A Homotopical Approach to Cryptography

Paventhan Vivekanandan
School of Informatics, Computing and Engineering
Indiana University Bloomington
Bloomington, USA
pvivekan@umail.iu.edu

***Abstract*—We present a new direction for the formal specification of cryptographic schemes using types. In this approach, we specify a cryptographic protocol using the tools of homotopy type theory. Homotopy type theory adds the notion of higher inductive type and univalence axiom to Martin-Löf's intensional type theory. A higher inductive type allows us to introduce constructors for paths and higher-dimensional paths in addition to points. The paths are then identified by equivalences in the universe through univalence. A higher inductive type can act as a front-end mapped to a concrete cryptographic implementation in the universe. By having a higher inductive type front-end, we can encode domain-specific laws of the cryptographic implementation as higher-dimensional paths. Due to univalence and functoriality, the path structure will be preserved in the mapping and realized by equivalence in the universe. Using this model we can achieve various guarantees on the correctness of the cryptographic implementation.**

***Index Terms*—Higher Inductive Type, Univalence, Functor, Homotopy, Groupoid, Universe, Equivalence, Quasi-inverse, Identity type**

I. Introduction

Formal verification of cryptographic protocols has become a significant research focus over recent years [1] [2]. Some widely used cryptographic implementations were found to be flawed after their deployment becoming vulnerable to various attacks. For example, the Heartbleed attack (CVE- 2014-0160) is a consequence of a simple coding error [3]. Even with skilled designers, developers and testers it is highly difficult to implement a cryptographic protocol without errors [4]. Theorem provers with a mathematical background such as Coq [5] and Agda [6] can be used for implementing cryptographic protocols and for designing attack models [1].

Type theory can be used both as a functional programming language and as a theorem prover. In type theory, the propositions can be interpreted as types whereas set theory has clear segregation between first-order logic and the axioms of ZFC. A type in type theory corresponds to a topological space in homotopy theory. Homotopy type theory [7] extends type theory by adding the notion of higher inductive type and univalence axiom. In homotopy type theory, the witness or proof element of a type can be viewed as a point in a topological space and a witness of an identity type can be viewed as a path in a topological space. A higher inductive type differs from an ordinary inductive type by providing constructors not only for points but also for paths. A cryptographic scheme can be specified in homotopy type theory using the notion of higher inductive types. A cryptographic system which expresses decryption as an inverse of encryption [8] can be defined using higher inductive type representing a graphical model in a topological space. Functions from a concrete implementation of the cryptographic system can be projected from this graphical model using univalence axiom. The higher inductive type acts as an abstract model for the encoded cryptographic system and enables us to model the correctness properties as paths or higher-dimensional paths in a topological space. Agda is a dependently typed programming language based on Martin-Löf's intensional type theory [6]. A higher inductive type can be specified in Agda using Dan Licata's method [9].

In this paper, we discuss specifying cryptographic schemes using higher inductive types in homotopy type theory, implemented in Agda, and how to project computational models from such specifications. In this paper, we make the following contributions.

- We show how to design a cryptographic construction using a higher inductive type and how to map the abstract type to a concrete implementation in the universe. Such developments give rise to interesting homotopies which are paths between paths or two-dimensional paths in a topological space.
- Paths in a higher inductive type are used to model correctness rules such as functional correctness [8], which says decryption inverses encryption, and this structure will be preserved in the mapping to the universe due to the functoriality of mappings in homotopy type theory.
- We can enforce various restrictions on the concrete implementation when a cryptographic protocol is modeled using a higher inductive type. We discuss designing a higher inductive type for a database model with multi-layered encryptions in the style of cryptDB [10].
- We discuss encoding of domain-specific properties related to homomorphic encryption, deterministic encryption, and order-preserving encryption as a path between paths or homotopies in a higher inductive type.

Designing cryptographic constructions as a higher inductive type has the following benefits.

- In type theory all functions are functorial. Therefore, the functional correctness and domain-specific properties of a cryptographic construction can be specified as paths or homotopies in a higher inductive type, and the functions will preserve the path structures in the mapping of the type to the universe.

- By specifying cryptographic properties as paths, we achieve guarantee on the correctness of the underlying concrete implementation with respect to the encoded properties.
- We can have a graphical representation of a cryptographic construction in a topological space, and we map it to a concrete implementation in the universe.
- By modeling a cryptographic construction as a higher inductive type, we can get the groupoid structure and the relevant coherence laws related to the higher inductive type for free.
- We will get a non-dependent eliminator also known as the recursion principle, and we can use it to define functions or to map the elements including the paths of the higher inductive type to elements of other types such as the universe.
- We will get a dependent eliminator also known as induction principle which can be used to formulate and prove theorems related to a cryptographic construction encoded as higher inductive type.
- Correctness and theorem proving become an inherent part of the system. So we can eliminate the cost involved in using an external theorem prover to reason about the correctness properties of the cryptographic protocol in development.

II. Background

A formal specification of a cryptographic scheme requires a programming language with support for theorem proving. Proof-assistants with a strong mathematical background such as Agda and Coq can be used to specify correctness and security properties of a cryptographic construction. There are works which use an embedded domain-specific language [1] [24] [20] on existing theorem provers to support defining and proving cryptographic properties. In this paper, we discuss a new approach to specify cryptographic protocols based on types. This approach involves correlating a type with a cryptographic implementation. By combining with the right type, we can guarantee on various correctness properties of the cryptographic application. In the remainder of this section, we discuss the tools of homotopy type theory which are instrumental in modeling and associating types with cryptographic implementations.

