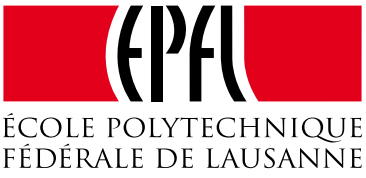
Improving the Coq proof automation tactics of the Verified Software Toolchain, based on a case study on verifying a C implementation of the AES encryption algorithm

A thesis presented for the degree of Master of Science in Computer Science
from École Polytechnique Fédérale de Lausanne

Thesis conducted from October 1, 2016 to March 31, 2017
at the Computer Science Department of Princeton University

Author:
Samuel Grütter
samuel.gruetter@epfl.ch

Princeton University
Department of Computer Science



School of Computer and
Communication Sciences IC

Supervisors:

Prof Andrew W. Appel
appel@princeton.edu
Department of Computer Science
Princeton University

Prof Viktor Kuncak
viktor.kuncak@epfl.ch
Laboratory for Automated
Reasoning and Analysis
École Polytechnique
Fédérale de Lausanne

Abstract

We present a (almost complete) machine-checked proof of functional correctness of a C implementation of the AES encryption algorithm. The proof is written in Coq, using the Verified Software Toolchain (VST), which connects to the CompCert verified C compiler. Since both of these come with a machine-checked correctness proof, we obtain an end-to-end proof linking the high-level specification of AES to the assembly code produced by CompCert.

Moreover, we report on the challenges encountered with VST's proof automation library, and identify and implement the improvements needed to write the AES proof, such as a generalization of the proof automation for memory loads and stores allowing accesses to nested fields inside arrays and/or structs whose access path is not explicitly written in the load or store statement, a more careful invocation of Coq's term simplification preventing unfeasibly slow proof script running times, and various usability improvements.

Contents

1 Introduction

The Verified Software Toolchain (VST) [5] is a Coq framework allowing one to prove that a C program conforms to a specification written in Coq. C programs are parsed into a Coq-defined abstract syntax tree data structure. Then, the VST user can define preconditions and postconditions for each function of the C program, and using the Hoare-style separation logic rules provided by VST, one can step through the parsed C code and prove that each execution satisfies the postcondition, if the precondition holds.

The first contribution of this work is a (almost complete) machine-checked VST proof of functional correctness of the AES encryption algorithm as implemented in the open-source library called mbedTLS.

We present a functional program written in Coq which closely follows the specification for the Advanced Encryption Standard [20], and we prove that the mbedTLS C implementation, which is heavily optimized and does not resemble the specification, produces the same output as the specification.

And since VST comes with a proof that it accepts no C programs with undefined behavior, a corollary of our proof is that the AES C program does not have any undefined behavior. This is an interesting result because often, undefined behavior can be exploited by attackers.

The other contributions are improvements to the proof automation infrastructure of the Verified Software Toolchain needed to write the AES proof. For instance, Figure 1b shows a snippet from the AES code containing a memory store where the "path" to the modified value is not completely contained in the store instruction, but distributed into two instructions: The first part of the path, `->rk`, is outside the loop, and the second part, `[i]`, is inside the loop.

Such memory accesses were originally not supported by the proof automation tactics of VST, and one had either to use the most low-level separation logic rule for memory stores, which would require several dozen lines of proof script code for just this single store instruction, or one had to rewrite the C code such that the whole path, `->rk[i]`, is contained in the store instruction, as in Figure 1c.

Overall, our contributions are the following:

- In Section 3, we present a low-level Coq specification of AES encryption closely following the mbedTLS implementation, and two machine-checked proofs: First, a proof using VST that the mbedTLS C implementation of AES conforms to the low-level Coq specification, and second, a proof that the low-level Coq specification is equivalent to a previously developed high-level specification in Coq [18] which closely follows the AES standard document [20].

- We present a generalization to VST's proof automation tactics for memory loads and stores in Section 4 and argue that, except for VST's known restrictions, there will be no further C code examples requiring a generalization of the proof automation for loads and stores.

- We discuss the challenges regarding term reduction that occur in large proofs about program behavior in Section 5, and show techniques to make sure terms are reduced enough so that they reach the form in which the desired lemmas can be applied, while ensuring at the same time that reduction is not too eager and creates terms of exponentially growing size.

```
typedef struct aes_context_struct {
    int nr; // number of rounds
    uint32_t rk[60]; // round keys
} aes_context;
aes_context *ctx;
```

(a) The `aes_context` type used in the subsequent examples

```
uint32_t *RK = ctx->rk;
for (int i = 0; i < 8; i++) {
    RK[i] = ...
}
```

(b) Code snippet which was not supported originally, but now is

```
for (int i = 0; i < 8; i++) {
    ctx->rk[i] = ...
}
```

(c) Equivalent code to which it had to be rewritten

Figure 1: Simplified code snippet from AES illustrating a shortcoming of the original load/store tactics

- We present various usability improvements to VST in Section 6.

Besides this, Section 2 gives some background on VST, Section 7 proposes directions for future work, Section 8 discusses related work, and Section 9 concludes.

2 VST preliminaries

This section presents the minimum VST preliminaries necessary to follow the subsequent sections about its proof automation. For a more detailed presentation, we refer to the VST book [4] and to the Verifiable C manual [3].

2.1 VST proof scripts

To prove that a C function conforms to a specification consisting of precondition *Pre* and postcondition *Post*, one has to prove the Hoare triple $\Delta \vdash \{Pre\}\ body\ \{Post\}$ using the separation logic rules of *Verifiable C* [4]. *body* is the parsed C code of the function's body, and Δ is a type context containing the types of function parameters, local and global variables, and specifications of global functions.

To do so, one performs forward symbolic execution, that is, step through the commands of *body* and for each command, apply the separation logic rule for that command, and continue with the postcondition of that rule as the new precondition.

Most such steps can be automated by Floyd's tactic called forward, which determines what rule should be applied, and solves all (or most) side conditions of that rule. If a side condition cannot be solved automatically, it is left as an additional open goal, and has to be proved manually.

There are three kinds of C commands which can only be stepped through automatically if the user provides some knowledge which is considered too hard to be inferred automatically: Loops, because they require a loop invariant, if-then-else statements, because they require a postcondition unifying the two branches, and function calls, because they require each parameter of the function specification to be instantiated. For those three cases, there are separate tactics called forward_for/forward_while, forward_if, and forward_call.

So, a proof in VST typically consists of many invocations of those tactics, and some manual proofs of nontrivial side-conditions.

2.2 Canonical form of VST Hoare triples

In order to make pre- and postconditions more readable for users, and more suitable for automation at the same time, they are represented in the canonical form $\mathrm{PROP}(\vec{P})\mathrm{LOCAL}(\vec{Q})\mathrm{SEP}(\vec{R})$, where $\vec{P}$ is a list of propositions of Coq's type Prop, $\vec{Q}$ is a list of local-variable definitions, and $\vec{R}$ is a list of separation-logic assertions about memory blocks. In this representation, the clauses in each of the three lists are separated by semicolons, but for the clauses in $\vec{P}$ and $\vec{Q}$, the semicolon actually stands for $\wedge$ (logical and), whereas in $\vec{R}$, it stands for $*$ (separating conjunction).

So, the canonical form of the proof goals that users (and tactics) generally see is

$$\Delta \vdash \{\mathrm{PROP}(\vec{P})\mathrm{LOCAL}(\vec{Q})\mathrm{SEP}(\vec{R})\}\, c\, \{\mathrm{PROP}(\vec{P'})\mathrm{LOCAL}(\vec{Q'})\mathrm{SEP}(\vec{R'})\}$$

where c is one or several C commands.[1] We will refer to the separation logic clauses in $\vec{R}$ as "SEP clauses".

[1] C's command-separating semicolon is an AST constructor in the Coq C representation, so a list of C commands can be represented as one single C command.

2.3 SEP clause predicates for structs and arrays

VST's basic separation logic rules are defined in terms of the mapsto operator. On paper, it's usually written as $v_1 \mapsto v_2$, and means that value v_1 is a pointer pointing to a memory location which stores value v_2. In VST, mapsto takes two additional arguments, a share sh indicating read/write permissions, and t, the type of v_2, so a mapsto predicate in VST has the form mapsto sh t v_1 v_2.

However, v_2 can only be a primitive value (i.e. a 32 or 64 bit integer or float, or a pointer, or uninitialized), so if we want to talk about structs and arrays, we have to give one clause for each primitive value of the struct or array. In pen-and-paper developments, this is usually written like $v_1.\mathit{first} \mapsto v_a * v_1.\mathit{second} \mapsto v_b$, or abbreviated into the notation $v_1 \mapsto (v_a, v_b)$. To make this more formal, one usually also adds the type of the value, so the notation becomes $v_1 \mapsto_\tau (v_a, v_b)$, where τ is a type.

This notation is called data_at in Coq, and the previous example would be written as data_at sh τ (v_a, v_b) v_1. To define data_at in Coq, we need to split the compound value into its primitive values, so we also need to define formally the notation $v_1.\mathit{first} \mapsto v_a$, where we have a "path" on the left-hand side of the maps-to operator. To do so, we make the following definitions:

A *general field* (written gfield in Coq) is an array index [i] or the name f of a struct field, and a *path* (written list gfield in Coq) is a list of general fields.

The field_at assertion is defined in Coq in terms of mapsto, and field_at sh t p v a means that the value a is a pointer pointing to an array or struct of type t with permissions sh, and that the path p exists in this array or struct, and "leads" to value v.

For instance, if ctx is a variable as defined in Figure 1a, then

field_at sh aes_context (DOT _rk SUB i) v ctx

means that the memory location denoted by the C expression `(*ctx).rk[i]` contains the value v.[2]

The Coq type of value v is a dependent type and depends on the type of the struct or array and on the path. This allows field_at to talk not only about primitive values, but also about the value of a whole struct or array at once. Struct values are represented using Coq pairs, and array values are represented using Coq lists.

So, to continue the example, the assertion field_at sh aes_context (DOT _rk) myList ctx can now talk about the whole contents of the round key array at once, saying that they equal the Coq-defined list myList.

Of course, we can also pass the empty path to field_at, in which case the provided value has to describe the whole structure stored at the given pointer. This special case of field_at is in fact data_at, i.e. we define data_at sh t v a := field_at sh t nil v a.

[2] DOT and SUB are Coq notations used to make lists of gfield more readable. DOT is used to select a field, and SUB is used to give an array index.

3 The AES encryption case study

The Advanced Encryption Standard (AES) is a symmetric encryption algorithm and is specified in the Federal Information Processing Standards (FIPS) Publication 197 [20]. It is written mostly in English, using some mathematical notations and some pseudo-code.

Lyubomirsky translated the FIPS publication into Coq [18], following it as closely as possible. We will refer to his specification as the *high-level specification*.

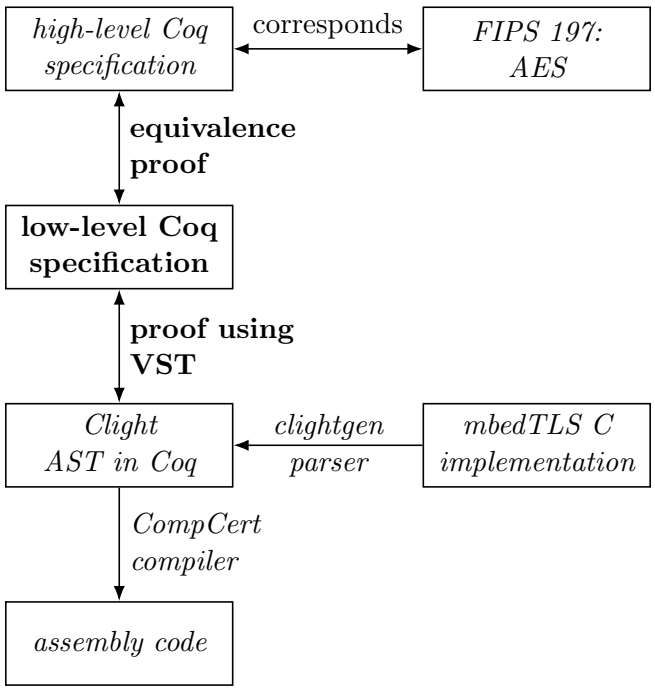


Figure 2: Overview of specifications and proofs. Items in bold font were developed in this project, items in italics are previous work by other authors.

