CIR: Applying Cryptographic IR for Lawmakers Targeting AES

Conor O'Brien
csobrien@wm.edu
https://conorobrien-foxx.github.io/CIR/

May 14, 2023

Abstract

The process of implementing cryptographic protocols to meet external standards is complex, time-consuming, and requires effective communication between legislators, cryptographers, and programmers. The implementation of the Advanced Encryption Standard (AES) involves referencing multiple specifications, testing against given examples, and passing NIST approval. While established, tested, and open-source cryptographic algorithm implementations are recommended, implementing new or obscure algorithms face similar issues as rolling one's own crypto. This paper aims to investigate several research questions related to the implementation of cryptographic protocols, the structure of a correct AES implementation, adapting the way legislators write law to an IR, ensuring the integrity of our static web application, and assessing the effectiveness of a well-designed IR skeleton. As a result of our research, we propose CIR 0.1, a prototype intermediate representation which empowers programmers to express partial code skeletons with varying possible levels of granularity. We showcase its syntax, as well as demonstrate and evaluate potential intermediate representations in CIR for writeability, legibility, and accuracy. We find CIR to be more flexible than shipping an implementation with a protocol, but neither general enough for application to any protocol, nor qualitatively excellent enough to contend against existing solutions and the status quo.

1 Introduction

Implementing cryptographic protocols to the specification and the satisfaction of an external review board is a time-consuming, difficult, and laborious process, at best involving communication and comprehension between the legislators who draft the requirements and the programmers who must implement them, and at worst no communication whatsoever beyond the actual approval process.

Consider the Advanced Encryption Standard (AES), a widely-used block cipher algorithm approved by the U.S. Government. Implementations of this algorithm wishing to conform to these standards must reference myriad specifications on the algorithm, test against given examples (See: [4] Appendix F), and pass NIST approval to pass as secure in government-regulated applications. Bridging the gap between programmers and legislators will not only ease the difficulty in communication, approval, and development surrounding implementation and codification of cryptographic protocols; the wider this gap, the more stunted progress and adoption in cryptography may be.

The breach which stands between lawmakers, cryptographers, and programmers is significant; each has their own unique specialties, yet depends fully on the expertise of each other. The AES selection process was a lengthy process from 1997 to 2001, requesting algorithm submissions from the general public, narrowing the selection from 15 to 5 algorithms, and finally selecting Rijndael's algorithm [13]. This selection process ultimately generated hundreds of pages of documentation spread across various sources, all available exclusively in English prose and mathematical notation (e.g., [11], [12]).

The initiated, security-oriented programmers knows well, "Don't Roll Your Own Crypto," and for good reason. Even seasoned cryptographers make mistakes; see the WEP algorithm contemporaneous to the AES selection process. In light of this, the safest and most popular approach to crypto is to use established, tested, often open-source cryptographic algorithm implementations. For AES, an algorithm widely adopted and tested, both in practice and in theory, there are no shortage of solutions available for most computational needs: C implementations exist in both lightweight (tinyAES [6]) and established (libgcrypt [5] and OpenSSL [1]) capacities.

There are two main issues with this accepted practice. First, the work of implementing these algorithms in practice falls to the work of third parties not necessarily directly involved in the creation and approval of such algorithms as AES, necessitating someone to initiate the process of reviewing existing documentation, organized and disseminated sporadically as mentioned. While not quite equivalent to rolling one's own crypto, as the difficult the-

ory is worked out beforehand, there is still much that can go wrong, and even more at stake; it is vital that the information constructed in the standard be as readable and approachable as possible without sacrificing clarity and security.

The second issue is that this presumes the availability and feasibility of such solutions. Thankfully, with AES, C solutions are available, translating directly to many computational machines due to the universality of C; thanks also to the nature of AES's design to be accessible across many devices with varying capacities [13]. There are many cases, though, where there will not always be an available, feasible solution; consider trivially cases where proprietary software harden themselves to third-party injection attacks by rejecting third-party code altogether. More topically, in the age of post-quantum cryptography [14], as we are preparing to engage in determination processes not unlike those in 1997 to select the final AES algorithm, there will initially not be available such open-source solutions, and naturally none of them well have any longevity at the release of the algorithm.

Implementing any new or relatively obscure cryptographic algorithm will face the same issues programmers face when rolling their own crypto: Someone must first venture to implement the cryptographic algorithm. Testing, breaking and fixing are inevitable; indeed, such is vital to maintaining security.

