# SecureScala: Scala Embedding of Secure Computations

Markus Hauck
codecentric AG [*]
markus.hauck@codecentric.de

Savvas Savvides
Purdue University
savvas@purdue.edu

Patrick Eugster
TU Darmstadt, Purdue University
peugster@dsp.tu-darmstadt.de

Mira Mezini
TU Darmstadt
mezini@cs.tu-darmstadt.de

Guido Salvaneschi
TU Darmstadt
salvaneschi@cs.tu-darmstadt.de

## Abstract

Cloud computing offers an attractive and cost-efficient computing platform and hence it has been widely adopted by the industry and the government. At the same time, cloud computing poses a serious security challenge because sensitive data must often be outsourced to third party entities that can access the data and perform computations on them.

Partial homomorphic encryption is promising for secure computation, since it allows programs to be executed over encrypted data. Despite advances in cryptographic techniques have improved the expressivity of such programs, integration with mainstream languages has seen little progress. To this end, we present SecureScala, a domain-specific language in Scala that allows expressing secure programs without requiring any cryptographic knowledge. SecureScala is based on a novel combination of free monads and free applicative functors and supports parallel execution and static analyzability. We evaluate our approach through several case studies, demonstrate its expressivity, and show that it incurs in limited performance overhead.

***Categories and Subject Descriptors*** D.3.3 [*Programming Languages*]: Language Constructs and Features

***Keywords*** Domain-specific language, Secure computation

## 1. Introduction

Cloud computing offers on-demand provisioning of resources, seemingly unlimited scalability and other desirable properties, e.g., fault tolerance, minimal maintenance and cost reduction. To take advantage of these opportunities, the program representing a computation at hand and the associated data is uploaded into the cloud and executed there. Moving computations to the cloud, however, forces developers to face a number of security challenges. While the cloud is a viable approach for non-sensitive and public data, it is problematic for scenarios where data is sensitive, requiring protection from both adversaries and the cloud provider. Traditional encryption targeted to protect data while passing communication channels does not offer a solution for the cloud-based computation scenario because the program running on the remote hardware is not allowed to decrypt the data during the execution – plain text data would be available to the owners of the remote host.

Homomorphic encryption schemes [10] allow computations over *encrypted* data. Hence, private data can be uploaded to the cloud and processed in the encrypted form, without leaking information. Available homomorphic encryption schemes differ in the operations they support. Fully homomorphic encryption schemes (FHE) support at least multiplication and addition, from which all other operations can be derived. While in theory FHE seems to be the ultimate solution to the problem of computing over encrypted data, the computational overhead makes it unviable in practice [11]. Partial homomorphic encryption (PHE) schemes are less expressive – they support computation over encrypted data with respect to specific operations, but in contrast to FHE, exhibit acceptable performance. Examples of such schemes are Paillier [20] (addition), ElGamal [9] (multiplication), OPE [6] (order comparisons) and AES [11] (equality comparisons).

PHE schemes have been used in several systems to provide practical and provable security [13, 21, 26]. Yet, these approaches adopt special-purpose languages, such as SQL [21] or PigLatin [26] to specify computations. A PHE embedding in a general-purpose language that enables reuse of existing libraries, fostering modularity and composition of secure computations, is still open research.

[*] Work done while at TU Darmstadt

To fill this gap, we present SecureScala, an embedded domain-specific language (DSL) for Scala, that allows developers to write computations over encrypted data without requiring sophisticated cryptographic knowledge, without forfeiting integration with a general-purpose language. SecureScala is based on a novel combination of a well known approach using free monads [24, 25] and more recent research on free applicative functors [7]. While applicative functors are less expressive than monads, the latter suffer from the lack of static analyzability and do not allow implicit parallelism. The resulting DSL allows programmers to combine these styles, retaining the benefits of both. Since each PHE scheme supports only a limited set of operations, conversions – transparent to the user – allow to express programs that include unsupported operations. Different interpreters allow processing locally for testing purposes, on remote hardware in a distributed fashion, or with an optional variety of optimization such as implicit parallelism and transformations achieved via static analysis.

We evaluate SecureScala demonstrating the benefits of embedding a DSL into a general-purpose language for secure computation. Our scenarios include queries over encrypted event streams – integrating with the complex event processing engine Esper [1] – and graphical user interfaces using encrypted data – based on the RxScala [2] reactive programming framework. More specifically, we make the following contributions:

- We analyze techniques for developing an embedded DSL for computations over encrypted data in Scala, contrasting a DSL using free monads with another using free applicative functors.
- We combine the two DSLs into SecureScala to get the best of both approaches by allowing programs to depend on previous effects, still exploiting implicit parallelism and static analyzability for the applicative parts.
- We evaluate SecureScala with case studies and benchmarks that show the performance of our solution and demonstrate the interaction with existing libraries due to PHE support integrated into a general-purpose language.

We structure this paper starting with an overview of our approach in Section 2 and then construct SecureScala in steps in Section 3, Section 4 and Section 5. We show an empirical evaluation in Section 6, we contrast our solution with related work in Section 7 before concluding in Section 8.

## 2. Overview

We give an overview of the execution model and the encryption schemes we use and introduce Cryptographic data types which form the basis of SecureScala.

### 2.1 Execution Model

Figure 1 shows an execution model of a program utilizing a third party cloud service. The user submits a program for execution through the client interface. The client deploys the program to the (untrusted) cloud service (along with any required encrypted data) which executes the program over encrypted data. The cloud service can utilize a trusted service to overcome limitations of PHE, by re-encrypting data as necessary, i.e., converting between encryption schemes (Section 3.4). Upon program completion, the encrypted results are returned to the client, decrypted and returned to the user in plain text.

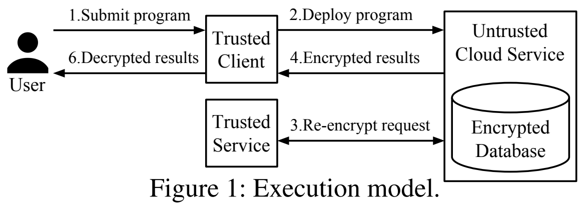


Figure 1: Execution model.