The mbedTLS library implements AES encryption, but does not directly follow the FIPS specification, because this would not be efficient enough. While the FIPS specification defines AES in terms of four kinds of operations on 4×4 matrices over the Galois Field GF(2^8), the mbedTLS implementation precalculates many values before doing encryption, merges the four kinds of operations into one, and only performs table lookups and bitwise XOR operations on 32-bit integers during encryption.

Therefore, it makes sense to split the proof that the mbedTLS implementation conforms to the high-level specification into two parts: First, we prove, using VST, that it conforms to a *low-level specification* written in Coq, which follows the mbedTLS implementation as closely as possible, and second, we prove that the low-level specification is equivalent to the high-level specification.

Now, since VST is compatible with the CompCert verified C compiler [17], we get a proof of the following end-to-end statement:[3]

[3]There are still several trusted parts: The Calculus of Inductive Constructions, on which Coq is based, the axioms of dependent functional extensionality and propositional extensionality, the Coq kernel, the OCaml compiler, because it is used to compile the Coq kernel, as well as the OCaml runtime, the high-level specification, the specification of CompCert (but not its specification of C), the assembler producing machine code from assembly code, as well as the hardware. Section 12 of Appel's SHA-256 article [2] discusses the trusted base in more detail.

If we run the verified AES program, compiled with CompCert, on hardware conforming to CompCert's specification of assembly language, its output in memory is the same as specified by the high-level specification in Coq.

Figure 2 summarizes the components described above.

3.1 The C Code

The mbedTLS implementation contains three functions relevant for AES encryption, shown in Figure 3.[4]

```
static void aes_gen_tables( void );
int mbedtls_aes_setkey_enc( mbedtls_aes_context *ctx,
                            const unsigned char *key,
                            unsigned int keybits );
void mbedtls_aes_encrypt( mbedtls_aes_context *ctx,
                          const unsigned char input[16],
                          unsigned char output[16] );
```

Figure 3: Signatures of the AES encryption functions

The type `mbedtls_aes_context` is (basically) the same as shown in Figure 1a and is used to store the key and the number of rounds.

The table generation function The mbedTLS implementation uses so-called forwarding tables storing key-independent precalculated values to speed up encryption. The function `aes_gen_tables` calculates these tables, and is invoked when the key expansion function is called the first time.

The key expansion function Before one can encrypt with AES, one has to set the key using `mbedtls_aes_setkey_enc`. The argument `keybits` has to be one of the supported key bit lengths of AES, i.e. 128, 192 or 256. In this project, we only treat AES-256, so the argument `key` has to contain a 32-byte (= 256-bit) key. Since each of the 14 encryption rounds needs its own 16 byte key, `mbedtls_aes_setkey_enc` has to perform *key expansion* to derive more round keys from the original key, and it stores them into the `ctx` struct.

The encryption function The `mbedtls_aes_encrypt` function takes a context with an expanded key and a 16 byte input, and encrypts it to produce a 16 byte output. It only performs table lookups and bitwise XOR operations, all the other operations were precalculated by `aes_gen_tables`.

3.2 The high-level specification

The high-level specification was developed by Lyubomirsky and is described in detail in his report [18].

[4]The full C code can be found at https://github.com/ARMmbed/mbedtls/blob/mbedtls-2.3.0/library/aes.c, and the changes needed to make it conform to the subset of C supported by VST can be found at https://github.com/PrincetonUniversity/VST/commits/master/aes/mbedtls/library/aes.c.

When extracting the Coq code to OCaml and running it with some sample inputs, we discovered and corrected a small bug in Lyubomirksy's specification of key expansion: Instead of doing an S-box substitution followed by a bitwise XOR with the round constant, these two operations were swapped.[5]

```
i = Nk
while (i < Nb * (Nr+1)]
   temp = w[i-1]
   if (i mod Nk = 0)
      temp = SubWord(RotWord(temp))
         xor Rcon[i/Nk]
   else if (Nk > 6 and i mod Nk = 4)
      temp = SubWord(temp)
   end if
   w[i] = w[i-Nk] xor temp
   i = i + 1
end while
```

(a) Pseudocode from FIPS 197

```
Definition GrowKeyByOne(w: list int): list int :=
  let i := Zlength w in
  let temp := (Znth (i-1) w Int.zero) in
  let temp' := if (i mod Nk =? 0) then
    Int.xor (SubWord (RotWord temp))
      (Znth (i/Nk) RCon Int.zero)
  else if (i mod Nk =? 4) then
    SubWord temp
  else temp in
    w ++ [Int.xor (Znth (i-Nk) w Int.zero) temp']
.
```

(b) Coq code for one loop iteration

```
Fixpoint pow_fun{T: Type}(f: T → T)(n: nat)(a: T): T := match n with
| O ⇒ a
| S m ⇒ f (pow_fun f m a)
end.
Definition KeyExpansion: list int → list int := pow_fun GrowKeyByOne (Z.to_nat (Nb*(Nr+1)-Nk)).
```

(c) Coq code needed to define the iteration

Figure 4: Key expansion code in the FIPS standard and in Coq

The second change to his specification is related to the problem that Coq's Gallina language is functional, whereas the specification of key expansion in FIPS 197 is done in an imperative way, as one can see in Figure 4a. Lyubomirsky's translation into Coq did not look similar to the code in Figure 4a any more, and was 42 lines long.[6]

We improve on this by defining a function GrowKeyByOne (see Figure 4b), which takes a partially expanded key and appends to it the new key entry calculated by one iteration of the loop in Figure 4a. To turn this step function into the key expansion function, we need to define the power function (repeated application of a function), and apply it to GrowKeyByOne, as shown in Figure 4c.

While there is a significant difference between the FIPS specification and the Coq specification regarding how the iteration is specified, we claim that at least between Figure 4a and 4b, there is a clearly recognizable correspondence between the imperative loop body and the functional GrowKeyByOne implementation.[7]

The complete high-level specification of AES is 229 lines long (without decryption),[8] so we will not print it here, but its core, the definition of one round of encryption, is

[5]The fix is trivial can be found at https://github.com/PrincetonUniversity/VST/commit/0ac40d29.

[6]https://github.com/PrincetonUniversity/VST/blob/eaf41469/aes/AES256.v#L238-L279.

[7]The AES encryption algorithm is parameterized by the number of cipher rounds Nr, the number of 32 bit words in the key Nk, and the number of 32 bit words in a block (state) Nb, which take different values depending on the key length. In this project, we only focus on the 256 bit key size, which implies that Nr = 14, Nk = 8 and Nb = 4. Therefore, Nk > 6 always holds and we removed it from the Coq specification.

[8]https://github.com/PrincetonUniversity/VST/blob/0ac40d29/aes/spec_AES256_HL.v#L7-L235

```
Definition round (s : state) (kb: block) : state :=
  AddRoundKey (MixColumns (ShiftRows (SubBytes s))) kb.
```

Figure 5: High-level specification of one round of AES encryption

```
Definition mbed_tls_fround_col (col0 col1 col2 col3 : int) (rk : Z) : int :=
  (Int.xor (Int.xor (Int.xor (Int.xor (Int.repr rk)
    (Znth (byte0 col0) FT0 Int.zero))
    (Znth (byte1 col1) FT1 Int.zero))
    (Znth (byte2 col2) FT2 Int.zero))
    (Znth (byte3 col3) FT3 Int.zero)).
Definition mbed_tls_fround (cols : four_ints) (rks : list Z) (i : Z) : four_ints :=
match cols with (col0, (col1, (col2, col3))) ⇒
  ((mbed_tls_fround_col col0 col1 col2 col3 (Znth i rks 0)),
  ((mbed_tls_fround_col col1 col2 col3 col0 (Znth (i+1) rks 0)),
  ((mbed_tls_fround_col col2 col3 col0 col1 (Znth (i+2) rks 0)),
   (mbed_tls_fround_col col3 col0 col1 col2 (Znth (i+3) rks 0)))))
end.
```

Figure 6: Low-level specification of one round of AES encryption

very concise, as one can see in Figure 5.

3.3 The low-level specification

The low-level specification consists of two parts: The first part is a functional program written in Coq following the C code as closely as possible. This part does not depend on any VST definitions, except that it uses the same 32 bit integer library as VST, namely CompCert's, and a list library which is part of VST. The second part is an API specification using VST's definitions to state the pre- and postconditions for each C function.[9]

The functional low-level specification follows the optimized C code as closely as possible, and therefore looks quite different from the high-level specification. For comparison, we show one round of encryption in the low-level specification in Figure 6. As one can see, it only performs bitwise XOR operations and lookups in the forward tables FT0, FT1, FT2, FT3, which were pre-calculated by the table generation function.

The functional low-level specification was developed by stepping through the C code with VST and copy-pasting the expressions that VST calculated from the C code. However, there was still a considerable manual effort needed to write the low-level specification, because to avoid unreasonably big expressions with code repetition, appropriate definitions had to be made and used, and loop invariants, which were sometimes even more complex than the low-level specification, had to be found manually.

[9] Lyubomirsky had developed partially finished definitions for both parts, but since he did not write any VST proofs, they did not really match the C code and had to be rewritten essentially from scratch.

```
Definition encryption_spec_low_level :=
  DECLARE _mbedtls_aes_encrypt
  WITH ctx : val, input : val, output : val, (* arguments *)
       ctx_sh : share, in_sh : share, out_sh : share, (* shares *)
       plaintext : list Z, (* 16 chars *)
       exp_key : list Z, (* expanded key, 4*(Nr+1)=60 32-bit integers *)
       tables : val (* global var *)
  PRE [ _ctx OF (tptr t_struct_aesctx), _input OF (tptr tuchar), _output OF (tptr tuchar) ]
    PROP(Zlength plaintext = 16; Zlength exp_key = 60;
         readable_share ctx_sh; readable_share in_sh; writable_share out_sh)
    LOCAL(temp _ctx ctx; temp _input input; temp _output output; gvar _tables tables)
    SEP(data_at ctx_sh (t_struct_aesctx) (
          (Vint (Int.repr Nr)),
          ((field_address t_struct_aesctx [StructField _buf] ctx),
          (map Vint (map Int.repr (exp_key ++ (list_repeat (8%nat) 0)))))
        ) ctx;
        data_at in_sh (tarray tuchar 16) (map Vint (map Int.repr plaintext)) input;
        data_at_ out_sh (tarray tuchar 16) output;
        tables_initialized tables)
  POST [ tvoid ]
    PROP() LOCAL()
    SEP(data_at ctx_sh (t_struct_aesctx) (
          (Vint (Int.repr Nr)),
          ((field_address t_struct_aesctx [StructField _buf] ctx),
          (map Vint (map Int.repr (exp_key ++ (list_repeat (8%nat) 0)))))
        ) ctx;
        data_at in_sh (tarray tuchar 16) (map Vint (map Int.repr plaintext)) input;
        data_at out_sh (tarray tuchar 16)
          (map Vint (mbed_tls_aes_enc plaintext (exp_key ++ (list_repeat (8%nat) 0)))) output;
        tables_initialized tables).
```

Figure 7: The low-level API specification of the AES encryption function

The API specification for the encryption function is given in Figure 7, while we refer to the GitHub repo for the API specifications of the other two functions.[10]

The definition in Figure 7 specifies a pre- and postcondition for the C function named `mbedtls_aes_encrypt`, whose C signature was given in Figure 3. It uses VST's notation

$$\mathsf{DECLARE}\ f\ \mathsf{WITH}\ \vec{w}\ \mathsf{PRE}\ [\ \vec{a}\ ]\ \phi_{pre}\ \mathsf{POST}\ [\ r\ ]\ \phi_{post}$$

where f is the name of the C function, $\vec{w}$ is a list of universally quantified variables on which both the pre- and postcondition may depend (which allows to say how values before execution of the function relate to values after execution of the function), $\vec{a}$ gives the types of the function arguments, r is the return type of the function, ϕ_{pre} is the precondition and ϕ_{post} is the postcondition. Both ϕ_{pre} and ϕ_{post} are in the $\mathrm{PROP}(\vec{P})\mathrm{LOCAL}(\vec{Q})\mathrm{SEP}(\vec{R})$ canonical form explained in section 2.2.

