# A Domain-Specific Language for Computing on Encrypted Data

**Alex Bain, John Mitchell, Rahul Sharma, Deian Stefan and Joe Zimmerman**

**Stanford University, Stanford, CA**

**Abstract**

In cloud computing, a client may request computation on confidential data that is sent to untrusted servers. While homomorphic encryption and secure multiparty computation provide building blocks for secure computation, software must be properly structured to preserve confidentiality. Using a general definition of *secure execution platform*, we propose a single Haskell-based domain-specific language for cryptographic cloud computing and prove correctness and confidentiality for two representative and distinctly different implementations of the same programming language. The secret sharing execution platform provides information-theoretic security against colluding servers. The homomorphic encryption execution platform requires only one server, but has limited efficiency, and provides secrecy against a computationally-bounded adversary. Experiments with our implementation suggest promising computational feasibility, as cryptography improves, and show how code can be developed uniformly for a variety of secure cloud platforms, without explicitly programming separate clients and servers.

**1998 ACM Subject Classification** D.3.3 Language Constructs and Features

**Keywords and phrases** Domain-Specific Language, Secret Sharing, Homomorphic Encryption

**Digital Object Identifier** 10.4230/LIPIcs.FSTTCS.2011.6

## 1 Introduction

Recent advances in secure multiparty computation and homomorphic encryption promise a wide range of new applications. In particular, it is cryptographically possible to protect data in the cloud from the servers manipulating it, subject to varying threat models. However, the practical widespread use of these cryptographic techniques requires a suitable software development, testing, and deployment infrastructure.

In this paper, we present the design, foundational analysis, implementation, and performance benchmarks for an initial embedded domain-specific language (EDSL) that allows programmers to develop code that can be run on different secure execution platforms with different security guarantees. Figure 1 shows how our separation of programming environment from cryptographically secure execution platforms can be used to delay deployment decisions or run the same code on different platforms.

While homomorphic encryption and secure multiparty computation are based on different cryptographic insights and constructions, there is a surprising structural similarity among them that we express in our definition of *secure execution platform*. This definition allows us to develop a single set of additional definitions, theorems, and proofs that are applicable to many platforms. In particular, we prove functional correctness and confidentiality, for an honest-but-curious adversary, across relevant platforms. We then show that fully homomorphic encryption satisfies our definition, as does a specific secret-sharing scheme, subject to assumptions on the number of potentially colluding servers. Moreover,

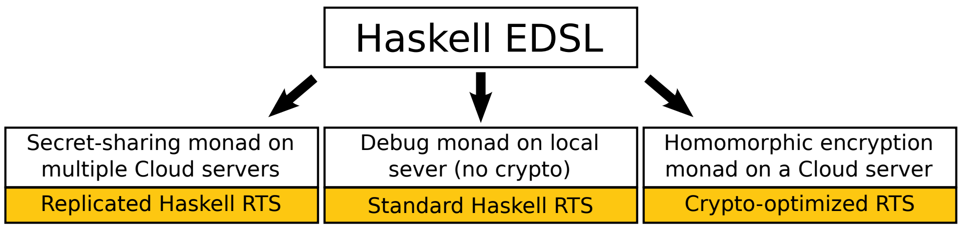
Haskell EDSL
Secret-sharing monad on multiple Cloud servers
Replicated Haskell RTS
Debug monad on local sever (no crypto)
Standard Haskell RTS
Homomorphic encryption monad on a Cloud server
Crypto-optimized RTS

**Figure 1** Multiple deployment options using different runtime systems (RTS)

our definition of secure execution platform is parameterized over the set of primitive operations on secret values, so that our language and our theoretical guarantees are applicable to partially homomorphic schemes, when they support the operations actually used in the program code. Our correctness theorems show equivalence with a reference implementation and therefore imply output equivalence for alternative secure execution platforms.