We have the following research questions we investigate through the course of this paper.

RQ1: *What is the essential structure of a correct AES implementation?*

RQ2: *How can the way legislators write law be adapted to an IR?*

RQ3: *How do we ensure the integrity of a static web application?*

RQ4: *How effective is the skeleton an adequately designed IR provides?*

The remainder of this paper proceeds as follows. Section 2 overviews our paper. Section 3 describes the design of CIR 0.1. Section 4 evaluates our solution. Section 5 discusses additional topics. Section 6 describes related work. Section 7 concludes.

2 Overview

While there are many parts of the entire cryptographic algorithm lifecycle subject to improvement, we will focus on bridging the gap between those who are responsible for legislatively codifying the algorithm, and those who are to implement it, by leveraging the well-established concept of architecture design and unit testing from the software engineering world, and applying it at the legislative level.

We identify three principal agents: Cryptographers, legislators, and implementers. As with the AES selection process, we anticipate cryptographers and legislators to work closely together. This close cooperation will produce an intermediate representation (IR) that codifies certain aspects of the protocol programatically. The implementers (those tasked with implementing the protocol) then take this IR and apply our compiler to it. We call this specific IR compiler CIR (**C**ryptographic **I**ntermediate **R**epresentation)[1].

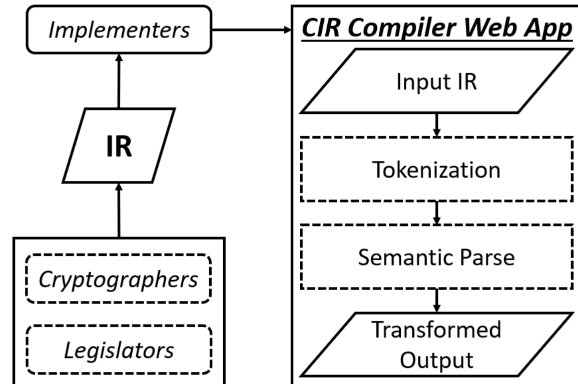

Figure 1: Usage flow of our approach, showcasing our architecture at a high level. Left: The implementation and adoption lifecycle. Right: The designed architecture.

Furthermore, we restricted our targets to specifically the AES algorithm, although in practice this IR should be able to adapted to a variety of circumstances and algorithms. We also restricted our output to specifically C skeleton codes, although the same principles could be applied to generating other artifacts and for other languages.

The compiler works by adapting the textual representation of the IR, which bears a similar resemblance to Python through the dynamic whitespace emblematic of the language, and producing various artifacts. For the scope of this paper, we examined the production of code skeletons from this IR.

3 Design

3.1 CIR 0.1

CIR 0.1 is a high-level, human-interpretable intermediate representation which allows the user to specify, with as much or as little granularity as they desire, a language-agnostic description of a protocol or algorithm.

3.1.1 Definitions

A CIR program consists of a series of *lines* which correspond directly to a series of tree structures.

[1] Alternatively, named after ***Ciriatto***, a demon from Dante's *Inferno*.

3.1.2 Tokenization

We define the following token structures identifiable in a CIR program:

1. *Keywords* are fully-uppercased strings of letter corresponding to one of the following: `CHOOSE`, `DEFAULT`, `DEFINE`, `ELSE`, `ELSEIF` (acceptable variants: `ELSIF`, `ELIF`), `FOR`, `IF`, `METHOD`, `MUTABLE`, `PASS`, `REPEAT`, `RETURN`, `SETMODE`, `STRUCTURE`, `TIMES`, and `TO`.

2. *Comments* begin with `//` and extend to the end of the line.[2]

3. *Spaces* are consecutive strings of spaces (0x20) and/or tabs (0x09).

4. Each *colon* (`:`) gets its own token.

5. *Return type indicators* are defined as the two-character string `->`, or the unicode character →.

6. *Words* are defined as any alphabetic character or underscore, followed by 0 or more alphanumeric characters and/or underscores.

7. *Numbers* are defined as 1 or more digits (0 to 9) in sequence.

8. *Linebreaks* are any sequence of linefeeds (0x0A) and/or carriage returns (0x0D).

9. *Operators* are defined to be any of the following: `-`, `+`, `*`, `/`, `!`, `~`, `^`, `|`, `&`, `<`, `<=`, `>`, `>=`, `or`, `and`, or `is`.