In the precondition, the PROP clause specifies the length of the plaintext and the expanded key, and requires that the shares for the inputs grant read permission, and that the share for the output grants write permission. The LOCAL clause links function arguments and global variables to Coq values listed in the WITH clause, and the SEP clause uses data_at (see section 2.3) to give separation logic formulae describing the requirements on the heap. The first data_at clause describes the context data structure, and besides some details, the important statement is that it stores the expanded key exp_key. The second data_at clause says that the input contains the plaintext, and the third data_at_ clause specifies the memory area where write access for the output is required. The trailing underscore in data_at_ means that the values of this memory area may be undefined before the function is invoked.

In the postcondition, the PROP clause is empty, because it could only talk about Coq values given in the WITH clause, but these are immutable, so everything which was true about them before the invocation of the function is still true after the invocation, even if we do not say it. The LOCAL clause is also empty, because function arguments are not visible after the invocation, and all global variables are immutable pointers, so nothing could change there. The first two data_at clauses inside the SEP clause are the same as in the precondition, because we expect the encryption function to leave these unmodified. The third data_at clause, however, changed, and contains the most important part of this whole specification: It says that the memory area pointed to by output now stores the result of applying the low-level Coq specification function named mbed_tls_aes_enc to the arguments plaintext and exp_key.

3.4 The VST proof

For each of the three considered functions, i.e. table generation, key expansion, and encryption, we prove that it conforms to its API specification.[11]

The proof statement for the encryption function looks as follows (the other two statemtents are similar):

Theorem body_aes_encrypt:
semax_body Vprog Gprog f_mbedtls_aes_encrypt encryption_spec_low_level.

[10] https://github.com/PrincetonUniversity/VST/blob/master/aes/verif_setkey_enc_LL.v
and https://github.com/PrincetonUniversity/VST/blob/master/aes/verif_gen_tables_LL.v

[11] The Coq proofs can be found at https://github.com/PrincetonUniversity/VST/tree/master/aes, in the files verif_gen_tables_LL.v, verif_setkey_enc_LL.v, and verif_encryption_LL.v, respectively.

Here, Vprog is the list of global variables of the program, Gprog is the list of all functions together with their specification, f_mbedtls_aes_encrypt is the parsed abstract syntax tree of the C function, and encryption_spec_low_level is the API specification given in Figure 7. And the meaning of semax_body is that if we derive an environment Δ from Vprog and Gprog, and call the precondition in Figure 7 ϕ_{pre} and the postcondition ϕ_{post}, and *body* is the body of f_mbedtls_aes_encrypt, then the following Hoare triple holds:

$$\Delta \vdash \{\phi_{pre}\}\ body\ \{\phi_{post}\}$$

To summarize the proofs, they consist of the following kinds of proof code:

- Calls to the VST tactics forward, forward_for, and forward_if. forward requires no argument, whereas forward_for requires a loop invariant, and forward_if requires a postcondition, and specifying these can result in lengthy code.
- Often, the above tactics cannot solve all separation logic entailments automatically. For these cases, some manual proving is needed.
  For instance, to keep the proof search space reasonably sized, the automatic entailment solver does not unfold data_at assertions which talk about the contents of a whole struct into one assertion per field, but this is necessary to prove an entailment if the data_at assertion was unfolded on one side of the turnstile, but not on the other side.
- Sometimes, it is useful to make the current proof goal "nicer" or more suitable for automation before invoking the next tactic.
  For instance, while an array is being initialized, it is best represented in Coq as a list val, because val can be either uninitialized or hold a value. But once the array is initialized, it's better to represent it in Coq as a list int, which makes it clear that the list cannot contain any undefined values. The decision to perform such a representation change has to be taken by the user.
- Code needed to replace the expressions inferred by VST from the C code with definitions of the low-level functional specification.
  For instance, in the proof of the encryption function, the value of a variable calculated by VST sometimes looks like the right-hand side of the definition of mbed_tls_fround_col (Figure 6), and has to be replaced by an invocation of mbed_tls_fround_col to keep the proof state at a reasonable size.

During the development of these proofs, many usability issues of the Floyd tactics library were found. For each of them, either a work-around in the proof script was found, and it was marked with a "TODO floyd" note, so that future contributors can search for this string and find examples where the Floyd tactics library could be improved, or the issue was addressed and solved as a part of this project. These improvements are described in sections 4, 5, and 6.

3.5 The low-level/high-level equivalence proof

To establish equivalence between the low-level and high-level specifications, we prove[12] the lemma stated in Figure 8.

Notice that the high-level encryption function, called Cipher, has a different input and output format than the low-level encryption function mbed_tls_aes_enc: To represent

[12] https://github.com/PrincetonUniversity/VST/blob/master/aes/equiv_encryption.v

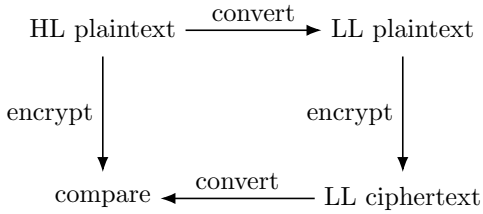
```
Lemma HL_equiv_LL_encryption: ∀ exp_key plaintext,
  Zlength exp_key = 15 →
  (mbed_tls_aes_enc
     (map Int.unsigned (state_to_list plaintext))
     ((blocks_to_Zwords exp_key) ++ (list_repeat (8%nat) 0))
  ) = output_four_ints_as_bytes (state_to_four_ints (Cipher exp_key plaintext)).
```

Figure 8: The low-level/high-level equivalence proof statement

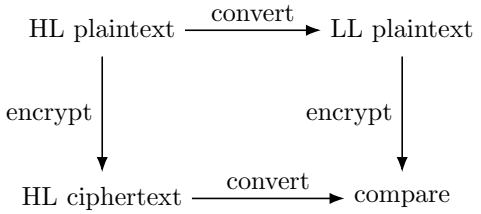
128-bit encryption states, the former uses 4×4 matrices of 8-bit integers, whereas the latter uses quadruples of 32-bit integers. Therefore, if the lemma quantifies over input data in high-level format, it can be directly fed to the high-level encryption function Cipher, but needs conversion before being passed to the low-level mbed_tls_aes_enc. And to compare the outputs in the low-level format, we can directly use the output of mbed_tls_aes_enc, but we have to convert the output of Cipher from the high-level format into the low-level format.

The proof consists mostly of unfolding the definitions on both sides of the equality, rewriting expressions using associativity, commutativity and distributivity rules, until the two sides of the equality match.

However, two difficulties were encountered while working on the proof:



(a) First approach: Starting with a plaintext in high-level format, and doing the comparison in the high-level format



(b) Simpler approach: Still starting with a plaintext in high-level format, but doing the comparison in the low-level format

Figure 9: Two ways of stating the equivalence proof between the high-level (HL) and low-level (LL) specification

The first difficulty was that originally, we used a different proof statement: Instead of comparing the output in the low-level format, we compared it in the high-level format. The difference between these two approaches is illustrated by Figure 9. This approach turned out to be much more complicated, because we not only need conversion functions from the high-level format to the low-level format, but also the inverse, as well as lemmas stating that these two functions are inverses of each other, and we have to apply these lemmas in appropriate places. Therefore, approach (a) was given up, and the proof was finished in approach (b). Note, though, that the proof statement in (b) is slightly weaker than in (a) in the sense that we have to trust that the two conversion functions are correct (i.e., they are part of the proof statement). To illustrate this with an extreme example, if the two conversion functions in approach (b) were just constant functions, and the low-level encryption was just the identity function, the proof would still work. In approach (a), however, it would not work, because the high-level encryption function is invertible, so each of the three functions on the path claimed to be equivalent, i.e. plaintext conversion, low-level encryption, and ciphertext conversion,

have to be invertible as well.

The second difficulty was that at some point in the proof, the two sides of the equality did not match, in a way which suggested that the statement we were trying to prove was false. To debug this problem, the QuickChick [23] tool proved to be very useful: Given a computable property, it evaluates the property with a large number of randomly generated inputs, and tells the user if the property holds for all tested inputs, or reports a counterexample otherwise. Using QuickChick, the problem could be tracked down to an admitted lemma which looked trivially true and seemed to be just tedious bit fiddling, but turned out to be false because of confusion about endian-ness.

3.6 Status of the proofs

During proof development, the Coq proof assistant allows users to pose axioms, or to assume an unfinished proof holds by ending it with the keyword Admitted. This allows one to focus on the interesting parts of the proofs first, and prove the less interesting lemmas later.

At the time of writing, some required lemmas were admitted, i.e. are not yet proved, and this section reports on what exactly is left to do.

First, we emphasize that all the VST-related parts are finished. That is, the admitted lemmas do not depend on any VST definitions, except for the floyd.sublist library (a generic list library shipped with VST), CompCert's integer type int, and CompCert's type for values val.

Library-like general lemmas Most unproved lemmas are general facts about bounded integers, the Galois Field GF(2^8), and lists. They were factored out into the following files:[13]

- bitfiddling.v: Contains 21 admitted lemmas about bit operations, such as packing 4 bytes into a 32-bit integer, etc.
- equiv_GF_ops.v: The equivalence between the low-level implementation of operations on the Galois Field GF(2^8) and the high-level specification has not yet been proved.
- GF_ops_LL.v: Contains 8 admitted lemmas about properties of the low-level implementation of operations on the Galois Field GF(2^8).
- list_lemmas.v: Contains one admitted lemma on lists.
- partially_filled.v: Contains two admitted lemmas on updating partially initialized lists.

The 3 verified functions (table generation, key expansion, encryption) The verification of the tables generation function (verif_gen_tables_LL.v) is complete, and can be checked with Coq.

The verification of the key expansion function (verif_setkey_enc_LL.v) is almost complete, because it contains 3 admitted lemmas about partially expanded keys, and one axiom (which actually does not hold) needed to resolve a specification mismatch regarding whether an array is uninitialized or initialized to zero.

The verification of the encryption function (verif_encryption_LL.v) contains no admits.

[13] All files can be found at https://github.com/PrincetonUniversity/VST/tree/b38098e0ef/aes

However, when checking the proof scripts for the key expansion function and the encryption function, Coq can check the proof just until before the Qed, that is, up to the point where no subgoals are left. However, to finish the proof, Coq's Qed command retypechecks the whole generated proof term, and since the proof term is quite large, this process does not finish within an hour. This problem was encountered previously in other proofs in VST, and can be solved by splitting the proof into several parts. For the AES proofs, this splitting should work as well, but has not yet been done due to time constraints.

Equivalence proofs The equivalence proof between the low-level and high-level encryption function (equiv_encryption.v) depends on the fact that the value stored in an unsigned byte is at most 256, and these hypotheses are not yet pushed carefully enough through all parts of the proof, so we have to admit this inequality in a few places.