Our embedded domain-specific language is implemented as a Haskell library, rather than as a completely new language, so that developers can use existing and carefully engineered Haskell development tools, compilers, and run-time systems. Programmers also have the benefit of sophisticated type-checking and general programming features of Haskell because we rely only on the Haskell type discipline, not *ad hoc* code restrictions. Further, we use the Haskell type system to impose an information-flow discipline that is critical to preserving confidentiality against cloud servers that could otherwise leak information through control-flow analysis or other forms of program monitoring. Our Haskell implementation also provides flexible data structures, since our information-flow constraints make secrecy-preserving operations on such such structures possible.

```
generalizedMillionaires = do
  -- Read worth from all users:
  allWorth ← withUsers readWorth
  -- Find richest user and her worth
  (richest, worth) ← foldlM1 maxWorth allWorth
  -- Notify the users of the status:
  richestUser ← reveal richest
  withUsers_ (λu →
    if u == richestUser
      then uPutStrLn u "You are the richest!"
      else uPutStrLn u "Keep working!")

    where readWorth u =  do
            w ← uRead u
            return (hide u, w)
          maxWorth (u1,w1) (u2, w2) = do
            b ← (w1 .> w2)
            sif b sthen (u1,w1) selse (u2, w2)
```

**Figure 2** Generalized Millionaires' Problem

As a working example, we consider the Generalized Millionaires' Problem: given a number of millionaires, request their net worth, identify the richest millionaire, and, finally, notify each one of their status without revealing their net worth. Figure 2 shows an example implementation, that highlights several key aspects of our Haskell EDSL. First, our language provides various primitives such as `withUsers`, `uRead`, and `reveal` that are respectively used to apply a function (e.g., `readWorth`) to each connected user, read a secret input from the user, and reveal (decrypt) a secret value. Second, the DSL embedding allows the programmer to use existing Haskell features including higher-order functions, abstract data types, general recursion, etc. An example use of recursion in our example is `foldlM1` which, with `maxWorth`, is used to find the richest millionaire. Finally, compared to languages with similar goals (e.g., SMCL [24]), where a programmer is required to write separate client and server code, using our EDSL, a programmer need only write a single program; we eliminate the client/server code separation by providing a simple runtime system that directs all parties.

We describe a Haskell implementation of secure execution platforms based on both secret sharing and fully homomorphic encryption, both using SSL network communication between clients and any number of servers. Our implementation effort produced 2500 lines of Haskell and 650 lines of C/C++ code. We developed sample applications and measured performance on benchmarks, as reported in Section 4.3. Because our implementation is packaged in the form of Haskell libraries, other researchers could use our libraries to implement other programming paradigms over the same forms of cryptographic primitives. Conversely, we could target our language to other run-time systems such as SMCR [24], for programmers only interested in that execution paradigm.

The contributions of this work include:

- We leverage the similarity between secure multiparty computation and homomorphic encryption, as captured in a precise definition of *secure execution platform.*
- We design, implement, and test an embedded DSL that allows programmers to develop code that runs on any secure execution platform supporting the operations used in the code. We avoid *ad hoc* language restrictions by relying only on the Haskell type system for information flow properties and other constraints.
- We prove general functional correctness and security theorems, beyond previous work on related languages for secure multiparty computation (SMC [27], Fairplay [22], SIMAP [4, 24] and VIFF [8]).
- We develop and evaluate distributed secret sharing and homomorphic encryption execution platforms, using SSL network communication, implemented in Haskell.

Although we develop our results using the commonly used honest-but-curious adversary model, there are established methods for assuring integrity, using commitments and zero-knowledge techniques [16]. Moreover, since these add communication and computation overhead, we can also consider the possibility of using techniques from [23] in future work. These methods employ *computational* commitment and proofs of knowledge to provide computations on ciphertexts with verifiable integrity and smaller overhead. While we focus on data confidentiality, we can also protect confidential algorithms by considering code as input data to an interpreter (or "universal Turing machine").

## 2 Background

We propose a domain-specific programming language (DSL) embedded in Haskell, drawing on previous languages (e.g., Cryptol), use of monads for cryptographic computation, and other works on programmable secure multiparty computation (e.g., Fairplay [22], SIMAP [4, 24]). In this section, we introduce Haskell, and review secure multiparty computation and homomorphic encryption.

