ISSN 2515-8260 Volume 08, Issue 02, 2021

A REVIEW ARTICLE ON CRYPTOGRAPHY USING FUNCTIONAL PROGRAMMING:HASKELL

Dikchha Dwivedi[1*] and Hari Om Sharan[2]

*[*1]Department of Computer Science & Engineering, Rama University Kanpur, 208016, India*
E-mail: deeksha14dec@gmail.com
[2]Department of Computer Science & Engineering, Rama University Kanpur, 208016, IndiaE-maildrsharan.hariom@gmail.com

Abstract: *A modern wave of programming technology has been at the frontline of functional languages, experiencing growing success as well as impact.* ***Hughes published an article entitled 'Why Functional Programming Matters', that has now been one of the most recent references in the field.****Safe programming defines the method used by software engineers to include multiple safety mechanism for their system.****Secure programming can be broken down into two subgroups to analyse its correlation with software design: access control, secure programme initialization, input validation, cryptography, secure networking, secure random number generation, and anti-tampering. In this paper we provide a review of various functional programming in Haskell for encryption. The inkling of functional programming is to make programming more closely related to mathematics. We describe crucial features and trade-offs that has to be well-thought-out while selecting the right method for secure computation.***

Keywords: Secure Computation,Cryptography, Haskell, Functional programming

1. Introduction

In the internet age, all the interactive and computing devices are interconnected across global and private networks. A vast majority of public and private agencies depend upon information system for their mission critical operations. Thus, network and system security are of paramount importance for efficient functioning of these systems. Cryptography is used to protect such sensitive and confidential information against undetected modification or unauthorized access during its storage and transmission [1].

Cryptography is the art and science of securing secret information[2]. However, these primitives alone cannot provide complete end to end security. In particular, we need network protocols such as TLS and SSH for end to end secure communication.

Cryptographic primitives are the low-level operations that act as the building blocks of any cryptographic scheme[3]. Common examples of these include hashes, ciphers, encryption-decryption routines, signing primitives, etc. Encryption is the process of hiding or encoding secret messages or information such that only the parties with authorization can read it. The

ISSN 2515-8260 Volume 08, Issue 02, 2021

keys used to encrypt a plaintext or decrypt a ciphertext are called encryption key and decryption key, respectively as shown in figure 1. If an adversary somehow gets access to the encryption key or the decryption key, the security of the whole system is compromised and its purpose is defeated.

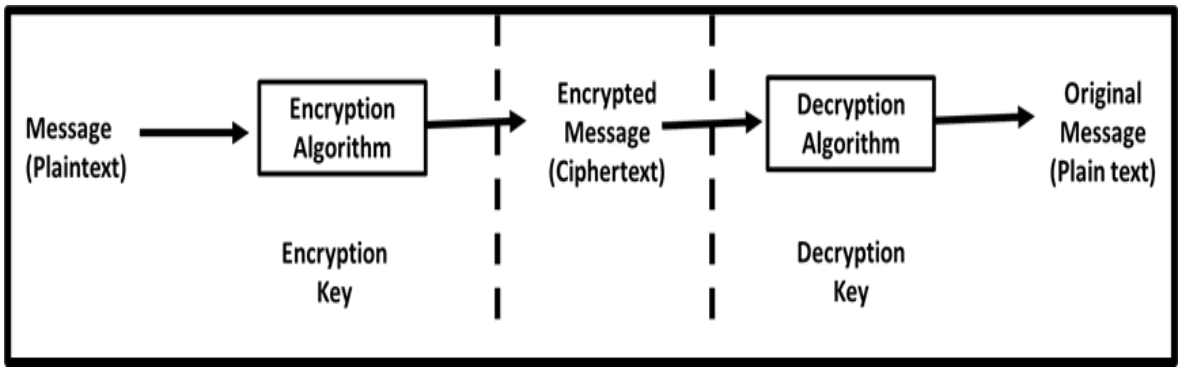

Figure 1: Encryption-Decryption Scheme

Symmetric key encryption/decryption schemes provide a secure communication channel to each pair of parties[4] as shown in figure 2. Symmetric key ciphers (encryption/decryption algorithms) are of two variants: stream ciphers and block ciphers. Block ciphers encipher in blocks of plaintext whereas stream ciphers encipher individual characters of plaintext. A major drawback of symmetric key schemes is that the secret key needs to be exchanged between parties beforehand via some secure mechanism. Moreover, in such a system, different key must share with distinct communicating entities.Therefore, this indicates that the number of network participants is raised by square. Another drawback of such a system is that it cannot be used for non-repudiation purposes, that is, for checking or proving which party (among the two communicating parties) had actually sent a particular message, without the involvement of trusted third party.