### 2.2 Encryption Schemes

Table 1 shows the encryption schemes used to perform operations over encrypted data, similar to those used in previous work [13, 21]. Some schemes are *symmetric* and use a single *secret* key, others are *asymmetric* and use a *public/private* key pair. Asymmetric schemes allow encrypting data on the untrusted cloud since encryption in asymmetric schemes only requires the public key. In contrast, symmetric schemes do not allow encryption operations on the untrusted cloud. Hence it is necessary to provide encrypted versions for all required values in advance , e.g., constants like 0 and 1. A `KeyRing(pub: PubKeys, priv: PrivKeys)` class encapsulates encryption keys for each scheme – `PubKeys` are available only in asymmetric schemes.

| Scheme | Operation | Input Data Types | Scheme Type |
|---|---|---|---|
| Paillier [20] | Addition | Integers | Asymmetric |
| ElGamal [9] | Multiplication | Integers | Asymmetric |
| AES [12] | Equality | Integers, Strings | Symmetric |
| OPE [6] | Ordering | Integers, Strings | Symmetric |

Table 1: Overview of the encryption schemes.

### 2.3 Cryptographic Data Types

A first step towards building a DSL for secure computation is to define a set of data types that capture the semantics of the operations supported by the encryption schemes described above. Since the input data types (plain text data types) of the encryption schemes we use are limited to integer and string, we start by defining `EncInt` and `EncStr` as `sealed trait`, representing an encrypted integer and an encrypted string respectively. For each, we further distinguish the concrete scheme type. We present a simplified representation of our scheme types for the case of `EncInt` below:

```scala
sealed trait EncInt
case class PaillierEnc(value: BigInt) extends EncInt
case class ElGamalEnc(x: BigInt, y: BigInt) extends EncInt
case class AesEnc(value: Array[Byte]) extends EncInt
case class OpeEnc(value: BigInt) extends EncInt
```

`EncInt` represents an encrypted integer and case classes represent an integer encrypted under a specific scheme.

Knowledge of the scheme indicates what operations are allowed on the encrypted data. Our functions use pattern matching to ensure operands are of the required scheme, or make appropriate re-encryptions before the required computation. For example, to add two encrypted values, we must ensure that the values are of type `PaillierEnc` which is the scheme that supports addition over encrypted data. A simplified version of the `addition` function is shown below.

```
def addition(lhs: EncInt, rhs: EncInt): Option[EncInt] =
    (lhs,rhs) match {
  case (PaillierEnc(l),PaillierEnc(r)) => Some(Paillier(l,r))
  case _ => None
}
```

In the first case, both operands are encrypted under a compatible scheme, so the implementation delegates to the internal function *Paillier* to perform the encrypted addition. We give more details on how we handle cases where one or more operands are not encrypted under the required encryption scheme in Section 3. Parameters of encryption schemes for numbers are chosen such that we are able to encrypt 64 bit integers, internally we represent the ciphertext as a `BigInt`. In the following sections we use Cryptographic data types to describe SecureScala.

## 3. A DSL Based on Free Monads

We present a simplified version of a DSL using free monads [24, 25] and show the design of an interpreter.

```
sealed trait CryptoF[+K]
case class Plus[K](lhs: EncInt, rhs: EncInt,
  k: PaillierEnc => K) extends CryptoF[K]
case class Mult[K](lhs: EncInt, rhs: EncInt,
  k: ElGamalEnc => K) extends CryptoF[K]
case class Equals[K](lhs: EncInt, rhs: EncInt,
  k: Boolean => K) extends CryptoF[K]
case class Compare[K](lhs: EncInt, rhs: EncInt,
  k: Ordering => K) extends CryptoF[K]
```

Listing 1: A simplified version of the `CryptoF` functor.

### 3.1 Free Monads

We define the base functor for the Scalaz [4] free monad and specify the possible operations in the DSL, reusing the definition of `EncInt`. We also define a new type `CryptoF` and add cases for every operation, i.e., addition (`Plus`), multiplication (`Mult`), equality (`Equals`) and order comparisons (`Compare`), as shown in Listing 1. We present details for the `Plus` operation and the corresponding smart constructor `add` below. Other cases are analogous.

```
1 def add(lhs: EncInt, rhs: EncInt): Crypto[EncInt] =
2       Free.liftF(Plus(lhs,rhs,identity))
3 def multiply(lhs: EncInt, rhs: EncInt): Crypto[EncInt] =
4       Free.liftF(Mult(lhs,rhs,identity))
5 def equal(lhs: EncInt, rhs: EncInt): Crypto[Boolean] =
6       Free.liftF(Equals(lhs,rhs,identity))
7 def compare(lhs: EncInt, rhs: EncInt): Crypto[Ordering] =
8       Free.liftF(Compare(lhs,rhs,identity))
```

Listing 2: Smart constructors perform the lifting.

The `add` constructor shown in Line 2 of Listing 2 takes the left- and right-hand operands for the addition operation and a function from `PaillierEnc` to `K`. The latter is a continuation which receives an argument of type `PaillierEnc` – the result of interpreting the `Plus` operation and the operands. Returning `PaillierEnc` instead of `EncInt` retains information useful for next steps. Classes representing operations are not exposed directly to users, instead we expose smart constructors for each case (Listing 2). Based on `CryptoF`, we define a type alias for our free monad: `type Crypto[A] = Free[CryptoF,A]`

### 3.2 Simple Programs

With the DSL described so far, we can write a first program that increments an encrypted integer. We assume all variables have been properly initialized to hold the required encrypted values.

```
lazy val one: EncInt = ...
def plus1(in: EncInt): Crypto[EncInt] = add(in,one)
```

The result type `Crypto[EncInt]` implies that given an `EncInt`, running the program results in a value of the Cryptographic data type `EncInt`. The use of a free monad enables monadic combinators, as well as Scala's for-comprehensions. To avoid encrypting constants (non-sensitive data) in advance we extend the DSL with a new encryption operation:

```
sealed trait Scheme
case object Paillier extends Scheme
case object ElGamal extends Scheme
case class Encrypt[K](s: Scheme, plain: Int,
  k: EncInt => K) extends CryptoF[K]
def encrypt(s: Scheme)(plain: Int): Crypto[EncInt] =
  Free.liftF(Encrypt(s,plain,identity))
```

We now present a program that adopts three different DSL primitives, namely addition, multiplication and equality comparison to return a `Boolean` value after interpretation shown in Listing 3. The program takes 2 encrypted parameters. It increments the first parameter and multiplies it by 2 before comparing it with the second parameter. Lines 2 and 3 of Listing 3 encrypt the values 1 and 2 under Paillier and ElGamal respectively. We stress that encrypting values during a program's execution does not offer confidentiality for these values since the values are already available in plain text form, and should hence be used only for non-sensitive values, e.g., constants `one` and `two`. These encrypted values can then be used in computations involving other encrypted sensitive information, e.g., `in1` and `in2`.

```
1 def isAnswer_(in1: EncInt, in2: EncInt): Crypto[Boolean] =
2   for {one <- encrypt(Paillier)(1)
3        two <- encrypt(ElGamal)(2)
4        x1 <- add(in1, one)
5        x2 <- multiply(x1, two)
6        x3 <- equal(x2, in2)
7   } yield x3
```

Listing 3: Using Scala's for-comprehension.