**Haskell and EDLs** Haskell is a widely used host language for EDSLs [17]. The language offers a strong, static type system that includes parametric and ad-hoc polymorphism (via type classes); first-class monads, with convenient syntactic sugar; and, the IO monad, strictly separating pure from impure computations. Haskell's type classes, lazy evaluation strategy (i.e., expressions are evaluated only when their values are needed), and support for monads makes it easy to define new data structures, syntactic extensions, and control structures—features commonly desired when embedding DSLs.

```
class Monad M where
  return :: α → Mα
  (≫=) :: Mα → (α → Mβ) → Mβ
```

**Figure 3** Monad operations

The main Haskell constructs used in embedding our DSL are monads and type classes. A monad $M$ provides a type constructor and related operations that obey several laws. Specifically, if $M$ is a monad and $\alpha$

is an arbitrary type, then $M\alpha$ is a type with operations `return`, and $\gg=$ (pronounced "bind"), whose types are shown in Figure 3. As shown in the figure, Haskell provides support for monads through the `Monad` *type class*. Type classes provide a method of associating a collection of operations with a type or type constructor. Programmers, then, declare *instances* of a given type class by naming the type or type constructor and providing implementations of all required operations. As explained later, type classes are also useful in 'overloading' arithmetic operations over secret and public data.

**Homomorphic encryption** A *homomorphic encryption scheme* $\langle \mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Eval} \rangle$ consists of a key generation algorithm, encryption and decryption algorithms, and an evaluation function that evaluates a function $f \in \mathcal{F}$ on one encrypted value to produce another. More specifically, if $c = \mathsf{Enc}(\mathrm{pk}, m)$ then $\mathsf{Eval}(\mathrm{pk}, c, f) = \mathsf{Enc}\big(\mathrm{pk},\ f(m)\big)$ for every $f \in \mathcal{F}$, where $\mathcal{F} \subseteq (\mathrm{Plaintext} \rightarrow \mathrm{Plaintext})$ is some set of functions on plaintexts. As stated here, $\mathcal{F}$ is a set of unary functions; however, we consider the more general case where each function has a specific arity and type. We say that the scheme is *homomorphic with respect to the set* $\mathcal{F}$ *of functions.*

While some homomorphic encryption schemes [6, 15, 20] are homomorphic with respect to a restricted class of functions, such as the set of quadratic multivariate polynomials or the set of shallow branching programs, recent research has produced an encryption scheme that is *fully homomorphic*, i.e., homomorphic with respect to all functions of polynomial complexity [11, 12, 28, 29]. Since this work has generated substantial interest, there is a rapidly growing set of fully homomorphic constructions. However, for efficiency reasons we remain interested in partially homomorphic schemes as well. Moreover, for any given program, it is only necessary to use a form of homomorphic encryption that is sufficient for the functions used by that program.

**Secure multiparty computation** Another approach to computing on ciphertexts makes use of generic 2-party or multi-party secure computation [30, 21, 2, 18, 7, 23, 19, 9, 1], in which the client, who has the plaintext $x$, communicates through some protocol with the server(s), who have the function $f$ to be computed on $x$. The standard conditions for secure multiparty computation guarantee that the client only learns $f(x)$ and the server learns nothing about $x$.

In Shamir secret sharing and the multi-party computation algorithm based on it (see [10]), a client $C$ shares a secret value $a_0$ from a finite field $F$ among $N$ other parties that we will refer to as *servers*. In an $(N, k)$ secret sharing scheme, $N$ servers can jointly perform computations on $a_0$ and other shared secrets, such that at least $k$ of the $N$ servers must cooperate to learn anything about $a_0$.

The client $C$ shares a secret value $a_0$ by choosing values $a_1, \ldots, a_{k-1}$ uniformly at random from $F$, and forms the polynomial $p(x) = \sum_{i=0}^{k-1} a_i x^i$. Then, $C$ computes and distributes the secret shares $s_1 = p(1), \ldots, s_N = p(N)$ to the servers $S_1, \ldots, S_N$, respectively.

