On the Use of Cryptol, a Cryptography Domain Specific Language

WILLIAM KOVACS

ACM Reference Format:
William Kovacs. 2017. On the Use of Cryptol, a Cryptography Domain Specific Language. 1, 1 (December 2017), 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 SUMMARY

The implementation of cryptographic algorithms faces numerous issues when being implemented in common languages, particularly due to the lack of constructs that lead to a more intuitive translation from idea to code. This can lead to issues when code is used as a reference for a particular algorithm due to the difficulty of parisng the meaning of the code itself. To combat this issue, Cryptol has been developed so that code better mirrors the functioning of the algorithm itself, as opposed to more abstruse code. In order to evaluate the utility of this language, I implemeted a series of increasingly difficult cryptographic algorithms (the Viginere cipher, the SHA1 hash function, and the DES symmetric key cipher) in both Cryptol and C. As with learning any new language paradigm, getting Cryptol code to work was initially difficulty, but once it became comfortable, it felt better than C. Its utility becomes increasingly apparent with code sections that require single bit manipulations, such as the permutation of a bit string. However, such a language is, as expected of a functional language, mostly useful in research or verification of other implementations due to the speed tradeoff, and the opacity of the language constructs.

2 BACKGROUND

Cryptographic algorithms require operations that can be rather difficult to succintly capture in many programming languages. Such an issue is best exemplified with any form of bit manipulation, where the programmer needs to manage the bits of a particular variable as opposed to the bytes. While it is possible to achieve this in most languages, its implementation can look a bit unwieldy. For example, when given a string to encrypt in C, it is necessary to employ shift operators to access the requisite bit. It would be possible to implement a bit string as its own array of chars to improve access to the bits, but such a representation would introduce its own difficulties when the bits need to actually represent a number, such as an index as in DES or an integer to be summed.

In order to construct more understandable representations of this class of algorithms, Galois Inc. developed a domain specific language (DSL) known as Cryptol[Galois, Inc. 2016], accompanied by a tool suite that can be used to verify the implementation of an algorithm written in another language, like C. To be clear, the intention behind this language is not to be used as a production ready implementation, but rather as an alternative that provides clarity in code coupled with a provably functional implementation. Such a system would be useful when prototyping, verifying, and creating a reference for these types of algorithms, which is in contrast with other DSLs, such as Tensorflow, that are meant for production usage.

The purpose of this project is to evaluate the utility of using such a DSL to implement cryptogtaphic functions, particularly by comparing it with a corresponding implementation in C, with the hypothesis being that using Cryptol provides an easier alternative to quickly implementing such a function, and that such an implementation is more easily understandable. As such, this relates to two of the themes of the course: expressiveness and correctness. Expressiveness clearly relates to this project because a major distinction between these two languages is the ability to represent the requisite operations and its impact on the ability to both generate and read the different code. Meanwhile, correctness is a major distinction between these two types of languages because while C can have a variety of issues with its pointers and explicit memory management, Cryptol is a strongly typed, functional language based on Haskell that does not encounter such issues. Furthermore, as mentioned above, it does offer a variety of tools to allow easier access to proving the correctness of these algorithms, though this is not touched upon in the constructed implementations. The third theme, performance, is not considered herein due to the purpose of Cryptol: its more focused on readability and correctness rather than speed (this is mentioned due to the noticeably longer time to run an iteration of the algorithm in Cryptol compared with those in C).

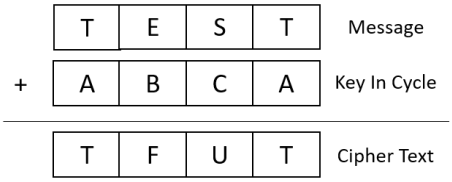
Before describing the project in further details, it is helpful to discuss a major distinction between the two languages that contributes to the aforementioned differences: the type system. In C, the smallest representation of data that you can reasonable work with is 8 bits, in a char (bitfields are excluded due to their availability only in structs, which makes it an untenable approach for this use case). However, in Cryptol, the smallest value you can have is a single bit (represented by "True" or "False"), and most other values are interpreted as sequences of these bits. For example, the character "a" is equivalent to [F,T,T,F,F,F,F,T]. This idea can then be extended to strings, which would be a sequence of such a sequence of bits. This particular difference is one that contributes a lot to the readability and ease of programming in Cryptol, as will be discussed later.

