.

Note that there is no physical data structure representing the attacker knowledge, which could be connected by a gluing invariant to the attacker set of the log. Instead all the information flowing in and out of the TPM should be considered known to the attacker. This property can be encoded in the pre- and postconditions of TPM functions that communicate with the outside world.

For an example, let us look at Create (Listing 13). Its signature reveals that it can pass information to the outside world through two buffers: public and private. Thus it has to be equipped with a *gluing postcondition*, stating that the content of each output buffer corresponds to a term known to the attacker. An intuition behind this postcondition is that whatever Create returns to the outside world is safe to publish, because, according to the results of L0, the attacker cannot use it to violate the security goals.

The gluing postcondition forces Create to update the log and the representation table, which is done by invoking its L1 counterpart, create_1. To make the connection between the two levels, we have to verify that the protocol messages computed by the physical code are the same as described by the ghost code. To achieve that for cryptographic terms Cipher and Hash we have to instrument the cryptographic library

with contracts in terms of the uninterpreted functions lib_encrypt and lib_hash introduced in L1 (see an example in Listing 13).

Our second TPM command, Load (Listing 13), receives data from the outside world; thus the content of its input buffers has to be related to the attacker knowledge through *gluing preconditions*. Note that these preconditions have somewhat special semantics: they do not express a contract with the client code, but rather our model of the client.

The gluing invariant of storage forces Load to update the loaded set of the log, which is accomplished through a call to load_1. The latter requires not only that the input strings be known to the attacker, but also that they have a correct format (in this case: that the integrity value matches). Those preconditions are established by the physical code that parses the input buffers and performs the integrity check.

We do not provide physical code for the attacker events on L2, because a concrete implementation of a TPM client would be of limited usefulness for our problem. Rather we would like to argue that whatever program a client can write using the given TPM interface, can be proved to refine a sequence of L1 events. This argument though remains informal and comes down to adequacy of the chosen attacker model.

Our technique has several benefits when it comes to the physical code level. First, it does not pose any requirements on the structure of the program and thus can work with pre-existing code. Second, the security-related specification only appears in a few top-level functions; the rest of the code is only concerned with memory safety and functional correctness, which considerably improves scalability.

5 Empirical Results

In this section we summarize the results of our case study, present some statistics and share our observations.

In the case study we managed to verify a slightly simplified version of the draft reference implementation of two TPM 2.0 commands, Create and Load, against the two security goals described in Sect. 3. Even though our objective was thorough verification of the protocol code, as opposed to finding bugs, we still discovered two faults in the original implementation, one related to memory safety and another one to security.

The security fault resided in the code performing the integrity check before loading a key. In the faulty version, if the first two bytes of the decrypted private area were zero, the integrity check was skipped and the rest of the private buffer was treated as the sensitive area. In this case, in order to load an incorrect object the attacker has to guess a value for the private area, such that its first two bytes would decrypt to zeros and the rest would fit the format of a sensitive area (which is easier than matching the integrity value). This fault shows up during verification as a failure to prove a postcondition of the function performing the integrity check, which in turn is needed to establish the precondition of load_1 and eventually to update the log in a consistent way. Note that in this case the error lies in the implementation rather than in the protocol itself, and thus could not be discovered by a high-level protocol checking tool.

One of the reasons we had to simplify the code is that VCC had problems reasoning about deep syntactic nesting of C structs, which TPM uses to store parameters and attributes of objects. Our simplified implementation only works with storage keys and

supports just one encryption and one hashing algorithm, which eliminates the need to store most of the object parameters. As a consequence we also removed the code that checks consistency of those parameters or works explicitly with other types of objects.

File	Specs	Code	Ratio
ByteString.h	127		
CryptUtil.h	47	31	151%
TPM_Types.h	43	37	116%
marshal.h	94	32	293%
marshal.c	33	199	16%
create_load_0.c	384		
create_load_1.c	488		
create_load_2.c	282	205	137%
TOTAL	1498	504	297%

The table in this section gives code metrics for our case study. The annotation overhead for files containing both physical code and specification is usually around 150%, which is consistent with previous results for VCC. However, we also have two extra "implementations" of the protocol containing just ghost code, which brings the overall overhead to about 300%. The overhead in [8] is roughly 150%, but does not include the Coq proofs. Note, that our refinement models should not be understood as just overhead, as they convey useful information about the system in an easy to understand, operational way, where the hints for the verifier only comprise a fraction of the code.

Running VCC on the whole case study takes about 120 seconds (on a slightly dated 2.8GHz Core2 CPU using a single core), and only one function takes longer than 10 seconds to verify (precisely 38 seconds), whereas two more take between 3 and 10. They are all physical level functions, which involve multiple struct assignments, numerous function calls reading and writing entire nested structs, and complex postconditions. All other functions (including all the ghost-only functions) take less than 3 seconds to verify. It thus seems that handling of relatively complex C structs in VCC needs to be improved, whereas reasoning about pure mathematical data structures (even if they involve complex invariants) works well.

In terms of development time, specification and verification effort can be estimated as 4 person-months, including studying the TPM source code, developing the refinement approach and writing reusable specifications (e.g., the byte string library).

The major verification road-blocker, and source of frustration, was understanding failed verification attempts when working with physical code, especially large functions that mutate a lot of state. One reason is the lack of immediate feedback: when verifying those functions VCC would rarely produce a quick negative result, but rather keep on running for indefinite time. The Z3 Inspector [5] tool, monitoring progress of the backend proof engine, was invaluable in those cases. Another reason is that error reports are often related to internal properties of the VCC memory model and are obscure to a non-expert user, as compared with errors expressed in terms of user-defined specifications.