### 3.3 Local Program Execution

Programs in the DSL issue a result of type `Crypto[A]`, where `A` is the concrete return type, e.g., `Int`, `Double` etc. The reason is that so far we showed only *descriptions* of programs which are either a suspended computation represented by a functor or they end with a concrete result – the final value of the computation based on the definition of the free monad. This is represented by Scalaz's `Either` equivalent `\/`, where `Left` is `-\/` and `Right` is `\/-`.

Running program descriptions, requires an interpreter for `Crypto` programs. We start with the assumption of access to *both private and public* keys. Listing 4 shows an interpreter for the `Plus` and `Encrypt` case of `CryptoF` – other cases differ with regard to the concrete encryption scheme required for the operation. We rely on helper functions (1) `convertToPaillier` to convert between `EncInt` and `PaillierEnc`, and (2) `asymEncrypt` to encrypt a value with the given encryption scheme. `asymEncrypt` only requires *public keys* because it supports only asymmetric schemes.

With these functions, `Encrypt` can be interpreted by calling `asymEncrypt` and passing the result into the continuation `k` of the current suspension. Interpreting `Plus` is straightforward if both operands are encrypted under Paillier, using the "+" in `PaillierEnc` – exploiting the homomorphic property. For operands not encrypted with the correct scheme, the `convertToPaillier` helper function performs the conversion, requiring access to the private keys.

```
def convertToPaillier(keyRing: KeyRing)(
  enc: EncInt): PaillierEnc = ...
def asymEncrypt(scheme: Scheme, keyRing: PubKeys)(
  plain: BigInt): EncInt = ...

def interpret[A](keyRing: KeyRing)(program: Crypto[A]): A =
  program.resume match {
    case \/-(x) => x
    case -\/(Encrypt(s,plain,k)) =>
      interpret(keyRing)(k(asymEncrypt(s,keyRing.pub)(plain)))
    case -\/(Plus(l,r,k)) => (l,r) match {
      case (lhs@PaillierEnc(_),rhs@PaillierEnc(_)) =>
        interpret(keyRing)(k(lhs + rhs))
      case _ =>
        val lhs = convertToPaillier(keyRing)(l)
        val rhs = convertToPaillier(keyRing)(r)
        interpret(keyRing)(k(lhs + rhs))
    }
    case _ => sys.error("Unhandled case")
}
```

Listing 4: Interpreting encryption and addition.

## 3.4 Remote Program Execution

We show an interpretation suitable to run on an untrusted party. In the remote interpreter in Listing 5, `remote` returns a `Future[A]` for a program with result `A` – we use `Futures` to model network communication to the `CryptoService`. The interpretation of `Encrypt` is analogous to local interpretation: Encryption with asymmetric schemes uses the public keys. In contrast, re-encrypting to the required scheme is not possible without accessing the private keys and the conversion is delegated to the `srv: CryptoService` argument of `remote`. `CryptoService` handles network communication with a trusted service requesting re-encryption of values. Apart of the `CryptoService`, the interpretation of the `Plus` case in the remote interpreter is similar to the local case. The difference

```
def remote[A](pub: PubKeys, srv: CryptoService)
 (program: Crypto[A])(implicit ec: ExecutionContext): Future[A] =
   program.resume match {
     case \/-(x) => Future.successful(x)
     case -\/(Encrypt(s,plain,k)) =>
       remote(pub,srv)(k(asymEncrypt(s,pub)(plain)))
     case -\/(Plus(l,r,k)) => for {
            lhs <- srv.toPaillier(l)
            rhs <- srv.toPaillier(r)
            res <- remote(pub,srv)(k(lhs+ rhs))
          } yield res
     case _ => sys.error("Unhandled case.")
  }
```

Listing 5: A remote interpreter.

is in for-comprehensions, because each call to `toPaillier` results in a `Future[PaillierEnc]`. The `CryptoService`'s task models a trusted access point for the program running on the untrusted party to perform conversions between schemes:

```
trait CryptoService {
  def toPaillier(enc: EncInt): Future[PaillierEnc] = ...
  def toElGamal(enc: EncInt): Future[ElGamalEnc] = ...
  // ...
}
```

Its interface defines methods to convert from `EncInt` to each of the concrete cases with known encryption scheme, e.g., `PaillierEnc` for Paillier and `ElGamalEnc` for ElGamal. If the interpreter requests a conversion between two encryption schemes, a concrete implementation of `CryptoService` has to talk to the trusted server that re-encrypts values.

## 3.5 Using Monadic Combinators

Free monads allow us to use monadic combinators. We have already used Scala's for-comprehensions to get syntactic sugar in Listing 5. For-comprehensions translate to functions such as `flatMap`, `map` and `filter`. In addition, functions derived from them are also available in the DSL. For example, Scalaz has a `Foldable` type class that provides, besides others, monadic folds, for instance. Scala's `List` type is an instance of `Foldable`, which allows use of *monadic* folds to sum a list of encrypted numbers (Listing 6).

```
def randomEnc: EncInt = ...
def sumList(zero: EncInt)(xs: List[EncInt]): Crypto[EncInt] =
  xs.foldLeftM(zero)(add(_,_))
lazy val result = sumList(zero)(List.fill(100)(randomEnc))
```

Listing 6: The Sum program using a monadic fold.

Similar to monadic folding, the DSL can be used with other methods such as `findM` to find elements based on a monadic predicate, `filterM` (the monadic version of the `filter` function), `takeWhileM`, `partitionM` and others, as defined in Scalaz for instances of corresponding type classes.

