Deductive verification of the s2n HMAC code

1002221

University of Oxford

A dissertation submitted for the degree of

MSc in Mathematics and Foundations of Computer Science

Trinity term 2016

Abstract

We use VCC, a deductive verifier, to prove that s2n (an implementation of the TLS security protocol) correctly implements the HMAC construction. Moreover, this proof includes showing that the code is free from errors such as memory errors and arithmetic overflows. Our proof has revealed an error in one function, which turned out to not be a bug in the current implementation but which could have been a bug in a different one.

Contents

Introduction

On the internet, communication flows over untrusted data connections and through untrusted mediums. To secure such communication, cryptography is used. Today, the cryptographic protocol most commonly in use is TLS [11]. s2n [19, 20] is an open-source implementation of the TLS protocol, and we have successfully proved, by using the program verification software VCC, that it correctly implements the HMAC construction (which is only one part of the s2n code) in an error-free way. We note that our proof is not about the HMAC construction itself, but about its implementation in s2n.

Our dissertation is structured as follows: in chapter 1 we provide some background on program verification and why it is necessary. In chapter 2, we describe the HMAC construction and show, by means of pseudocode, how s2n implements it. Next, in chapter 3, we provide readers with the tools necessary to read our VCC proofs (for a tutorial on how to write them, see [6]). Chapter 4 contains explanations of the ideas behind our proofs, as well as the complete proof of one particular function, complete with explanations and comments. Our full proofs can be found in [12]. We assume basic knowledge of the C programming language (see [13] for a complete guide), but include, after the conclusion, a brief overview of pointers, structs and unions as an appendix.

Chapter 1

Background

As cryptographic protocols need to be implemented in an efficient way, it is necessary that TLS be implemented in a low-level language, such as C. Despite having the desirable trait of allowing for high performance execution, C comes with potential memory safety issues.

For example, a buffer overflow occurs when program execution can result in writing beyond the end of an allocated memory buffer. In the following code (example from [1]), this will happen if `p` is not properly null-terminated, or if its length is greater than `N`:

```
char buf[N];
char *q = buf;
while (*p)
*q++ = *p++;
```

Buffer overflows are only one of many kinds of memory errors that can cause a C program to exhibit *undefined behaviour*, which in C parlance means that it can the program can end up crashing or behaving arbitrarily. Other examples include null pointer dereference, signed arithmetic overflows, division by zero and accessing a union through the wrong union member. To prove anything about a program, we must at least prove the absence of all such errors. Once we've proved the absence of errors causing undefined behaviour, we might want to know whether the program code correctly does the job it was meant to do. Low-level code might be written in a way to do the computation efficiently, which may be different from the way it's actually defined.

The traditional method of checking programs is to test them. The problem with this approach is that bugs can easily go unnoticed, as when testing it is difficult to avoid assuming normal user behaviour [14]. Errors are more common than would be desirable - for example, in 2002, just in the USA, it is estimated that industries lost 59 million dollars just due to poor software quality [15]. The Heartbleed bug [9] in OpenSSL [16], the most widely used open-source implementation of TLS, is an example of a bug that was caused by a memory safety error in the code and serves as a reminder of the importance of having safe code.

In the past few years, there has been a growing interest in program verification softwares, whose objective is to come as close as possible to having the following three features [5]:

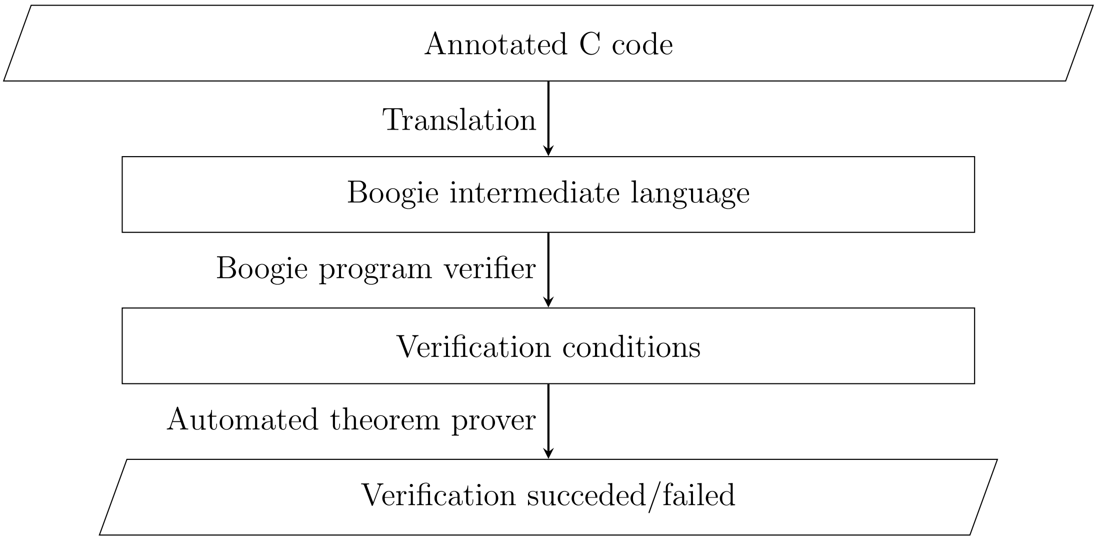
- automatic verification;
- no false positives;
- no unnoticed bugs.

Satisfying any two of these three is easy. If we want an automatic verifier that gives no false positives, it just needs to return 'verification successful' for any program. An automatic verifier that doesn't miss any bugs can just return 'verification failed' for any program, and a verifier that gives no false positives and doesn't miss any bugs could not return anything message at all and require that the user do all the work.

Deductive verification, the approach we employ in this project, aims to satisfy the second and third properties. In deductive verification, the program code, along with additional annotations, are used to generate a set of mathematical formulas, the proof of which suffices to show the correctness of the program. Because this process is complicated and results in a large number of formulas to be proved, it requires mechanical assistance. We use the program verifier VCC [7, 8] to do these proofs for us.

The first thing VCC does with the annotated C code we provide it with is to translate it into Boogie [2], an intermediate language also used by program verifiers Havoc [18] and Spec# [3]. Next it is fed to a Boogie program verifier, which outputs some verification conditions which are checked by the automated theorem prover Z3

[10].


Annotated C code
Translation
Boogie intermediate language
Boogie program verifier
Verification conditions
Automated theorem prover
Verification succeded/failed

Chapter 2

HMAC construction

2.1 MAC: message authentication codes

A MAC [17] provides a way of authenticating a message, thus ensuring that it has not been tampered with along the way. If a sender wants to send a message, they can send that message along with a MAC, which is computed using both the message and a private key which only the sender and the receiver have. MAC's are designed to be hard to forge, so that when the receiver receives the message and computes the MAC for herself, she knows that if she obtains the same MAC that came along with the message, then it is highly unlikely that anyone else could have computed it without knowing the private key. The idea is that even if someone else had a load of messages and their MAC's, they still wouldn't be able to forge a MAC for a new message.

A MAC consists of a function which generates a key for a given message, a function which generates a tag based on a message and a key, and a function which verifies a message and a corresponding tag. Formally:

Definition 2.1.1. MAC triple A triple $(\texttt{Gen} : 1^n \mapsto k, \texttt{Tag} : m \mapsto \sigma, \texttt{Var} : k \to \{\texttt{accept,reject}\})$ over a message space $\{0,1\}^n$ (where k is a key, m is a message and σ is a tag) is a MAC if `Gen`, `Tag`, `Var` are probabilistic polynomial time algorithms and the following equation is satisfied for a given key k, message m and tag σ:

$$(\sigma = \texttt{Tag}_k(m)) \implies (\texttt{Ver}_k(m, \sigma) = \texttt{accept})$$

We consider a MAC to be safe when the probability that the reverse implication doesn't hold is statistically negligible.