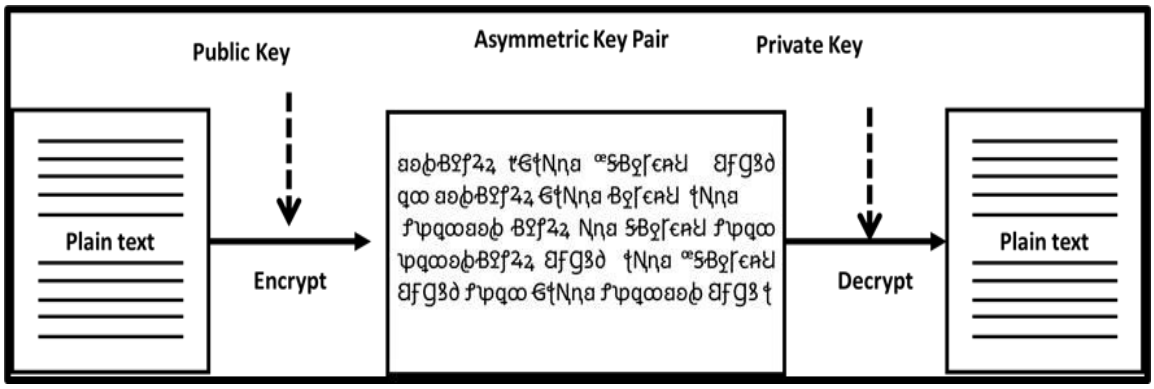

Figure 2: Symmetric Key Scheme

Public key (or asymmetric key) cryptography refers to a class of algorithms that include two different mathematically key, one is private key and the other one is public key. The public key cryptosystem is based on the fact that the key is created such that the private key computed cannot be determined from the public key. Although they demonstrated that they are linked[5]. In this system, the public key may be distributed freely, whereas the private key is kept secret. Common examples of asymmetric key techniques include RSA, Digital Signature Standard, ElGamal technique, etc. Figure 3 shows the asymmetric key scheme.

ISSN 2515-8260 Volume 08, Issue 02, 2021

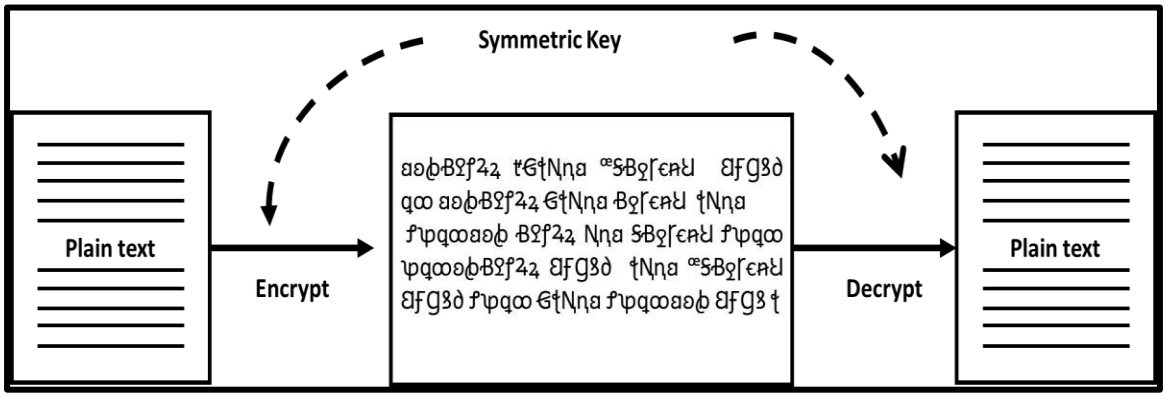

Figure 3: Asymmetric Key Scheme

Encryption alone can only provide confidentiality of messages. However, there is a need for making sure that the encrypted messages sent by a party are not altered by an eavesdropper thereby changing the meaning of decrypted text. Therefore, authentication is needed to provide integrity and authenticity assurances on the message. A message authentication code (MAC) is a small piece of information which helps to achieve this purpose.