## 3.6 Limitations of the Monadic Approach

There are two main limitations of the monadic approach. First, the execution of a program such as Listing 6 is *sequential*, i.e., starting from the head of the list, *if the underlying scheme is not Paillier, convert the head, add it to the accumulator of the fold, repeat for the tail.* The program has poor performance, especially for a computation that is embarrassingly parallel. The issue is inherent to the monadic approach: Use of the bind operator depends on previous monadic values, forcing the computation to progress *sequentially*. Second, programs in monadic style limit static inspection. With monadic binds the analysis is stuck after the first argument, because the second argument is a continuation expecting an argument before it can continue – the analysis cannot progress into the function value.

## 4. DSL Based on Free Applicative Functors

In this section, we present a different version of the DSL based on free applicative functors [7] that supports static analysis and implicitly parallel program execution.

### 4.1 Moving to Free Applicative Functors

Applicative functors support lifting *pure* functions such that they accept *effectful* arguments. Contrarily to monads, they forbid dependencies on previous effectful computations, but provide the ability to evaluate effects in parallel. We follow the approach of Capriotti and Kaposi [7], defining a DSL that can be analyzed statically based on a free applicative functor. We use Scalaz's free applicative functor `FreeAp` as well as our previously defined functor `CryptoF`. Instead of providing smart constructors that lift into the *free monad* arising from `CryptoF`, we change those to lift into the *free applicative functor*. An example of such change is shown in Listing 7 where `Free` is replaced with `FreeAp` and `Free.liftF` with `FreeAp.lift`. This approach is applied to each smart constructor. We change the `Crypto` type alias to use the free applicative `FreeAp` instead of the free monad `Free` and rename the `Crypto` type alias to `CryptoM`.

```
type Crypto[K] = Free[CryptoF,K] // free monad
def add(lhs: EncInt, rhs: EncInt): Crypto[EncInt] =
  Free.liftF(Plus(lhs,rhs,identity))

type Crypto[K] = FreeAp[CryptoF,K] // free applicative functor
def add(lhs: EncInt, rhs: EncInt): Crypto[EncInt] =
  FreeAp.lift(Plus(lhs,rhs,identity))
```

Listing 7: Lifting free monads and free applicative functors.

### 4.2 Interpreting Free Applicative Functors

Switching from free monads to free applicative functors, requires new interpretation functions. As changes are independent of the interpretation being local or remote, we only discuss the local case, showing the `Encrypt` and `Plus` operations (Listing 8). We pass to `foldMap` a natural transformation [5] from `CryptoF` to the result `A`, written infix as `CryptoF  > Id` and match on the cases of the `CryptoF` type. This version is identical to Section 3.3 except that, here, `foldMap` handles recursion and the usage of `Id`'s `Applicative` instance is explicit. Using, for example, `Future` as target `Applicative` operands in the `Plus` case is converted in parallel.

### 4.3 Using the Applicative DSL

Since we rely on a free *applicative* functor over `CryptoF`, Scala's for-comprehensions cannot be used because a monadic

```
def interpret[A](kr: KeyRing)(p: Crypto[A]): A = {
  p.foldMap(new (CryptoF ~> Id) {
    def apply[B](fa: CryptoF[B]): B = fa match {
      case Encrypt(s,plain,k) =>
        k(asymEncrypt(s,kr.pub)(plain)).point(Id.id)
      case Plus(l,r,k) => (l,r) match {
        case (l@PaillierEnc(_),r@PaillierEnc(_)) => k(l+r)
        case _ =>
          val l = convertToPaillier(kr)(l)
          val r = convertToPaillier(kr)(r)
          k(l+r).point(Id.id)
      }
      case _ => sys.error("Unhandled case.")
}})}
```

Listing 8: Interpretation with free applicative functors.

bind `flatMap` cannot be defined. Without the ability to depend on effectful previous values, programs like `isAnswer` (Listing 3), cannot be expressed. We remedy this limitation later and investigate which programs we can express.

Scalaz's type class `Traversable` allows traversing and performing an action on each element. Using `traverse` and extending the DSL with explicit conversions:

```
case class ToPaillier[K](v: EncInt, k: PaillierEnc => K)
  extends CryptoF[K]
def toPaillier(v: EncInt): Crypto[PaillierEnc] =
  FreeAp.lift(ToPaillier(v,identity))
// same for ElGamal, Aes and Ope
```

We can write `sumList` as shown in Listing 9. We traverse the list with the effectful conversion and use `map` to sum the converted list of numbers inside `Crypto` of type `List[PaillierEnc]`, exploiting the homomorphic property of the Paillier scheme.

```
def sumList(zero: PaillierEnc)(
  xs: List[EncInt]): Crypto[PaillierEnc] =
    xs.traverse(toPaillier).map(_.foldLeft(zero)(_+_))
```

Listing 9: Summing the elements of a list with `traverse`.

### 4.4 Static Analysis

Free applicative functors enable static analysis because an applicative program can always be modeled as a *pure* function that receives arguments, represented by *independent* effectful computations, thereby allowing inspection of each argument without running the program. `FreeAp` provides two functions to express the analysis, `foldMap`, which gives the program the semantics of a provided applicative functor and `analyze`, which performs a monoidal analysis. As an example, we show how to count the required number of conversions in a program *without interpreting* it (Listing 10) but using the additive monoid for integers. The following example demonstrates the usage:

```
lazy val (x,y,z): (EncInt,EncInt,EncInt) = ...
val conversions: Int = requiredConversions {
  (toPaillier(x) |@| toPaillier(y) |@| toPaillier(z)){_+_+_}
}
```

The program converts the encrypted integers to the Paillier scheme and exploits the homomorphic property to perform the sum. In addition to analysis, this approach allows program transformation as well as partial evaluation. For example, `preconvert` in Listing 11 performs all required conversions before running the program: Given a key ring it

```
def requiredConversions[A](p: Crypto[A]): Int =
  p.analyze(new (CryptoF ~>   [   => Int]) {
    def apply[B](fa: CryptoF[B]): Int = fa match {
      case ToPaillier(PaillierEnc(_),_) => 0
      case ToPaillier(_,_) => 1
      // for each encryption scheme ...
      case Plus(PaillierEnc(_),PaillierEnc(_),_) => 0
      case Plus(_,PaillierEnc(_),_) => 1
      case Plus(PaillierEnc(_),_,_) => 1
      case Plus(_,_,_) => 2
      // for every operation in the DSL ...
        }}
)
```

Listing 10: Counting conversions via static analysis.

matches all conversion instructions and performs the conversions – possibly in parallel.

```
def convert(keyRing: KeyRing)(s:EncScheme,v:EncInt): EncInt = ...
def preconvert[A](keyRing: KeyRing): Crypto[A] => Crypto[A] =
  _.foldMap(new (CryptoF ~> Crypto) {
    def apply[B](fa: CryptoF[B]): Crypto[B] = fa match {
      case ToPaillier(v,k) =>
        val r@PaillierEnc(_) = convert(keyRing)(Additive, v)
        FreeAp.point(k(r))
        // same for ElGamal, Aes and Ope
      case x => FreeAp.lift(x)
}})
```

Listing 11: Static analysis for scheme conversions.

