Cryptol: A Domain Specific Language for Verification of Cryptographic Algorithms

Zhiyuan Lin

July 22, 2016

Abstract

Contents

Chapter 1

Introduction

The aim of this project is to investigate domain specific languages that enhance reliability of cryptographic algorithms. The focus of this work is on Cryptol, however other languages with similar facilities will also be covered as related works.

May 25th - June 2nd:	Surveying Cyptol and similar languages
June 2nd - June 9th:	In-depth investigation of Cryptol's design and theory
June 9th - June 23rd:	Implementing AES algorithm in Cryptol
June 23rd - July 30th:	Verification of implementaion using Cryptol and SAW.
July 30th - July 7th:	Finishing up evaluation and writing report
July 7th - July 14th:	Preparing for presentation

Figure 1.1: Time Line of The Project

Therefore the first part of the project will be a brief survey of Cryptol and similar languages. This part will also cover high-level design of the Cryptol language and theory behind its functions for formal verification.

The second part of the project will be empirical study of the Cryptol language with an implementation. For the implementation part, the AES symmetric-key algorithm [1, ?] is to be implemented and properties related to the algorithm will be defined and checked in Cryptol to evaluate whether the language can efficiently verify the implementation. AES is chosen because it is the modern standard for symmetric-key encryption is widely used. The focus in this part is the evaluation of the language, rather than actual implementation of the algorithm, therefore other algorithms written in Cryptol, if available, will also be used to conduct empirical evaluations.

Another tool provided as a part of the Cryptol project is called The Software Analysis Workbench (SAW). SAW also provides formal verification for properties of programs written in Cryptol. SAW utilizes symbolic execution to translate programs into formal models. This tool will also be used to verify the AES implementation in Cryptol in order to see if it provides better functionalities for verification.

Figure 1.1 provides a time frame for the project. This is just a rough estimation, but the project will follow the steps specified. As mentioned before, because the focus is investigation of the Cryptol language, more time will be spent on evaluating the language functionalities.

1.1 Organizations

Chapter 2

Literature Survey

In this chapter we present previous works that uses programming language as a means to improve reliability of cryptographic applications. In Section 2.1, we give broad overview of several different works, whereas in Section 2.2 we dig into the details of the Cryptol programming language and the guarantees it provides.

2.1 Languages for Cryptographic Applications

The idea of using language features to enhance cryptogprahic applications have been investigated for over a decade. Some works, such as [2] builds upon existing languages, and provides extensions, e.g. libraries and frameworks, for efficient implementation of cryptographic protocols. Other studies [3] create brand-new domain specific programming languages dedicated to cryptographic applications. These works also focus on different aspects of implementation. Some focus on relaibility and correctness guarantees, while others emphasize ease of use and performance.

Charm [2] is an extensible framework in Python designed for rapid prototyping of cryptographic schemes. Charm promotes modularity and reusability of cryptographic primitives, and successfully increases inter-operability of existing numeric libraries such Sage and the Stanford Pairing-Based Crypto (PBC). It also provides benchmarking and profiling utilities for determining the performance of cryptographic algorithms.

NaCl [4] is a C/C++ library for implementing cryptographic protocols that provides security guarantee through features such as no data flow from secrets to load address, and no padding oracles.

ZKPDL [5] is an interpreted description language for specifying zero-knowledge protocols, motivated by applications such as electronic cash. Although the language is designed specifically for implementing prover and verifer of zero-knowledge, the language itself also have potentials for specifying other types of privacy-preserving systems. The ZKPDL interpreter also performs optimizations for protocols.

Similar to ZKPDL, TASTY [6] is a novel compiler designed specifically for generating efficient two-party computation protocols. TASTY provides a high-level domain specific language in which the user can specify the computation to be performed on encrypted data, and the compiler would translate that directly to a secure protocol. Moreover, TASTY uses the FairPlay [7] system to evaluate the protocol generated.

The Ceritified computer-aided cryptography [8] project provides a computer-aided framework for proving concrete security for cryptogprahic implementations. It extends EasyCrypt, an interative framework for verifying the security of cryptographic applications, to provide formal verification for cryptographic applications implemented in a C-like language. The framework also supports generation of optimized machine code based on the high-level language while retaining the security properties.

One of the study that is perhaps closest to Cryptol is CAO [9], a language designed to facilitate high-level, performant implementation of the AES algorithm. The CAO compiler utilises advanced techniques to improve performance of the implementation, but provides no specific functions for verifying the correctness of the algorithm. Agosta et al. [10] also proposed a domain specific language for cryptography based on Python. The major benefit that this work provides, however, is syntactic.

cPLC [11] is a more recent attempt at providing a domain specific languages for cryptogprahy. Instead of borrowing the syntax of existing programming languages, cPLC provides a language that is closed to the mathematical notations used in the cryptography community to describe protocols. Moreover cPLC provides native support for mathematical entities and operations such as groups that are often used in cryptography.