Addition is easy for the servers to compute, since they can simply add their shares of two values pointwise: if the values $s_i$ form a sharing of $a_0$ via $p$, and $t_i$ form a sharing of $b_0$ via $q$, then $s_i + t_i$ form a sharing of $a_0 + b_0$ via $p + q$. Similarly, if the values $s_i$ form a sharing of $a_0$ via $p$, then, for a constant $c$, $c \cdot s_i$ form a sharing of $c \cdot a_0$ via $c \cdot p$. Multiplication of two secret values is more complicated, because multiplication of polynomials increases their degree. The solution involves computing and communicating a new sharing, which increases the cost because the servers must communicate.

## 3 Language design for Secure Cloud Computing (SCC)

For the purpose of analysis, we present a functional language whose definition is parameterized by a set of given operations over some given type of encryptable values. This language, $\lambda_{\vec{\mathrm{P}},\mathrm{S}}$, is a form of simply-typed lambda calculus, with labeled types as used in information flow languages (see, e.g., [26]). Our implementation, described in Section 4, embeds an extension of this language in Haskell, and provides specific operations over encryptable integer values. From the programmer's standpoint, different cryptographic backends that support the same operations provide the same programming experience. However, our analysis of security and correctness depends on the number of servers, the form of cryptography used, and the form and extent of communication between servers.

In order to provide a uniform analysis encompassing a range of cryptographic alternatives, we formulate both a standard reference semantics for $\lambda_{\vec{\mathrm{P}},\mathrm{S}}$ and a distributed semantics that allows an arbitrary number of servers to communicate with the client and with each other in order to complete a computation. Correctness of each distributed cryptographic semantics is proved by showing an equivalence with the reference semantics. Security properties are proved by analyzing the information available to each server at every point in the program execution.

Before presenting the definition of $\lambda_{\vec{\mathrm{P}},\mathrm{S}}$, we summarize the semantic structure used in our analysis. As shown below, our semantic structure is sufficient to prove correctness and security theorems for $\lambda_{\vec{\mathrm{P}},\mathrm{S}}$, and general enough to encompass secret sharing, homomorphic encryption, and other platforms.

**Reference semantics primitives** In the reference semantics, the private values used in computation are interpreted using a set $Y$ of base values, together with primitive operations $\mathrm{op}_1, \ldots, \mathrm{op}_r : Y \times Y \to Y$. For simplicity, we consider only binary operators over a single set of base values. The generalization to arbitrary typed operations over several types of base values is straightforward. We note that this parameterization allows our language (and its Haskell implementation) to easily encompass a variety of platforms, including cryptosystems that are only additively or multiplicatively (rather than fully) homomorphic.

**Randomness** Because cryptographic primitives used by each of $N$ servers in the distributed semantics may require randomness, we assume a set $\mathcal{R}$ of tuples of sequences, where each $R = (R_C, R_{S_1}, \ldots, R_{S_N}) \in \mathcal{R}$ provides $N+1$ infinite sequences of elements of some finite set $Z$ (such as $Z = \{0,1\}$). As the notation suggests, if there are $N+1$ parties, the sequence $R_P$ is assumed available to the party $P \in \{C, S_1, \ldots, S_N\}$. Since security relies on correct random sequences, we let $U_{\mathcal{R}}$ be a *uniform randomness source:* $U_{\mathcal{R}} = ((U_{\mathcal{R}})_C, (U_{\mathcal{R}})_{S_1}, \ldots, (U_{\mathcal{R}})_{S_N}) = (U_Z^\omega, U_Z^\omega, \ldots, U_Z^\omega)$, where $U_Z^\omega$ denotes an infinite sequence of uniform random variables over $Z$.