A MAC algorithm as shown in figure 4 takes a secret key and an arbitrary-length message (which is to be authenticated) as input and outputs a MAC (also known as tag). MAC algorithms can be constructed from cryptographic hash functions (HMAC - Hashed MAC), or from block cipher algorithms (OMAC, CBC-MAC, etc). The MAC values are both generated and verified using the same secret key. Therefore, the sender and the receiver must agree on same key similar to the case of symmetric encryption. Hence, MACs cannot offer non-repudiation.

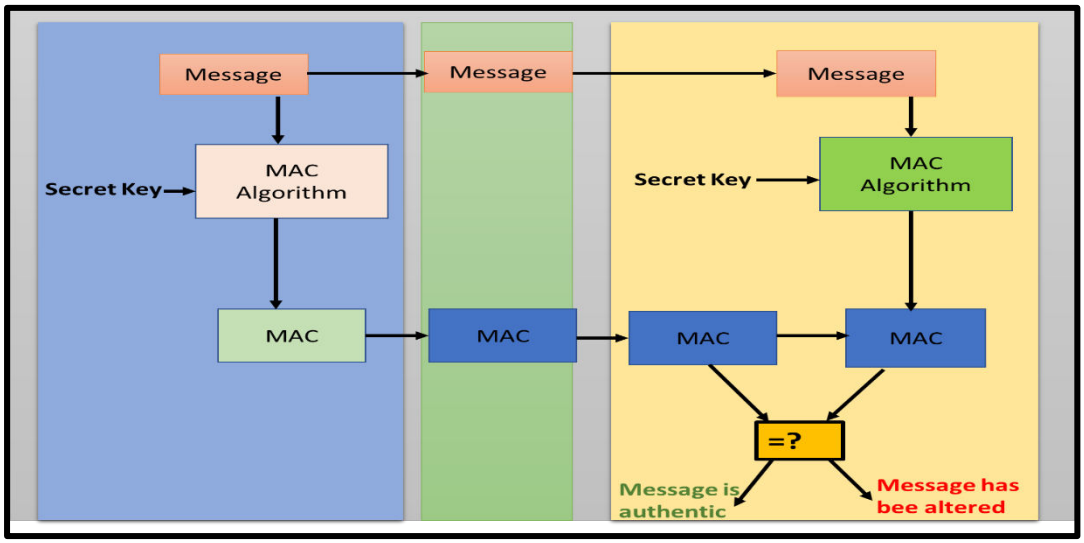

Figure 4: Message Authentication Scheme

A digital signature is another form of message authentication which also offers nonrepudiation (that is, the sender cannot deny having sent the message), unlike MAC algorithms. It also offers integrity. Hence, it is commonly used for software distribution, financial transactions and cases where detection of forgery and tampering is important. A digital signature is expected to be unforgeable by a third party. Some common digital signature algorithms include RSA-based schemes like RSA-PSS, DSA, ECDSA, etc.

ISSN 2515-8260 Volume 08, Issue 02, 2021

Although public key encryption schemes offer non-repudiation, message authentication and other benefits, the encryption using public key cryptographic schemes is much slower than symmetric key encryption schemes. The security level associated with similar bitlength keys is much higher for symmetric key encryption than public key cryptographic algorithms. Hence, in general, a shared secret key is usually established using public key cryptographic schemes so that an adversary, even by looking at the network traffic, cannot figure out the shared secret key. This shared secret key is then used for secure communication between the two parties using symmetric key encryption schemes. One of the first such key exchange protocols, Diffie-Hellman key agreement scheme, is discussed in the section 2.

1.1 Diffie-Hellman Key Agreement Scheme

Diffie-Hellman algorithm provides the capability for two communicating parties to establish a shared secret between them over an insecure channel[6]. This shared secret is then used for symmetric key encryption in further communication between these parties. The protocol for key exchange between two parties *A* and *B* can be described as follows:

1. *A* and *B* agree on a finite cyclic group *G* and a generator $g \in G$. (This step is assumed to have taken place long before the rest of the protocol; and *g* is publicly known).
2. *A* picks a random natural number *x* and sends g^x to *B*.
3. Similarly, *B* picks a random natural number *y* and sends g^y to *A*.
4. A computes $(g^y)^x$ and B computes $(g^x)^y$. Now, both *A* and *B* both possess the group element g^{xy}, which can serve as a shared secret key.