Unlike set theory, which is an interplay between propositions and sets, type theory is based on the interpretation of propositions-as-types. According to this interpretation, a proposition stating that two elements of a type $a, b : A$ are equal corresponds to a type known as the *identity type* given by $a =_A b$ or $Id_A(a, b)$. In homotopy type theory, elements of the identity type $a =_A b$ are used to model the notion of paths or equivalences between a and b in the space A. An element of the type $a =_A b$ is a witness or a proof stating that a and b are propositionally equal. *Propositional equality* is a proof relevant notion of equality expressed by identity types. There is also a proof-irrelevant notion of equality in type theory known as *judgmental equality* or *definitional equality*. Definitional equality is not internal to the theory, and it is used to express equality by definition. For example, when we have a function $f : Nat \to Nat$ defined as $f(x) = x^3$ then $f(2)$ is definitionally equal to 2^3.

Homotopy type theory extends Martin-Löf's intensional type theory by adding univalence axiom and higher inductive types. It introduces the notion of viewing type as a topological space in homotopy theory or a higher-dimensional groupoid in category theory. Because of this correspondence, we can observe an element of the identity type $x =_A y$ for $a, b : A$ as a path in a topological space or a morphism in a groupoid. Also, an element of the iterated identity types $m =_{x =_A y} n$ and $p =_{m =_{x =_A y} n} q$ can be viewed as a 2-dimensional and a 3-dimensional path respectively in a topological space or a morphism between morphism and a higher-level morphism respectively in a groupoid and so on.

A morphism at a level k in a groupoid is called a k-morphism. A k-morphism has a groupoid structure defined by identity, composition, and inverse operations. These operations satisfy the groupoid laws which are associativity of composition, identity as a unit of composition and cancellation of inverses through a weak sense of equality but only up to a morphism at the next level $k+1$. We can view the k-morphism as a k-dimensional path in a topological space. Similarly, we can observe the elements of an iterated identity type at level k as k-dimensional paths. Therefore a proof element of the type $x =_A y$ acts like a one-dimensional path between endpoints x and y and a proof element of type $m =_{x =_A y} n$ acts like a 2-dimensional path or a homotopy between paths of type $x =_A y$ and so on. A homotopy between non-dependent functions $f_1, f_2 : A_1 \to A_2$ is given by the following equation.

$$f_1 \sim f_2 :\equiv \prod_{x : A_1} (f_1(x) =_{A_2} f_2(x)) \tag{1}$$

Moreover, the paths also satisfy the groupoid laws up to homotopy at the next level in the following sense.

- $refl \circ x = x \circ refl = x \longrightarrow$ identity as a unit of composition
- $(x \circ y) \circ z = x \circ (y \circ z) \longrightarrow$ associativity of composition
- $!x \circ x = x \circ !x = refl \longrightarrow$ cancellation of inverses

where $refl$ is an element of type $x =_A x$

Because of the correspondence of types to a topological space or a higher-dimensional groupoid, we can map the elements of an identity type, which are paths in homotopy type theory, to equivalences between types in a universe. Equivalence can be relaxed to a bijection when types behave like sets. The following defining equations give an equivalence between type A and type B.

$$A \simeq B :\equiv \sum_{f : A \to B} isequiv(f) \tag{2}$$

$$isequiv(f) :\equiv \Big(\sum_{g : B \to A} (f \circ g \sim id_B) \Big) \times \Big(\sum_{h : B \to A} (h \circ f \sim id_A) \Big) \tag{3}$$

In (3), the composite $f \circ g$ is homotopic to the identity function id_B, and the composite $h \circ f$ is homotopic to the identity function id_A. The mapping of a path to equivalence is made possible by the univalence axiom which describes that we may identify equivalent types A and B in the following sense.

$$ua : (A \simeq B) \to (A =_U B) \tag{4}$$

In (4), the type U is the *universe* or the *type of types*. The univalence axiom states that when we have a proof of type $A \simeq B$, we can obtain a path between A and B. Axiomatization of univalence weakens the computational properties of type theory. But there are works in progress, such as $cubical\, type\, theory$, which provides a computational interpretation of the univalence axiom.

There is also a reduced notion of equivalence called *quasi-inverse*. A quasi-inverse for a function $f : A \to B$ is given by

$$qinv(f) :\equiv \sum_{g:B\to A} \Big((f \circ g \sim id_B) \times (g \circ f \sim id_A) \Big) \tag{5}$$

Also, we have a function that maps an element of quasi-inverse $qinv(f)$ to $isequiv(f)$ for $f : A \to B$ [7].

$$mkqinv : qinv(f) \to isequiv(f) \tag{6}$$

For examples described in this paper, we will use $mkqinv$ to obtain a proof of equivalence from quasi-inverse. For a path $p : A =_U B$, we have a function coe [12] that coerces along p. The following equation gives the type of coe.

$$coe : (A =_U B) \to (A \to B) \tag{7}$$

In the presence of univalence, we also have a computation rule for coe [12] defined as follows.

$$coe\,(ua\,(f\,,\, isequiv(f)))\, x = f(x) \tag{8}$$

where $x : A$, $f : A \to B$ and $(f\,,\, isequiv(f)) : A \simeq B$.