μCryptol [12, 13] is a language derived from Cryptol. The study focuses providing a verifying compiler that proves the correctness of the code tranformation process.

Cryptol applies techniques for formal verification of cryptographic protocols, a subject that have been studied for decades. We refer to [14] for these works.

2.2 The Cryptol Language

As has been mentioned before, Cryptol is a high-level programming language designed for cryptogpraphy. It provides a formal methods-based approach to cryptographic developments.

The Cryptol language offers several benefits to development of cryptographic protocols:

- Cryptol is designed to be the standard language of cryptography
- The Cryptol language provides high assurance of the correctness of the implementation
- The Cryptol source program can be used as source for code generation to multiple target platforms

We introduce the features that bring about these benefits in detail below.

2.2.1 The Standard Language of Cryptography

Cryptol is ambitiously designed to become the standard language of cryptography. Implementation of cryptographic algorithms in Cryptol are expected to serve as a high-level formal specification or at least reference implementation of the algorithms. This means that the language allows for algorithms specified in academic papers and standards to be translated into Cryptol source code in a manner that is straightforward and readable. Moreover, Cryptol frees developers from machine level detail so that they could focus on developing new algorithms.

To achieve this goal, Cryptol is designed to be a pure functional programming language similar to Haskell. The Cryptol syntax, just like Haskell's, is heavily inspired mathematical notations, and therefore can express computations in cryptographical protocols easily. It is also argued that functional programs are usually shorter and easier to understand. The persistent data structures provided in Cryptol allows for easier analysis and optimization of computation. Furthermore, the functional style naturally helps to create simple and clean abstractions so that the program is well-structured.

2.2.2 High Assurance Programming

To serve as authoritative specifications, Cryptol programs need to be correct first. Cryptol comes with several features that provides strong guarantees of correctness of functions:

- Type System
- Formal Verification
- Automated Testing

Type System

```
encrypt: {n} (0 < n, n < 3) => [8*n] -> [8*n]
```

Figure 2.1: Cryptol Type Signature

Cryptol uses a type system based on the Hindley-Milner type system [15], extend with size-polymorphism and arithmetic type predicates [3]. The type system is designed capture constraints that naturally arise from cryptographic algorithms such as fixed-size keys and input blocks.

See Figure 2.1 for an example of type signatures in Cryptol. The signature specifies a function *encrypt* that takes as input an 8-bit or 16-bit integer and output a number of the same size. The quantified type variable n in the type signature are bounded by the predicates $0 < n$ and $n < 3$. The predicates put a limit on the values of the type variable. Any arithmetic operations can be used in such predicates. Type signatures of these kinds rule out a large number of illegal input statically and provides strong guarantees to the correctness of the program. The AES algorithm [1] for example operates on 128, 192 or 256-bit keys. Such a constraint can precisely specified and checked in Cryptol type system.

Formal Verification and Automated Testing

The Cryptol language provides native formal verification utilities designed for equivalence and safety-checking. Correctness properties can be specified in Cryptol as part of the source code accompanying the algorithm. The Cryptol checker then checks for correctness of property with the SAT/SMT solvers. The Z3 prover is used by default.

Figure 2.2 shows a correctness property defined in Cryptol. The property states simply that decryption after encryption with the same key should result in the original message. We can see that a correctness property in Cryptol is but another function that returns a Bit (boolean) type value.

The Cryptol language provides an interactive mode similar to Haskell's with which developers can test functions interactively. Cryptol can conduct verification of the above property automatically with a simple *:prove* command issued in its interative prompt, as shown in Figure 2.3.

In cases where the property is invalid, the Cryptol checker provides an input for which the property does not hold true as counterexample. This gives developers a concrete scenario to look into and debug.

Because Cryptol properties are just functions, it is possible to write conditions in the property to construct proofs efficiently for more restricted scenarios. For polymorphic functions, we can also restrict its type signatures when proving properties.

Of course it is possible that external theorem prover used by Cryptol could not finish the proof within a reasonable amount of time. Therefore Cryptol also

```
encrypt: {n} [8] -> String n -> String n
// implementation of encryption

decrypt: {n} [8] -> String n -> String n
// implementation of decryption

encryptCorrect: {n} [8] -> String n -> Bit
property encryptCorrect =
         decrypt key (encrypt key msg) == msg
```

Figure 2.2: Correctness Property in Cryptol

```
>> :prove encryptCorrect
>> Q.E.D
```

Figure 2.3: Proving Correctness Properties in Cryptol

provides another command *:check* that can be used in the same way as *:prove* to conducted automated testing on the property. Figure 2.4 demonstrates how this utility can be used in Cryptol.