10. Each of the *parentheses* (`(` and `)`), the comma (`,`), and the equals sign (=) get their own token.

We tokenize by greedily scanning the input string in the above order for the equivalent regular expressions (e.g., `[A-Za-z_][A-Za-z0-9_]*` to match words). We statically assert spaces and tabs cannot both exist in the same spaces string, as well as error on any character not described above. Tokenization yields a list of Tokens, each of which has a type (corresponding exactly to one of the above categories) and a raw string (corresponding to its lexical value).

3.1.3 Semantic Parsing: Trees

We identify the following tree structures in CIR programs:

[2] Comments beginning with `///` are reserved for fall-through comments (which are additionally represented in the generated output), although this is not currently enabled due to the not-necessarily-linear nature of compilation.

1. Both *declarations* and *method evaluations* consist of a word, followed by an open parenthesis, occurring at the base level of code. We then infer what is meant depending on the word prefixing the parenthesis: If it is a Type word (i.e., `Int`, `Byte`, or `Array`), we infer a *declaration*, else a *method evaluation*. In either case, the structure is finished by reading tokens until a close parenthesis.

2. *Assignments* consist of a word followed by an equals sign, and are terminated by a line break.

3. *Structure* and *method* definitions are both determined by the presence of the corresponding keyword (`STRUCTURE` or `METHOD`), followed by a word (the name of the definition), and an option list of parameters, terminated by a colon; then, the definition expects a list of indented statements.

4. *If*, *elseif*, *else*, and *while* structures are the corresponding keyword (`IF`, `ELSEIF`, `ELSE`, or `WHILE`), followed by a condition demarcated by a colon.

5. Define and default statements function similar to `#define` in C, but follow the syntax of an assignment: They are the corresponding keyword (`DEFINE` or `DEFAULT`), followed by a word followed by an equals sign followed by the value. Unlike in C, however, these are defined more like enumerators.

We greedily scan for the appropriate patterns from left to right in the tokenized code. Each statement is either a *leaf* or a *tree*. Each has a head value and a type value identifying which kind of structure it represents. Trees have 1 or more children. (By implementation, leafs and trees are both nodes with 0 children and 1 or more children, respectively.)

3.1.4 Skeletonization

This step operates on the list of trees produced by the above step. Each tree corresponds to a block of skeleton code in the output.

3.1.5 Syntax

Modes. `SETMODE` denotes the existence of a constant, in the following example, named Mode, which can take on one of the values of Rad, Cool, or Square. We can specify its default value using `DEFAULT`, or explicitly set its value with `DEFINE`.

```
SETMODE Mode(Rad, Cool, Square)
DEFAULT Mode = Cool
DEFINE Mode = Cool
```

Variables. A type followed by one or more variables declares those variables of that type. By default, variables are immutable. Mutable variables are explicitly tagged with the `MUTABLE` keyword. You can also create advanced variable types with `STRUCTURE`.

```
Int(IterationCount)
MUTABLE Int(Counter)
STRUCTURE Name:
   Array(Bytes, 256)
Name(MyName)
```

Methods. The `METHOD` keyword denotes a method, which has an optional parameter list (assumed `void` if omitted) and an optional return type (assumed `void` if omitted).

```
METHOD Foo:
   PASS
METHOD Bar -> Int:
   RETURN 621
METHOD Multiply(Int x, Int y) -> Int:
   RETURN x * y
```

Iteration. Iteration is accomplished via either `WHILE` or `REPEAT...TIMES`.

```
MUTABLE Int(other)
other = 15
WHILE other > 10:
   other = other - 2
REPEAT other * 2 TIMES:
   PASS
```

Branching. Branching is accomplished with `IF`, `ELSEIF`, and `ELSE`.

```
IF other is 0:
   PASS
ELSEIF other > 5:
   IF other > 10:
      PASS
ELSE:
   PASS
```

3.2 Essential design of AES

To answer **RQ1**, we surveyed the three AES implementations mentioned earlier ([1], [6], and [5]), as well as the existing AES documentation ([11], [12]). This design is reflected in our designed IR in Section 4.

3.3 Web application integrity

The service is served statically through GitHub Pages. It is susceptible to the same kinds of attacks most client interfaces are, such as man-in-the-middle attacks. Being a static web application, however, means it could be ported to an offline client application (e.g. by serving it as an Electron application, or even just downloading the webpage), which can be useful in an environment which needs a secure information flow. We are less susceptible to denial of service attacks, as there is no server-side activity besides serving a webpage, which is cached; furthermore, GitHub Pages is equipped to handle large server loads already. Thus, we deem the application fairly secure, at least as secure as existing security options for vending applications, thereby answering **RQ3**.