2.2 Hash functions and HMAC

One way to generate a MAC is with a hash function [4]. Hash functions are algorithms which take keys, and return digested versions of them of a fixed length (for the examples in this section, we will call this length L). In order to be safe, the probability of two distinct inputs returning the same output should be sufficiently small. It can be proved that if the underlying hash function is safe, then the HMAC construction must be safe too. The HMAC of a message `M` is given by

$$\texttt{H(K'} \oplus \texttt{ opad || H(K'} \oplus \texttt{ ipad || M))},$$

where H is the hash function, K' is a modified version of the key (which either has 0's appended to it if it's shorter than L or is hashed if it's longer), $\oplus$ is the XOR operation, || denotes concatenation and `ipad` and `opad` are strings composed of the bytes `0x36` and `0x5c` (respectively) repeated L times.

2.3 How s2n implements the HMAC construction

To implement the HMAC construction, s2n uses some structures called HMAC states, which store abstractions of the HMAC being computed for a given message. It operates on them with three functions: `s2n_hmac_init`, `s2n_hmac_update` and `s2n_hmac_digest`. Full details will be given in the next chapter, but for now we only look at what happens to a message `M` when we feed it into them. Throughout this section, we use the same notation as in section 2.2. `L` is the length of the array `xor_pad`. The first function, `s2n_hmac_init` doesn't use the message at all - it only prepares a construction which will be used by the other two functions. Any objects we mention, apart from the private key `K`, are fields of HMAC states. `inner`, `inner_just_key` and `outer` aren't arrays, but other types of objects - we have been a little imprecise in the following pseudocode, as what gets updated isn't the object but its abstract value (this will become clearer in section 4.2).

Algorithm 1: s2n_hmac_init

```
sets L according to the hash algorithm being used;
if $length(K) > L$ then
    outer ← K;
    digest_pad ← H(outer);
    xor_pad ← digest_pad;
else
    xor_pad ← K;
end
xor_pad ← xor_pad⊕ $0x36^{size(xor\_pad)}$ ;
xor_pad ← xor_pad || $0x36^{L-size(xor\_pad)}$; //append 0x36’s to xor_pad until it
    becomes of length L
inner_just_key ← xor_pad;
xor_pad ← xor_pad ⊕ 0x6a;
inner ← inner_just_key;
```

We note that there is no difference between first computing K’ from K and then XOR’ing it with ipad, and first XOR’ing K (or H(K), if K’s length is greater than L) with a string of 0x36’s and then appending some 0x36’s to get a a string of length L. This is because 0x0 ⊕ 0x36 = 0x36 (N.B. 0x0 is hexadecimal for 0). Furthermore, we note that

$$\begin{aligned}\texttt{0x36} \oplus \texttt{0x6a} &= \texttt{0x36} \oplus (\texttt{0x36} \oplus \texttt{0x5c}) \\ &= (\texttt{0x36} \oplus \texttt{0x36}) \oplus \texttt{0x5c} \\ &= \texttt{0x0} \oplus \texttt{0x5c} \\ &= \texttt{0x5c},\end{aligned}$$

as the XOR operation is associative and idempotent. So at this point, we have that:

- xor_pad = K’ ⊕ opad;
- inner_just_key = K’ ⊕ ipad;
- inner = K’ ⊕ ipad.

Next, we call s2n_hmac_update. Since the message we want to compute an HMAC of might be extremely long, we divide it into chunks M1, M2, etc.

Algorithm 2: s2n_hmac_update

```
inner ← inner || Mi;
```

So if we repeatedly call s2n_hmac_update with all the various chunks of the message, eventually we will end up with inner = (K’ ⊕ ipad)||M. inner_just_key and `xor_pad` don't change. Finally, we have `s2n_hmac_digest`:

Algorithm 3: `s2n_hmac_digest`

```
digest_pad ← H(inner);
outer ← xor_pad;
outer ← outer || digest_pad;
HMAC ← H(outer);
```

It can be seen, by substituting in what we knew those fields were equal to before this function began, that by computing `H(outer)` we have in fact computed an `HMAC`:

1. `digest_pad = H((K' ⊕ ipad)||M);`
2. `outer = K' ⊕ opad;`
3. `outer = (K' ⊕ opad)||H((K' ⊕ ipad)||M);`
4. `HMAC = H((K' ⊕ opad)||H((K' ⊕ ipad)||M)).`

Chapter 3

An introduction to VCC

To prove something about a program using VCC, you need to give it a C code to which you have added annotations which carry information both about what the program does, and why you think it works. Like this, you implicitly describe a proof, which VCC checks for soundness. The key concept that VCC uses is computational induction: if everything you've said at the beginning is true, and if we have that for any control point, if we have that whenever what you've said at that control point is true then what you've said at the next control point is true, then the proof works and VCC returns 'verification succeeded'. In this chapter, we introduce the various types of annotations. We will use the word 'concrete' to refer to anything seen by the C compiler, and 'abstract' to refer to anything which only VCC operates on.

3.1 Function contracts

The contract is the part of a function that appears before the first curly bracket. There are four types of annotations that can make up a function's contract:

- `_(requires ...)`
- `_(writes...)`
- `_(ensures...)`
- `_(decreases...)`

The first one specifies the preconditions which a function's arguments need to meet whenever that function is called. The second one specifies which fields of the arguments the function is allowed to modify. The third one specifies the postconditions which are expected to hold when we exit the function, and the last one is used to prove termination (see section 3.2). All annotations follow the syntax `_(...)`.

We now present a simple example which allows us to introduce various types of annotations, and to illustrate how contracts work:

```
#include <vcc.h>

void copy(int *from, int *to)
    _(requires \thread_local(from))
    _(writes to)
    _(ensures *to == \old(*from))
{
    *to = *from;
}

void main()
{
    int x,y;
    copy(&x,&y);
    _(assert x==y)
}
```

First, let's look at the function `copy`. `\old` refers to the value of its argument before the function was called. We will see the definition of `thread_local` in section 3.4 - for now, it's enough to note that it assures us that the argument doesn't change while we read it. When verifying this function, VCC assumes the preconditions (in this case, that `from` doesn't change while we use it), applies updates as specified in the function's body (as long as the objects to be updated appear in the `writes` clause), and then checks that the postconditions hold. In this example, there isn't much to check, so verification succeeds very quickly.

The `main` thread starts with integers `x` and `y`, which could be any pair of values. Then, it calls `copy` - but what does it know after the call? A central feature of VCC is *modularity*, meaning that when a function is called, VCC checks its preconditions are met, assumes an arbitrary update to whatever is in its `writes` clause, and then assumes its postcondition. It doesn't even look inside the function's body - all it looks at is the contract. Let's go through these steps for this example:

- it checks that the preconditions of `copy` hold (`x` is a local variable, so its value can't change during the execution of this thread);

- it assumes an arbitrary update to `y`;

- it assumes that `y` is equal to what `x` was equal to before the function was called (this assumption overrides the previous one).

After the call, there is an assertion `_(assert ...)` which asks VCC to check whether `x` is equal to `y`. Seeing as `x` wasn't modified by the function call, and VCC now knows that `y` is equal to what `x` was equal to before the call, the assertion verifies.