For the key expansion, no low-level specification has been written yet, so there's no equivalence proof either, and while we do successfully step through the whole C code with VST, we assume that the terms inferred by VST match the high-level specification.

Note that for table generation, no equivalence proof is needed, because the high-level specification does not need any tables.

4 Generalizing memory load/store tactics

Because of examples like the one shown in Figure 1, some improvements to the proof automation for memory loads and stores were needed to make the AES proof work. This section first precisely defines what kind of memory loads and stores are supported by VST (4.1), then explains why VST needs proof automation at all (4.2), describes VST's original rules and tactics for memory loads and stores (4.3 and 4.4), and then explains the shortcomings detected during the AES case study and how they were addressed in the remainder of the section.

4.1 VST's restrictions on memory loads/stores

To define which loads and stores are supported, we need the following definitions:

We use *primitive type* to denote all integer types, floating point types and pointer types. A *nonaddressable* local variable is a local variable of primitive type whose address is never queried in the program. A *primary l-value expression* refers to an address in memory where computation of the address does not involve any memory dereferences or function calls. A *primary r-value expression* is an expression not containing any memory dereferences or function calls.

Now, a *load command* in VST has to be an assignment where the left hand side is a nonaddressable local variable and the right hand side is a primary l-value expression, and a *store command* has to be an assignment where the left hand side is a primary l-value expression and right hand side is a primary r-value expression.

Note that these restrictions do not limit the expressive power of C, because every C program can be rewritten to one supported by VST through some local rewriting factoring out subexpressions and assigning them to nonaddressable local variables.

4.2 The need for proof automation

In pen-and-paper presentations of separation logic, the Hoare rules are relatively simple. For instance, the rule for memory loads looks as follows in the original presentation by O'Hearn et al. [21]:

$$\{x = v_1 \wedge E \mapsto v_2\}\ x := [E]\ \{x = v_2 \wedge E[v_1/x] \mapsto v_2\}$$

It says that if the local variable x stores the value v_1, and the (pure) expression E points to a memory location containing value v_2, running the command which dereferences E and assigns it to x causes x to store v_2, and requires us to update the proposition $E \mapsto v_2$ if x occurs in E.

While this rule looks simple on paper, its corresponding rule in the C separation logic module of VST needs four additional side conditions.[14] Moreover, in order to be able to reason about C structs and arrays (and any nesting thereof), VST defines and proves several rules derived from the basic load rule. This results in a final, most high-level rule for memory loads with about a dozen side conditions.

Most of these side conditions are trivial to prove and should not be exposed to the VST user. Therefore, VST provides a tactics library called *Floyd*, written in Ltac, which can solve most of these side conditions automatically.

[14] These side conditions are needed because of some typechecking conditions, to deal with read/write permissions, and because in Coq, we have to be more explicit about converting C expressions into expressions which can occur in propositions.

4.3 Hoare rules for memory loads and stores

VST's basic separation logic rule to reason about memory loads can be stated as follows:[15]

$$\textsc{Load-0}\ \frac{e \text{ evaluates to } a \qquad \mathsf{sh} \text{ grants read permission} \qquad P \vdash \mathsf{mapsto\ sh\ t}\ a\ v_2}{\Delta \vdash \{P \wedge \text{local var } x = v_1\}\, x = e\, \{[v_1/x]P \wedge \text{local var } x = v_2\}}$$

The rule for stores looks as follows:[16]

$$\textsc{Store-0}\ \frac{e_1 \text{ evaluates to } a \qquad e_2 \text{ evaluates to } v_{new} \qquad \mathsf{sh} \text{ grants write permission}}{\Delta \vdash \{P * \mathsf{mapsto\ sh\ t}\ a\ v_{old}\}\, e_1 = e_2\, \{P * \mathsf{mapsto\ sh\ t}\ a\ v_{new}\}}$$

Both rules are defined in terms of the mapsto operator (see section 2.3), which only talks about primitive values.

In order to reason about structs and arrays, VST defines and proves the following rule, which uses field_at (see section 2.3) instead of mapsto:[17]

$$\textsc{Load-1}\ \frac{\begin{array}{c} e \text{ evaluates to } a \qquad \mathsf{cpath} \text{ evaluates to path } p \qquad p \text{ can be split into } p_0.p_1 \\ \vec{R}_i = \mathsf{field\_at\ sh\ t}\ p_0\ v'\ a \qquad \mathsf{sh} \text{ grants read permission} \qquad v'.p_1 = v \\ \vec{Q}' = (\vec{Q} \text{ with the value for } x \text{ updated to } v) \end{array}}{\Delta \vdash \{\mathrm{PROP}(\vec{P})\mathrm{LOCAL}(\vec{Q})\mathrm{SEP}(\vec{R})\}\, x = e.\mathsf{cpath}\, \{\mathrm{PROP}(\vec{P})\mathrm{LOCAL}(\vec{Q}')\mathrm{SEP}(\vec{R})\}}$$

Since field_at is defined as a separating conjunction of several mapsto assertions, Load-1 can be proven by unfolding field_at and applying Load-0. Note that contrary to Load-0, there's no need for a substitution in the propositions of the postcondition, because $\vec{Q}$ is updated to $\vec{Q}'$ above the line, and because the canonical form prevents $\vec{P}$ and $\vec{R}$ from referring to local variables.

And in a similar way, Store-0 is turned into Store-1 to reason about arrays and structs:[18]

$$\textsc{Store-1}\ \frac{\begin{array}{c} e_1 \text{ evaluates to } a \qquad e_2 \text{ evaluates to } v \\ \mathsf{cpath} \text{ evaluates to path } p \qquad p \text{ can be split into } p_0.p_1 \\ \vec{R}_i = \mathsf{field\_at\ sh\ t}\ p_0\ v_{old}\ a \qquad \mathsf{sh} \text{ grants write permission} \\ v_{new} = v_{old} \text{ with the substructure denoted by } p_1 \text{ updated to } v \\ \vec{R}' = (\vec{R} \text{ with } R_i \text{ replaced with } \mathsf{field\_at\ sh\ t}\ p_0\ v_{new}\ a) \end{array}}{\Delta \vdash \{\mathrm{PROP}(\vec{P})\mathrm{LOCAL}(\vec{Q})\mathrm{SEP}(\vec{R})\}\, e_1.\mathsf{cpath} = e_2\, \{\mathrm{PROP}(\vec{P})\mathrm{LOCAL}(\vec{Q})\mathrm{SEP}(\vec{R}')\}}$$

4.4 The tactics

To do forward symbolic execution of a memory load or store statement, the VST user simply calls the forward tactic, which calls the tactic for memory loads or stores, named load_tac and store_tac, respectively, and in most cases, these tactics can apply the above rule and prove all side conditions fully automatically.

We will use the example given in Figure 1c to explain how store_tac works, and we will not describe load_tac because it works very similarly.

To give the ellipsis (of Figure 1c) a name, suppose the command is `ctx->rk[i] = expr`, where `expr` is a primary r-value. Moreover, suppose that $\vec{R}$ contains the SEP clause

[15] This is a slightly simplified presentation omitting some details for better readability. The full precise rule in Coq is called semax_load and can be found in the file https://github.com/PrincetonUniversity/VST/blob/fa30d27a/veric/SeparationLogic.v, or in the PLCC book [4] on page 161.

[16] Called semax_store in Coq, in the same file.

[17] Called semax_SC_field_load in Coq, in the file https://github.com/PrincetonUniversity/VST/blob/fa30d27a/floyd/sc_set_load_store.v

[18] Called semax_SC_field_store in Coq, same file.

field_at sh aes_context (DOT _rk) keyWords ctx

where sh is a share granting write permission and keyWords is a Coq list representing the values currently stored in the round key array of the AES context struct.

Then, store_tac does the following steps to determine the values of the parameters of the rule STORE-1: First, it splits the left-hand side of the assignment into a root expression e_1 (the C variable ctx in our example) and a cpath ("`.rk[i]`"). Next, it evaluates e_1, e_2 and cpath by looking up symbolic values in the variable list $\vec{Q}$. In our example, e_1 simply evaluates to the pointer ctx, and cpath evaluates to $p = $ (DOT _rk SUB i). Next, it loops through all SEP clauses to find one about ctx whose path is a prefix of p. Note that this must be unique, because the SEP clauses talk about disjoint memory areas, and no two distinct memory areas can be accessed by the same path. Once the SEP clause is found, it's clear how to split p into $p_0.p_1$; in our case we have $p_0 = $ (DOT _rk) and $p_1 = $ (SUB i). The SEP clause also gives the value for v_{old}, which is keyWords in our case, and store_tac now has to select the substructure of v_{old} according to the remainder of the path, p_1, to obtain the part of v_{old} to be updated. Now, all parameters needed to apply STORE-1 are known, and the generated subgoals are (usually) easy to solve automatically, or if some are not, those subgoals are left open and have to be proven by the user.

4.5 A shortcoming detected in the AES case study

During the case study on verifying AES encryption (section 3), we found examples where load_tac and store_tac as described before did not work. One of them is shown in Figure 1b.

Now, assuming that we have the same field_at assertion as in 4.4, store_tac will fail on the statement `RK[i] = ...`, because the root expression e_1 that it picks is `RK`, which evaluates to ctx+ofs, where ofs is the offset of the field rk within the struct aes_context. So, store_tac will look for a SEP clause of the form field_at ? ? ? ? (ctx+ofs), which it will not find, because we only have a SEP clause of the form field_at ? ? ? ? ctx.[19]

As we can see, the problem is that the LOAD-1 and STORE-1 rules assume that the whole access path appears in the source code of the load or store instruction, which is not always the case.

4.6 More general Hoare rules and tactics

Therefore, we have to come up with a more general Hoare rule which does not require that the path be visible in the syntax:[20]

$$\text{LOAD-2}\ \frac{\begin{array}{c} e \text{ evaluates to } a.p \qquad p \text{ can be split into } p_0.p_1 \\ \vec{R}_i = \mathsf{field\_at\ sh\ t}\ p_0\ v'\ a \qquad \mathsf{sh} \text{ grants read permission} \qquad v'.p_1 = v \\ \vec{Q}' = (\vec{Q} \text{ with the value for } x \text{ updated to } v) \end{array}}{\Delta \vdash \{\mathrm{PROP}(\vec{P})\mathrm{LOCAL}(\vec{Q})\mathrm{SEP}(\vec{R})\}\, x = e\, \{\mathrm{PROP}(\vec{P})\mathrm{LOCAL}(\vec{Q}')\mathrm{SEP}(\vec{R})\}}$$

[19] We use the question mark to denote a wildcard.

[20] Called semax_SC_field_load' in Coq, in the file https://github.com/PrincetonUniversity/VST/blob/fa30d27a/floyd/sc_set_load_store.v

And similarly for stores:[21]

$$\textsc{Store-2}\ \frac{\begin{array}{c}e_1 \text{ evaluates to } a.p \qquad e_2 \text{ evaluates to } v \qquad \mathsf{sh} \text{ grants write permission} \\ p \text{ can be split into } p_0.p_1 \qquad \vec{R}_i = \mathsf{field\_at\ sh\ t}\ p_0\ v_{old}\ a \\ v_{new} = v_{old} \text{ with the substructure denoted by } p_1 \text{ updated to } v \\ \vec{R}' = (\vec{R} \text{ with } R_i \text{ replaced with } \mathsf{field\_at\ sh\ t}\ p_0\ v_{new}\ a)\end{array}}{\Delta \vdash \{\mathrm{PROP}(\vec{P})\mathrm{LOCAL}(\vec{Q})\mathrm{SEP}(\vec{R})\}\, e_1 = e_2\, \{\mathrm{PROP}(\vec{P})\mathrm{LOCAL}(\vec{Q})\mathrm{SEP}(\vec{R}')\}}$$

We can rewrite load_tac and store_tac to use these rule instead of the previous ones, but that makes them more brittle in one point. This is best illustrated with the AES example: When evaluating e_1, that is `RK[i]`, we might obtain a pointer of the form ctx + ofs + i, where ofs is the offset of the field rk within the aes_context struct. But the above rule expects that e evaluates to something of the form $a.p$ (where p is a path), so we have to prove that ctx + ofs + i equals ctx.p, where $p = (\mathsf{DOT\ \_rk}, \mathsf{SUB\ i})$. This might seem trivial, but so far, we have not yet been able to automate all proofs of this kind, because field addresses need to satisfy many well-formedness conditions such as field names conforming to struct types, array indices being within array bounds and not overflowing, etc.