4 Evaluation

4.1 Case studies

Given's CIR structure, we can implement an IR for the AES protocol in varying degrees of granularity.

4.1.1 Least granularity

```
Int(KeySize, RoundCount)

SETMODE Mode(Mode128, Mode192, Mode256)
DEFAULT Mode = Mode128
CHOOSE Mode:
   OPTION Mode128:
      KeySize = 128
      RoundCount = 10
   OPTION Mode192:
      KeySize = 192
      RoundCount = 12
   OPTION Mode256:
      KeySize = 256
      RoundCount = 14

Int(RowCount, ColumCount)
RowCount = 4
ColumCount = 4
STRUCTURE State:
   Array(Byte, RowCount, ColumCount)

STRUCTURE RoundKey:
   TODO
METHOD Encrypt:
   TODO
METHOD KeyExpansion:
   TODO
METHOD SubBytes:
   TODO
METHOD ShiftRows(State state):
   TODO
METHOD MixColumns(State state):
   TODO
METHOD AddRoundKey:
   TODO
```

This provides the following C structure:

```c
#include <stdint.h>
#define true (1)
#define false (0)

/** Mode: Mode_Mode128 | Mode_Mode192 |
    Mode_Mode256 **/
#if !defined(Mode_Mode128) && !defined(
    Mode_Mode192) && !defined(Mode_Mode256
    )
  #define Mode_Mode128
#endif
void Encrypt(void);
void KeyExpansion(void);
void SubBytes(void);
void ShiftRows(State state);
void MixColumns(State state);
void AddRoundKey(void);

int KeySize, RoundCount;
#ifdef Mode_Mode128
   #define KeySize ((int) 128)
   #define RoundCount ((int) 10)
#endif
#ifdef Mode_Mode192
   #define KeySize ((int) 192)
   #define RoundCount ((int) 12)
#endif
#ifdef Mode_Mode256
   #define KeySize ((int) 256)
   #define RoundCount ((int) 14)
#endif
int RowCount, ColumCount;
#define RowCount ((int) 4)
#define ColumCount ((int) 4)
typedef uint8_t State[RowCount][ColumCount
    ];
typedef /* TODO: FILL */ RoundKey;
void Encrypt (void) {
   //TODO:
}
void KeyExpansion (void) {
   //TODO:
}
void SubBytes (void) {
   //TODO:
}
void ShiftRows (State state) {
   //TODO:
}
void MixColumns (State state) {
   //TODO:
}
void AddRoundKey (void) {
   //TODO:
}
```

4.1.2 Increased granularity

Similar to above, but we instead implement `METHOD Encrypt` as:

```
METHOD Encrypt:
   MUTABLE State(state)
   KeyExpansion()
   // initial round key addition
   AddRoundKey()
   // interior rounds
   REPEAT RoundCount - 1 TIMES:
      SubBytes()
      ShiftRows(state)
      MixColumns()
      AddRoundKey()
   // final round
   SubBytes()
   ShiftRows()
   AddRoundKey()
```

which produces the following C skeleton (snippet):

```c
void Encrypt (void) {
   State state;
   KeyExpansion();
   AddRoundKey();
   for(int _temp_0 = 0; _temp_0 <
       RoundCount - 1; _temp_0++) {
      SubBytes();
      ShiftRows(state);
      MixColumns();
      AddRoundKey();
   }
   SubBytes();
   ShiftRows();
   AddRoundKey();
}
```

4.2 Results

The design of the IR itself is potentially too programmatic in style, and not flexible enough to be intuitive to the layperson. Although it was modeled after Python, and we hope that the analytic mind of a legislator might parse such a structure easier than a layperson, it is still unashamedly a programming language's syntax. That being said, it is much more approachable than writing pure C code. With increased granularity, however, the current design of CIR more closely approaches C code, both in expression and syntax.

Our results show that CIR, while capable of expressing certain levels of granularity for algorithms, is currently neither general enough to express all kinds of algorithms and data structures, nor is it natural enough to be friendly enough for a non-programmer. Thus, to answer **RQ4**, we claim CIR is somewhat effective, although leaves a lot to be desired, and is far from market-viable.