3.2 Loop invariants

When loops are involved, verification becomes a little bit trickier. Rather than trying to determine by itself what it knows upon the entry to each loop iteration (which at least partially depends on what it knew just before the end of the previous loop iteration), VCC forgets the values of all the variables that have been modified. It is the user's responsibility to write loop invariants to tell VCC what it what it can assume at the top of each loop iteration. It is good practice to also include a `decreases` clause, which VCC can use to assert that the loop terminates.

It then checks that:

- the loop invariants are guaranteed to hold upon loop entry;
- if they hold at the start of the nth iteration, they are guaranteed to hold at the end of it.
- in going from iteration n to $n + 1$ (as long as the loop guard isn't broken), the argument in the `decreases` clause is guaranteed to decrease.

We now look at an example[1]:

```
#include <vcc.h>
#include <stdint.h>

void main()
    _(decreases 0)
{
    uint8_t a[128];
    for (int i = 0; i < 128; i++)
        _(writes \array_range(a,128))
        _(invariant i>=0 && i <= 128)
        _(invariant \forall int j; j>=0 && j<i ==> a[j] ==
            \old(a[j])^0x6a)
```

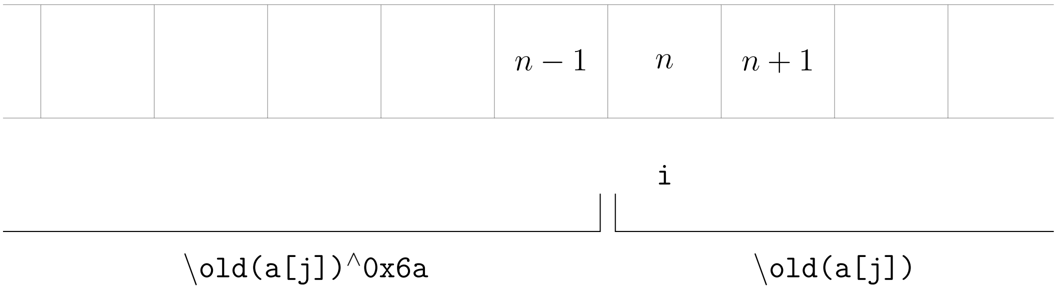
[1] for ease of readability, we remove any type-casting that takes place

```
        _(invariant \forall int j; j>=i && j<128 ==> \
          unchanged(a[j]))
        _(decreases 128 - i)
        {
            a[i] ^= 0x6a;
        }
}
```

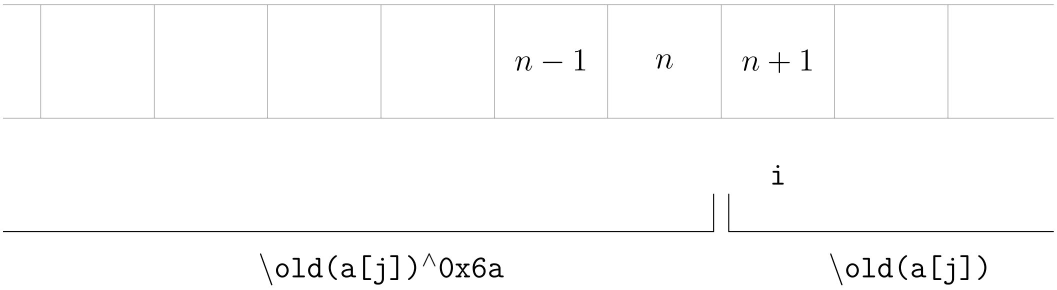
Before describing how VCC verifies this loop, let's try to make sense of what it does. The ^ symbol is C syntax for the XOR ("exclusive or") operation, so all this function does is it XOR's with `0x36` every entry of the array `a` that has an index smaller than 128. `\old` and `\unchanged` refer to the value of the argument before the loop started.

Upon loop entry, `i` is equal to 0, and so the third invariant amounts to saying that every entry in `a` is exactly what it was before the loop started. As nothing has been modified yet, it obviously holds. The second invariant has a precondition that can never be satisfied: no integer can be both greater or equal to 0 whilst simultaneously being less than it. Hence, the statement is always true, and this invariant also holds on loop entry. The first invariant is self-evident, but recall that VCC forgets the value of everything it modifies in the loop, and this includes the value of the index `i` - hence, it is necessary to include it.

Next, we check that VCC will be able to justify that if the invariants hold after n iterations, they also hold after $n+1$ iterations. This time, it's not enough to look at the last two invariants separately: they need to be considered together in order for their truthfulness to be asserted. Let's take `i` equal to n such that it is strictly less than 128. Upon loop entry, by the inductive hypothesis, every element of 128 with an index greater than i is exactly what it was before the loop started, and everything with an index smaller or equal to i is equal to its old value XOR'd with `0x36`.



At the end of this iteration `a[i]` is XOR'd with `0x36`, and i is increased by one, so this is what we end up with:



Again, everything with an index greater than `i` is equal to its old value, while everything with an index less than or equal to `i` is equal to its old value XOR'd with `0x36`, and so the invariants hold.

For simple loop like this one, it isn't strictly necessary to provide a `decreases` clause - VCC can infer termination on its own. It would be necessary to include one if the loop included a nested loop inside it.

3.3 Object invariants

In general, when we talk about objects, we refer to anything whose identity can be specified with an a memory address and a type. When we define compound objects like structs or unions (see appendix 4), there often are some properties that we would like them to satisfy. We tell VCC what these properties are by writing object invariants. For an example, we now create `interval`, which is just a struct with two cursors (`start` and `end`). Ideally, we would like the first cursor to be no greater than the second one, so we introduce an invariant saying exactly that:

```
struct interval{
    int start;
    int end;
    _(invariant start<=end)
}interval;
```

If an object's invariants are known to hold, then that object is said to be closed. If it's not closed, it's open. Each object has an abstract (i.e. not seen by the C compiler) boolean field `\closed` which is set to 1 if that object is closed and is set to 0 otherwise. If a thread owns a closed object, then that object is said to be 'wrapped', while if a thread owns an open object, then it's said to be 'mutable'.

If a thread owns an object, then only that thread is allowed to write (modify) its fields[2]. In order to be able to modify a wrapped object, we need to temporarily stop assuming that the properties specified by its invariants are known to hold. We do so by unwrapping it. When we unwrap an object, we gain write-access and ownership of that object's fields (more precisely, we obtain ownership of its span - see section 3.4). Typically, when we've finished modifying an object, we will want to wrap it, and so we will have to make sure that its invariants still hold so that VCC verification can run successfully.

As an example, let's take another look at `interval` and write a function to shift its end-points by 1:

```
void shift_interval(struct interval *x, int a)
    _(maintains \wrapped(x))
    _(requires x->end < INT_MAX)
    _(writes x)
    _(ensures x->start == \old(x->start)+1)
    _(ensures x->end == \old(x->end)+1)
{
    _(unwrap x)
    x->start = x->start + 1;
    x->end = x->end + 1;
    _(wrap x)
}
```

Before explaining what's going on, we note that `maintains` is a combination of `requires` and `ensures`, and that the first precondition is there to prevent arithmetic overflow.

Upon function entry, we know that `x`'s invariant holds because `x` is wrapped (which implies that it's closed i.e. that its invariants hold). It is also easily checked to hold just before we re-wrap `x` and the end of the function body: as they have both increased by 1, `start` can't suddenly have become greater than `stop`. VCC has no trouble asserting either of these conditions on its own, and verification succeeds.