Higher inductive types are a general schema for defining new types in homotopy type theory. It extends an ordinary inductive type by providing constructors for generating paths and higher paths. In homotopy type theory, we define a higher inductive type by specifying its introduction, elimination, and computation rules. The introduction rule of a type specifies its constructors. The elimination rule of a type defines how to use its elements, and the computation rule describes the action of the elimination rule on the constructors of the type. A simple example for higher inductive type is the interval type I. It consists of two point constructors 0_I and 1_I and a path constructor $seg : 0_I =_I 1_I$. The following declaration[1] specifies the introduction rule for I.

```
data I : Set where
    -- point constructors
    0_I : I
    1_I : I
    -- path constructors
    seg : 0_I =_I 1_I
```

[1]In this paper, we have given a reduced declaration of higher inductive types for better understanding. In Agda, we use Dan Licata's method [9] to define higher inductive types where we specify the path constructors as postulates.

The non-dependent elimination rule or the recursion principle of I states that when given a type C along with constructors $c_0, c_1 : C$ and $cseg : c_0 =_C c_1$, there is a function $f : I \to C$ such that $f(0_I) = c_0$, $f(1_I) = c_1$ and $ap_f(seg) = cseg$ where ap_f defines the action of functions on paths. The equalities $f(0_I) = c_0$, $f(1_I) = c_1$ and $ap_f(seg) = cseg$ are the computation rules for the type I. The computational rules for the point constructors 0_I and 1_I hold definitionally, but the computation rule for path constructor seg holds only propositionally, and we specify it as an axiom which is a limitation of homotopy type theory.

Similarly, the dependent eliminator or the induction principle of I states that when given a type $D : I \to U$ along with constructors $d_0 : D(0_I)$, $d_1 : D(1_I)$ and $dseg : d_0 =^D_{seg} d_1$, there is a dependent function $f : \prod_{(x:I)} D(x)$ with computation rules $f(0_I) = d_0$, $f(1_I) = d_1$ and $apd_f(seg) = dseg$. Here $dseg$ is a heterogeneous path transported over seg and apd_f defines the action of functions on heterogenous paths [7].

Another important concept of homotopy type theory which is central to understand the idea proposed in this paper is that the functions behave functorially on paths. It means that a function $f : A \to B$ respects equality and it preserves the path structure in the mapping from type A to type B. Now we can give the type of ap_f which defines the action of non-dependent functions on paths as follows.

$$ap_f : (x =_A y) \to (f(x) =_A f(y)) \tag{9}$$

The following equation gives the action of dependent functions of type $f : \prod_{(x:A)} B(x)$ on paths.

$$apd_f : \prod_{p:x=y} (p_*(f(x)) =_{B(y)} f(y)) \tag{10}$$

In (10), $p_*(f(x))$ lying in space $B(y)$ can be thought of as an endpoint of a path obtained by lifting p from f(x) to a path in the total space $\sum_{(x:A)} B(x) \to A$ [7]. The following equation gives the type of $p*$ also known as $transport$.

$$transport^B_p : B(x) \to B(y) \tag{11}$$

where $p : x = y$ for $x, y : A$.

In section 3 we will give an example of encoding a cryptographic scheme using higher inductive type and explain how to map this higher inductive type to a concrete implementation of the scheme in the universe. In section 4 we will discuss how to design higher dimensional paths to enforce restrictions on the implementation of a cryptographic protocol.

III. Higher Inductive Type front-end for OTP

In this section, we will discuss an encoding of the one-time pad using a higher inductive type with a path constructor to specify the encryption function [2]. We will construct a proof for an equivalence which reflects the encryption path of the higher inductive type in the universe. The functional correctness property, which states that decryption inverts encryption, will be part of the construction of the proof for the equivalence. We will then map this higher inductive type, with the encryption path, to a concrete implementation of the one-time pad, with the equivalence reflecting the encryption path, in the universe. The encryption and the decryption functions are then projected from the concrete implementation in the universe using the higher inductive type which acts as a front-end. By accessing the concrete implementation of the one-time pad through a higher inductive type, we can get a certificate or a guarantee on the functional correctness of the system. Some other property such as homomorphic encryption requires introducing higher-dimensional paths to act as a certificate. We will discuss higher-dimensional paths in section 4.

A. *One-time Pad*

The following Agda code gives the higher inductive type encoding of the one-time pad.

```
data OTP (n : Nat) : Set where
  -- point constructors
  message : OTP n
  cipher  : OTP n

  -- path constructors
  otp-encrypt : {n : Nat} → (key : Vec Bit n) →
    message {n} ≡ cipher {n}
```

The higher inductive type `OTP` has two point constructors `message` and `cipher` representing the plain-text and the cipher-text respectively. The path constructor `otp-encrypt` represents the encryption function of the one-time pad. We parameterize the type `OTP` with the length `n` of the data. `otp-encrypt` uses the same length parameter `n` to specify the length of the key which encodes another restriction, namely the length of the key for the one-time pad should be equal to the length of the message, which is crucial for the security of the one-time pad.