**Distributed computing infrastructure** We assume $N$ servers, $S_1, \ldots, S_N$, execute the secure computation on behalf of one client, $C$; the extension to multiple clients is straightforward. (In many natural cases, such as homomorphic encryption, $N = 1$). The $(N+1)$ parties will communicate by sending messages via secure party-to-party channels; we denote by $M$ the set of possible message values that may be sent. A *communication round* is a set $\{(P_1^{(i)}, P_2^{(i)}, m^{(i)})\}_{1 \le i \le r}$ of triples, each indicating a sending party, a receiving party, and a message $m \in M$. A *communication trace* is a sequence of communication rounds, possibly empty, and $\mathcal{T}$ is the set of communication traces.

If $A \subseteq \{S_1, \ldots, S_N\}$ is any subset of the servers, the *projection of trace $T$ onto $A$*, written $\Pi_A(T)$, is the portion of the trace visible to the servers in $A$, i.e., $\Pi_A(\varepsilon) = \varepsilon$ and:

$$\Pi_A(\{(S_1^{(i)}, S_2^{(i)}, m^{(i)})\} \| T) = \{(S_1^{(i)}, S_2^{(i)}, m^{(i)}) \mid \{S_1^{(i)}, S_2^{(i)}\} \cap A \neq \emptyset\} \| \Pi_A(T) \,.$$

**General form of cryptographic primitives** We work with a two-element security lattice, $\mathscr{S} = \{\mathrm{P}, \mathrm{S}\}$ (with $\mathrm{P} \sqsubseteq \mathrm{S}$), representing (respectively) "public" values, which are transmitted in the clear and may be revealed to any party; and "secret" values, which are encrypted or otherwise hidden, and must remain completely unknown to the adversary. We assume a set $\mathscr{E}_{\mathrm{S}}(Y)$, holding "secret equivalents" of base values in $Y$; for notational uniformity, we also define $\mathscr{E}_{\mathrm{P}}(Y) = Y$, signifying that the "public equivalent" of a value is just the value itself.

We also assume a cryptographic protocol operation $\mathrm{Init} : \mathcal{R} \to \mathcal{I} \times \mathcal{R}$ that establishes the initial parameters of the platform (e.g., it may generate a public/private key pair for use throughout the computation). Init is a randomized operation, taking a random source in $\mathcal{R}$ (and returning the modified source after potentially consuming some values). We define $\hat{\iota}$ to be the random variable over $\mathcal{R}$ that is derived from running Init on a uniformly random source ($\hat{\iota} = \pi_1(\mathrm{Init}(U_{\mathcal{R}}))$). Further, we assume a projection operator from the initial parameters onto any collection of servers $A \subset \{S_1, \ldots, S_n\}$, writing $\Pi_A(\iota)$ to mean, intuitively, the portion of the initial parameters $\iota$ that servers in $A$ should receive.

The other cryptographic operations used in the distributed semantics return secret or public values, but may also consume random values ($\mathcal{R}$), read from the initial parameters ($\mathcal{I}$), and/or result in communication among the parties ($\mathcal{T}$). We assume the following operations:

- $\mathrm{Enc}_{\mathrm{S}} : Y \times \mathcal{R} \times \mathcal{I} \to \mathscr{E}_{\mathrm{S}}(Y) \times \mathcal{R} \times \mathcal{T}$, "hiding" $y \in Y$.
- $\mathrm{Dec}_{\mathrm{S}} : \mathscr{E}_{\mathrm{S}}(Y) \times \mathcal{R} \times \mathcal{I} \to Y \times \mathcal{R} \times \mathcal{T}$, "unhiding".
- $\mathrm{Enc}_{\alpha,\beta}(\mathrm{op}_i) : \mathscr{E}_{\alpha}(Y) \times \mathscr{E}_{\beta}(Y) \times \mathcal{R} \times \mathcal{I} \to \mathscr{E}_{\alpha \sqcup \beta}(Y) \times \mathcal{R} \times \mathcal{T}$ (when $\alpha \sqcup \beta = \mathrm{S}$), evaluating a primitive.