Functional programming is a programming paradigm in which functions are the building blocks of the program and computations are treated as evaluating expressions while avoiding mutable state and side effects, in contrast to imperative programming where programs are composed of statements which can modify the global state. Haskell is a functional language with non-strict semantics and a modern strong static type system. The different features of Haskell language are described in this work.

1.2 Referential Transparency

Referential transparency means an expression always represents the same value independent of number of times and scope in which it is evaluated[7]. Thus, the expression in a referentially transparent program can be replaced with its value without changing program's behaviour. Mathematical functions (for example $f(x) = x*x$) are referentially transparent.

Referential transparency allows programmer and compiler to reason about the behaviour of the program. This helps to prove program correctness, perform optimizations like memorization, lazy evaluation, automatic parallelization. Thus, compiler can generate much efficient code of a referentially transparent program. Programmes with mutable variables are not straightforward referential since, in most imperative languages, they could change the values of the inner variable to have new values in each call.

```
// A non-referentially transparent function.
int global;
int nonref(int x){
        global += 1;
        return (x + global);
}
```

ISSN 2515-8260 Volume 08, Issue 02, 2021

```
// A referentially transparent function
int ref(int x){ return x + 1; }s
```

1.3 Functions Of Higher Order

A higher order function is a function that take a function as an argument and/or return a function as result[8]. Haskell treating functions as first-class values does not differentiate between a value and a function, thus functions can be stored in data structures, passed to other functions or returned from a function. An example of a higher order function in Haskell is map which applies the given function on a list and returns the new list. In imperative languages like C, a limited higher order function behaviour can be simulated using function pointers[9].

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs
-- adds 1 to all elements of a list
add1List ::[Int] -> [Int]
add1List = map add1
-- adds 1 to the given number
add1:: [Int]-> [Int]
add1 a = a + 1
```

In lazy evaluation, an expression is not evaluated until its value is needed. Sincereferentially transparent functions yield the same value every time they are executed, their execution can be deferred until the result is needed and thus can be evaluated lazily. In Program, function to calculate minimum is implemented using quicksort. Due to lazy evaluation minimum is much more efficient as only the head of the list is calculated and rest of the list in never sorted[10].

```
quicksort :: [Int] -> [Int]
quickSort [] = []
quickSort (x:xs) = quickSort (filter (< x) xs) ++ [x]
++ quickSort (filter (>= x) xs)
minimum = head. quicksort
```

Infinite data structures can be constructed using lazy evaluation. In Program, all Even is an infinite list of all even numbers. Lazy evaluation can also be used to construct cyclic structures such as graphs which would otherwise be difficult in pure languages like Haskell. However, lazy evaluation does not provide a panacea and it has its weaknesses too[11]. The extra book-keeping has its overhead on the program performance. It is also difficult to predict required memory consumption and speed of the program . A recent shift from lazy IO to iteratee based IO in Haskell is because of this unpredictability in releasing resources [12].