Similar to proving correctness, we can check in Cryptol whether a property is satisfiable through the *:sat* command. The command finds a satisfying input for the property using an off-the-shelf SAT solver. Finding satisfying assignments are interesting in cryptography because it can be used to formulate attacks to the cryptographic protocol. Figure 2.5 provides an example of known plaintext attacks specified in Cryptol. Such a property would certainly have a satisfying assignment. The point, however, is that Cryptol should never be able to find that assignment in a feasible amount of time in order for the encryption function to be safe.

2.3 How Formal Verification Works in Cryptol

Under the hood, the Cryptol source is translated to a *symbolic bit-vector* language, for which there are existing methods that can the decide if the code segments satisfies a certain property. Translation is done through symbolic evaluation, by executing the function in question with symbolic variables.

```
>> :set tests=10000
   // setting number of test cases generated
>> :check encryptCorrect
```

Figure 2.4: Testing Correctness Properties in Cryptol

```
    >> :sat (\key -> encrypt key msg == cipher)
```

Figure 2.5: Known Plaintext Attack in Cryptol

Language	Programming Paradigm	Features
Cryptol	Purely Functional	high-level specification, formal verification
CAO [9]	Imperative	high-level specification, performance
cPLC [11]	Imperative	high-level specification, mathematical syntax, numerical libraries

Figure 2.6: Comparing Cryptographic Programming Languages

The symbolic bit-vector program can be reduced to a SAT (boolean satisfiability) instance and solved with an off-the-shelf SAT solver such as *lingeling*. For checking equivalence property such as the one in Figure 2.2, we ask the SAT solver if there exists an assignment that would make the property return 0 (false). If there is, then the property does not hold, and part of the assignment will be translated to an input and returned as counterexample for the property. The SAT-based property checking approach has been used to prove equivalence of Cryptol program before and after compiler optimization [16].

Cryptol also supports using SMT (satisfiability modulo theory) solvers in place of SAT solvers to decide the bit-vector programs. The reason why SMT solvers might be better than SAT solvers in this case is that they tend to natively support higher-level structural information and arithmetic operations better than SAT solvers. Many SMT solvers come with dedicated mode for bit-vector programs. In practice, it was found that SMT solvers work better when proving properties that involve algebraic equalities.

2.4 Comparing Cryptol with Other Cryptographic Languages

Figure 2.6 compares Cryptol with CAO and cPLC, the languages we have covered in Section 2.1. These two languages are chosen because, like Cryptol, they focus on cryptographic applications in general. As shown in the table, Cryptol is the only when that employs a functional programming style, and also the only one that faciliates formal verification. The other two languages, although both high-level languages, put a strong focus on performance of the implementations.

Chapter 3

Implementation and Evaluation

This section presents the implementation part of the project. For the purpose of empirically evaluating the Cryptol language in practice, we created an implementation of the AES algorithm and devised properties to verify the implementation. The AES algorithm is the current standard of symmetric key encryption and is well specified in [1]. A brief introduction to the algorithm and the implementation is provided in Section 3.1. The correctness properties and results of verification is dicussed in Section 3.2. In Section 3.3 we summarize the observations of the language from the implementation process.

3.1 The AES Algorithm

We start by providing a brief introduction to the AES algorithm in the section to faciliate our discuss of the implementation.

As a block cipher, the AES algorithm runs on a 4 × 4 column-major order matrix of bytes. The matrix is formally called a *state* and it has a fixed size of 128 bits. See Figure 3.1 for an example of states.

The other input that the algorithm takes is the encryption key, which can be

$$\begin{bmatrix} b_0 & b_4 & b_8 & b_{12} \\ b_1 & b_5 & b_9 & b_{13} \\ b_2 & b_6 & b_{10} & b_{14} \\ b_3 & b_7 & b_{11} & b_{15} \end{bmatrix} \tag{3.1}$$

Figure 3.1: A State in AES

128, 192, or 256-bit long. In AES, a sequence of substitution and permutation steps are repeated for a number of rounds, and the number of rounds is decided by the key size. For 128-bit keys, 10 rounds of repetition is required, whereas 12 and 14 rounds are needed for 192-bit and 256-bit keys respectively. The number of rounds are needed to use the encryption key efficiently. More specifically, the algorithm does not simply reuse the encryption key for every round of operations. Instead, the key is expanded into (number of rounds + 1) round keys, which of 128 bits. The expansion is done using Rijndael's key schedule. This way a different round key is used in each round and there is no repetition. Note that because each entry in the state matrix is one byte, most arithmetic operations in Cryptol are defined over a finite field $GF(2^8)$ (GF stands for Galois Field).