Suppose we have a wrapped object `d` of the following type:

```
struct disjoint_intervals{
    struct interval x;
    struct interval y;
    _(invariant &x.end < &y.start)
};
```

[2] strictly speaking, it's only allowed to modify its non-volatile fields. However, in this project, we don't look at volatile fields at all, and so in general will not point this out

If we would like to apply shift_interval to y, then simply unwrapping d isn't enough, because we would only gain ownership and write-access to the its fields (which are pointers to x and y, rather than x and y themselves). So we need to somehow gain ownership of y.

We can achieve this by adding an invariant to d specifying that it owns the address of the pointer to its intervals. In order to be able to do this, VCC requires the object to have a dynamic_owns tag. So now, our definition of disjoint_intervals looks like this:

```
_(dynamic owns) struct disjoint_intervals{
    struct interval x;
    struct interval y;
    _(invariant &x.end < &y.start)
    _(invariant \mine(&x) && \mine(&y))
};
```

and if we try verifying

```
void main(struct disjoint_intervals *d)
    _(maintains \wrapped(d))
    _(requires (&d->y)->end < INT_MAX)
    _(writes d)
{
    _(unwrap d)
    shift_interval(&d->y);
    _(wrap d)
}
```

then verification succeeds.

3.4 Ownership

We have already introduced the concept of ownership in section 3.3 - it's what allows us to modify objects. If we want to modify an object, we need to own it. But what if we only want to read the information it stores? Then we need to be assured that it doesn't change while we're reading it. One way to ensure this is to require that object to be thread-local, which means that it's owned by a wrapped object. The examples from this section come from the HMAC functions, which we will look at in more detail in sections 4.2 and 4.3. For now, it's not important to know what any of the objects are or what any of the functions do - the reader should just focus on the thread_local commands.

We start with the OpenSSL function MD5_update, whose contract reads[3]:

```
int MD5_Update(MD5_CTX *c, uint8_t *data, size_t len)
    _(requires \wrapped(c) && \thread_local_array(data,len
      ))
    _(writes c)
    _(ensures \wrapped(c))
;
```

Seeing as we read data, we need to know it's thread local, so we list this requirement as a precondition.

Next, we look at the contract of the OpenSSL function MD5_final, which also takes a string as an argument, but unlike MD5_update, it modifies it. The only way an object can be writable is if it is thread local, so we can omit this requirement from the contract and only include the string in the writes clause:

```
int MD5_Final(uint8_t *md, MD5_CTX *c)
    _(maintains \wrapped(c))
    _(writes c, \array_range(md, MD5_DIGEST_LENGTH))
;
```

Now, we look at a couple of instances where VCC doesn't behave as we might expect it to. Specifically, these examples are correctly annotated, but VCC requires some extra hints, reminding it of things it already knows, in order for verification to be successful.

Part of the contract of a function in s2n_hmac.h (see 4.3) reads

```
    _(requires \extent_mutable(state))
    _(requires \thread_local_array(key,klen))
```

The function body includes some calls to other functions, including one that reads key. If we leave the preconditions of the function like this, then verification fails - not because the implicit proof is wrong, but because VCC thinks that key might be changing during the function body. In order for verification to succeed, we need to add the precondition

```
    _(requires \wrapped(\domain_root(\embedding((uint8_t
      *)key))))
```

What do the commands in this annotation mean? The domain of an object o is the set containing all the objects owned by o, as well as all the objects owned by all the objects owned by o, and so on. That is to say, it contains all the objects transitively

[3] the original function is written in a way that requires type-casting, so for the sake of this example we have simplified it slightly

owned by o. The root of a domain is the object in the transitive closure that isn't owned by any other object. The embedding of an object is the object that owns it. So this annotation amounts to saying that key is in the domain of a wrapped object - but didn't VCC already know this, from the precondition that key was thread-local? The answer is: yes. This new precondition doesn't add anything new, but acts as a more explicit hint to VCC that key doesn't change while it's being read. We now present another case of a correct proof that fails, where it is necessary to remind VCC of something it already knows in order for it to succeed. Here's a call in s2n_hmac_update, to which we add a couple of assertions:

```
_(assert (&state->outer)->hashState == repeat(0x0,0))
int res = s2n_hash_update(&state->inner, in, size);
_(assert (&state->outer)->hashState == repeat(0x0,0))
```

Even without knowing what any of these objects are, or what the called function does, readers should be able to convince themselves of the following:

- we assert something about some object;
- we call some function that updates a different object;
- we repeat the first assertion.

Seeing as the assertion is about an object that's not the object being called, we would expect the second assertion to verify if and only if the first one does. However, if we try verifying this code with VCC, the first one will succeed and the second one will fail.

To get around this, we need to remind VCC that everything inside state->outer (which is wrapped) is inside its domain, and hence that it can't be modified by a call to a different object. The annotation \wrapped_with_deep_domain does exactly that, and if we rewrite the above code like this

```
_(assert \wrapped_with_deep_domain(&state->outer))
_(assert (&state->outer)->hashState == repeat(0x0,0))
int res = s2n_hash_update(&state->inner, in, size);
_(assert (&state->outer)->hashState == repeat(0x0,0))
```

then it verifies. Like in previous example, we haven't added any new information, we've just reminded VCC of what it already knows.

3.5 Ghosts

Sometimes, it's useful to pretend that there exist objects which aren't actually there. For example, if we have a complicated data structure that is used to compute the abstraction of a set, then it's helpful to have a field that actually is that set because this gives you a direct way to talk about it. As this field wouldn't be seen by the compiler, we can write it as a ghost. We have already seen an example of a ghost field that is common to every VCC object in section 3.3: `\closed`, which keeps track of whether that object is closed or not.

Ghosts can also be used to keep track of the values of objects and their fields at particular points during execution. This could come in handy if we want to reason about how an object behaves in the middle of a function's body - for example, we may want to check that a function call correctly modifies an object which we passed as an argument, or that it doesn't modify an object which shouldn't get updated by the called function. As an example, we show how we could rewrite the last example from section 3.4 using a ghost state:

```
_(assert \wrapped_with_deep_domain(&state->outer))
_(ghost \state s = \now())
int res = s2n_hash_update(&state->inner, in, size);
_(assert (&state->outer)->hashState == \at(s,(&state->
   outer)->hashState))
```

A state is the set of values that all the function's objects and fields take at a particular time during execution. So the meaning of the above assertion is quite intuitive: the value of `(&state->outer)->hashState` is exactly what it was before the call to `s2n_hash_update`.

3.6 Block contracts

Sometimes, when verifying large functions, verification can be quite slow even when the assertions should be relatively simple to check. This is because VCC is provided with a lot of irrelevant information which only slows it down. We can overcome this problem by dividing the code into blocks.

A block behaves exactly like a call to another function: we provide it with a contract, and when VCC is verifying what's outside of the block, it only looks at the block's contract. The advantage is that we can tell VCC which information to use, effectively filtering out everything that's irrelevant.