3 APPROACH

Two things needed to be accomplished within the scope of this project. The first is to learn a new language, in this case Cryptol itself. Cryptol is a functional language built on top of Haskell, and as such, requires a different paradigm of thinking about programming. Of particular note, is the strictness of the type system in which there can be no ambiguity of what goes in or out of a function. This translates to knowing the exact lengths of these input/output sequences, which took time to recognize. In order to acclimate myself to these changes, I read through the beginning section of the Cryptol documentation[Galois, Inc. 2016] to get a sense of how it

Author's address: William Kovacs.

Fig. 1. Example of Vigenere Cipher with Message "TEST" and key "ABC"



works. To facilitate the learning process while achieving the next goal is to implement the algorithms in an increasingly difficult order.

The second goal is to perform a comparison of the effectiveness, in terms of expressivity and readability, of the two languages, Cryptol and C, in implementing working representations of cryptographic algorithms. To this end, I implemented three sets of algorithms with increasing difficulty, with each one having a focus on different types of operations used. The order of Cryptol then C is important because it would be expected that the initial attempt would be more difficult due to the novel introduction. With the hypothesis suggesting that Cryptol provides an easier interface, if I still found it easier to construct something in Cryptol after working on it in the more difficult setting, then it is more likely to attribute it to the language than the familiarity of implementing the algorithm. As for the actual algorithms, the first was the Viginere cipher, followed by the SHA1 hash function, and finally the DES cipher. The reasoning behind these and overviews of the algorithms themselves will be described in the following sub-sections.

3.1 Vigenere Cipher

The Vigenere cipher is a very simple 'classic' cipher that provides essentially no security this day and age, but provides a useful vehicle to begin learning how to effectively program in this language. Originally, there was going to be a progression of the Caesar cipher followed by the Vigenere cipher; however, as I was following the tutorial in the Cryptol documentation, they provided the code for it as an example, but left the Vigenere cipher to the reader. Seeing as the Vigenere cipher can be viewed as a more complicated Caesar cipher, solely using the Vigenere cipher proved to be an apt introduction.

As with any cipher, in order to encrypt a message, a key is needed. Essentially, the message and the key are aligned. If the key is smaller than the message, it is repeated until each letter of the message is aligned with one of the key. Then, each letter of the message is right-shifted according to the numeric value (a=0, b=1, etc...) of the aligned letter in the key (i.e. if the message's letter is 'c', and the key's is 'b', corresponding to a shift of 1, then the resulting cipher text would be 'd'). See figure 1 for an example of this algorithm. Decryption of this cipher is simply reversing the shift direction based on the key.

3.2 SHA1 Hash Function

In order to provide a broader comparison of cryptographic functions, I have chosen to implement a hash function in addition to the cipher described in the following subsection. While the SHA1 hash function is technically not a secure hash function any more (Google recently found a collision), this function contains the core complex ideas to implement, such as how to pad the message, how to construct the main word array based on the message, and how to set up the recursive hash cycle. That is to say, implementing a stronger variant of this hash like SHA256 mostly results in the change of the initial hash values, longer blocks, and some slight changes to the calculations in the core functionality, which is essentially more copy-paste rather than testing the more difficult sections.

The way I learned about the design of the SHA1 hash function is by reading RFC 3174[Eastlake and Jones 2001]. Because it's more complex details are easily accessible (and the algorithm itself is not a focus of this report), I will only provide a basic overview. It should be noted that this RFC also has corresponding C code, which was not referenced during my own implementation, but the initial hash values were used, as they are a part of the algorithm itself.

To start the algorithm, the message is padded with a '1', followed by '0's, follwed by the message length, such that it can be easily divided into 512 blocks of bits. 80 32-bit values are generated, where the first 16 are simply the current block's bits, and the remaining are a series of XORs of specific prior blocks. The current hash value is represented as 5 blocks. For 80 iterations, the first block's value is modified based on the aforementioned values, and a function (that varies based on the iteration number) of the three following segments. The remaining blocks are then shifted block-wise (with the second block also being shifted bitwise). The remaining values are then combined to form the current hash value, which is then fed into the next block, if any, or used as the final result.

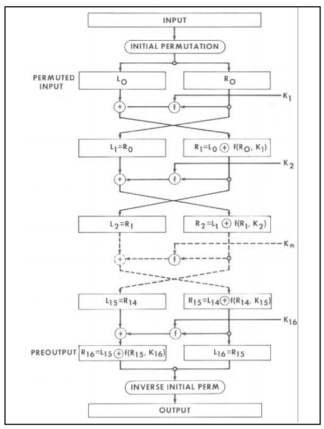
3.3 DES Cipher

The last cryptographic function I implemented was the DES cipher, which, like any cipher, requires both a key and a message to encrypt. As with the prior function, this one also is no longer secure, but still provides a good example comparison between Cryptol and C due to its use of permutations and a unique form of indexing a table dependent on disjoint bits. This was also implemented last as it seemed to be the most complex variant, and so follows the desired difficulty curve. For this section, I was originally going to implement the RSA cipher, but the problem with using that as a comparison is that its difficulty does not rely on the complexity of its operation, but rather the ability to represent large primes and perform exponentiation on them. This would both make it more difficult to test whether I have a working example, while not providing an interesting implementation to work on, though the representation of the large required numbers might have been an interesting concept to compare.