```
allEven :: [Int]
allEven = filter even [1..]
```

2. Functional language: Haskell

Haskell is a strictly functional language so we test functions (syntactic terms) by using all computation techniques to derive values. All Haskell meaning is "first level," can be

ISSN 2515-8260 Volume 08, Issue 02, 2021

transferred to programs as statements, retrieved as outcomes, inserted in programming languages, etc. In another hand, Haskell forms aren't the first class. Types in one manner define the meanings, and the meaning is referred to as a typing. However, series of equation represent Haskell function .It also represent a co-relation between values and types. Vytiniotis, Weirich et al. (2008) defines polymorphic types, which is universally quantifies over all other types. Their result demonstrate thatpolymorphic phrases classify mainly families of types. Defining functions by pattern matching is quite common in Haskell, and the user should becomefamiliar with the various kinds of patterns that are allowed. Although, Haskell is a computational language, functionality must play a significant role. Lambdaabstraction can also we used as alternative to equations for defining functions. Perhaps by applying equation one can analyse infix operators . Although infix operators are basically just functionality, it allows sense to simply can implement it selectively as well. Section is defined in Haskell for partial application of an operator infix. If a function f is applicable to a non-terminating language, its findings show that function f does not eliminate the languages in programmes. [13]. It shows that function “f” shows strict properties and perpendicular results for “f”bot.

One advantage of the non-strict nature of Haskell is that data constructors are non-strict, too. Non-strict constructors permit the definition of (conceptually) infinite data structures[14].This is a rather odd function: it appears to look like if it returns a valuation of a polymorphic type over which it understands very little, because it never obtains such a benefit as an argument. Haskell utilises an inherently statement-based 2-D syntax layout which depend for explaining with "lined up in columns." In fact, layout is syntax for an implicit sorting function, that should be listed and can be helpful even under scenarios. The use of layout greatly reduces the syntactic clutter associated with declaration lists, thusenhancing readability. One of the finest qualities of the Haskell framework is that it differentiates it from other programming languages and, relative to other languages, is considered innovative in layout. In past research it was resulted that polymorphism that we have discussed in this study is known as parametric polymorphism. Parametric polymorphism is usually referred to as the form of polymorphism that has been discussed so far. There is yet another form of polymorphism named adhoc, referred mainly as overloading [15]. Haskell even defines various succession so there could be someone superclass of classes.Conflicts with names are eliminated by limiting a single activity to have become a component of around single category in every major review.

Ideally, arrays can simply be called functions from indices to values in a functional language. Butpragmatically, we need to be confident that together we can use advantage of the different features of the structures of such features, that are isomorphic to limited contiguous subgroups of the numeric subgroups, in order to achieve effective access to array elements [16]. Furthermore, Haskell doesn't really consider arrays as basic features for an application operation. Different analyses are characterise for functional array, these are monolithic and incremental. In the incremental case, there is a feature that generates a null array of a set quantity and the other, which takes an array, an index, and a value, introducing a unique array that varies mostly at the specified index from its existing one. In addition, an array was condensed at once by the monolithic system, without responding to intermediary array

ISSN 2515-8260 Volume 08, Issue 02, 2021

values. Even though Haskell has a gradual array upgrade operator and consider to be monolithic.

Haskell's I / O method is strictly practical, and it has all the interactive visualizations of traditional operating systems. [17]. Haskell relies critically on lazy estimation and higher-order functions, the core building blocks of any usable application, to accomplish this. In Haskell, there are two related methods of doing I / O: the stream-based technique and the continuation-based technique [18]. The prior is presumably structurally easier to show, and the latter is also described in terms of the former. Although, continuation techniques is used for practical programming.

2.1 Haskell libraries

Haskell is known for libraries providing much better type safety than other languagesas most of the programming errors are caught because of the type system[19]. Types are used to avoid typical programming errors such as endian mismatch, length conversion and buffer overflows. Types are also used extensively to distinguish between similar data with different intent to avoid illegal use by the library user. For example, a SHA Hash value is distinguished from a Blake hash value as both have different types and thus can never be used interchangeably even though they might have the similar representation when transmitted over network.

Data kinds are used, to further increase type safety of library. For example, a primitive such as RSA can work in different modes as captured by the type RSA mode. But this doesn't constraint users to create illegal types such as RSA Int, which is not a meaningful primitive. Using the Haskell kinds one can constraint polymorphic mode to be only of certain types (Sign, Encrypt etc) and any other type usage such as RSA Int is reported as compile error.