In summary, the DSL based on free applicative functors is restricted to computations that do not depend on previous effectful values, but resulting programs support implicit parallelism, as well as static analysis.

### 4.5 Recovering Monadic Expressivity

We combine the DSLs developed so far such that programs can depend on previous effects, while being amenable of static analysis for the parts that only use the applicative interface. We introduce a new instruction `Embed`, which embeds programs based on the free applicative functor into a new DSL based on free monads. We use the technique from [24] to combine our previous `CryptoF` and the new `Embed` functor into Scalaz's `Coproduct`. We update the type alias `CryptoM` to it's final definition:

```
type Crypto[A] = FreeAp[CryptoF, A]
type CryptoM[A] = Free[Coproduct[CryptoF,Embed,?], A]
```

In the applicative version of `sumList` (Listing 9) the argument has to be of type `PaillierEnc`, requiring an input with a known encryption scheme because conversion is an effect and a dependency on previous effects is not allowed in the applicative version. We remedy this issue with the embedding, allowing arbitrary encrypted numbers:

```
def sumList2(zero: EncInt)(xs: List[EncInt]): CryptoM[EncInt] =
  for { z <- toPaillierM(zero)
        result <- embed(sumList(z)(xs))
  } yield result
```

The program performs an explicit conversion into the Paillier scheme and embeds the previously defined applicative `sumList` program. Arguments to `embed` have to be written using the applicative interface, enforced by Scala's type system. In summary, the interpretation of the `sumList2` program (1) converts the first argument to the Paillier scheme (2) runs the embedded applicative program converting all numbers, *in parallel* and finally performs the additions. The result of `sumList2` is of type `CryptoM[PaillierEnc]`, a monadic program, which can no longer be embedded using `Embed` and cannot be statically analyzed. We achieve a strict separation of the monadic and applicative DSL with the ability to use an applicative program inside a monadic program – but not vice versa – guaranteeing that programs of type `Crypto[A]` can use implicit parallelism during interpretation and can be statically analyzed.

## 5. SecureScala

Previous sections show simplified versions of SecureScala. In this section, we combine the monadic and applicative DSL, and extend the operation set.

```
sealed trait CryptoF[+K]
sealed trait CryptoRatio[+K] extends CryptoF[K]
case class CeilRatio[K](r: EncRatio, k: EncInt => K)
  extends CryptoRatio[K]
... // also: floor
sealed trait CryptoString[+K] extends CryptoF[K]
case class EqualsStr[K](lhs: EncStr, rhs: EncStr, k: Boolean
              => K )
... // also: compare, split, concat, conversions
sealed trait CryptoNumber[+K] extends CryptoF[K]
case class Mult[K](lhs: EncInt, rhs: EncInt, k: ElGamalEnc => K)
... // also: +,-,/,==,compare,even,odd,encryption,conversions
abstract case class Embed[K]() {
 type I
 val v: Crypto[I]
 val k: CryptoM[I] => CryptoM[K]
}
```

Listing 12: The complete DSL.

### 5.1 Operations

We start with the full version of `CryptoF` as shown in Listing 12. The `CryptoF` functor can be divided into three parts. (1) the `CryptoRatio` trait, defining rounding operations on ratios of encrypted integers (2) `CryptoString`, defining operations for Strings and `CryptoNumber` supporting the already introduced operations on encrypted integers as well as additional operations not used before like subtraction, division, etc., and (3) the `Embed` class to embed applicative programs into the monadic DSL. As before, we do not expose the constructors directly, instead we define corresponding smart constructors for both the monadic and applicative DSL, as well as infix operators where applicable. As an example, for `Mult`, we define the following:

```
def multiply(lhs: EncInt, rhs: EncInt): Crypto[EncInt]
def multiplyM(lhs: EncInt, rhs: EncInt): CryptoM[EncInt]
implicit class InfixOps(self: EncInt) {
  def *(that: EncInt) = multiply(self,that)
}
```

We discuss operations that are not supported by any encryption scheme, like `isEven`, later in this section.

### 5.2 Interpretation

Interpreters extend the `CryptoInterpreter` trait which defines two different interpretation functions `interpret` and `interpretA` which are responsible for the interpretation of monadic and applicative programs:

```
trait CryptoInterpreter[F[_]] {
 def interpret[A](p: CryptoM[A]): F[A]
 def interpretA[A](p: Crypto[A]): F[A]
}
```

It is possible to use the implementation of `interpret` just as well for `interpretA`, but the distinction allows to exploit the fact that we can interpret applicative programs in a more flexible way, as shown in previous sections. `CryptoInterpreter` is parameterized over a higher kinded type constructor `F`, which appears in the result type of the interpretation functions. This higher kinded type parameter allows different result types for different interpretation styles, e.g., `Id` for pure evaluation or `Future` for parallelism.

```
trait CryptoService {
  def publicKeys: Future[PubKeys]
  def toPaillier(in: EncInt): Future[PaillierEnc]
  // ... explicit conversions for each scheme
  def convert(s: Scheme)(in: EncInt): Future[EncInt]
  def encrypt(s: Scheme)(in: Int): Future[EncInt]
  def batchConvert(xs: List[(Scheme,EncInt)]):
                        Future[List[EncInt]]
  def batchEncrypt(xs: List[(Scheme,Int)]):
                        Future[List[EncInt]]
  def decryptAndPrint(v: EncInt): Unit
  def println[A](a: A): Unit
}
```

Listing 13: The full `CryptoService` interface.