The following code gives the recursion principle and its action on constructors or the computation rules for `OTP`.

```
otp-rec : {n : Nat} →
  (B : Set) →
  (b-msg : B) →
  (b-cipher : B) →
  (b-encrypt : (key : Vec Bit n) → b-msg ≡ b-cipher) →
  OTP n → B
otp-rec B b-msg b-cipher b-encrypt message = b-msg
otp-rec B b-msg b-cipher b-encrypt cipher = b-cipher

postulate
  β-otp-rec : {n : Nat} →
    (B : Set) →
    (b-msg : B) →
    (b-cipher : B) →
    (b-encrypt : (key : Vec Bit n) → b-msg ≡ b-cipher) →
    {key : Vec Bit n} →
    ap (otp-rec B b-msg b-cipher b-encrypt)
       (otp-encrypt key) ≡ (b-encrypt key)
```

The recursion principle `otp-rec` states that when given a type `B` with point constructors `b-msg` and `b-cipher` and path constructor `b-encrypt`, there exists a function of type `OTP n → B`. `otp-rec` maps `message` and `cipher` to `b-msg` and `b-cipher` respectively. `β-otp-rec` gives the action of `otp-rec` on the path `(otp-encrypt key)` which maps it to the path `(b-encrypt key)`. Equation (9) gives the type of `ap`. The computation rules for point constructors `message` and `cipher` are given as definitional equalities specified as part of `otp-rec`. The computation rule for the path `otp-encrypt` is postulated as propositional equality.

The following code gives the induction principle and its computation rules for `OTP`.

```
otp-ind : {n : Nat} →
  (B : OTP n → Set) →
  (b-msg : B (message)) →
  (b-cipher : B (cipher)) →
  (b-encrypt : (key : Vec Bit n) →
     transport B (otp-encrypt key) b-msg ≡ b-cipher) →
  (x : OTP n) → B x
otp-ind B b-msg b-cipher b-encrypt message = b-msg
otp-ind B b-msg b-cipher b-encrypt cipher = b-cipher

postulate
  β-otp-ind : {n : Nat} →
    (B : OTP n → Set) →
    (b-msg : B (message)) →
    (b-cipher : B (cipher)) →
    (b-encrypt : (key : Vec Bit n) →
      transport B (otp-encrypt key) b-msg ≡ b-cipher) →
    {key : Vec Bit n} →
    apd (otp-ind B b-msg b-cipher b-encrypt)
        (otp-encrypt key) ≡ (b-encrypt key)
```

The induction rule `otp-ind` states that when given a type `B : OTP n → Set` along with points `b-msg`, `b-cipher` and path `b-encrypt`, there exists a dependent function `(x : OTP n) → B x`. The computation rule for path `b-encrypt` is postulated as propositional equality. Equation (10) gives the type of `apd` and equation (11) gives the type of `transport` where p is the path `(otp-encrypt key)`.

B. *Implementation of one-time pad in the universe*

The functional programming aspect of homotopy type theory allows us to implement any cryptographic schemes. In this section, we will develop a concrete model for the higher inductive type `OTP` described in section 3.1. The encryption function for the one-time pad is straightforward, and it is implemented using `xor`. The encryption of one-time pad is defined using the following function.

```
OTP-encrypt : {n : Nat} →
```

[2] See **https://github.com/pavenvivek/FCS-2018**

```
  (key : Vec Bit n) →
  (message : Vec Bit n) → Vec Bit n
OTP-encrypt {n} key message = message xorBits key
```

where xorBits perform xor on two vectors of equal length.

Similar to keys, we have chosen to use the type Vec Bit n to represent the point constructors message and cipher of the higher inductive type OTP in the universe. Therefore, the path otp-encrypt should be mapped to an equivalence formed by the function OTP-encrypt between types Vec Bit n and Vec Bit n. To create an equivalence for the function OTP-encrypt, we need a proof element of type given by equation (5). To construct a proof element of (5), we need a function g : Vec Bit n → Vec Bit n, a proof element of fog ∼ id, and a proof element of gof ∼ id. For the one-time pad, the encryption function is also its inverse. So both f and g are represented by OTP-encrypt in this case. Therefore, the types fog ∼ id and gof ∼ id are definitionally the same. The equivalence formed by OTP-encrypt is defined as follows.

```
OTP-equiv : {n : Nat} →
  (key : Vec Bit n) →
  Vec Bit n ≃ Vec Bit n
OTP-equiv key = ((OTP-encrypt key) ,
                 equiv₁ (mkqinv (OTP-encrypt key)
                        (α-OTP key) (α-OTP key)))

(α-OTP key) : (OTP-encrypt key (OTP-encrypt key msg))
              ≡ msg
```

In the above code, (OTP-equiv key) is of the type given by equation (2). equiv_1 forms a proof element of the type given by equation (3). The type of mkqinv is given by equation (6) which takes an element of (5) as input and gives an element of (3) as output. (α-OTP key) is a proof which says the encryption of msg, implemented by OTP-encrypt, followed by its decryption, which is also implemented by OTP-encrypt in this case, is the same as msg.