Therefore, we designed a "hint" interaction system: If load_tac fails to make e evaluate to something of the form $a.p$, it displays an error message asking the user to prove an equality of the form $e' = a.p$, where e' is the result of evaluating e, and a and p are to be given by the user. After proving such a hint, the user can invoke forward again, which invokes load_tac or store_tac, and the tactic will use the hint and thus succeed.

However, this solution is not satisfying, because many cases which worked fully automatically before now require a user hint. The problem is that we do not make use of path evaluation (i.e. the clause "cpath evaluates to path p" in Load-1 and Store-1) any more to turn paths given in the load or store instruction into Coq paths.

But we can prove yet another version[22] of the load and store rules which combine the automation friendliness of Load-1/Store-1 with the generality of Load-2/Store-2:

$$\textsc{Load-3}\ \frac{\begin{array}{c}e \text{ evaluates to } a.p_a \qquad \mathsf{cpath} \text{ evaluates to path } p_b \qquad p_a.p_b = p_0.p_1 \\ \vec{R}_i = \mathsf{field\_at\ sh\ t}\ p_0\ v'\ a \qquad \mathsf{sh} \text{ grants read permission} \qquad v'.p_1 = v \\ \vec{Q}' = (\vec{Q} \text{ with the value for } x \text{ updated to } v)\end{array}}{\Delta \vdash \{\mathrm{PROP}(\vec{P})\mathrm{LOCAL}(\vec{Q})\mathrm{SEP}(\vec{R})\}\, x = e.\mathsf{cpath}\, \{\mathrm{PROP}(\vec{P})\mathrm{LOCAL}(\vec{Q}')\mathrm{SEP}(\vec{R})\}}$$

$$\textsc{Store-3}\ \frac{\begin{array}{c}e_1 \text{ evaluates to } a.p_a \qquad e_2 \text{ evaluates to } v \qquad \mathsf{cpath} \text{ evaluates to path } p_b \\ p_a.p_b = p_0.p_1 \qquad \vec{R}_i = \mathsf{field\_at\ sh\ t}\ p_0\ v_{old}\ a \qquad \mathsf{sh} \text{ grants write permission} \\ v_{new} = v_{old} \text{ with the substructure denoted by } p_1 \text{ updated to } v \\ \vec{R}' = (\vec{R} \text{ with } R_i \text{ replaced with } \mathsf{field\_at\ sh\ t}\ p_0\ v_{new}\ a)\end{array}}{\Delta \vdash \{\mathrm{PROP}(\vec{P})\mathrm{LOCAL}(\vec{Q})\mathrm{SEP}(\vec{R})\}\, e_1.\mathsf{cpath} = e_2\, \{\mathrm{PROP}(\vec{P})\mathrm{LOCAL}(\vec{Q})\mathrm{SEP}(\vec{R}')\}}$$

The key insight is that there might be two different ways of splitting the whole path p: The first, $p = p_0.p_1$, is the one that the SEP clause prefers, and the second, $p = p_a.p_b$, is the one chosen by the C instruction.

If we rewrite the tactics to use these rules, we derive as much of the path as possible from the C instruction, so they work in all cases where they worked originally, but also in cases like the example in Figure 1b.

[21] Called semax_SC_field_store_without_nested_efield in Coq, same file.

[22] Called semax_SC_field_load'' and semax_SC_field_store_with_nested_field_partial in Coq, same file.

4.7 Yet another shortcoming

The version above was used successfully until another case study on verifying a garbage collector revealed a problem: In the system of this case study, each object on the heap was preceded by a 32-bit header, so if `p` is a pointer to the first field of an object, `p[-1]` is a valid expression and returns the header of the object. Now, if the tactic matches `p[-1]` with e_1.cpath, and evaluates cpath, it obtains (SUB (-1)), which is never a valid path, because all arrays in C start with index 0. So, the side condition that the path is valid[23] does not hold, and the tactics fail.

Later, another example of the same kind was found in the AES case study, and a simplified version of it is shown in Figure 10.

```
uint32_t *RK = ctx->rk;
for(int i = 0; i < 7; i++, RK += 8) {
    RK[11] = ...
}
```

Figure 10: AES code snippet where the first load/store tactics improvement still failed

Both examples have in common that the path seen in the load or store instruction is not the actual path through which the value would normally be accessed.

4.8 The final version of the memory load/store tactics

Fixing the above problem is simple: We can just apply LOAD-2 (or STORE-2) instead. So overall, our final tactic for a memory store of the form e_1.cpath $=e_2$ now works as follows (the tactic for memory loads is similar):

- Evaluate the whole expression e_1.cpath and check if the context contains a user-defined hint on how to bring it into the form $a.p$.
  - If yes, find a SEP clause about a whose path is a prefix of p, and apply STORE-2 and solve its side conditions.
  - Otherwise, evaluate the root expression e_1 and check if the result already has the form $a.p_a$ or if the context contains a user-defined hint on how to bring it into the form $a.p_a$.
    - If yes, evaluate cpath to obtain a path p_b, find a SEP clause about a whose path is a prefix of $p_a.p_b$, and apply STORE-3 and solve its side conditions.
    - Otherwise, evaluate cpath to obtain a path p, and check if there is a SEP clause about a whose path is a prefix of p.
      - If yes, apply STORE-1 and solve its side conditions.
      - Otherwise, fail with an error message containing the form of hints which were not found before.

Note that it uses all three STORE rules, even though STORE-2 would be general enough to be applicable in all cases. The only reason to use the other store rules is to overcome the automation brittleness described in section 4.6. In fact, instead of attempting to use tactics to perform the tricky task of bringing the result of evaluating

[23] This side condition was omitted in the rules presented before, but is present in the actual rules in Coq.

an expression into the form $a.p$, we prefer to do this work with lemmas, by having three specialized lemmas, STORE-1, STORE-2 and STORE-3, and depending on the case, picking the one which is easiest to apply automatically.

Now, one might wonder if this is the end, or whether at some time in the future, we might come across yet another C code example where these tactics fail. We believe that, except for the deliberately chosen restrictions described in section 4.1, there will be no further such C code examples, because LOAD-2 and STORE-2 do not impose any restriction on the form of the expression which denotes the memory location, and if evaluating this expression cannot turn it into something of the form $a.p$ automatically, the tactics can fall back to the user hint mechanism, allowing to request that the user proves the tricky part in such a way that all the rest can still be solved automatically.

5 Controlling term reduction

Term reduction in Coq is the process of applying *conversions* [27] such as β-reduction, simplification of match expressions on constructors, and unfolding definitions, to change terms into other terms considered equal by Coq's type system.

There are several situations where term reduction has to be performed, and it has been (and still is) a challenge to control how much reduction is done: If too little reduction is done, the term does not reach the shape needed to prove the goal, but if too much reduction is done, it might reduce user-defined expressions that the user would not like to be simplified, or worse, reduction might take more time than acceptable, and in some cases, it can also consume all the RAM available on the user's computer.

5.1 When term reduction is invoked

We can distinguish three situations when term reduction has to be performed:

First, several tactics have to perform term reduction in order to prove simple side conditions. Coq provides a number of tactics for this, such as simpl, cbn, cbv, lazy, and hnf, and several options to control their behavior [29]. However, even with all those options at hand, it is not always easy to control how much reduction is done.

Second, Coq's typechecker also peforms reduction, because whenever it has to unify two types, it runs its conversion algorithm [27]. If the conversion algorithm reduces the "wrong" terms, typechecking can take a very long time. This is particularly noticeable when invoking the set tactic, which requires the current proof state to be re-typechecked, and when closing a proof with Qed, which requires the whole proof to be typechecked. For both of these situations, excess time and memory consumption was experienced in our case studies.

And third, there are built-in Coq tactics where one might not immediately expect that they perform reduction. An example for such a tactic is Coq's inversion tactic. In the AES case study, unfeasibly slow running times for it were observed.

5.2 Restricting reduction inside VST

In order to discuss what VST could and should do to restrict term reduction, we will consider each of the three situations described in section 5.1 separately:

Tactics performing reduction Examples for this are the situations involving VST's typechecking function for values, tc_val. It takes a C type and a Coq value, and returns a proposition which is true iff the Coq value can be represented as the C type. In most cases, reducing the terms of the form (tc_val t v) yields True, so it seems reasonable that VST's tactics automatically reduce such terms, because it allows them to solve trivial side-conditions automatically.

However, when typechecking a value v as an unsigned char, the term (tc_val tuchar v) reduces to a proposition testing whether v is less than 256. Now, if this proposition is further reduced, the user-defined expression v is reduced, and this never finished in the AES case study, because reduction basically tried to symbolically execute AES encryption for arbitrary inputs.

So, we see that it is important that only the functions needed to define the typechecking should be simplified, but not the user-defined expressions inside the values being typechecked.

This could be achieved by providing Coq's reduction tactic simpl or cbv a "whitelist" of functions it is allowed to unfold. Some parts of the *Floyd* library use this approach, but these whitelists tend to become very long and tedious to maintain, so it was not chosen in this case.

Another approach to prevent user-defined subexpressions from being simplified is to call Coq's remember tactic for each user-defined subexpression, so that each of them is replaced by a fresh variable, and to substitute all these variables by the remembered value after the reduction. This approach is also used in some places in the *Floyd* library, but it tends to be very slow, and increases the proof size as well as the time it takes to typecheck proofs at the Qed.

And yet another approach is to experiment with the Coq's different reduction strategies and their options. A lot of effort was spent during this project on fine-tuning the reduction behavior, and four attempts were successful,[24] while many other attempts were unsuccessful.

So, VST is now in a state where undesired reduction of user-defined terms happens less often than before, but still does happen. In particular, it happens in the tc_val example described above.

Reduction in Coq's typechecker Since the slowness of Coq's typechecking mostly comes from the complexity of user-defined expressions, there's not much the VST library can do against this.

Built-in Coq tactics So far, the only case where a built-in Coq tactic did too much reduction was the inversion tactic, and this could be solved by avoiding its use, because it turned out that it was not necessary at all.[25]

5.3 Term reduction controlling options for VST users

Now, given that in the first and second situation described before, VST cannot always solve the problem of controlling the amount of reduction correctly, we have to discuss what users can do to solve it:

Using remember to prevent subterms from being simplified, as described before, can also be done by VST users. The same limitations apply, but less seriously, because the user can choose to only remember those subterms which are really too expensive to simplify, instead of all user-defined subterms of a given term. Note that in some cases, for instance before invoking entailer! (see section 6.3), calling remember is not enough, because entailer! will call subst, which undoes the remember, so the equation generated by remember has to be hidden inside some wrapper.

Another strategy for VST users to control term reduction is to prevent some definitions from being unfolded during reduction, and there are several ways to do so: Using the command Arguments ... : simpl never., one can prevent simpl and cbn from unfolding the definition. Using Opaque, one can also prevent the other reduction tactics from unfolding the definition. And if one wants to prevent the typechecker from unfolding it

[24] The corresponding commits can be found, respectively, at
https://github.com/PrincetonUniversity/VST/commit/d04a191b,
https://github.com/PrincetonUniversity/VST/commit/f2ea465d,
https://github.com/PrincetonUniversity/VST/commit/58f506bf,
https://github.com/PrincetonUniversity/VST/commit/be011e12.