In brief there are 5 major functions used in AES:

- *KeyExpansion*: for expanding encryption key to round keys, as we have discussed before.
- *AddRoundKey*: Applying round key to the state using bitwise *xor*.
- *SubBytes*: Substituting each byte in the state according to a lookup table called the S-box.
- *ShiftRows*: Shifting the last three rows of the state cyclically each by different number of bytes.
- *MixColumns*: Transform each column (treated as a polynomial on the field $GF(2^8)$) through matrix multiplication.

We refer to [1] for more detailed description of these operations. Algorithm 1 shows the pseudo-code for the algorithm using the operations introduced.

Algorithm 1 The AES Algorithm

```
roundKeys = expandKey(key)
                ▷ roundKeys is an $n + 1$ array where $n$ is the number of rounds.
state = addRoundkey(state, roundKeys[0])
for i = 1 to $n - 1$ where $n$ is the number of rounds do
    state = subBytes(state)
    state = shiftRows(state)
    state = mixColumns(state)
    state = addRoundKey(state, roundKeys[i])
end for
state = subBytes(state)
state = shiftRows(state)
state = addRoundKey(state, roundKeys[n])
                ▷ The last round does not involve the MixColumns operation.
return state
```

3.2 Proving Correctness of the AES algorithm

3.3 Review of the Cryptol Language

Chapter 4

Conclusion and Future Works

Bibliography

[1] N.-F. Standard, "Announcing the advanced encryption standard (aes)," *Federal Information Processing Standards Publication*, vol. 197, pp. 1–51, 2001.

[2] J. A. Akinyele, C. Garman, I. Miers, M. W. Pagano, M. Rushanan, M. Green, and A. D. Rubin, "Charm: a framework for rapidly prototyping cryptosystems," *Journal of Cryptographic Engineering*, vol. 3, no. 2, pp. 111–128, 2013.

[3] J. R. Lewis and B. Martin, "Cryptol: High assurance, retargetable crypto development and validation," in *Military Communications Conference, 2003. MILCOM'03. 2003 IEEE*, vol. 2, pp. 820–825, IEEE, 2003.

[4] D. J. Bernstein, T. Lange, and P. Schwabe, "The security impact of a new cryptographic library," in *International Conference on Cryptology and Information Security in Latin America*, pp. 159–176, Springer, 2012.

[5] S. Meiklejohn, C. C. Erway, A. Küpçü, T. Hinkle, and A. Lysyanskaya, "Zkpdl: A language-based system for efficient zero-knowledge proofs and electronic cash.," in *USENIX Security Symposium*, vol. 10, pp. 193–206, 2010.

[6] W. Henecka, A.-R. Sadeghi, T. Schneider, I. Wehrenberg, *et al.*, "Tasty: tool for automating secure two-party computations," in *Proceedings of the 17th ACM conference on Computer and communications security*, pp. 451–462, ACM, 2010.

[7] D. Malkhi, N. Nisan, B. Pinkas, Y. Sella, *et al.*, "Fairplay-secure two-party computation system.," in *USENIX Security Symposium*, vol. 4, San Diego, CA, USA, 2004.

[8] J. B. Almeida, M. Barbosa, G. Barthe, and F. Dupressoir, "Certified computer-aided cryptography: efficient provably secure machine code from high-level implementations," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pp. 1217–1230, ACM, 2013.

[9] A. Moss and D. Page, “Bridging the gap between symbolic and efficient aes implementations,” in *Proceedings of the 2010 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, pp. 101–110, ACM, 2010.

[10] G. Agosta and G. Pelosi, “A domain specific language for cryptography.,” in *FDL*, pp. 159–164, Citeseer, 2007.

[11] E. Bangerter, S. Krenn, M. Seifriz, and U. Ultes-Nitsche, “cplca cryptographic programming language and compiler,” in *2011 Information Security for South Africa*, pp. 1–8, IEEE, 2011.

[12] M. Shields, “A language for symmetric-key cryptographic algorithms and its efficient implementation,” *Available from the authors website*, 2006.

[13] L. Pike, M. Shields, and J. Matthews, “A verifying core for a cryptographic language compiler,” in *Proceedings of the sixth international workshop on the ACL2 theorem prover and its applications*, pp. 1–10, ACM, 2006.

[14] C. A. Meadows, “Formal verification of cryptographic protocols: A survey,” in *International Conference on the Theory and Application of Cryptology*, pp. 133–150, Springer, 1994.

[15] R. Hindley, “The principal type-scheme of an object in combinatory logic,” *Transactions of the american mathematical society*, vol. 146, pp. 29–60, 1969.

[16] L. Erkök and J. Matthews, “Pragmatic equivalence and safety checking in cryptol,” in *Proceedings of the 3rd workshop on Programming Languages meets Program Verification*, pp. 73–82, ACM, 2009.