2.2 Avoiding Timing Attacks

Cryptographic implementations whose execution time depend on the message contentare prone to timing attacks[20]. A common source of such vulnerability is using naïve equality comparisons. For example, while comparing two strings, the naive algorithmwill return failure as soon as it finds the first non-matching character. Although beingefficient, the time taken by such comparison leaks information about the content ofthe strings. A timing resistant comparison would take same time for messages ofsame sizes independent of its content.

Haskell provides ad-hoc polymorphism through the use of type classes where theuser can define new type classes or overload functions provided by the default typeclasses such as Eq for equality comparison, Ord for ordered comparison etc[21]. Ourextensive use of types allow us to define our own timing independent operationson them. We carefully implement equality comparison on all types to be timingattack resistant.

A common source of bugs in cryptographic implementations is due to incorrecthandling of endian sensitive data[22]. As the endian conversion only takes place whenthe data is loaded from the buffer or when it is written out, endian conversion can be completely separated from the implementation of the underlying primitive.

Libraries are usually we provided with the EndianStore class which takes care of endian conversion automatically. Moreover, it is enforced that any data type which can potentially be serialized should be an instance of this class. By using the correct endian sensitive types in

ISSN 2515-8260 Volume 08, Issue 02, 2021

the description of the algorithm, the endian conversion is automatically taken care of. Explicitly endianness encoded versions of Word32 and Word64 are provided, which are instances of EndianStore as Word32LE, Word32BE, Word64LE and Word64BE. As far as the user is concerned these types can be used in the same manner as their parent because they inherit their parent type's Num instance (besides Ord, Eq etc).

2.3 Type safe lengths

The other source of errors is when there are length conversions. Sometimes there is need of length in bits (for example when appending the pad bytes in a hash), in other instances there is need in bytes (for example while allocating buffers) or sometimes there is even need in blocks (for example when applying a gadget on a buffer). The use of simple type such as Int for all these instances makes it more error-prone as the programmer has to remember to do the necessary conversion and if he forgets to do so, the error might go unnoticed in the production code until properly tested. A similar bug due to improper length conversion was found in OpenSSL [23] .

Length units is generally captured in a type safe manner, and any wrong unit usage is reported as a type error at compile time. Inter-conversion between these types is handled by the Rounding class and any conversion is explicit. The programmer doesn't need to do any arithmetic himself to convert between types as that is also error-prone. The functions provided by the Rounding class makes it explicit whether the user wants to round up or down while converting depending on the requirement [24].

There are the following length types:

• Bytes a where a is any integral type.

• Bits a where a is any integral type.

• Blocks p where p is the underlying Primitive.

Libraries are designed in way such that all the functions requiring any length argument uses the type safe lengths and all length conversions are explicit but handled by the Rounding class automatically. Thus, a lot of length conversion boilerplate is avoided.

Buffer overflows is another source of error in cryptographic systems[25]. The programmer have to be careful not to access data beyond the size of the buffer. Typesafe lengths play a crucial role in avoiding buffer overflows as all buffer allocationsand pointer manipulations are done using type safe lengths.

2.4 Core

The core of library usually mainly focuses on providing abstractions to implement cryptographic primitives and network protocols. A common design to capture cryptographic primitives is mainly centred around two concepts [26]

• Primitives to capture any cryptographic primitive.

• Gadgets to compute those primitives

The typical cryptographic primitives are hashes, MACs, ciphers, public keyencryption schemes, signature algorithms, random number generators etc[27]. A gadgeton the other hand is a device or algorithm that implements the primitive. Gadgets are not simple functions, but have internal memory elements associated with them with explicit initialization and finalization steps. Almost all the primitives currently have at least two gadgets, where one is for reference and the other is meant to be used in the production code. The reference gadget

ISSN 2515-8260 Volume 08, Issue 02, 2021

emphasizes on correctness rather than speed or security and are usually implemented in Haskell. They are used to verify the correctness of other gadgets for the same primitive. For use in production, there is the recommended gadget and by default all library functions are tuned to use this gadget. To provide a flag to auto tune the recommended gadget during the library compilation to choose the best from the available gadgets for the given primitive. For example, in systems supporting AES-NI, the gadget which uses AES-NI instructions is the recommended gadget, while on other machines the C portable can be the recommended gadget.

3. Haskell Features