## 5.3 Conversions and Unsupported Operations

`CryptoService` is responsible for handling conversion requests and can be asked to perform computations that are not supported by any scheme. In Listing 13 we show the interface that provides the necessary functions to convert between encryption schemes. We provide the functions `batchConvert` and `batchEncrypt` to be able to divide a large number of conversion requests efficiently into batches of requests. The two functions `decryptAndPrint` and `println` are used to communicate between the remote and local hosts for testing.

For some operations in the DSL (Listing 12), e.g., `SplitStr`, we fall back on respective methods from the `CryptoService`, which should be run by a trusted party, because the operation is performed locally on unencrypted data. While undesirable from a design point of view, we gain a lot of expressivity, allowing many programs to be written which are impossible in other systems [13, 21].

## 5.4 Importance of Separation

Our system fosters a clear distinction between the monadic and the applicative DSL. Since every monad is also an applicative functor, we could combine the two and provide only the monadic interface. As an example, we consider Haxl, Facebook's Haskell library for remote data access [17]. Haxl's `Fetch` datatype is an instance of `Monad` and therefore it is also an instance of `Applicative`. To exploit implicit parallelism, the `Applicative` instance deviates from one of Haskell's rules for `Control.Applicative`: *If f is also a Monad, it should satisfy `pure = return` and `(<*>) = ap`.* The rule forbids concurrency in the `Applicative` instance, because `ap` uses `(>>=)`, which is inherently sequential. By breaking this rule and having only one interface that allows both monadic and applicative styles, programmers have to be well behaved and make use of the applicative style as much as possible to benefit from parallelism and optimization.

Our implementation explicitly distinguishes between applicative and monadic programs. A program of type `Crypto[A]` is *guaranteed* to be statically analyzable and exhibits concurrent behavior during runtime. `CryptoM[A]` instead does not give those guarantees, but provides more expressive power in the sense that it is possible to depend on previous effectful results.

```
1  lazy val one: EncInt = ...
2  lazy val two: EncInt = ...
3  def genFib(x1: EncInt, x2: EncInt)(n: EncInt):
4    CryptoM[EncInt] = for  {
5    r <- if (n <= one) {
6      x1.lifted
7    } else for {
8      (n1,n2) <- (n-one).tuple(n-two)
9      (f1,f2) <- genFib(x1,x2)(n1).tuple(genFib(x1,x2)(n2))
10     s <- f1 + f2
11   } yield s
12 } yield r
```

Listing 14: Secure Generalized Fibonacci series.

## 5.5 An Example SecureScala Program

We present a larger program which calculates the n-th number in a Generalized Fibonacci series shown in Listing 14. The necessary call to `embed` is added automatically by an implicit conversion. In Line 14 the program performs the two operations, "`n-one`" and "`n-two`" in parallel. In contrast, the use of `tuple` in Line 14 does *not* allow parallel execution, because `genFib`'s result type is `CryptoM`, the monadic DSL inhibiting parallelism. This demonstrates the need for developers to write programs that minimize the monadic parts to get better performance. The value of `n` can be potentially inferred by observing the number of invocations of the `genFib` function but, importantly, the final output remains secure.

## 6. Evaluation

In this section, we evaluate our contributions along two dimensions: we compare the performance of different interpretation styles based on micro-benchmarks and present several case studies. The goals of the evaluation are:

- Evaluate the performance of SecureScala and of the combination of the monadic and applicative styles.
- Provide evidence for our claim that PHE support should be integrated into general-purpose languages, showing the effects of the combination of styles in more realistic scenarios than the micro-benchmarks.

Experiments are executed on a Dell Latitude E6540, Intel(R) Core(TM) i7-4800MQ CPU quad core, with frequency scaling disabled, using 16GB of RAM, which simulates both the client and the untrusted cloud. We use Scala 2.11.7, OpenJDK JRE (IcedTea 2.5.6), OpenJDK 64-Bit

| Style | Description |
|---|---|
| local | local interpretation |
| remote | use `CryptoService` |
| remoteOpt | use `CryptoService` and parallelism |
| remoteOptAnalysis | like *remoteOpt*, batch requests into chunks of up to 15, if total number exceeds (15) |

Table 2: Comparison of interpretation styles.

Server VM (build 24.79-b02) with 2 GB of min/max heap and ScalaMeter [3] v0.6 with `PerformanceTest.OfflineReport` configuration, reporting the average over 10 runs.

### 6.1 Micro Benchmarks

We compare performance of a program that sums an increasing number of encrypted integers using the monadic style Listing 6 versus applicative style Listing 9 (Figure 2a). Numbers are encrypted under a randomly chosen encryption scheme with equal probability among schemes. For both the applicative and monadic case, we consider the interpretation styles in Table 2.

The results show that for the "local" and "remote" styles there is virtually no performance difference between the applicative and the monadic version. For the "remoteOpt" case there is a difference of about a factor of two due to the applicative version using implicit parallelism to perform the encryption scheme conversions for all numbers in parallel.

Function `remoteOptAnalysis` performs similarly despite using static analysis in addition to exploiting implicit parallelism. Since in this setting there is no network delay involved in encryption scheme conversions, the interpreter does not gain any benefit from grouping conversion requests. Note that some variance is due to the fact that the encryption scheme for each randomly generated number is also randomly selected and each has significantly different costs for decyption/encryption, hence times necessarily differ between benchmark runs. In the benchnmark, we introduce a conversion request delay of *75ms* (taken from literature on estimating run times for secure computation [22]), which simulates network latency when performing conversion requests over the network, i.e., talking to a remote `CryptoService` (Figure 2b). Results show that the two optimizing interpreters "remoteOpt" and "remoteOptAnalysis" perform significantly better than the naive "remote" interpreter. With the delay for conversion requests, the interpreter using static analysis outperforms the interpreter that exploits only implicit parallelism for input size $>= 15$ which is the threshold. As input size grows, static analysis further increases performance, because the "remoteOpt" interpreter performs *all* conversion requests in parallel – with a per-request network overhead – while "remoteOptAnalysis" batches requests into groups of up to 15.