[25] https://github.com/PrincetonUniversity/VST/commit/a81ecb77

	simpl, cbn	cbv, lazy, hnf	inversion, type checker
Arguments ... : simpl never.	no	yes	yes
Opaque ...	no	no	yes[26]
abstract module	no	no	no

Table 1: Different levels of preventing definitions from being unfolded

as well, one can hide the definition behind the signature of an abstract module. Table 1 summarizes which method prevents which components from unfolding definitions.

Using the Arguments command, we got the tc_val example to work by marking some operations occurring in user-defined expressions as simpl never. Other situations required Opaque, and while the abstract module solution is not strictly necessary to make proofs work, it allowed us to worry less about controlling reduction during a first iteration of writing the AES encryption function proof.

[26]The unfolding is done as late as possible [28].

6 Various usability improvements

During the AES case study, many usability issues with the Floyd tactics library were found, and some of them were fixed during this project, and are described in the following sections.

6.1 Avoid swallowing error messages

Background The forward tactic is used in VST to do forward symbolic execution, and it works for all statements which do not need any additional user inputs (such as e.g. loop invariants). It first brings the proof goal into the form $\Delta \vdash \{P\}\, c\, \{?Q\}$, where c is one single command, and $?Q$ is a Coq evar, i.e. a hole which can be instantiated later. Then, depending on what c is, it calls the appropriate C statement specific tactic to solve the goal, e.g. store_tac that we saw in Section 4.

Originally, it used Coq's tactic first to do this dispatch. first[$t_1 \mid \ldots \mid t_n$] calls tactic t_1, if that succeeds, it stops, otherwise, it calls t_2, and so on, until a tactic succeeds. If none succeeds, it fails with the error message "No applicable tactic."

The problem This means that if anything goes wrong in a C statement specific tactic, the error will always be "No applicable tactic."

This was worked around by defining some inductive datatypes whose names were an error message, e.g. Cannot_find_function_spec_in_Delta, and if a tactic wanted to present an error to the user, it changed the goal to be of such a type. But this only works for explicitly handled errors, and unhandled errors still only displayed "No applicable tactic."

The solution The improvement made in this project is to refactor the forward tactic so that it does not use the first tactic any more, but instead does a pattern match on the proof goal to determine which C statement specific tactic should be applied.[27]

Thanks to this change, errors occurring in the C statement specific tactics now can bubble up and are displayed to the user. In particular, the error stack traces generated by Coq are shown, as one can see in Figure 11.

```
Ltac call to "forward" failed.
Error: No applicable tactic.
```

(a) Error message before the refactoring

```
In nested Ltac calls to "forward",
"forward1", "store_tac", "store_tac",
"store_tac", "store_tac_with_full_path_hint",
"store_tac_with_root_path_hint",
"sc_new_instantiate", "sc_new_instantiate" and
"sc_new_instantiate", last call failed.
Error: No matching clauses for match.
```

(b) Stack trace printed after the refactoring

Figure 11: Avoiding the first tactic results in better error messages

Of course, ideally all possible errors would be handled explicitly by the tactics and turned into a nice error message. If this was the case, the above change would be less relevant, but the work on the AES case study has shown that there are still many unhandled errors that can occur, and in these cases, it is very useful to get at least a stack trace of the error.

[27] https://github.com/PrincetonUniversity/VST/commit/01aa7fa1

```
for( i = ( tmp >> 1 ) - 1; i > 0; i-- ) {
    AES_FROUND( Y0, Y1, Y2, Y3, X0, X1, X2, X3 );
    AES_FROUND( X0, X1, X2, X3, Y0, Y1, Y2, Y3 );
}
```

Figure 12: Main loop of AES encryption with two macro invocations

The refactoring pattern proposed here was later adopted by Appel to enable error propagation in the tactics for forward_call.[28]

6.2 Enabling folding of macro-generated code in goal display

Background VST displays the C commands in its proof goals as ASTs (abstract syntax trees). Experience has shown that most users quickly get used to reading the ASTs, but the still, the AST of a whole function body can be very long, and should not be completely shown in the proof goal. To this end, VST has a mechanism to display only the first command of the current sequence of commands, and hiding the remaining commands behind a variable called MORE_COMMANDS. That is, instead of displaying (Ssequence c_1 c_2), where Ssequence is the name of the AST for C's semicolon, and c_1 and c_2 are C commands, it displays (Ssequence c_1 MORE_COMMANDS). Often, c_2 consists of many nested Ssequence constructors and is very long, so this makes the proof goal much shorter and easier to read, without hiding any relevant information, because in forward symbolic execution, we only focus on the first command of a sequence of commands.

The problem The main loop of the AES encryption function consists of two macro invocations, as shown in Figure 12. The `AES_FROUND` macro expands to a sequence of 24 C commands, so the body of this loops is represented as (Ssequence c_1 c_2), where both c_1 and c_2 consist of 23 nested Ssequence constructors. However, as described above, only c_2 will be hidden behind MORE_COMMANDS, so the user still sees a huge proof goal, which is very inconvenient to handle.

The solution VST already contains the following rule called semax_unfold_Ssequence

$$\frac{\Delta \vdash \{P\}\, c_1\, \{Q\} \qquad \mathsf{unfold}\ c_1 = \mathsf{unfold}\ c_2}{\Delta \vdash \{P\}\, c_2\, \{Q\}}$$

where the unfold function takes an AST and flattens all Ssequence constructors into a list of statements.

It was used in this project to implement a tactic called reassoc_seq,[29] which replaces an arbitrary nesting structure of Ssequence by the canonical form

$$(\mathsf{Ssequence}\ c_1\ (\mathsf{Ssequence}\ c_2\ (\ldots\ (\mathsf{Ssequence}\ c_{n-1}\ c_n)\ \ldots)))$$

where all c_i are other AST nodes than Ssequence. This structure makes sure that the folding into MORE_COMMANDS works as expected, only exposing the first command, and hiding all others.

[28] https://github.com/PrincetonUniversity/VST/commit/ba8881dd
[29] https://github.com/PrincetonUniversity/VST/commit/29f250a6

Special attention had to be paid to make sure that for-loops and while-loops, which are defined as a special case of a more general loop AST, are preserved by this operation.

6.3 Making the tactics library user-configurable

Background Many separation logic rules have entailments of the form $P \vdash Q$ as side conditions, where P and Q are two separation logic formulae. Most of them can be solved automatically by a tactic called entailer!, and many other tactics in VST (e.g. load_tac and store_tac) call entailer! to solve side conditions.

The problem However, in some situations, entailer! is very slow, and the VST user has domain-specific knowledge which could be used to solve the entailments much faster. This caused some VST users to copy-paste the whole definition of load_tac into their own files, renaming it to load_tac', and replacing the call to entailer! by their domain-specific entailment solving strategy.[30] From a software engineering point of view, this is clearly not a desirable solution. Since this problem also appeared in the AES case study, the following solution was proposed (and adopted).

The solution The solution depends on a little-known Coq feature allowing one to *globally redefine* a tactic with the "::=" operator (using double instead of single colon).

The implementation of load_tac is changed to call entailer_for_load_tac instead of entailer! directly, and entailer_for_load_tac is just an alias for default_entailer_for_load_tac, and default_entailer_for_load_tac calls entailer! in the same way as load_tac originally did. That is, nothing has changed so far except that a few seemingly unnecessary levels of indirection were added.

But this now allows VST users to do the following: When they have a call to forward which takes too long, and they suspect that it could be because of an entailer! call by load_tac, they can redefine entailer_for_load_tac e.g. with idtac, which does nothing and just leaves the goal open, or with a custom strategy using domain-specific knowledge. The implementation of load_tac will then call the fast user-defined strategy, so the slowness is gone. Once the user wishes to revert the behavior of load_tac back to the default, they can re-redefine entailer_for_load_tac to be default_entailer_for_load_tac.

Besides load_tac, this pattern was also implemented for store_tac[31] and another tactic called canon_load_result[32], and adopted by William Mansky for calls to the cancel tactic invoked after function calls.[33]

6.4 Enabling automated typechecking of array elements

Background CompCert defines an inductive datatype called val, shown in Figure 13, to represent the values stored in C memory, and VST uses this datatype as well.[34] Note that there is the value Vundef, which stands for an uninitialized value.

The contents of arrays are represented using Coq lists, so if myInts is a list of integers, the expression (map Vint myInts) represents the contents of an array whose elements are all initialized and correspond to the values in myInts. To index elements of a list, VST uses the function Znth. Besides an implicit type arguments, it takes an index, a list, and

[30] https://github.com/PrincetonUniversity/VST/blob/244a1a5d/progs/verif_mailbox.v#L2123
[31] https://github.com/PrincetonUniversity/VST/commit/0a66f6f5
[32] https://github.com/PrincetonUniversity/VST/commit/6d39c4b4
[33] https://github.com/PrincetonUniversity/VST/commit/783b1541
[34] https://github.com/PrincetonUniversity/VST/blob/e3c5ae84/compcert/common/Values.v#L27-L42

```
Inductive val: Type :=
  | Vundef: val
  | Vint: int → val
  | Vlong: int64 → val
  | Vfloat: float → val
  | Vsingle: float32 → val
  | Vptr: block → int → val.
```

Figure 13: CompCert's datatype for values

as a third argument, a default value to return in case the index is out of bounds. So, the value that VST's forward symbolic execution computes for a load from an integer array typically looks like (Znth i (map Vint myInts) Vundef).

The problem This form of expressions starting with Znth is not automation-friendly, because many tactics need to prove typechecking side conditions that many Hoare rules of VST impose: For instance, the condition (tc_val tint v) requires that v is of the form (Vint x), and if v is (Znth i (map Vint myInts) Vundef), this only holds if i is within bounds, and needs an additional reasoning step. Using the lemma Znth_map (and a proof that i is within the bounds), (Znth i (map Vint myInts) Vundef) can be turned into (Vint (Znth i myInts Int.one)), which is much more automation-friendly. But this step had to be done manually by the user, and the user had to know that doing this step before invoking further tactics allows them solve more goals automatically, and moreover, in the case of load_tac, the Znth_map lemma should have been applied even before the user could interact the next time.

The improvement made in this project is to apply the Znth_map lemma inside load_tac in the earliest possible place, guaranteeing the maximum number of subsequent tactic invocations to benefit from the more automation-friendly form.[35]

This simple change allowed for a considerable simplification[36] of the proof body for the AES encryption function: Before, it was 15201 non-whitespace characters long, and after the change, only 12760 characters of user proof script were required, which corresponds to an improvement by 16%.

[35] https://github.com/PrincetonUniversity/VST/blob/6d39c4b4/floyd/forward.v#L1743-L1754
[36] https://github.com/PrincetonUniversity/VST/commit/dd5d55af

7 Possibilities for future work

7.1 VST cryptography

The most immediate addition to the work done in this project would be to verify AES decryption as well. This would involve verifying the inverse key scheduling function and the decryption function. The table generation code is the same for both encryption and decryption. However, we expect that not much insight regarding VST would be gained from this, because decryption is very similar to encryption. Next, one could also verify the various block cipher modes which turn the 256-bit cipher into a cipher for data of arbitrary length.

Given encryption and decryption functions, one could write (VST-independent) proofs that they are the inverse of each other.

This work is part of a bigger effort to use VST to verify C implementations of some of the most commonly used cryptographic primitives. So far, the OpenSSL SHA-256 hash function was verified [2], as well as the hash-based message authentication code (HMAC) implementation based on it [9] and a deterministic random bits generator (DRBG) using the HMAC function [25].