Haskell is a language that is not simple. A relatively broad structure involves the decision to highlight those elements like pattern recognition and user-defined data forms and the need for a complete and functional vocabulary that incorporates topics such as I / O and modules. In both the static and dynamic actions of the language, the Haskell review also offers denotational semantics; it is probably more complicated than Paul Hudak [28].

Haskell is a strictly functional programming language for general purposes that exhibits most of those latest advances in computational (as well as other) structured programming analysis, include complex cognitive operations, slow comparison, static polymorphic coding, user-defined data forms, pattern recognition, and list understanding.

This is a very thorough language in that one has a system service, a more well-defined system I / O model, and a large collection of basic data types, like lists, tables, numbers of arbitrary and constant accuracy, and amounts of floating points.

Haskell has many other intriguing extra models, including prominently a systemic initialising approach, an orthogonal conceptual data source service, a standard and strictly operational I / O method, and a sort of array understandings by contrast to list understandings.

4. Functional programming

If variables are regarded in a system as first-class values-allowing them to be stored, transferred as statements, and retrieved as effects in data structures-they may refers to as higher-order functions.Up to this point, I did not say so much about the utilisation higher - level structures, but they appear in almost all of the programming languages I've spoken about, along with the lambda calculus, of example [29].

For higher-order functions, the major philosophical claim would be that operations are concepts like every other, then why not grant themselves some first-class status? However for requiring higher-order features, there will still be compelling pragmatic explanations. As the potential measure of inference over value systems, the functionality is explicitly described; hence promoting the use of constructs strengthens the use of that form of abstraction [30].

As an example of a higher-order function,consider the following:

twicefx=f(fx)

which takes its first argument, a function

f, and applies it twice to its second argument, *X*. The syntax used here is important:twice as written is curried, meaning thatwhen applied to one argument it returns afunction that then takes one more argument, the second argument above. For example, the function add2.

ISSN 2515-8260 Volume 08, Issue 02, 2021

add2 = twice succ

where succ x = n+lis a function that will add 2 to its argument.

This method promotes the association of the function implementation to the left, since (twice succ X) is equal to ((twice succ) x), so it works perfectly.

Functions can be produced in many ways in conventional programming languages. One approach is to use calculations to call them, as above; other way is to simply construct them as lambda abstractions, thereby making them nameless, as in the Haskell word.

\x + x+1

in lambda calculus this would be writtenXx.x + 11, which is the same as the successorfunction succ defined above. add2 can thenbe defined more succinctly asadd2 = twice (Lx + x+1)From a pragmatic viewpoint, we can understand the use of higher-order functionsby analysing the use of abstraction in general. As known from introductory programming, a function is an abstraction of valuesover some common behaviour (an expression). Limiting the values over which theabstraction occurs to nonfunctions seems unreasonable; lifting that restriction results in higher-order functions. Hughesmakes a slightly different but equally compelling argument in[29] wherehe emphasizes the importance of modularity in programming and argues convincingly that higher-order functions increasemodularity by serving as a mechanism forglueing program fragments together. Thatglueing property comes not just from theability to compose functions but also fromthe ability to abstract over functional behaviour[31]as described above.As an example, suppose in the course ofprogram construction we define a functionto add together the elements of a list asfollows:

sum [] = 0

sum(x:xs) = add x (sum xs)

Then suppose we later define a function tomultiply the elements of a list as follows:

prod [ ] = 1

prod(x:xs)= mu1 x (prod xs)

But now we notice a repeating pattern andanticipate that we might see it again, so weask ourselves if we can possibly abstractthe common behaviour. In fact, this is easyto do: We note that add/mu1 and O/l arethe variable elements in the behaviour, andthus we parameterize them; that is, wemake them formal parameters, say f andinit. Calling the new function fold, theequivalent of sum/prod will be “fold f init”,and thus we arrive at

(fold f init) [ ] = init

(fold f init)(x:xs) = f x ((fold f init) xs)

where the parentheses around “fold f init”are used only for emphasis, and are otherwise superfluous.From this we can derive new definitionsfor sum and product:

sum = fold add 0

prod = fold mu1 1

Now that the fold abstraction has beenmade, many other useful functions can bedefined, even something as seemingly unrelated, as append:append xs ys = fold (:) ys xs

[An infix operator may be passed as anargument by enclosing it in parentheses;thus (:) is equivalent to \x y + x :y.] Thisversion of append simply replaces the [ ] atthe end of the list xs with the list ys.

ISSN 2515-8260 Volume 08, Issue 02, 2021