5 Discussion

There are many holes left by assumptions throughout this project. IR input parsing and correction, which is vital to any language, is currently heavily under supported and under reported. Although the language does have a lot of syntax-level and some semantic-level error checking, there is still much that could be added, such as enforcing correct if/else/else-if structure. The writer of the IR is assumed to be, not only a programmer, but one well-versed with the quirks and oddities of CIR, which currently lacks documentation beyond the source code and that information outlined in this paper. This falls short of the goal of letting legislators write this code who are not versed in programming. However, it does meet the goal of being legislator legible, reading very nearly like pseudocode.

6 Related Work

The Catala programming language [9] attempts a similar task, albeit on the contract level, and is intended for intralegislative communication.

Source code summarization is a common adjacent task to source code skeleton generalization [8] [10] [2] [7], although such attempts are for intraprogrammatic communication.

In general, there seems to be very little research in this exact area, namely, the intersection between legislative and programmatic communication.

7 Conclusion

Often, a solution attempting at unifying people, unless universally adopted on the spot, merely creates a new standard which competes with the others, thereby contributing to the disunity.[3]

The idea behind CIR is to help legislators, cryptographers, and implementers communicate. Although it enjoys some theoretical success to this end, the language is useless—even harmful—if not adopted. Therefore, the concept must be iterated upon much more before even attempting to market it.

Therefore, instead of introducing a new standard, it might be best to revise the current practice. We should reflect: Why is there such a gap between legislation and implementation? Perhaps because the legislation needs to be correct, specific, and set in stone, whereas implementations need to be dynamic and change with the times and use cases to remain secure. The legislation is an ideal, of which the implementation is sometimes only a partial realization. Could an implementation be shipped with legislation? Could we see a government-endorsed or even mandated implementation? If mandated, it would have to be general enough for virtually any purpose, even those not yet existent. Even if a merely endorsed implementation exists (e.g., physically through microchips, or digitally through software), the niche that CIR attempted to fill could still exist, especially depending on the nature of the government solution; such involvement on the government's end on programming implementations might even be overbearing and unhelpful.

If we were, however, to pursue the new standard route, there are a few avenues available that might provide a more fruitful standard. First, one might consider allowing for goals and tasks to be expressed, not imperatively, but functionally. Integrating concepts about what the program SHOULD and SHOULD NOT do (in the style of RFC 2119 [3]), might be useful in some cases.

To allow for more natural language inputs, one might consider the use of large language models to interpret human speech. However, for such a sensitive area as security, much progress into making models deterministic and reliable would need to be made, most likely culminating in a specific large language model suited to this end.

[3] As eloquently expressed in `https://xkcd.com/927/`.

References

[1] openssl.git. `https://git.openssl.org/?p=openssl.git;a=summary`, Mar. 2023.

[2] M. P. Arthur. Automatic source code documentation using code summarization technique of NLP. *Procedia Computer Science*, 171:2522–2531, 2020.

[3] S. Bradner. RFC2119: Key words for use in RFCs to Indicate Requirement Levels, 1997.

[4] M. Dworkin. Recommendation for Block Cipher Modes of Operation: Methods and Techniques, Dec. 2001.

[5] W. Koch. libgcrypt. `https://dev.gnupg.org/source/libgcrypt/`, Mar. 2023.

[6] kokke. Tiny AES in C. `https://github.com/kokke/tiny-AES-c`, 2021.

[7] D. Kramer. API Documentation from Source Code Comments: A Case Study of Javadoc. In *Proceedings of the 17th Annual International Conference on Computer Documentation*, SIGDOC '99, page 147–153, New York, NY, USA, 1999. Association for Computing Machinery.

[8] P. W. McBurney. Automatic Documentation Generation via Source Code Summarization. In *2015 IEEE/ACM 37th IEEE International Conference on*

Software Engineering, volume 2, pages 903–906, 2015.

[9] D. Merigoux, N. Chataing, and J. Protzenko. Catala: a programming language for the law. *Proceedings of the ACM on Programming Languages*, 5(ICFP):1–29, 2021.

[10] M. Moser, J. Pichler, G. Fleck, and M. Witlatschil. RbG: A documentation generator for scientific and engineering software. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 464–468, 2015.

[11] National Institute of Standards and Technology. Advanced Encryption Standard (AES), Nov. 2001.

[12] National Institute of Standards and Technology. Security Requirements for Cryptographic Modules, Mar. 2019.

[13] National Institute of Standards and Technology. Cryptographic Standards and Guidelines, May 2023.

[14] National Institute of Standards and Technology. Post-Quantum Cryptography, May 2023.