C. Mapping OTP into the universe

The higher inductive type OTP defined in section 3.1 can now be mapped into the universe using univalence. The abstract nature of higher inductive types also means that we can map the same type to more than one concrete implementation in the universe whenever compatible. The equivalence (OTP-equiv key) respects the path structure specified by the constructor otp-encrypt. Because of this, a path formed by univalence given by (ua (OTP-equiv key)) represents the path structure of otp-encrypt in the universe. This correspondence allows us to define a mapping I-OTP which maps the points message, cipher of OTP to type Vec Bit n and a mapping I-OTP-path which maps the path (otp-encrypt key) to (ua (OTP-equiv key)).

```
I-OTP : {n : Nat} →
  OTP n → Set
I-OTP {n} bits = otp-rec Set
                  (Vec Bit n)
                  (Vec Bit n)
                  (λ key → ua (OTP-equiv key)) bits

I-OTP-path : {n : Nat} →
  (key : Vec Bit n) →
  ap I-OTP (otp-encrypt {n} key)
  ≡ ua (OTP-equiv key)
I-OTP-path {n} key = β-otp-rec Set
                      (Vec Bit n)
                      (Vec Bit n)
                      (λ k → ua (OTP-equiv k))
```

I-OTP is defined using the recursion principle otp-rec of the higher inductive type OTP. It maps the points of OTP to the type Vec Bit n in the universe represented by Set. I-OTP-path maps the path (otp-encrypt key) to (ua (OTP-equiv key)) using β-otp-rec. Now we can define an interpreter function ITP using coe given by equation (7) as follows.

```
ITP : {n : Nat} →
  {a b : OTP n} →
  (p : a ≡ b) →
  (I-OTP a) → (I-OTP b)
ITP {n} {a} {b} p = coe (ap I-OTP p)
```

When we give the path otp-encrypt as input, the interpreter ITP returns the encryption function OTP-encrypt. By accessing a concrete implementation in the universe using a higher inductive type, we get the certificate or guarantee specified by the path structures of the higher inductive type. In the case of OTP, the functional correctness property is part of the equivalence (OTP-equiv key) given by (α-OTP key), and the path otp-encrypt will reflect this through the mapping specified by I-OTP-path.

We will consider an example of using ITP to extract OTP-encrypt and its application on a vector.

```
pf : (ITP
       (otp-encrypt (1b :: (0b :: [])))
       (1b :: (1b :: [])))
     ≡ (0b :: (1b :: []))
```

In the above code, ITP takes otp-encrypt as input with key (1b :: (0b :: [])) and plain-text (1b :: (1b :: [])) and returns the cipher-text (0b :: (1b :: [])) as output.

IV. Encoding Properties as Higher Dimensional Paths

The path otp-encrypt described in the previous section is one-dimensional. We can also encode domain-specific cryptographic properties as higher dimensional paths. In this section, we will design properties of a database model with multi-layered encryptions in the style of cryptDB [10] as higher dimensional paths. CryptDB has different layers of encryption known as *onion layers* of encryption. The idea of cryptDB is to allow computation on top of encrypted data without the need to decrypt them. For example, homomorphic encryption can be used to implement addition, and deterministic encryption can be used to perform equality comparison on top of encrypted data. Similarly, order-preserving encryption can be used to implement inequality comparisons on encrypted data. A higher inductive type can be used to define the computational behavior of cryptDB. We will consider the following higher inductive type specification to discuss encoding domain-specific laws of cryptDB as higher dimensional paths [3]. CryptDB involves non-bijective functions, and can be implemented using singleton types [12]. In this section, we will not be focusing on the implementation details or mapping types into the universe.

```
data encDB : Set where
  -- point constructors
  tab : encDB
  tabDET : encDB
  tabHOM : encDB
  tabOPE : encDB

  -- one-dimensional paths
  hom-enc : tab ≡ tabHOM
  det-enc : tab ≡ tabDET
  ope-enc : tab ≡ tabOPE
```

The higher inductive type `encDB` specifies a lot of restrictions and a mapping to a concrete implementation should respect those restrictions. For example, it says that homomorphic encryption is a function that takes a plain-text table `tab` as input and gives an encrypted version of the table `tabHOM` as output. The inverse path `(! hom-enc)` specifies the decryption function. Similarly, the paths `det-enc` and `ope-enc` specifies the deterministic and order-preserving encryption schemes respectively. The higher inductive type `encDB` acts as a single interface giving a lot of information on underlying implementation of a cryptographic setting. It provides us with a graphical model composed of points, paths, paths between paths or higher dimensional paths to specify about correctness properties and various domain-specific laws of a cryptographic construction. In the remainder of this section, we will discuss homotopies or path between paths describing properties specific to homomorphic encryption, deterministic encryption, and order-preserving encryption.

A. Homomorphic Encryption

Homomorphic encryption can be used to perform computations on cipher-text. In cryptDB, homomorphic encryption

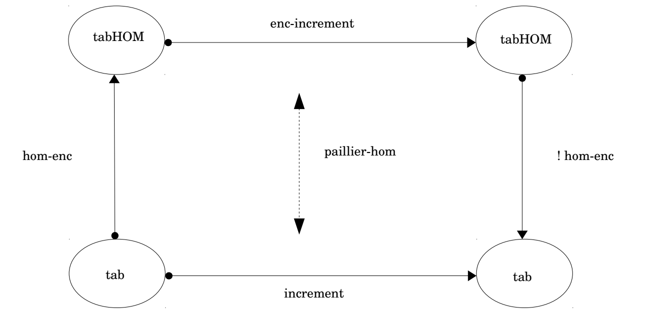

Figure 1. Homotopy representing the homomorphic property of paillier cryptosystem. The path `hom-enc` concatenated with `enc-increment` and `(! hom-enc)` is equal to the path `increment`.

is implemented using paillier cryptosystem. According to the homomorphic property of paillier cryptosystem [11], the addition of two plain-texts will be equal to the multiplication of their corresponding cipher-text. We can express this property as a two-dimensional path saying homomorphic encryption of a plain-text concatenated with a path expressing homomorphic multiplication concatenated with homomorphic decryption is the same as the regular addition performed on the plain-text.