It is useful to identify that, utilizing basic numerical logic and inference, the new concepts are identical to the old. Although It is vital to know because they did nothing from the ordinary before coming at the central abstraction. They only enforce the principles of classical data abstraction in as unregulated a manner as possible, that further ensures that mechanisms are first-class citizens.

5. Conclusions

This review provides the reader with makes it very important into the basic nature of object - oriented programming languages and the technique of design they advocate. It starts with a discussion cryptography, followed by cryptographic primitives in general, A review of the distinctive features of existing functional languages, as well as a discussion of existing areas of study. We can bring into context both the importance and shortcomings of the functional programming model through this analysis. Here we provide a survey of various functional programming in Haskell for encryption. The inkling of functional programming is to make programming more closely related to mathematics. We also describe crucial features and trade-offs that has to be well-thought-out while selecting the right method for secure computation.

References

1. Barker, E.B., W.C. Barker, and A. Lee, *Guideline for implementing cryptography in the federal government.* 2005.
2. Katz, J. and Y. Lindell, *Introduction to modern cryptography*. 2014: CRC press.
3. Applebaum, B., et al. *Fast cryptographic primitives and circular-secure encryption based on hard learning problems*. in *Annual International Cryptology Conference*. 2009. Springer.
4. Sindhuja, K. and S.P. Devi, *A symmetric key encryption technique using genetic algorithm.* international journal of computer science and information technologies, 2014. **5**(1): p. 414-416.
5. Potlapally, N.R., et al. *Optimizing public-key encryption for wireless clients*. in *2002 IEEE International Conference on Communications. Conference Proceedings. ICC 2002 (Cat. No. 02CH37333)*. 2002. IEEE.
6. Herzog, J.C. *The Diffie-Hellman key-agreement scheme in the strand-space model*. in *16th IEEE Computer Security Foundations Workshop, 2003. Proceedings.* 2003. IEEE.
7. Scholz, E. *Four concurrency primitives for Haskell*. in *ACM/IFIP Haskell Workshop*. 1995. Citeseer.
8. Haftmann, F. *From higher-order logic to Haskell: there and back again*. in *Proceedings of the 2010 ACM SIGPLAN workshop on Partial evaluation and program manipulation*. 2010.
9. Bourdoncle, F. *Abstract debugging of higher-order imperative languages*. in *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*. 1993.

ISSN 2515-8260 Volume 08, Issue 02, 2021

10. Maessen, J.-W., *Hybrid eager and lazy evaluation for efficient compilation of haskell*. 2002, Massachusetts Institute of Technology.
11. Karczmarczuk, J., *Generating power of lazy semantics.* Theoretical Computer Science, 1997. **187**(1-2): p. 203-219.
12. Harrison, W., T. Sheard, and J. Hook. *Fine control of demand in Haskell*. in *International Conference on Mathematics of Program Construction*. 2002. Springer.
13. Hudak, P., et al., *Report on the programming language Haskell: a non-strict, purely functional language version 1.2.* ACM SigPlan notices, 1992. **27**(5): p. 1-164.
14. Turner, D.A. *Miranda: A non-strict functional language with polymorphic types*. in *Conference on Functional Programming Languages and Computer Architecture*. 1985. Springer.
15. Vytiniotis, D., S. Weirich, and S. Peyton Jones, *FPH: First-class polymorphism for Haskell.* ACM Sigplan Notices, 2008. **43**(9): p. 295-306.
16. Jay, C.B., et al. *A monadic calculus for parallel costing of a functional language of arrays*. in *European Conference on Parallel Processing*. 1997. Springer.
17. Achten, P. and S.P. Jones. *Porting the Clean object I/O library to Haskell*. in *Symposium on Implementation and Application of Functional Languages*. 2000. Springer.
18. Spivey, J.M. and S. Seres. *Embedding Prolog in Haskell*. in *Proceedings of Haskell*. 1999.
19. Tsai, T.-c., A. Russo, and J. Hughes. *A library for secure multi-threaded information flow in Haskell*. in *20th IEEE Computer Security Foundations Symposium (CSF'07)*. 2007. IEEE.
20. Barbosa, M., et al. *Type checking cryptography implementations*. in *International Conference on Fundamentals of Software Engineering*. 2011. Springer.
21. Hall, C.V., et al., *Type classes in Haskell.*ACM Transactions on Programming Languages and Systems (TOPLAS), 1996. **18**(2): p. 109-138.
22. Mouha, N., et al., *Finding bugs in cryptographic hash function implementations.* IEEE transactions on reliability, 2018. **67**(3): p. 870-884.
23. Hanson, R.J. and K.H. Haskell, *Algorithm 587: Two algorithms for the linearly constrained least squares problem.* ACM Transactions on Mathematical Software (TOMS), 1982. **8**(3): p. 323-333.
24. Weirich, S., et al., *A specification for dependent types in Haskell.* Proceedings of the ACM on Programming Languages, 2017. **1**(ICFP).
25. Dockins, R. and S.Z. Guyer, *Bytecode verification for Haskell*. 2007, Citeseer.
26. Sloan Jr, R.M., *Constructive synthesis of optimized cryptographic primitives*. 2017, Massachusetts Institute of Technology.
27. DELLEDONNE, L., *A methodology based on functional languages for the design of hardware cryptographic primitives resistant to side-channel attacks.* 2017.
28. Marlow, S., *Haskell 2010 language report.* Available on: https://www. haskell. org/onlinereport/haskell2010, 2010.
29. Hughes, J. and J. Sparud. *Haskell++: An object-oriented extension of Haskell*. in *Proc. Haskell Workshop*. 1995.

ISSN 2515-8260 Volume 08, Issue 02, 2021

30. Davie, A.J., *Introduction to Functional Programming Systems Using Haskell*. Vol. 27. 1992: Cambridge University Press.
31. Algehed, M. and A. Russo. *Encoding dcc in haskell*. in *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security*. 2017.