The example for this section also comes from s2n_hmac.c and only serves the purpose of illustrating how a block works. Rather than trying to understand what's inside the block (which will become clearer after reading section 4.3), readers should note the syntax of the block (i.e. a contract and a body), and that if something (such as state->alg) isn't part of the block's writes clause, then VCC knows that its value after the block is the same as what its value was before the block (hence the assertion after the block verifies):

```
_(ghost \state s= \now())

_(requires hash_alg == hmac_to_hash(alg))
_(writes \extent(&state->inner_just_key), \extent(&
   state->outer))
_(ensures \wrapped(&state->outer) && \wrapped(&state->
   inner_just_key))
{
    s2n_hash_init(&state->outer, hash_alg);
    s2n_hash_init(&state->inner_just_key, hash_alg);
}

_(assert state->alg == \at(s,state->alg))
```

Chapter 4

Verifying the HMAC construction in VCC

4.1 Nums

To carry out our proof we create and use the abstract object `Num`. We would like nums to be like structs, but seeing we only use them abstractly, we can define them as records. The main difference between structs and records is that objects of the latter type don't occupy a specific memory location. The advantage is that we are allowed to use the == command to compare two records, where we get 1 if every entry in each of their fields is the same.

```
_(typedef _(record) struct Num {
    \natural len;
    uint8_t val[\natural];
}Num;)
```

We would like `nums` to have every entry outside of their range equal to zero, so we also define the boolean

```
_(def \bool valid_num(Num n1)
{
    return (\forall \natural i; i>= n1.len ==> n1.val[i
       ]==0x0);
})
```

and will make sure that the nums which keep track of what's going on are valid. This is all we need to define our first function on nums, which simply takes two nums and XOR's them entry by entry[1]:

```
_(def Num xor(Num n1, Num n2)
```

[1] for ease of reading, in this and the following example, we don't include the type-casting

```
_(requires n1.len == n2.len)
_(requires valid_num(n1) && valid_num(n2))
_(ensures valid_num(\result))
    {
        return n1 / {.val = (\lambda \natural i; i<n1.len?
            n1.val[i] ^ n2.val[i] : 0)};
    }
)
```

The last line might seem a bit strange - why return a num that includes a substitution, rather than first modifying a num and then returning it? The reason is that as records don't occupy specific memory locations, they can't be passed to functions to be updated.

As we will see, in `s2n.hmac.c`, we will first XOR each entry in an array with `0x36`, and then later with `0x6a`. As stated in section 2.3, the reason for this second number is that, as the xor operation is idempotent and associative, XOR'ing with `0x36` and then with `0x6a` is the same as directly XOR'ing with `0x5c`. VCC knows that the XOR operation is idempotent but not that it is associative - hence, we need to write a ghost function which we will use as a theorem[2]:

```
_(def void xor_ass()
    _(ensures \forall uint8_t a,b,c; a^(b^c) == (a^b)^c)
{
    _(assert {:bv} \forall uint8_t a,b,c; _(unchecked)((_(
       unchecked)(a^b)^c))==_(unchecked)((a^_(unchecked)(b
       ^c))))
})
```

The assertion in the function body uses two things we haven't yet seen: bitvector reasoning (indicated by {`:bv`}), and the `unchecked` command. The first simply asks VCC to prove the assertion using boolean reasoning at the level of bits. The second is used to assure VCC that no overflow occurs: operations are evaluated modulo the size of the type. In this case, as all we are doing is XOR'ing, no overflow can occur and we use `unchecked` we need to remind VCC of this. We also define the following functions to operate on nums (for full definitions, see [12]):

- `make_num`: makes a valid num from a given pointer to `uint8_t` and a size;
- `repeat`: given a value and a size, makes a valid num composed of that value repeated throughout the array range;

[2] type-casting removed for better readability

- concatenate takes two valid nums and places the second one after the first one to form a new valid num;
- deconcatenate takes a valid num and removes everything in the num whose index is less than a given non-negative integer;
- num_resize replaces entries / appends zeros to resize a valid num, keeping it valid.

4.2 s2n_hash.h

The main object used here is the struct s2n_hash_state. s2n_hash_state has a concrete field alg which keeps track of the hash algorithm being used (which can be MD5, SHA1, SHA224, SHA256, SHA384 or SHA512), and a concrete field called hash_ctx, which is a union of various structs called 'contexts' which are used by OpenSSL functions. As the various members of a union all occupy the same memory location, they can't all be usable at the same time, so we have an invariant specifying which union member should be the active one according to which hash algorithm is being used.

To each context we add a ghost Num field val which stores the abstract value of that context, and a ghost boolean field valid that indicates whether it's usable or whether it needs to be reset before being used. We also add a ghost Num field hashState to s2n_hash_state to keep track of the hash state's abstract value, and by means of an invariant require that the context's abstract value (val) be the same as the hash state's one (hashState). The contexts are modified by calling the following functions from OpenSSL[3]:

- init: initialises a context, which we keep track of by setting its valid flag to 1, setting its abstract value to a Num of length zero, and wrapping it;
- update: takes a valid, wrapped context and appends some data, which we keep track of by appending that data to the context's abstract value. As we keep the context valid and wrapped, we note that this function can be called repeatedly;
- final: hashes a valid, wrapped context onto a string and destroys the context. We keep track of this by not saying anything about its valid flag in its postconditions (when validity appears in a precondition, that precondition is

[3] the original functions have prefixes depending on which hash algorithm is being used

always that the object be valid, so for verification purposes, not knowing an object is valid is just as good as knowing it's not valid). We note that we keep the context wrapped, but will only explain why later in this section. To keep track of what's happened to the string we've hashed the context onto, we use the function

```
_(ghost _(pure) Num hashVal(Num state,
   s2n_hash_algorithm alg)
    _(ensures \result.len == alg_digest_size(alg))
      )
```

We never define this function's body, because we're not interested in what it does - we just care about how the s2n functions use it. This function will only be used in ghost statements and doesn't modify any concrete fields - VCC requires such functions to have a `pure` tag.

As we will see in `s2n_hmac_init`, it will be useful to be able to temporarily wrap a hash state in order to speed up verification. We would like to be able to do this without having to modify the concrete fields, so we only require the invariants of the hash state to hold if the `valid` flag is set to 1. This will allow us to wrap it without having to worry about its invariants failing, and hence without having to modify its concrete fields.

The `s2n_hash.h` file has functions that mirror the OpenSSL ones:

- `s2n_hash_init` initialises a hash state, which we keep track of by setting its `valid` flag to 1 and by wrapping it;
- `s2n_hash_update` updates the abstract value of a valid and wrapped hash state by appending some data to it. As it keeps the hash state usable and wrapped it can be called repeatedly;
- `s2n_hash_digest` hashes a usable and wrapped hash state to some given string, thus destroying the hash state's context. We keep track of this by setting the hash state's abstract value to a `Num` to length 0, and by not including its `valid` flag in the postconditions. We keep the hash state wrapped, and to keep track of what's happened to the string we hashed the hash state onto, we again make use the `hashVal` function (see discussion about contexts earlier in this section).

We noted earlier that digesting a hash state doesn't maintain it as valid. Does this mean that it can no longer be used? The answer is: no. However, we need a function to make it usable again. Hence, the file also includes the function s2n_hash_reset, which takes a wrapped hash state, completely unwraps it and then initalises it again, making it usable. This is why we keep hash states (resp. contexts) wrapped after having passed them to s2n_hash_digest (resp. final) - it allows us to pass them to s2n_hash_reset.

4.3 s2n_hmac.h

The file is written to handle both the most recent OpenSSL implementation and the sslv3 one. We have proved functional correctness for both, though here we only describe the former. This is partially because conceptually they are similar, and partially because the sslv3 implementation is now pretty much obsolete.

The main object used in this file is s2n_hmac_state. Its concrete fields include three hash states inner, outer and inner_just_key; a field alg to keep track of the algorithm being used, an array xor_pad to store K'$\oplus$ opad (see section 2.2) and an array digest_pad which is used to temporarily hash the abstract states of its hash states onto.