The way I learned about the DES cipher is by reading the federal information processing standards (now outdated due to the insecurity of DES)[National Institute of Standards and Technology 1995] that described this algorithm in detail. Again, I'll only provide a summary of the algorithm and leave the more specific details to the reference. The base algorithm functions on 64 bit blocks of data that can be chained together external to the DES algorithm itself, so I only focus on the core functions.

Before the encryption takes place, the key used to encrypt is broken into 16 different subkeys that are generated via a series of permuatations and left-circular shifts on each half of the key.

Fig. 2. Overview of DES encryption, originally in [National Institute of Standards and Technology 1995]



With these subkeys, encryption of the data can begin. The message first undergoes a permutation, followed by 16 iterations. In each iteration, the left half of the block is XOR-ed with a function of the right hand side and one of the 16 subkeys. This modified half becomes the new right side, while the right side (unmodified in this iteration) becomes the new left side. See figure 2 for clarity. The function of the key and right side involves permuting the message into a 48 bit block, which is XORed with the corresponding subkey. This is then split into 8 6-bit blocks, where the end bits and the four middle bits provide the indices to a table of values that is used to replace them. After all 16 iterations have been performed, a final permutation is performed.

Given the encryption algorithm, decryption is trivial: just run the encryption algorithm with the order of subkeys reversed. This works because the two initial and final permutations of the message are each other's inverse.

4 RESULTS

Overall, this project was successful in the two main goals I set to achieve: to learn how to use Cryptol to implement cryptographic algorithms, as well as garner an understanding of the tradeoffs in using a DSL like Cryptol versus a production ready language like C. At a high level, the barrier of entry to Cryptol was slightly higher than I had anticipated because of its functional paradigm, and the only experience I've had with that so far is OCaml. In particular, understanding how the type system worked took a bit of getting used to. However, once I had become familiar with it, implementing the algorithms proved to be a better experience than C. In the subsequent sections, I will give a more detailed description for each algorithm, wiht a discussion first on the process of learning Cryptol, followed by a comparison with the experience of implementing it in C.

4.1 Vigenere Cipher

Implementing the Vigenere cipher was not intended to be a difficult problem to solve, but rather a simple way to ease into the language, and it proved as much. It allowed me to get a better feeling of how one of Cryptol's core constucts, comprehensions work. While not unique to Cryptol, in fact it's style is seemingly directly taken from the underlying language, Haskell, it fits very cleanly with the desired applications. A comprehension looks very similar to set notation:

```
[shiftL (keyL, messL) | messL <- message | keyL <- (cycle key)]
```

Instead of constructing a set, it builds up an array iteratively, and is used in place of any loops. It takes in letters one at a time from the message and a cycle of the key (generated via lazy evaluation of an infinitely looping function called 'cycle,' implementation not shown), feeds it into the shifting function, and uses that value as the next letter of the cipher text. This style of generating arrays is what I used most in the subsequent implementations, as it seems to be the sole method of constructing arrays (aside from explicit appends). One thing to note about this style is that it is very readable.

Likewise, the C implementation was also fairly easy because C allows chars to be cast to ints without any issues, so obtaining the letter shifts was not a problem. Performing the shift itself required using modular arithmetic as opposed to the use of an infinite cycle, so the two implementations required two different styles but neither were particularly advantageous over the other with such a simple design.

4.2 SHA1 Hash Function

Implementing the SHA1 hash function is where the difficulties of learning a new language, particularly one that is strongly typed, comes into focus. The first major roadblock was implenting the padding function; however, doing so granted a better understanding of the type system. As discussed in Section 2, types are expressed in terms of length, but you have to ensure that there is no ambiguity in the type of a function, i.e. [n]->[m] may be invalid because m could be anything. Instead, a slightly awkward expression is needed: [n] -> [((n+65)/512)*512+512], which means that the input is of size n, and the output is the total size of the 512 bit blocks used after padding (65 referring to the minimum number of bits that are added). While it is a bit awkward, it does force the user to be more aware of what the function is doing, and helps to promote correctness, at least to the degree of getting the lengths correct.