### 6.2 Case Studies

***Word count.*** The first case study is a program to count the words in a document encrypted under the OPE scheme. The output associates each encrypted word with its (plain) occurrence. The main functionality can be expressed as follows:

```
def wordCountText(e: EncStr): CryptoM[List[(OpeStr,Int)]] =
  e.split("""\s+""") >>= wordCount_

def wordCount_(es: IList[EncStr]): Crypto[List[(OpeStr,Int)]] =
  es.traverse(toOpeStr).map(
    _.groupBy(x => x).map(_.length).toList)
```

The program (1) reads the file with encrypted text, (2) splits the text into words, (3) groups by word, (4) associates each word with its occurrences, (5) writes the result into a file or prints it. Figure 2c shows the performance of `wordCountText` as text size increases. We include measurements for two interpretation styles, (1) local without any optimization and (2) remote using implicit parallelism. For reference, we show (3) an implementation in plain Scala. Results show that the remote interpretation style with implicit parallelism performs more than 2x faster than the local interpreter. The plain Scala version without any encryption is faster and is included to show the impact of the encryption on performance.

***Complex event processing.*** The CEP case study combines the CEP engine Esper [1] with SecureScala to perform queries over events holding encrypted fields. Esper supports calling Scala methods from within queries – enabling the use of SecureScala for encrypted data. The scenario of the case study considers a track on a road where the speed of cars is measured at three checkpoints that check if the speed is higher than a threshold. Events consist of one plain and two encrypted attributes:

```
sealed trait LicensePlateEventEnc {
  @BeanProperty def car: EncStr
  @BeanProperty def time: Long
  @BeanProperty def speed: EncInt
}
```

an encrypted string for the license plate, an encrypted integer as the car speed and a plain integer as a timestamp. Case classes model events:

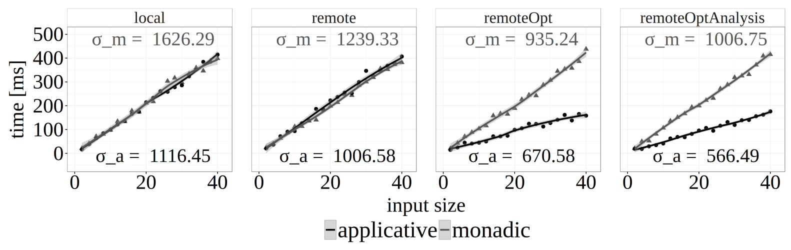
```
case class CarStartEventEnc(...) extends LicensePlateEventEnc
case class CheckPointEventEnc(...) extends LicensePlateEventEnc
case class CarEndEventEnc(...) extends LicensePlateEventEnc
```

We show an Esper query that for each license plate, tracks the start time, the three checkpoint times and the end time. At each checkpoint, the car speed is compared to the threshold.

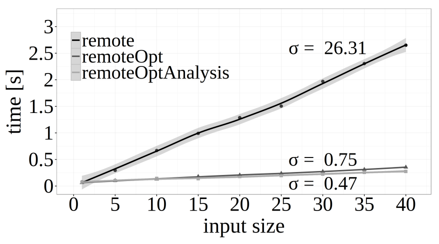
```
SELECT car AS license, number, speed
FROM CheckPointEventEnc
WHERE Interp.isTooFast(speed)
```

In the `WHERE Interp.isTooFast(speed)` clause, the `Interp` object defines the `isTooFast` method which runs the interpreter to check the result of the DSL program, indicating if the speed is higher than the allowed `speedLimit`.

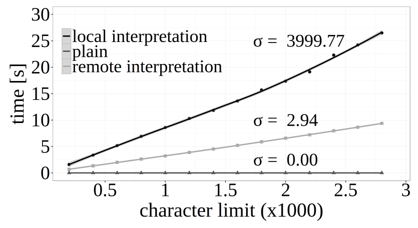
```
object Interp {
  val keyRing: KeyRing = ...
  val interpret: LocalInterpreter = ...
  val speedLimit: EncInt = ...
  def isTooFast(s: EncInt): Boolean = interpret(s > speedLimit)
}
```



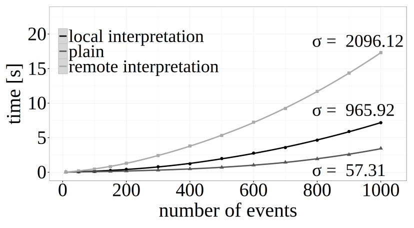
(a) Microbenchmark for different interpretation styles.



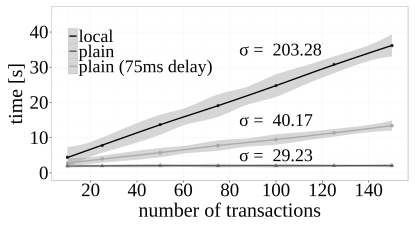
(b) Microbenchmark: 75ms conversion delay.



(c) Word Count case study.



(d) Complex event processing case study.



(e) RxBank case study.

Figure 2: Performace results for microbenchmarks and case studies.

Figure 2d compares the performance of a version using plain events and one with encrypted speed and encrypted license plates. The encrypted version runs with the *local* interpreter, without implicit parallelism, as well as the `remote` interpreter, talking to a remote service on the same machine. High variance is caused by calling the interpreter once per event to evaluate the query. The case study shows that when SecureScala is applied in the context of a more complex computation – like event correlation – the overhead is significant but not prohibitive. Thanks to embedding, SecureScala can easily interoperate with systems interfacing with Scala.

***Reactive programming.*** The RxBank case study combines RxScala [2] with SecureScala to implement a GUI visualizing transactions among bank accounts. The GUI exists in two versions, one displays the balance of bank accounts and the transferred amount in plain text, in the second one the balance and the transactions are encrypted – the only visible information is the color of the account from dark red (negative) over orange (neutral) to light green (positive). For example, to calculate the color for the account based on a mapping from balances to colors, we filter the ascending list of balance thresholds and set the color in the GUI:

```
def redrawFor(account: String, amt: EncInt): Unit = {
  val cs = interp { colorMap.filterM { amt > _._1 } }
  if (cs.nonEmpty) fieldFor(account).background = cs.last._2
}
```

We evaluate the performance of RxBank measuring the time required to process an increasing number of transactions, which update the GUI after each account change. We measure three versions. *plain*: Plain Scala, process as fast as possible, *delay*: Plain Scala, 75ms delay per transaction, *encrypted*: Encrypted version with a local interpreter, no limit on transaction processing. The plain version *without any delay* is about 17x faster than the encrypted version. While this performance difference is large, it is important to consider that transactions are processed as fast as possible. Throttling the rate of transactions using a delay of 75ms (Figure 2e) shows that the encrypted version is able to process transactions with a rate of 4.15 transactions/s – fast enough to display the changes in the GUI. Additionally, the case study shows that programs in SecureScala can be seamlessly integrated with RxScala and Scala Swing.