The structure of the file also mirrors the OpenSSL functions, with a function to initialise an HMAC state (s2n_hmac_init), a function to append new data to it (s2n_hmac_update) and a function to hash it onto a string. Just like in hash states, HMAC states have a boolean field valid, and just like in s2n_hash.h, initialising an HMAC state makes it valid, updating it keeps it valid, digesting it doesn't maintain its validity, and resetting it makes it valid again.

The abstract value of an HMAC state is stored in the ghost Num field message. Its validity is tied to the validity of its hash states via an object invariant saying that inner_just_key is always valid, and inner is valid if the HMAC state is. We don't need to impose any restrictions on outer's validity because s2n_hmac_update doesn't use it, and s2n_hmac_digest begins by resetting it - so even if it isn't valid on function entry, it is for the rest of the function's body.

Finally, we give HMAC states the ghost Num field key, which stores the private key with which the HMAC state was initialised. So the complete definition (excluding anything related to the sslv3 implementation) of s2n_hmac_state is [4] :

```
#define _IPAD repeat(0x36,block_size)
#define _OPAD repeat(0x5c,block_size)
#define _Kprime num_resize(key,block_size)
```

[4] where possible, we have used macro variables to respect the notation from 2.2

```
#define _Kprime2 num_resize(hashVal(key,hmac_to_hash(alg))
   ,block_size)
#define _L block_size

#define IPAD repeat(0x36,state->block_size)
#define OPAD repeat(0x5c,state->block_size)
#define Kprime num_resize(state->key,state->block_size)
#define Kprime2 num_resize(hashVal(state->key,hmac_to_hash
   (state->alg)),state->block_size)
#define H(NAME) hashVal(NAME ## ,hmac_to_hash(state->alg))
#define H1(NAME) hashVal(NAME ## ,hmac_to_hash(state->alg)
   )
#define L state->block_size

#define WRAPPED_VALID_HMAC_STATE \wrapped(state) && state
   ->valid
#define UNCHANGED_HMAC_ALG_KEY \unchanged(state->alg) && \
   unchanged(state->key)

struct s2n_hmac_state {
    s2n_hmac_algorithm alg;

    uint16_t hash_block_size;
    _(invariant hash_block_size==hash_block_size_alg(alg))
    uint32_t currently_in_hash_block;
    uint16_t _L;
    _(invariant _L == block_size_alg(alg))
    uint8_t digest_size;
    _(invariant digest_size == digest_size_alg(alg))

    _(ghost Num key)
    _(invariant valid_num(key))

    _(ghost \bool valid)
    _(invariant valid ==> (&inner)->valid)
    _(invariant (&inner_just_key)->valid)

    struct s2n_hash_state inner;
    struct s2n_hash_state inner_just_key;
    struct s2n_hash_state outer;

    _(ghost Num message)
    _(invariant valid ==> valid_num(message))
    _(invariant valid ==> concatenate((&inner_just_key)->
       hashState,message) == (&inner)->hashState)
    _(invariant key.len <= _L && alg  ==> (&inner_just_key
       )->hashState == xor(_Kprime,_IPAD))
    _(invariant key.len > block_size && alg  ==> (&
       inner_just_key)->hashState == xor(_Kprime2,_IPAD))
    _(invariant !alg ==> (&inner_just_key)->hashState ==
       repeat(0x0,0))
    _(invariant (&outer)->hashState == repeat(0x0,0))

    uint8_t xor_pad[128];
    _(invariant key.len>_L && alg ==> make_num(xor_pad,_L)
        == xor(_Kprime2,_OPAD))
    _(invariant key.len>_L && !alg ==> make_num(xor_pad,_L
       ) == _OPAD)
    _(invariant key.len<=_L && !is_sslv3(alg) ==> make_num
       (xor_pad,_L) == xor(_Kprime,_OPAD))

    uint8_t digest_pad[SHA512_DIGEST_LENGTH];
    _(invariant alg>=0 && alg <= 8)
    _(invariant valid==>(&inner)->alg == hmac_to_hash(alg)
       )
    _(invariant (&inner_just_key)->alg == hmac_to_hash(alg
       ))
    _(invariant (&outer)->alg == hmac_to_hash(alg))
    _(invariant \mine(&inner) && \mine(&outer) && \mine(&
       inner_just_key))
};
```

4.4 s2n_hmac_init.c: full VCC proof

This function takes an unitialised HMAC state (state), a hash algorithm (algorithm) and a key (key). See section 2.3 for an overview of what it does to the HMAC's concrete fields. Here is the full[5] annotated code fors2n_hmac_init:

```
int s2n_hmac_init(struct s2n_hmac_state *state,
   s2n_hmac_algorithm alg, const void *key, uint32_t klen)
    _(requires \wrapped(\domain_root(\embedding((uint8_t
       *)key))))
    _(requires \thread_local_array((uint8_t *)key,klen))
    _(requires is_valid_hmac(alg))
    _(writes \extent(state))
```