The next confusing segment came with stringing together the hash functions. One of the common patterns in Cryptol is to repeatedly append to the end of an array. This is done through the use of comprehensions, as described in the aforementioned section. However, there are multiple ways to use these; one is to repeatedly construct the full arrays and append that to the end of an array, thereby creating an array of ararys. The other is to do repeatedly construct the elements themselves, then put them in an array after. My first instinct was to construc the arrays each time; however this results in severe slowdown. The SHA1 hash function performs 80 iterations of the core hashing function. However, with my initial implementation, after 12 iterations, there would be a slowdown of 5 seconds, and by 18 iterations, it would perpetually hang. Only by calculating each block of the current hash individually could I make it run in a reasonable time. This points to one of the frustrating things about functional languages: its opaque structure. Even now,

I am still not sure why my initial method caused such a massive slowdown (perhaps a constant reallocation of arrays needed).

The C implementation proved trickier due to its data representation in memory. Again, implementing the padding function is where the initial difficulty began; however, in C, it is because the implementation is not as intuitive as in Cryptol, particularly with appending the length to the end of the padded block. In my implementation, I used strlen() to retrieve the length of the message. Too late did I realize the slight error I made via this method: strlen() returns an unsigned int, which only supports up to 32 bits, but the specification allows for up to 64 bit lengths. While I lack the time to correct this error, the major change needed would be to not use strlen(), but rather crawl over the bits, keeping track of its length in an unsigned long, until a null character is reached. Contrast this approach with that of Cryptol's, whose type system already takes care of retrieving this length. However, I do not know how its memory management is handled in such a case, or even how to test sending over 500 Mb to be hashed. For most reasonable applications, supporting only up to 32 bits is still sufficient though.

Again regarding padding, appending its length to a message requires more memory management (to set up the new space), as well as lots of bit shifting that needs to be done carefully in order to ensure that the endianness of the int representation does not intefere with the transfer (as in the case of a naive transfer). While there were other spots that shared this need for such delicacy in C, this particular example is a solid demonstration of the difference in expressivitiy and readability of the code as seen with the C code below:

```
char* padMessage(char* toPad){
  unsigned int msgLen = strlen(toPad);
  size_t addPad=(512 - (msgLen*8+96) % 512)/8;
  addPad = addPad + msgLen + 96/8;
  unsigned char* padded = calloc(addPad,1);
  int i = 0;
  for(i = 0; i < msgLen; i++){
    padded[i] = toPad[i];
  }
  padded[i]=0x80;
  msgLen*=8;
  //Append length to end
  padded[addPad-4] = (msgLen >> 24) & 0xff;
  padded[addPad-3] = (msgLen >> 16) & 0xff;
  padded[addPad-2] = (msgLen >> 8) & 0xff;
  padded[addPad-1] = msgLen & 0xff;

  return padded;
}
```

and the Cryptol code:

```
padMessage: {n} (fin n, 64 >= width n)
        => [n] -> [((n+65)/512)*512+512]
padMessage msg = msg # [True]
        # (zero:[rem]) # ( (n):[64])
        where type rem = 512 - (n + 65) % 512
```

Clearly, the Cryptol code not only is more succint, but even at a glance, it is understandable what is happening, as long as you know that # is the append operator. As mentioned in the discussion of types, Cryptol represents a bit as either "True" or "False", so trying to append [1] instead of [True] results in type error, which is a slight detriment due to its readibility. Overall though, its clear how the append functions in a more concise manner.

In order to verify my code as I was running it, I referenced example digests to see what the values should be at the corresponding steps provided by NIST[National Institute of Standards and Technology 2017b]. In order to verify my final versions, I compared my output to that of an online SHA-1 generator[SHA [n. d.]], and found that the final implementations match those of the generator. Inputs were tested to stress the different aspects of the algorithm, ensuring that it works for input that fits in 512 bits with the padding, those with multiple blocks, and those with padding causing the block size to be extended.

4.3 DES Cipher

While the SHA1 hash function was a good followup for exploring Cryptol, requiring thorough understanding of the basics of Cryptol and its type system, the DES cipher introduces even greater complexity with its dependencies on individual bit manipulation via permutations and the table lookup indices. However, this difference was more noticeable in the C implementation than the Cryptol one. In Cryptol, because everything is defined in terms of bits these types of interactions were very intuitive. For instance, when performing any of the permutations, all that is needed is a comprehension like: "permKey = [key@x | x:[8]<-keyP]". A translation of this is: for each bit of the permuted key, take the x'th bit of the original key according to the permutation matrix. Succint and clear.