Instead of basing the DRBG on the HMAC function, the NIST standard [6] also allows DRBG to be based on AES because it is faster, so the present work could be connected to the existing DRBG stack in the future. Ye [32] proves pseudo-randomness for the HMAC-based DRBG, and explains how the proof could be linked to the VST proof of HMAC-based DRBG, which applies in a similar way to AES-based DRBG.

7.2 Specialized AES hardware instructions

There are several processors with built-in instructions for AES, such as an instruction to perform one round of AES encryption. The mbedTLS library supports two such instruction sets by inlining some assembly code into their C library: The VIA C3 x86 processor for embedded devices provides an instruction set called PadLock,[37] and many Intel and AMD processors x86 processors have AES instructions grouped under the name AES-NI.[38]

It would be interesting to verify implementations using these specialized hardware instructions, because most of today's processors have such instructions. A first approach would be to use CompCert's possibility to call one assembly instruction as if it was an external function, and to pose the correctness of the AES round function as an axiom, while verifying the correctness of the surrounding code.

It would also be interesting to connect this work to efforts on inferring specifications for AES hardware instructions in the instruction-level abstraction (ILA) language [26].

7.3 Cache timing attacks

Another important threat to secure encryption are side channels such as timing attacks. For instance, Bernstein [10] presents a simple attack to extract the secret AES key of a server if the attacker can measure precise encryption timings for a large number of known plaintexts. It is based on the observation that the time it takes to access an array element can depend on the array index because of caches, and since the AES implementation he considered (as well as the mbedTLS implementation) contains array

[37] https://github.com/ARMmbed/mbedtls/blob/mbedtls-2.3.0/library/padlock.c
[38] https://github.com/ARMmbed/mbedtls/blob/mbedtls-2.3.0/library/aesni.c

accesses where the index is calculated from the secret key, it is possible to infer the secret key.

Osvik et al [22] exploit the same effect under a different threat model: They consider a server running an AES encryption process, as well as a malicious process, which one would expect to be unable to interfere with the AES process, because the virtual memory system prevents it from reading other processes' memory. However, by mere observation of the effects that the encryption has on the state of the cache, without any knowledge of plaintexts or ciphertexts, it is possible to extract the secret key, as they show.

VST does not provide any mechanism to reason about timings, and this problem is even harder because it is difficult to know whether a given instruction will run in constant time on the hardware, because detailed hardware specifications are typically not public.

7.4 Future work on VST's proof automation

During the AES case study, many situations where VST's proof automation could be improved were encountered. While some of them were fixed as part of this project, there remain 37 issues to be resolved, which can be found by searching the Coq files in the AES subdirectory[39] for the phrase "TODO floyd".

[39] https://github.com/PrincetonUniversity/VST/tree/master/aes

8 Related work

The domain of verifying cryptographic software is vast. Figure 14 is an attempt to classify the various approaches. In the following, we will use lower-case letters to refer to the nodes and edges of Figure 14.

We can distinguish between cryptographic primitives, such as encryption and hashing (in the left half of Figure 14) and protocols using them (in the right half). Moreover, we can distinguish various levels of abstraction (from top to bottom): One can give an abstract specification of cryptographic properties (c) by stating the properties we expect them to have, such as confidentiality of encrypted data, pseudo-randomness, invertibility of encryption, etc. One can also specify how the input of a cryptographic primitive is related to its output in a concrete way (f), or in a more low-level, but still formal way (o), which is as close to a runnable implementation as possible. The runnable implementation (t) should be in a programming language supporting input and output and network communication.

On the protocol side, at the very top, we have the final goal of all cryptography verification efforts: The desired properties of the cryptographic protocol (a). Examples for desired properties include confidentiality, integrity, authenticity, and examples for protocols are TLS (transport layer security), on which HTTPS is based, the Signal protocol, which is used in several chat applications, the SET protocol for credit card transactions, etc. To reason about the protocols, one creates a model of them (d), and of course, it also needs to be implemented (u). Depending on the approach, there might be intermediate models between (d) and (u), but since they vary between different approaches, we omit them for simplicity.

At the end, both the cryptographic primitives and the protocol implementations have to be compiled into machine code (w) and run on hardware (z).

VST-related proofs The proofs done in the present project are a VST-based equivalence proof (r) between a C implementation of AES (t) and a low-level specification (o), as well as an equivalence proof (m) between the low-level (o) and the high-level specification (f). In fact, (f), (o), (t), (w) correspond to Figure 2 presented earlier.

The other VST proofs mentioned in section 7.1, as well as the VST verification of the TweetNaCl cryptographic library [11] all are in this same area, except Ye's proof (e) of pseudo-randomness (c) for the HMAC-based DRBG [32].

Nothing in VST is cryptography-specific, so VST can also be used to verify other C code. For instance, Mansky specifies and verifies a concurrent exclusive-write buffer system [19].

Extraction and synthesis based verified cryptography While VST can be used to verify *existing* C implementations being widely used in practice, there are several other verification tools which *generate* a verified implementation of cryptographic primitives, possibly by synthesizing (n) it from a specification, or by extracting (q) a proven correct implementation from a proof language to a runnable language. These approaches differ from VST in two ways: First, cryptography users only benefit from their correctness proof if they choose to use this code, whereas the VST proofs are about commonly used C libraries such as OpenSSL and mbedTLS, so many cryptography users benefit from VST's correctness proofs without noticing it.[40] And second, the code generated by

[40]However, VST does not (yet) support all of C (see section 4.1), so the code actually running in production slightly differs from the code verified in VST.

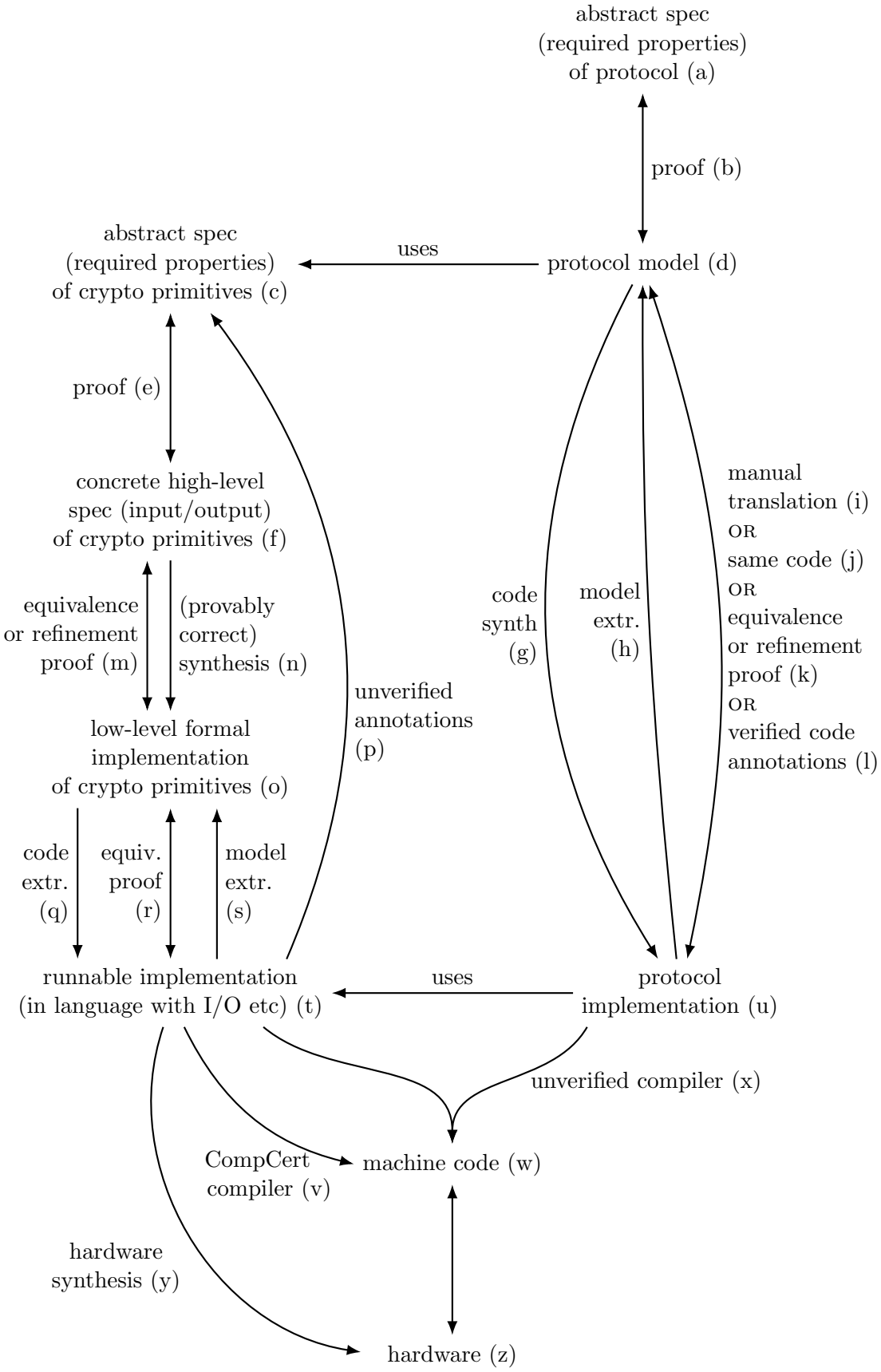


Figure 14: Formalization and verification of cryptographic software

these approaches, e.g. OCaml, Haskell or C code, is compiled by an unverified compiler (x), whereas VST's proofs connect to the correctness proof [17] of the CompCert verified C compiler (v). In fact, of all other approaches discussed here, the VST approach is the only *foundational* one, where the tools themselves are verified as well, and an end-to-end proof statement relating the high-level formal specification and the actual assembly code are related by a machine-checked proof.

Erbsen et al [15] start from a high-level Coq specification of elliptic curve cryptography with arbitrary field moduli (f) and automatically generate (n) efficient low-level code (o) from it. Their low-level Coq code consists of sequences of let bindings of primitive operations on native integers, which can be extracted (q) from Coq to C by a simple syntactic transformation, so there's only a minimal amount of unverified extraction. However, their approach could not easily be adapted to AES, because AES involves loops and some management code surrounding the cryptographic code, so one would have to extract the Coq code to Haskell or OCaml, which is a more complex extraction phase, and produces less efficient code.

Toma and Borrione [30] implement a hardware circuit for the SHA-1 hash function using VHDL (t), extract (s) an ACL2 model (o) from it and prove functional correctness (f) in the ACL2 theorem prover.

Verification of protocols Much work has been done on verifying protocols (the right half of Figure 14). Some, but not all researchers, also connect their protocol verification to the verification of the cryptographic primitives (the left half).

Project Everest intends to build a verified secure HTTPS stack, and they recently presented [12] a verified implementation of the underlying TLS 1.3 protocol, including the verification of the cryptographic primitives they need. They use the F* language, a higher-order effectful ML-like functional programming language designed for program verification. F* allows to write specifications as well as implementations, and verifying the implementation reduces to typechecking. This allows them to have the implementation of the protocol in the same language as the model (j), but the compilation of F* code still is unverified (x) and uses C or OCaml as an intermediate language.

An earlier TLS 1.0 verification project [16] by an overlapping set of authors uses F# instead of F*, and achieves less integration of the primitives and the protocol: They extract (h) cryptographic models from the F# code, and define a symbolic (c) and a concrete (t) implementation of primitives such as hashing and encryption with the same API, and the protocol implementation can be linked against either of them: To run the protocol, it is linked against the concrete implementation, and for verification, it is linked against the symbolic implementation. To establish cryptographic properties, they use the ProVerif and CryptoVerif tools for protocol and cryptography verification, respectively.

Another project using these two tools is the verification of the Signal chat protocol, as implemented in a chat called CryptoCat [16]: They automatically extract (h) a ProVerif model from JavaScript code, and verify it with ProVerif and CryptoVerif.