[5] although the code is written to handle many different hash algorithms, to save space here we assume the algorithm is SHA1 and remove all irrelevant lines of code

```
    _(ensures state->message == repeat(0x0,0))
    _(ensures \result == 0)
    _(ensures \wrapped(state))
    _(ensures state->alg == alg)
    _(ensures state->key == make_num((uint8_t *)key,klen))
    _(ensures state->valid)
    _(decreases 0)
{
    _(ghost xor_ass()) //Used to establish that the XOR
       operation is associative
    s2n_hash_algorithm hash_alg = S2N_HASH_NONE;
    state->currently_in_hash_block = 0;
    state->digest_size = 0;
    state->block_size = 64;
    state->hash_block_size = 64;
    switch (alg) {
    case S2N_HMAC_SHA1:
        hash_alg = S2N_HASH_SHA1;
        state->block_size = 40;
        state->digest_size = SHA_DIGEST_LENGTH;
        break;
    /*other cases removed for this example*/
    default: _(assert 0) /*this asks VCC to verify that
       execution will never reach this point */
    }

    state->alg = alg;

    /*We put initialisation inside a block in order to
       speed up verification*/
    _(requires hash_alg == hmac_to_hash(alg))
    _(writes \extent(&state->inner_just_key), \extent(&
       state->outer), \extent(&state->inner))
    _(ensures \wrapped(&state->outer) && \wrapped(&state->
       inner_just_key) && \wrapped(&state->inner))
    _(ensures (&state->outer)->alg == hmac_to_hash(alg))
    _(ensures (&state->inner_just_key)->alg ==
       hmac_to_hash(alg))
    _(ensures (&state->outer)->valid && (&state->
       inner_just_key)->valid)
    _(ensures (&state->outer)->hashState == repeat(0x0,0))
    _(ensures (&state->inner_just_key)->hashState ==
       repeat(0x0,0))
    {
        s2n_hash_init(&state->outer, hash_alg);
    s2n_hash_init(&state->inner_just_key, hash_alg);
    _(ghost (&state->inner)->valid = 0)
    _(wrap &(&state->inner)->hash_ctx)
    _(ghost (&state->inner)->\owns = {&(&state->inner)
       ->hash_ctx})
    _(wrap (&state->inner))
}

uint32_t copied = klen;

/*Copy K (or H(K), if K's length is greater than L) to
    state->xor_pad*/
if (klen > state->block_size) {
    s2n_hash_update(&state->outer, key, klen);
    _(assert (&state->outer)->hashState == make_num((
       uint8_t*)key,klen))
    s2n_hash_digest(&state->outer, state->digest_pad,
       state->digest_size);
    memcpy(state->xor_pad, state->digest_pad, state->
       digest_size);
    copied = state->digest_size;
} else {
    memcpy(state->xor_pad, key, klen);
}
_(ghost \state t = \now()) /*used later to remind VCC
   that key hasn't changed*/

for (int i = 0; i < (int) copied; i++)
    _(writes \array_range(state->xor_pad,copied))
    _(invariant i>= 0 && i<= (int)copied)
    _(invariant \forall int j; j>=0 && j<i ==> state->
       xor_pad[j] == \old(state->xor_pad[j]^(uint8_t)0
       x36))
    _(invariant \forall int j; j>=i && j<(int)copied
       ==> state->xor_pad[j] == \old(state->xor_pad[j
       ]))
{
    state->xor_pad[i] ^= 0x36;
}
for (int i = (int) copied; i < state->block_size; i++)
    _(writes \array_range(state->xor_pad,state->
       block_size))
    _(invariant \forall int j; (j>=0 && j<(int)copied
       || j>=i && j<state->block_size) ==> \unchanged(
       state->xor_pad[j]))
    _(invariant \forall int j; j>=(int)copied && j<i
       ==> state->xor_pad[j]== 0x36)
    _(invariant i>=(int)copied && i<= state->
       block_size){
    state->xor_pad[i] = 0x36;
}
/*At this point, state->xor_pad = K' XOR ipad*/
s2n_hash_update(&state->inner_just_key, state->xor_pad
   , state->block_size);

for (int i = 0; i < state->block_size; i++)
_(writes \array_range(state->xor_pad,state->block_size
   ))
_(invariant i>=0 && i <= state->block_size)
_(invariant \forall int j; j>=0 && j<i ==> state->
   xor_pad[j] == \old(state->xor_pad[j]^(uint8_t)0x6a)
   )
_(invariant \forall int j; j>=i && j<state->block_size
    ==> state->xor_pad[j] == \old(state->xor_pad[j]))
{
    state->xor_pad[i] ^= 0x6a;
}

/*As 0x6a = 0x36 XOR 0x5c, at this point state->
   xor_pad = K' XOR opad. The following two assertions
    are necessary, as they provide VCC with the hints
   necessary for it to deduce that state->xor_pad = K'
    XOR 0x5c*/
_(assert make_num(state->xor_pad,L) == xor(num_resize
   (\at(t,make_num(state->xor_pad,copied)),L),repeat((
   uint8_t(0x36^0x6a),L)))
_(assert make_num(state->xor_pad,L) == xor(num_resize
   (\at(t,make_num(state->xor_pad,copied)),L),OPAD))

_(assert make_num((uint8_t *)key,klen) == \at(t,
   make_num((uint8_t *)key,klen)))

_(ghost
{
    state->key = make_num((uint8_t *)key,klen);
    state->valid = 0;
    state->message = repeat(0x0,0);
    state->\owns = {&state->inner, &state->
       inner_just_key, &state->outer};
})
```

```
    _(assert \inv(state))
    _(wrap state)
    return s2n_hmac_reset(state);
}
```

When wrapping the `s2n_hmac_state`, VCC needs to successfully verify that its invariants hold. As the `valid` flag is set to zero, it only has to check that:

- the `inner_just_key` and `outer` hash states are valid: as they were initialised with `s2n_hash_init` which, as we described in section 4.2, makes its argument valid, this holds;
- the abstract value (`hashState`) of `inner_just_key` is `K'`⊕`ipad`: this holds because when we set its value to be equal to `xor_pad`, that is exactly what the latter was equal to;
- `xor_pad` equals `K'`⊕`opad` - this holds as `0x36`⊕`0x6a = 0x5c`;
- `state` owns its three hash states - this holds, as we set it as their owner just before the wrap;
- `state->inner_just_key` and `state->outer`'s algorithms are the same as the `state`'s - this holds, as we initialised them with it.

So the wrap succeeds. Next, we return `s2n_hmac_reset`, so we first need to check that its preconditions hold. Here is the contract of `s2n_hmac_reset`:

```
int s2n_hmac_reset(struct s2n_hmac_state *state)
    _(maintains \wrapped(state))
    _(writes state)
    _(ensures state->message == repeat(0x0,0))
    _(ensures UNCHANGED_HMAC_ALG_KEY)
    _(ensures \result == 0)
    _(ensures state->valid)
    _(decreases 0)
;
```

The only precondition this function has is that the HMAC state be wrapped, and we have just checked that the wrap succeeds, so this holds. Therefore, we get to assume its postconditions. We note that `s2n_hmac_reset` writes `state`, and hence none of the `state`'s fields should be assumed to stay the same unless it is explicitly mentioned in the contract that they do. Now, we need to check that the postconditions of `s2n_hmac_init` hold. Looking back at its, the postconditions that we need to check are:

- state is wrapped, state->_M is blank (i.e. a Num of length 0) and state is valid: these all hold because they are postconditions of s2n_hmac_reset;
- state->alg = algorithm - this was true when we exited s2n_hmac_init, and as s2n_hmac_reset has the postcondition that it leaves state->alg unchanged, it must still be true;
- state->key = key - this was true before exiting s2n_hmac_init, and as s2n_hmac_reset has the postcondition that it leaves state->key unchanged, it must still be true.

4.5 Testing s2n_hmac.h

We've looked at what the s2n HMAC functions do, but how would we use them if we wanted to construct an HMAC of a message? For the purpose of the next example, we will use a structure which we haven't defined yet: s2n_blob. We don't go into details here (though include its full definition in our Github repository) - it's enough to note that it's some chunk of memory whose size we keep track of. Here's an example, where we initialise three blobs: key (with which we initialise an HMAC state), message (the message which we would like to compute an HMAC of) and outt (onto which we write the hmac). Specifically, this is what we will do:

1. we allocate strings of data and uninitialised blobs;
2. we initialise those blobs (using the function s2n_alloc, from s2n_mem.h, which we haven't verified as part of this project but which we include a contract for in our Github repository);
3. we initialise an HMAC state with the blob key;
4. we update the HMAC state with the blob message;
5. we hash the the HMAC state onto the blob outt.

```
int testing()
{
    /*initialise the data with which to initalise the
       blobs, which will be used to initialise, update and
        store digested data
    from hmac states*/
    uint32_t size = 50;
```

```
uint32_t size1 = 20;
uint32_t size2 = 20; //constrained to be equal to the
   digest size corresponding to the algorithm

/*initialise blobs*/
struct s2n_blob *key = malloc(sizeof(*key));
if(key==NULL) return -1;
s2n_alloc(key,size);
struct s2n_blob *m = malloc(sizeof(*m));
if(m==NULL) return -1;
s2n_alloc(m,size1);
struct s2n_blob *outt = malloc(sizeof(*outt));
if(outt==NULL) return -1;
s2n_alloc(outt,size2);

/*choose an algorithm, initalise hmac state*/
struct s2n_hmac_state *state = malloc(sizeof(*state));
if(state==NULL) return -1;
s2n_hmac_algorithm r = S2N_HMAC_SHA1;
_(assert \wrapped_with_deep_domain(key)) //used to
   prove that key->data and key->size form a thread-
   local array
GUARD(s2n_hmac_init(state,r,key->data,key->size));
_(assert key->size>L ==> M == repeat(0x0,0))

/*update with data (can be called repeatedly*/
_(assert \wrapped_with_deep_domain(m))
GUARD(s2n_hmac_update(state,m->data,m->size));
_(assert \inv(state))
_(assert key->size>L ==> M == make_num(m->data,m->size
   ))

/*digest HMAC of message*/
_(assert \wrapped_with_deep_domain(outt))
_(unwrap outt)
_(unwrap blob_data(outt))
GUARD(s2n_hmac_digest(state,outt->data,outt->size));
_(assert \inv(state))
_(assert key->size>L ==> make_num(outt->data,outt->
   size) == H1(concatenate(xor(Kprime2,OPAD),H(
   concatenate(xor(Kprime2,IPAD),M)))))
_(assert key->size<=L ==> make_num(outt->data,outt->
   size) == H1(concatenate(xor(Kprime,OPAD),H(
   concatenate(xor(Kprime,IPAD),M)))))
/* what we end up writing to outt with can be seen to
```

```
        be equal to H((K' XOR 0X5C) || H((K' XOR 0X36) || M
        )),
      where M is the message, K' is the key (hashed or with
        0s appended to be the same length as the digest
        size), and
      H is the hash function. */
}
```

If we try verifying this function with VCC, verification succeeds. Hence we have proved that s2n correctly implements the HMAC protocol, and that it does so in an error-free way.