## 7. Related Work

***Homomorphic encryption.*** Several systems have been proposed that make use of homomorphic encryption to define confidentiality preserving computations. CryptDB [21] can automatically transform SQL queries to queries that use cryptographic user defined functions to operate over encrypted data but it cannot execute queries that require conversions between schemes. Monomi [27] improves on the design of CryptDB by introducing a split client/server execution. Queries are split into sub-queries some of which are executed on the untrusted server over encrypted data and the rest are executed on the trusted client after decryption of intermediate results. Both CryptDB and Monomi focus on SQL queries and require a set-up phase to identify what encryption schemes are required. MrCrypt [26] uses static analysis on Java programs to find suitable PHE schemes for the operations applied on each input field. After selection of the scheme, a source-to-source transformation generates a program working on encrypted data. Unlike our approach, MrCrypt cannot handle programs with input that requires multiple operations not supported by a scheme. JCrypt[8] pushes further on static analysis. Type-based information flow analysis isolates program parts to be encrypted, inter-procedural data-flow analysis deduces the appropriate scheme. SPR [13, 23] has been proposed as a Pig Latin [19] runtime. SPR can utilize a trusted client to perform subcomputations or re-encrypt between schemes, similarly to our approach, but limited to Pig Latin scripts. Mitchell et al. [18] propose a core language – implemented as an EDSL in Haskell – to write secure applications without cryptographic knowledge. The language also offers protection against control flow observations of the server that is running the program operating on encrypted data. This approach is flexible with regard to the underlying cryptographic system, but at the cost of interoperability – it does not support interfacing with the rest of the language.

***Functional programming.*** Swierstra et al. [25] demonstrate the use of Free Monads to embed a DSL using monadic combinators to separate program construction from interpretation. Contrarily to monads in general [14–16], free monads compose, because functors compose. This technique can be used to achieve modularity and to compose languages defined by free monads over individual functors into a free monad combining multiple languages into one [24]. Haxl [17] unifies the monadic and applicative DSL into one. This approach violates a rule of the Applicative type class – which forbids the exploitation of concurrency in the `Applicative` instance. We consciously do not follow the same approach as discussed in Section 5. Free Applicative Functors – on which the analysis and transformation abilities of SecureScala are based – have first been studied by Capriotti et al [7], then implemented in open source libraries as Scalaz [4].

## 8. Conclusion

We define an embedded DSL in Scala, allowing developers to write programs that work on encrypted data without requiring cryptographic knowledge. Our approach combines the advantage of free monads – expressivity – with the advantage of free applicative functors – static analysis and parallelism. The evaluation shows the advantage of implicit parallelism as well as the performance improvements from the possibility to perform static analysis and program transformation of programs written in the applicative DSL.

## 9. Acknowledgments

This work has been supported by European Research Council grants #FP7-321217 and #FP7-617805, the German Science Foundation grants #SFB-1119 and #SFB-1053, the Northrop Grumman Cybersecurity Research Consortium, Amazon EC2, Cisco Systems grant #585544, DARPA grant #N11AP20014, NSF grants #1117065, #1421910, and #1618923.

## References

[1] esper CEP engine. `http://www.espertech.com/esper/`.

[2] RxScala. `http://reactivex.io/rxscala/`.

[3] Scalameter. `https://scalameter.github.io`.

[4] Scalaz. `https://github.com/scalaz/scalaz`.

[5] Shapeless. `github.com/milessabin/shapeless/`.

[6] A. Boldyreva, N. Chenette, Y. Lee, and A. O'Neill. Order-preserving symmetric encryption. In *EUROCRYPT*, 2009.

[7] P. Capriotti and A. Kaposi. Free applicative functors. *Electronic Proceedings in Theoretical Computer Science*, 2014.

[8] Y. Dong, A. Milanova, and J. Dolby. Jcrypt: Towards computation over encrypted data. PPPJ '16. ACM, 2016.

[9] T. ElGamal. A Public-Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. *IEEE Transactions on Information Theory*, 1985.

[10] C. Gentry. Fully homomorphic encryption using ideal lattices. In *STOC'09*. ACM, 2009.

[11] C. Gentry, S. Halevi, and N. P. Smart. Homomorphic evaluation of the AES circuit. In *CRYPTO 2012*. Springer.

[12] S. Halevi and P. Rogaway. *A Tweakable Enciphering Mode*. Springer Berlin Heidelberg, 2003.

[13] J. James Stephen, S. Savvides, R. Seidel, and P. Eugster. Program analysis for secure big data processing. ASE '14. ACM, 2014.

[14] M. P. Jones and L. Duponcheel. Composing Monads. Technical report, 1993.

[15] D. J. King and P. Wadler. Combining monads. In *Functional Programming, Glasgow 1992*. Springer, 1993.

[16] O. Kiselyov, A. Sabry, and C. Swords. Extensible effects. In *Haskell '13*. ACM, 2013.

[17] S. Marlow, L. Brandy, J. Coens, and J. Purdy. There is no fork. *ACM SIGPLAN Notices*, 2014.

[18] J. C. Mitchell, R. Sharma, D. Stefan, and J. Zimmerman. Information-flow control for programming on encrypted data. CSF '12. IEEE Computer Society, 2012.

[19] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin. In *SIGMOD'08*. ACM Press, 2008.

[20] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. EUROCRYPT'99. Springer, 1999.

[21] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. SOSP '11. ACM, 2011.

[22] A. Schroepfer and F. Kerschbaum. Forecasting run-times of secure two-party computation. QEST '11. IEEE, 2011.

[23] J. J. Stephen, S. Savvides, R. Seidel, and P. Eugster. Practical confidentiality preserving big data analysis. In *HotCloud '14*. USENIX, 2014.

[24] W. Swierstra. Data types a la carte. *Journal of Functional Programming*, 2008.

[25] W. Swierstra and T. Altenkirch. Beauty in the beast. In *ACM SIGPLAN - Haskell '07*. ACM, 2007.

[26] S. D. Tetali, M. Lesani, R. Majumdar, and T. Millstein. Mrcrypt: Static analysis for secure cloud computations. OOPSLA '13. ACM, 2013.

[27] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich. Processing analytical queries over encrypted data. *VLDB Endowment*, 2013.