The connection between the implementation and the model of the protocol can also be made in the other direction: For instance, Cadé and Blanchet [13] start from a ProVerif model, and extract (g) it to OCaml code.

And finally, there are also projects which do not deal with implementations at all, but focus on proving (b) the high-level properties of the protocol: For instance, Paulson [24] verifies a simplified abstract version of TLS in Isabelle, where hashing and encryption are assumed to be secure. Authentication and secrecy are proven, but MACs and integrity

checks are not even modeled. His group also verified other protocols [7] such as the SET protocol for credit card transactions [8].

Verifying cryptographic C code There are also other tools for verifying C code, but contrary to VST, none of them is foundational. Like VST, these tools can be used to verify any C code, but we will focus on their case studies on verifying cryptography:

Frama-C uses Hoare logic, but no separation logic, to reason about C programs which were annotated in the ANSI-C Specification Language (ACSL) using a special form of source code comments. It extracts verification conditions and feeds them to different automated theorem provers, but can also interface with Coq for proofs which need user interaction. Almeida et al [1] use Frama-C to verify functional correctness of the RC4 stream cipher, as implemented in OpenSSL. It is interesting to note that their specification is also written in C, and their proof consists of showing equivalence between the high-level C implementation (which serves as specification) and the optimized, low-level OpenSSL C implementation.

VCC is another annotation-based C verifier with macro-based annotations, and Dupressoir et al [14] use it to prove both memory safety and security properties for a simple authenticated remote procedure calls protocol. They prove security properties in Coq, and translate them manually (i) into annotations that they add as axioms to VCC.

Yet another C verifier is VeriFast, which is also based on annotations in comments, and uses separation logic. Vanspauwen and Jacobs [31] annotate the cryptographic primitives of mbedTLS with axioms (p) and use this to prove security properties (a) of some protocols implemented on top of these primitives.

9 Conclusion

We have presented a machine-checked proof in VST that the mbedTLS implementation of AES encryption conforms to a high-level specification closely following the standard. We showed how to obtain a modular proof by splitting it into a VST-related part and a VST-unrelated part proving equivalence between the high-level specification and a low-level specification following the C implementation.

Moreover, we reported on the shortcomings of VST's proof automation detected during this case study, and fixed a number of them:

We generalized the tactics for memory loads and stores, so that memory accesses whose path is not completely written out in the load or store command are supported as well, and we added support even for cases where parts of the path written in the program are not the ordinary path through which the value nested in structs/arrays would normally be accessed. The tactics for loads and stores are now as general as possible (within the known restriction of VST that only one top-level load or store or function call can be done per statement), but still support the simpler cases equally well as before.

We reported on the difficulties in controlling how much term reduction is performed, and we presented several improvements to VST's proof automation regarding this, but also discuss the situations where we cannot (yet) solve this problem inside the VST library. For these cases, we presented strategies that VST users can employ to control term reduction, so that the proofs do work, albeit requiring more user interaction and experience than ideal.

We improved the error propagation in the proof automation, enabled better folding of C code in the proof goal display, introduced a technique for VST users to customize the tactics library if it is too slow, and enabled a more automatic typechecking of values loaded from arrays.

Finally, we pointed out that many computers perform the AES encryption on specialized hardware and that therefore, verification of this hardware should be done in future work, and that functional correctness is not the only guarantee that we need for reliable cryptography, because we also need the absence of side channels such as timings, as well as protocols correctly using the cryptographic primitives.

This project shows that it is possible today to produce machine-checked foundational proofs about the behavior of real-world C programs. We think that this is an exciting result, but the effort required to produce such proofs is considerable, because when stepping through a C program, the calculations of VST not only generate strongest postconditions, but also a proof term for each claim. This makes programming VST's proof automation, as well as using it, considerably more laborious than in a tool which only generates strongest postconditions without accompanying proof terms. Such systems exists (e.g. frama-c), but their assertion language has less expressive power. If we consider the cost to benefit ratio, VST gives the strongest results, but at such a high cost that for practical use, we would probably prefer an unverified tool, which gives a slightly weaker result, but for a much lower cost. However, as a research project, trying to push the boundary of what we can prove formally, this was a very interesting and exciting experience.

10 Acknowledgments

I would like to thank Andrew Appel for supervising me on this project. Whenever I was stuck or had questions, he would take time to explain me the necessary background and give me instructions, enabling me to solve the problem, and to make contributions to the VST framework. This was very motivating. Also, he was a great mentor during the process of writing up my results in this report.

Moreover, I thank Qinxiang Cao for explaining me all the background on VST's memory load/store proof infrastructure, and Lennart Beringer and William Mansky for helping me with learning VST.

I thank Leonidas Lampropoulos for helping me to get QuickChick to work on my proofs, which was very helpful for the equivalence proofs.

I thank Viktor Kuncak for being my home university supervisor and providing pointers to relevant related work.

And finally, I would like to thank my office mates and neighbors, the Coq ninjas, the rubber ducks, weasels and chinchillas, for being awesome people and for introducing me to Princeton, Philadelphia, New York, to magic, homotopy type theory, the grad school application process, board games, restaurants, American culture, origami, and many more.

This research was supported by NSF grant CCF-1521602.

11 References

[1] J. B. Almeida, M. Barbosa, J. S. Pinto, and B. Vieira. Verifying Cryptographic Software Correctness with Respect to Reference Implementations. In *Formal Methods for Industrial Critical Systems*, pages 37–52. Springer, Nov. 2009.

[2] A. W. Appel. Verification of a Cryptographic Primitive: SHA-256. *ACM Transactions on Programming Languages and Systems*, 37(2):1–31, Apr. 2015.

[3] A. W. Appel. Verifiable C. 2016. https://github.com/PrincetonUniversity/VST/blob/master/doc/VC.pdf.

[4] A. W. Appel, R. Dockins, A. Hobor, L. Beringer, J. Dodds, G. Stewart, S. Blazy, and X. Leroy. *Program Logics for Certified Compilers*. Cambridge University Press, New York, NY, USA, 2014.

[5] A. W. Appel and others. The Verified Software Toolchain. https://github.com/PrincetonUniversity/VST.

[6] E. B. Barker and J. M. Kelsey. Recommendation for Random Number Generation Using Deterministic Random Bit Generators. Technical Report NIST SP 800-90Ar1, National Institute of Standards and Technology, June 2015. http://dx.doi.org/10.6028/NIST.SP.800-90Ar1.

[7] G. Bella. *Inductive Verification of Cryptographic Protocols*. PhD thesis, University of Cambridge, Computer Laboratory, 2000. http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-493.html.

[8] G. Bella, F. Massacci, and L. C. Paulson. An overview of the verification of SET. *International Journal of Information Security*, 4(1-2):17–28, Feb. 2005.

[9] L. Beringer, A. Petcher, K. Q. Ye, and A. W. Appel. Verified Correctness and Security of OpenSSL HMAC. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, pages 207–221, Berkeley, CA, USA, 2015. USENIX Association.

[10] D. J. Bernstein. *Cache-Timing Attacks on AES*. Technical report, 2005. http://www.hamidreza-mz.tk/files/cachetiming-20050414.pdf.

[11] D. J. Bernstein, B. van Gastel, W. Janssen, T. Lange, P. Schwabe, and S. Smetsers. TweetNaCl: A Crypto Library in 100 Tweets. In *Progress in Cryptology - LATINCRYPT 2014*, pages 64–83. Springer, Sept. 2014. https://tweetnacl.cr.yp.to/.

[12] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, J. Pan, J. Protzenko, A. Rastogi, N. Swamy, S. Zanella-Béguelin, and J. K. Zinzindohoué. Implementing and Proving the TLS 1.3 Record Layer. *Cryptology ePrint Archive, Report 2016/1178*, 2016. https://eprint.iacr.org/2016/1178.pdf.

[13] D. Cadé and B. Blanchet. From Computationally-proved Protocol Specifications to Implementations. In *2012 Seventh International Conference on Availability, Reliability and Security*, pages 65–74, Aug. 2012.

[14] F. Dupressoir, A. D. Gordon, J. Jurjens, and D. A. Naumann. Guiding a General-Purpose C Verifier to Prove Cryptographic Protocols. In *Proceedings of the 2011 IEEE 24th Computer Security Foundations Symposium*, CSF '11, pages 3–17, Washington, DC, USA, 2011. IEEE Computer Society.

[15] A. Erbsen, J. Philipoom, J. Gross, R. Sloan, and A. Chlipala. Systematic Synthesis of Elliptic Curve Cryptography Implementations. June 2017. https://people.csail.mit.edu/jgross/personal-website/papers/2017-fiat-crypto-pldi-draft.pdf.

[16] N. Kobeissi, K. Bhargavan, and B. Blanchet. Automated Verification for Secure Messaging Protocols and their Implementations: A Symbolic and Computational Approach. In *2nd IEEE European Symposium on Security and Privacy (EuroS&P'17)*, Paris, France, Apr. 2017. IEEE. To appear.

[17] X. Leroy. Formal Verification of a Realistic Compiler. *Commun. ACM*, 52(7):107–115, July 2009.

[18] S. Lyubomirsky. Toward the Formal Verification of AES-256. Independent work report, 2015.

[19] W. Mansky, A. W. Appel, and A. Nogin. A Verified Mailbox Protocol (Draft), 2017.

[20] National Institute of Standards and Technology. Advanced Encryption Standard. *NIST FIPS PUB 197*, 2001.

[21] P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *International Workshop on Computer Science Logic*, pages 1–19. Springer, 2001.

[22] D. A. Osvik, A. Shamir, and E. Tromer. Cache Attacks and Countermeasures: The Case of AES. In *Topics in Cryptology – CT-RSA 2006*, pages 1–20. Springer, Feb. 2006.

[23] Z. Paraskevopoulou, C. Hriţcu, M. Dénès, L. Lampropoulos, and B. C. Pierce. Foundational Property-Based Testing. In *Interactive Theorem Proving*, pages 325–343. Springer, Aug. 2015.

[24] L. C. Paulson. Inductive Analysis of the Internet Protocol TLS. *ACM Trans. Inf. Syst. Secur.*, 2(3):332–351, Aug. 1999.

[25] N. Sanguansin. Verification of a Deterministic Random Bits Generator. Independent work report, 2015.

[26] P. Subramanyan, Y. Vizel, S. Ray, and S. Malik. Template-based Synthesis of Instruction-level Abstractions for SoC Verification. In *Proceedings of the 15th Conference on Formal Methods in Computer-Aided Design*, FMCAD '15, pages 160–167, Austin, TX, 2015. FMCAD Inc.

[27] The Coq development team. The Coq proof assistant reference manual, Chapter 4: Calculus of Inductive Constructions. https://coq.inria.fr/refman/Reference-Manual006.html, 2016.

[28] The Coq development team. The Coq proof assistant reference manual, Chapter 6: Vernacular Commands. https://coq.inria.fr/refman/Reference-Manual008.html, 2016.

[29] The Coq development team. The Coq proof assistant reference manual, Chapter 8: Tactics. https://coq.inria.fr/refman/Reference-Manual010.html, 2016.

[30] D. Toma and D. Borrione. Formal Verification of a SHA-1 Circuit Core Using ACL2. In J. Hurd and T. Melham, editors, *Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science, pages 326–341. Springer, Aug. 2005.

[31] G. Vanspauwen and B. Jacobs. Verifying Protocol Implementations by Augmenting Existing Cryptographic Libraries with Specifications. In *Software Engineering and Formal Methods*, pages 53–68. Springer, 2015.

[32] K. Ye. The Notorious PRG: Formal verification of the HMAC-DRBG pseudorandom number generator. Technical report, 2016. https://www.cs.cmu.edu/~kqy/resources/thesis.pdf.