4.6 s2n_digest_two_compression_rounds: an potential bug?

A potential problem with s2n_hmac_digest is that, depending on the space left in the HMAC state's current hash block, the time that the function takes to run differs and may be exploited by hackers when attempting to perform a timing attack. s2n_hmac_digest_two_compression_rounds overcomes this problem by passing the inner hash state to s2n_hash_update if the size of current hash block is too small, thus taking up time with what is effectively a useless operation.

Because of the way the function is written, our verification would not (and could not) succeed. This is because, before potentially passing the inner hash state to s2n_hash_update, it passes the HMAC state to s2n_hmac_digest. As mentioned in section 4.2, s2n_hash_update requires its argument to be valid - however, there is no guarantee that this is the case after the call to s2n_hmac_digest, which doesn't keep its argument valid.

As it turns out, closer analysis of the code revealed that this isn't a problem in the current implementation of the HMAC construction - however, it might have been a problem in a different implementation

Conclusion

We have proved that s2n correctly implements the HMAC construction, and that the implementation is free of bugs. We stress that we have not proved that the HMAC construction is safe - just that the s2n code implements it correctly. We have also found a potential bug in the function `s2n_hmac_two_compression_rounds`, which fortunately turned out to not be a problem in the current implementation. Time-permitting, we could have verified all the `utils` functions, which include `s2n_blobs` which we use in `2n_hmac_testing` and which make up an important part of s2n.

Overall, VCC is not a hard tool to use, and only requires that users fully understand the code they're verifying and that they are familiar with first-order logic. It is the second attribute, however, that software engineers may sometimes lack. Nonetheless, mathematics students who are not programmers (like me) can pick it up and become productive verifiers in a matter of months.

Acknowledgements

Many thanks to Ernie Cohen for his invaluable contribution to helping me learn to use VCC and in provoding general guidance for the project, Michael Brain for a useful conversation about program verification as a whole, and to my supervisor Daniel Kroening.

Appendix

Pointers

When a variable is passed to a function and that function updates it, then as soon as we exit that function, the update it lost. This is because the function only modifies a local copy of the variable. For example, if we write

```
void make_it_zero(int a)
{
    a=0;
}

int main()
{
    int x=1;
    make_it_zero(x);
}
```

then, as far as the current function is concerned, x will still be equal to 1 after the function call.

If we want to make a permanent update, we need to pass a pointer to some memory address - this way, the function will update the physical location of the variable and hence the update won't be limited to the scope of the function. We indicate a pointer with * and its inverse operation, which returns the address of a variable, with &. So to make the above example work as we would like it to, we can change it as follows:

```
void make_it_zero(int *a)
{
    a=0;
}

int main()
{
    int x=1;
    make_it_zero(&x);
}
```

Structs and unions

Primitive types include `int, double, float` and many more, such as `uint8_t`, which stores a byte. These types are great for storing individual values, but don't allow us to store multiples values together. Say we have an array and want to keep track of how much of it we're using. A simple way to achieve this would be to write a struct:

```
struct my_struct {
        uint32_t size;
        uint8_t *data;
    };
```

If we compare with with `s2n_blob`, as seen in section 3.4, we will notice that they do the same thing. A union is similar, except that only one of its members is allowed to occupy memory at a given time. For example,

```
struct my_struct {
        uint32_t tag;
        union my_union{
        my_struct1 x;
        my_struct2 y;
        my_struct3 z;
        };
    };
```

We could then use a tag to specify which member of the union we would like to be active, just like do in `s2n_hash_state` (see our Github repository for full definition).

Bibliography

[1] Periklis Akritidis. Practical memory safety for C. Technical Report UCAM-CL-TR-798, University of Cambridge, Computer Laboratory, June 2011.

[2] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. *Boogie: A Modular Reusable Verifier for Object-Oriented Programs*, pages 364–387. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[3] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The spec# programming system: An overview. In *Proceedings of the 2004 International Conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, CASSIS'04, pages 49–69, Berlin, Heidelberg, 2005. Springer-Verlag.

[4] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '96, pages 1–15, London, UK, UK, 1996. Springer-Verlag.

[5] Martin Brain and Daniel Kroening. The Software Verification Pyramid. *Philosophical transactions A*. under submission.

[6] Ernie Cohen, Mark A. Hillebr, Stephan Tobies, Micha Moskal, and Wolfram Schulte. Verifying C Programs: A VCC Tutorial, 2011.

[7] Ernie Cohen, Michal Moskał, Wolfram Schulte, and Stephan Tobies. A Practical Verification Methodology for Concurrent Programs. Technical Report MSR-TR-2009-2019, February 2009.

[8] Ernie Cohen, Michał Moskal, Wolfram Schulte, and Stephan Tobies. Local Verification of Global Invariants in Concurrent Programs. In *Proceedings of the 22Nd International Conference on Computer Aided Verification*, CAV'10, pages 480–494, Berlin, Heidelberg, 2010. Springer-Verlag.

[9] Cox, Mark, 2014. https://plus.google.com/+MarkJCox/posts/TmCbp3BhJma [Online; accessed May 3, 2016].

[10] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[11] T. Dierks and E. Rescorla. The transport layer security (TLS) protocol. In *IETF RFC 4346*, 2006.

[12] Marco Gorelli. VCC proofs, 2015. https://github.com/Gorelli/Dissertation [Online; accessed August 30, 2016].

[13] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988.

[14] Gerwin Klein. Operating system verification — an overview. *Sādhanā*, 34(1):27–69, February 2009.

[15] National Institute of Standards and Technology. 2 software errors cost U.S. economy $59.5 billion annually, 2002. , http://www.nist.gov/publicaffairs/releases/n02-10.htm [Online; accessed August 30, 2016].

[16] OpenSSL Software Foundation. Openssl, 2015. https://www.openssl.org/ [Online; accessed May 3, 2016].

[17] Shelat Pass. *A Course in Cryptography*. 2010. note = https://www.cs.cornell.edu/courses/cs4830/2010fa/lecnotes.pdf [Online; accessed August 1, 2016].

[18] Microsoft Research. The HAVOC property checker. http://research. microsoft.com/projects/havoc [Online; accessed August 30, 2016].

[19] Schmidt, Stephen. Introducing s2n, a New Open Source TLS Implementation, 2015. https://blogs.aws.amazon.com/security/post/TxCKZM94ST1S6Y/Introducing-s2n-a, [Online; accessed May 3, 2016].

[20] Amazon Web Services. s2n, 2015. https://github.com/awslabs/s2n/ [Online; accessed August 30, 2016].