A similar statement can be used to extract the bits required for the table lookup during each iteration. Moreover, when this extraction occurs and used to index, the new arrays (2 and 4 bits in length), are interpreted as the corresponding integer values.

Meanwhile, in C the difficulty curve from SHA1 to DES increased a fair amount due to this extra bit manipulation. For instance, my implementation of the permutation function contains a for loop iterating over the following lines:

```
newBitVal = (toPerm[charInd]
          & (1<<(7-bitInd))) >> (7-bitInd);
toReturn[i/8] |= newBitVal << (7-i%8);
```

Contrasted against the one liner of Cryptol, this version is much clunkier and more difficult to parse, particularly due to the numerous bit shifts required. Furthermore, having to split up up the permutation index into a char index and the bit index of that char adds a level of abstraction which further removes the implementation from the underlying idea. However, by C's design, both of these are necessary because char's have to be 8 bits long.

A similar issue is experienced with the 6 bit block extraction, but a more challenging aspect is the shifting of a key to generate the sub keys. The length of the original key is 56 bits, which is then split into 28 bit halves for processing. I chose to represent these halves as

arrays of chars, which makes the splitting of the key easier, but adds additional overhead to the acual shifting. An alternative would be to represent them as ints, and add overhead between splitting and joining, but making the shifts easier (with care taken to zero out any excess bits). In retrospect, the latter may have been an easier approach, as with the way I did it, there was a need to keep track of the overflow between chars that seems more difficult than the extra string to ints conversion.

To further emphasize the expressiveness of Cryptol lets look at the number of lines in each implementation: Cryptol had 151 lines, and C had 239 (granted about 50 are representations over the necessary permutation). While the number of lines of an implementation may not always be a valid comparison, in this case I feel like it presents a decent heuristic to demonstrate the succintness with which Cryptol can express an algorithm vs C. This is because this increase in line number seems to be from requisite expansions as can be gathered from the code above, as setting up the loop and indices is alo needed. Indeed, an extra 90 lines just from the expected extra lines, such as brackets for if statements and loops, seems quite unlikely.

In order to verify my results, I compared them to sample input and output for the 3DES algorithm provided by NIST [National Institute of Standards and Technology 2017a], which is possible because 3DES is simply DES applied three times to a block of code, using a different key each time. Because the DES algorithm only acts on 64 bits and there is no standard for padding, only this type of core algorithm can be verified.

5 CONCLUSION

Overall, the Cryptol language presents itself as a very expressive language for the purposes of implementing cryptographic algorithms, especially when compared to those used commercially, such as C. This difference boils down to how Cryptol effectively captures the requisite ideas to allow for easy bitwise manipulations. The strict type system employed also helps to focus the development process by forcing cognizance of the resulting lengths of the bits.

However, this comes at a price of speed: even with just the basic correctness tests that I performed, I noticed that the Cryptol implementation would be slightly, but noticeably, longer than their C counterpart. Even so, such a slowdown could be expected, as it is designed for the purposes of prototyping and serving as a reference. Even if not used for the final implementation, having a Cryptol counterpart aids in the development of a C one as it allows for easy comparisons of the varoius functions. Furthermore, there is an entire tool suite behind Cryptol that helps with verifying algorithms in C that I did not have the time to explore, which furthers its usefulness. Cryptol seems to be a strong alternative for the use of research and using as a reference.

REFERENCES

[n. d.]. SHA1 online. http://www.sha1-online.com/. ([n. d.]). Accessed: 2017-12-8.

D Eastlake and P. Jones. 2001. *US Secure Hash Algorithm 1 (SHA1).* RFC 3174. RFC Editor. 1–8 pages. https://www.rfc-editor.org/info/rfc3174

Galois, Inc. 2016. *Programming Cryptol.* Portland, OR. https://cryptol.net/files/ProgrammingCryptol.pdf

National Institute of Standards and Technology. 1995. *FIPS PUB 46-3: Date Encryption Standard (DES).* https://csrc.nist.gov/publications/detail/fips/46/3/archive/1999-10-25

National Institute of Standards and Technology. 2017a. *Block Cipher Modes of Operation.* https://csrc.nist.gov/CSRC/media/Projects/Cryptographic-Standards-and-Guidelines/documents/examples/TDES_Core.pdf

National Institute of Standards and Technology. 2017b. *Secure Hash Algorithm.* https://csrc.nist.gov/CSRC/media/Projects/Cryptographic-Standards-and-Guidelines/documents/examples/SHA1.pdf