The encoding of cryptDB in homotopy type theory involves non-bijective queries. Mapping a non-bijective query into the universe is not possible in the current type-theoretic setting. However, we can map a non-bijective path to singleton types in the universe [12]. Such a mapping holds because any function between singleton types is automatically a bijection.

B. Deterministic Encryption

Deterministic encryption generates the same cipher-text on multiple encryptions of the same plain-text. In cryptDB, a deterministic encryption scheme is used to perform equality comparisons on encrypted data. The correctness property of deterministic encryption requires $DET(m1) \equiv DET(m2)$ when $m1 \equiv m2$. We can specify this property as a heterogenous path over a path of type $m1 \equiv m2$. For example, when `tab` and `det-enc` encode the plain-text as an implicit argument given by `tab : {m} → encDB` and `det-enc : {m} → tab ≡ tabDET` respectively, we can define the following two-dimensional path.

```
det-correctness : (p : m1 ≡ m2) →
   transport (λ x → tab {x} ≡ tabDET) p (det-enc {m1})
   ≡
   (det-enc {m2})
```

where `det-correctness` says that the path `(det-enc {m1}) ≡ (det-enc {m2})` lies over `p : m1 ≡ m2`.

C. Order-Preserving Encryption

Order-preserving encryption [13] allows inequality comparisons on encrypted data without the need to decrypt them. Order-preserving encryption requires, for plain-texts x and y, if $(x < y)$ then $OPE(x) < OPE(y)$. We cannot specify

[3]We have simplified the higher inductive type `encDB` for ease of understanding. Many details have been omitted from the specification given in this paper. Please refer to **https://github.com/pavenvivek/FCS-2018** for detailed implementation of cryptDB.

this property in the style of `det-correctness` because inequality relation does not form paths. However, we can use a different approach to model this restriction in a higher inductive type. For example, consider a function `bigE (m1, m2)` which returns the biggest of two elements. When there exists a path `p' : bigE(m1, m2)` $\equiv$ `bigE(c1, c2)`, where c1 and c2 are the OPE cipher values of m1 and m2 respectively, lying in the space `encDB`, we can design a two-dimensional path saying `ope-encrypt` is the same path as `p'`. This two-dimensional path will hold only when the order-preserving encryption respects the inequality relation between the plain-texts.

The two-dimensional paths discussed above capture different domain-specific laws that should be respected by any concrete implementation of a multi-layered database in the style of cryptDB. By specifying the above paths as constructors of `encDB` and by mapping `encDB` to a concrete implementation in the universe similar to `OTP` in section 3, we can achieve various guarantees on the correctness of the implementation. The mapping of the higher inductive type into the universe alone is enough to guarantee on the correctness of properties specified by the path constructors because of univalence and functoriality. By having a higher inductive type front-end for a cryptographic implementation, we eliminate the need to generate individual proofs for different domain-specific properties. Also in a higher inductive type framework, we have a way to relate proofs of different properties because of the encoding of proofs as paths or higher dimensional paths of a single type.

V. Limitations and Future Work

A limitation of homotopy type theory is that the univalence can be added only as an axiom. This limitation weakens the good computational properties of type theory. We would like to develop the framework described in this paper using *cubical type theory* [14]. In cubical type theory, the univalence computes and is no longer an axiom.

Another limitation is that the mapping of higher inductive type into the universe requires the functions represented by paths to be bijective. We cannot specify all functions as bijections. In the case of cryptDB, functions implementing queries like insert and delete are not bijective and therefore cannot be encoded as paths in a higher inductive type. The functions with simple retractions are not acceptable. Every function should have inverses to be expressed as paths. One way to work around this problem is to encode functions as mappings between singleton types in the universe [12]. Any function mapping between two singleton types is automatically a bijection. So a path representing a non-bijective function in a higher inductive type can be mapped to bijection formed by a function between singleton types in the universe. Future work in this direction would be to characterize mapping of partial bijections to paths using the tools of homotopy theory. Another direction is to develop type theory with non-symmetric paths based on *directed type theory* [15]. In the current setting, since homotopy type theory is also a functional programming language, the non-bijective functions can be used along with higher inductive types. So the benefits of having a higher inductive type representing bijective functions can still be achieved. Homotopy type theory also allows us to postulate bijections as axioms and work with them. When we have a proof that a function is bijective in a different setting, then the function can be postulated as a bijection in homotopy type theory and can be encoded using a higher inductive type.