6 Related work

The introduction (Sect. 1) compares our work with previous work [8] on using VCC for security verification, which was in turn based on *invariants for cryptographic structures* [3].

There exist special-purpose tools for verifying security properties of C code, using abstract interpretation, like Csur [9], or model-checking techniques, as in ASPIER [4].

ASPIER only considers a limited number of protocol sessions, whereas Csur does not prove memory safety.

A number of tools use various static analysis and symbolic execution techniques to extract protocol description from code, or check conformance of code with a specific protocol. These tools are useful for finding bugs but their usability for sound verification is limited. In particular, various static analysis techniques tend to fail when confronted with slightly unusual or more complex codebases. On the other hand, in VCC proving correctness of complex code is a matter of putting in enough annotations, which is admittedly a difficult but manageable task.

Stepwise refinement has been used before to systematically construct cryptographic protocols from security goals [14]. Our approach complements this work, starting from a symbolic protocol definition (the final result of their technique) and refining it even further into an implementation.

There are other examples of encoding refinement techniques within a general-purpose program verifier [11], however they have mostly been applied to constructing relatively simple algorithms, rather than verifying pre-existing real-world software systems.

We believe that our approach could be implemented in any verification environment with expressive enough specification language. For the C programming language this includes Frama-C [12], VeriFast [10], and KeyC [13].

7 Conclusions

We proposed a novel approach to verifying implementations of security protocols based on stepwise refinement. To this end we developed an encoding of refinement in a general-purpose program verifier, which we believe can also be used in other problem domains.

Our refinement strategy for security protocols separates specification and verification into three levels of abstraction. Security goals can be expressed straightforwardly and concisely on the most abstract level. In general, all the specifications that have to be trusted (validated against the informal requirements) are explicit and relatively concise. They include security invariants, pre- and postconditions of events, gluing invariants between different levels, and symbolic assumptions. All other annotations are checked by the verifier, which makes our approach less error-prone.

The proposed technique is flexible and scalable enough to handle real pre-existing C code, which we confirmed by applying it to the draft reference implementation of TPM 2.0.

One direction of future work is extending the TPM case study to remove the code simplifications and include more TPM commands. The auxiliary invariants of the log would need to be extended to allow the additional command behavior, and the existing commands would need to be reverified against those invariants.

Another direction is applying the refinement approach to other security protocols. In a network-based protocol there is no shared physical state between the participants, however the ghost state can still be shared, which enables the use of our approach. In multi-message protocols honest agents and the attacker have to be executed concurrently. This does not affect the ghost code, as ghost events represent atomic actions of

sending and receiving single messages. A physical function that implements a protocol role, can call multiple ghost events and havoc the ghost state in between. Because of the flexibility of the general-purpose verifier, we believe that our approach can be naturally extended to handle other kinds of security properties and attacker models.

Acknowledgements We appreciate the collaboration of François Dupressoir, Paul England, Cédric Fournet, Andy Gordon and David Wooten on the TPM project. We are grateful to François Dupressoir, Carlo Furia and Andy Gordon, as well as the anonymous referees, for their comments on the draft versions of this paper.

References

1. J.-R. Abrial and S. Hallerstede. Refinement, decomposition, and instantiation of discrete models: Application to Event-B. *Fundam. Inform.*, 77(1-2):1–28, 2007.
2. 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 2005*, volume 4111 of *LNCS*, pages 364–387. Springer, 2005.
3. K. Bhargavan, C. Fournet, and A. D. Gordon. Modular verification of security protocol code by typing. POPL '10, pages 445–456, New York, NY, USA, 2010. ACM.
4. S. Chaki and A. Datta. Aspier: An automated framework for verifying security protocol implementations. In *Proceedings of the 2009 22nd IEEE Computer Security Foundations Symposium*, pages 172–185, Washington, DC, USA, 2009. IEEE Computer Society.
5. 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 2009*, volume 5674 of *LNCS*, pages 23–42. Springer, 2009.
6. L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS 2008*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
7. D. Dolev and A. Yao. On the security of public key protocols. *Information Theory, IEEE Transactions on*, 29(2):198–208, 1983.
8. F. Dupressoir, A. D. Gordon, J. Jürjens, and D. A. Naumann. Guiding a general-purpose C verifier to prove cryptographic protocols. In *IEEE Computer Security Foundations Symposium*, 2011.
9. J. Goubault-Larrecq and F. Parrennes. Cryptographic protocol analysis on real C code. In *VMCAI*, pages 363–379, 2005.
10. B. Jacobs and F. Piessens. The VeriFast program verifier. Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven, 2008.
11. K. R. M. Leino and K. Yessenov. Automated stepwise refinement of heap-manipulating code, 2010.
12. Y. Moy. *Automatic Modular Static Safety Checking for C Programs*. PhD thesis, Université Paris-Sud, Jan. 2009.
13. O. Mürk, D. Larsson, and R. Hähnle. KeY-C: A tool for verification of C programs. In *CADE*, volume 4603 of *LNCS*, pages 385–390. Springer, 2007.
14. C. Sprenger and D. A. Basin. Developing security protocols by refinement. In *ACM Conference on Computer and Communications Security*, pages 361–374, 2010.