Probabilistic encryption schemes are not bijective. It might not be possible to map them to singleton types in the universe because they compute to different values during each execution with overwhelming probability and does not uniquely identify the contents of a singleton type. So the probabilistic encryption schemes have to be encoded as regular mappings and can be used along with higher inductive types. Another limitation is the difficulty involved in deriving proofs for bijections. This limitation increases development time and effort. But after application development, we can achieve overwhelming guarantee on the correctness of the application. In the real-world applications, bug fixing has taken much more effort than the original development effort [16] [17] [18]. So the cost of the increase in development effort can be ignored considering the benefits achieved. It can be very significant especially when implementing cryptographic protocols because a flawed implementation of cryptographic protocol leads to serious security issues resulting in the compromise of the entire application. Agda also has a robust reflection library which can be used to automate the generation of proofs [19] and elimination rules for higher inductive types. Automated code generation can reduce the development effort to some extent. In the future, we would like to encode the security properties of a cryptographic scheme as paths in a higher inductive type and explore how to achieve security guarantees using this setting.

VI. Related Work

The work discussed in this paper takes a first step towards formal specification of cryptographic protocols based on higher inductive types. There are other works which support formal specification of cryptographic constructions using different settings for handling cryptographic primitives including shared-key and public-key cryptography, signatures, hash functions, message authentication codes etc. In this section, we will review few of those works.

A. Foundational Cryptography Framework

The Foundational Cryptography Framework (FCF) [1] implements a probabilistic programming language embedded inside Coq [5] proof assistant. Unlike Agda, the Coq proof assistant is based on the $Calculus\, of\, Inductive\, Construction$. However, the recent version of Coq allows the sort `Set` to be predicative. The probabilistic programming language defined by FCF enables the specification of cryptographic schemes, security definitions, and hard problems. A shallow embedding of the probabilistic language allows FCF to have access to the capabilities of the metalanguage (Coq) including dependent types and higher-order functions. It also allows any theory developed in the host language accessible to the embedded language. The technique described in this paper uses a higher inductive type to specify cryptographic protocols in a formal setting. Coq does not have a built-in mechanism to support the definition of higher inductive types. However, we can still work with higher inductive types in Coq [9] [5]. With further work, the shallow embedding can make the constructions involving higher inductive types visible to the Foundational Cryptography Framework.

B. CryptoVerif

The work of [22] implemented in CryptoVerif provides a mechanized prover for showing correspondence assertions which are useful to express authentication properties for cryptographic protocols in the computational model. The proof construction follows the sequences of games approach in cryptography. CryptoVerif is based on $Process\,Calculus$ extended with parametric events to serve in the definition of correspondences. The work also discusses proving mutual authentication and authenticated key exchange using correspondences. CryptoVerif incorporates efficient automation reducing the proof development effort but lacks interactive proof development features which makes it more specific to only a subset of cryptographic constructions when compared to FCF or EasyCrypt.

C. ProVerif

ProVerif [23] is a cryptographic protocol verifier for the automated reasoning of security properties based on Dolev-Yao model. It can be used for proving secrecy, authentication, and equivalences between processes differing only by terms. The input protocols to ProVerif are modeled using $Pi\,Calculus$ and internally translated using Horn clauses. The security properties which needs to be proved are translated to derivability queries on these clauses. ProVerif can handle different cryptographic primitives including shared-key and public-key cryptography, hash functions, and Diffie-Hellman key agreements.

D. EasyCrypt

CertiCrypt [21], a framework built upon the Coq proof assistant, enables machine-checked construction and verification of cryptographic schemes. The proof development in CertiCrypt is time-consuming, and EasyCrypt [20] was developed to address this limitation by speeding up the construction of proofs using automation based on SMT solvers. Both CertiCrypt and EasyCrypt has a deep embedding of a probabilistic programming language which is used for proof construction. The deep embedding makes them inaccessible to the cozy features of the host language (Coq) such as dependent-types, higher-order functions, modules, etc.

E. Verypto

Verypto [24], a framework implemented in Isabelle proof-assistant [25], provides a formal language for the specification and verification of game-based cryptographic security proofs. Verypto includes a probabilistic higher-order functional programming language with recursive types, references, and events to express constructs of a game-based security proof. The language handles stateful higher-order objects such as oracles, arbitrary data types and supports event-based reasoning patterns. Like CertiCrypt and EasyCrypt, the probabilistic programming language used for proof construction in Verypto follows a deep embedding.

VII. Conclusion

We have shown how to implement a cryptographic scheme using the tools of homotopy type theory. The higher inductive type acts as a front-end and provides us with a graphical computational model. The points, paths, and paths between paths are mapped to concrete implementation in the universe using univalence. The groupoid laws come for free [12], and we don't have to prove them explicitly. We use the elimination rule of the higher inductive type for mapping them into the universe. Because of the functoriality of functions, the two-dimensional paths describing the domain-specific properties related to homomorphic encryption, deterministic encryption, and order-preserving encryption are realized by equivalences or bijections respecting the corresponding properties. The higher inductive type front-end can be used to project functions from the underlying computational model by providing paths using coercion. By having a higher inductive type front-end, we can achieve various guarantees on the correctness of the underlying cryptographic implementation.

The limitations of homotopy type theory, namely having univalence only as an axiom and the requirement for functions to have inverses has restricted us to only a subset of cryptographic schemes to be benefitted by the model described in this paper. Nevertheless, there is a lot of work going on to improve type theory to allow for univalence to compute and mapping of non-bijective functions into the universe which can reduce the restrictions and enable us to encode more interesting cryptographic constructions using the higher inductive type model. However, this paper introduces the tools of homotopy type theory to the cryptographic community and also acts as a precursor of more interesting type theoretical settings to follow which can significantly improve the framework described in this paper.

References

[1] Petcher, A., Morrisett, G.: The foundational cryptography framework. In: Focardi R., Myers A. (eds) Principles of Security and Trust (POST). Lecture Notes in Computer Science, vol 9036. Springer, Berlin, Heidelberg (2015)

[2] Berg, M.: Formal verification of cryptographic security proofs. Ph.D. thesis, Saarland University (2013) http://www.infsec.cs.uni-saarland.de/~berg/publications/thesis-berg.pdf

[3] Durumeric, Z., Kasten, J., Adrian, D., Halderman, J.A., Bailey, M., Li, F., Weaver, N., Amann, J., Beekman, J., Payer, M., Paxson, V.: The matter of Heartbleed. In: Proceedings of the 2014 Conference on Internet Measurement Conference. Vancouver, BC, Canada (2014)

[4] Lazar, D., Chen, H., Wang, X., Zeldovich, N.: Why does cryptographic software fail?: a case study and open problems. In: Proceedings of 5th Asia-Pacific Workshop on Systems (APSys). Beijing, China (2014)

[5] Coq Development Team. The Coq proof assistant reference manual, version 8.2. INRIA (2009). http://coq.inria.fr/

[6] Norell, U.: Towards a practical programming language based on dependent type theory. PhD thesis, Chalmers University of Technology, Sweden (2007)

[7] The Univalent Foundations Program, Institute for Advanced Study. Homotopy Type Theory: Univalent foundations Of Mathematics (2013). Available from homotopytypetheory.org/book

[8] Duan, J., Hurd, J., Li, G., Owens, S., Slind, K., Zhang J.: Functional correctness proofs of encryption algorithms. In: Proceedings of the 12th international conference on Logic for Programming, Artificial Intelligence, and Reasoning, pp. 519–533. Jamaica (2005)

[9] Licata, D.: Running circles around (In) your proof assistant; or, Quotients that Compute (2011). Unpublished. http://homotopytypetheory.org/2011/04/23/running-circles-around-in-your-proof-assistant

[10] Popa, R.A., Redfield, C.M.S, Zeldovich, N., Hari Balakrishnan, H. : CryptDB: Protecting confidentiality with encrypted query processing. In: Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP), Cascais, Portugal (2011)

[11] Paillier, P.: Public-key cryptosystems based on composite degree residuosity classes. In: Proceedings of the 18th Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT). Prague, Czech Republic (1999).

[12] Anguili, C., Morehouse, E., Licata, D., Harper, R.: Homotopical patch theory. In: International Conference on Functional Programming (ICFP), Gothenburg, Sweden (2014)

[13] Agrawal, R., Kiernan, J., Srikant, R., and Xu, Y.: Order preserving encryption for numeric data. In Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data. Paris, France (2004).

[14] Cohen, C., Coquand, T., Huber, S., Mörtberg, A.: Cubical Type Theory: a constructive interpretation of the univalence axiom. In: Leibniz International Proceedings in Informatics. pp. 1–33. Germany (2015)

[15] Licata, D., Harper, R.: 2-dimensional directed type theory. In: Mathematical Foundations of Programming Semantics (MFPS). (2011)

[16] Ben Othmane, L., Chehrazi, G., Bodden, E., Tsalovski, P., Brucker, A.D., Miseldine, P.: Factors impacting the effort required to fix security vulnerabilities. In: Lopez J., Mitchell C. (eds) Information Security (ISC). Lecture Notes in Computer Science, vol 9290. Springer, Cham (2015)

[17] Hamill, M., Goseva-Popstojanova, K.: Software faults fixing effort: analysis and prediction. In: Technical report 20150001332, NASA Goddard Space Flight Center. Greenbelt, MD United States (2014)

[18] Cornell, D.: Remediation statistics: what does fixing application vulnerabilities cost? In: Proceedings of the RSAConference. San Fransisco, CA, USA (2012)

[19] Kokke, P., Swierstra, W.: Auto in Agda. In: Hinze R., Voigtländer J. (eds) Mathematics of Program Construction (MPC). Lecture Notes in Computer Science, vol 9129. Springer, Cham (2015)

[20] Barthe, G., Grégoire, B., Heraud, S., Zanella Béguelin, S.: Computer-aided security proofs for the working cryptographer. In: Advances in Cryptology - CRYPTO 2011. Lecture Notes in Computer Science, vol. 6841, pp. 71–90. Springer (2011)

[21] Barthe, G., Grégoire, B., Zanella Béguelin, S.: Formal certification of code-based cryptographic proofs. In: 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009. pp. 90–101. ACM (2009)

[22] Blanchet, B.: Computationally sound mechanized proofs of correspondence assertions. In: 20th IEEE Computer Security Foundations Symposium (CSF'07). pp.97–111. IEEE, Venice, Italy (Jul 2007)

[23] Blanchet, B.: Modeling and Verifying Security Protocols with the Applied Pi Calculus and ProVerif. In: Foundations and Trends in Privacy and Security. vol. 1, num. 1-2, pp.1–135. (Oct 2016)

[24] Berg, M.: Formal Verification of Cryptographic Security Proofs. Ph.D. thesis. Saarland University (2013) http://www.infsec.cs.uni-saarland.de/~berg/publications/thesis-berg.pdf

[25] Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL - A Proof Assistant for Higher-Order Logic. In: Lecture Notes in Computer Science, vol. 2283. Springer (2002).