For notational uniformity, as above, we also define the corresponding operations in the degenerate case of "hiding" public values: $\mathrm{Enc}_{\mathrm{P}}(y, R, \iota) = (y, R, \varepsilon)$, $\mathrm{Dec}_{\mathrm{P}}(y, R, \iota) = (y, R, \varepsilon)$, and $\mathrm{Enc}_{\mathrm{P},\mathrm{P}}(\mathrm{op}_i)(y_1, y_2, R, \iota) = (\mathrm{op}_i(y_1, y_2), R, \varepsilon)$.

In reasoning about the distributed semantics, we require that all of the protocol operations consume randomness sources correctly, i.e., when given random sources $R = (R_C, R_{S_1}, \ldots, R_{S_N})$, each operation returns a tuple $R' = (R'_C, R'_{S_1}, \ldots, R'_{S_N})$, where each $R'_P$ is a suffix of $R_P$ and the entire result of the operation depends only on the prefix consumed (and thus independent of $R'_P$). As a corollary, any operation given uniform randomness $U_{\mathcal{R}}$ must return $U_{\mathcal{R}}$.

**Cryptographic functional correctness** We assume the usual encryption and homomorphism conditions, augmented for cryptographic primitives that depend on explicit randomness and that may communicate among servers to produce their result. More precisely, for every $y \in Y$, and every choice of initial parameters $\iota \in \mathcal{I}$, we assume a family of *safe sets* $\mathcal{E}_{\alpha}(y, \iota)$: intuitively, any value $l \in \mathcal{E}_{\alpha}(y, \iota)$ can safely serve as the "hiding" of $y$ under the initial parameters $\iota$ (at secrecy level $\alpha \in \{\mathrm{P}, \mathrm{S}\}$). More precisely:

- $\pi_1(\mathrm{Enc}_{\alpha}(y, R, \iota)) \in \mathcal{E}_{\alpha}(y, \iota)$

We also require that unhiding ("decryption") is the left-inverse of hiding ("encryption"), and hiding commutes homomorphically with the primitive operations:

- $\pi_1(\mathrm{Dec}_{\alpha}(\pi_1(\mathrm{Enc}_{\alpha}(y, R_1, \iota)), R_2, \iota)) = y$
- $\pi_1(\mathrm{Enc}_{\alpha,\beta}(\mathrm{op}_i)(l_1, l_2), R_3, \iota) \in \mathcal{E}_{\alpha \sqcup \beta}(\mathrm{op}_i(y_1, y_2))$ whenever $l_1 \in \mathcal{E}_{\alpha}(y_1, \iota)$ and $l_2 \in \mathcal{E}_{\beta}(y_2, \iota)$

**Cryptographic statistical correctness** Analogous to functional correctness, for every $y \in Y$, and every choice of initial parameters $\iota \in \mathcal{I}$, we assume a family of *safe distributions* $\hat{\mathcal{E}}_\alpha(y, \iota)$ over the safe sets $\mathcal{E}_\alpha(y, \iota)$: intuitively, any distribution $l \in \hat{\mathcal{E}}_\alpha(y, \iota)$ can safely serve as the "hiding" of $y$ under the initial parameters $\iota$ (at secrecy level $\alpha \in \{\mathrm{P}, \mathrm{S}\}$), assuming randomness is uniform at all stages. We require that "hiding" a base value using a uniform randomness source must yield a safe distribution:

- $\pi_1(\mathrm{Enc}_\alpha(y, U_\mathcal{R}, \iota)) \in \hat{\mathcal{E}}_\alpha(y, \iota)$

In addition, for any two base values $y_1$ and $y_2$, we require that evaluating a primitive operation $\mathrm{op}_i$ on safe distributions of these two values must yield a safe distribution of $\mathrm{op}_i(y_1, y_2)$:

- $\pi_1(\mathrm{Enc}_{\alpha,\beta}(\mathrm{op}_i)(l_1, l_2, U_\mathcal{R}, \iota)) \in \hat{\mathcal{E}}_{\alpha\sqcup\beta}(\mathrm{op}_i(y_1, y_2), \iota)$ whenever $l_1 \in \hat{\mathcal{E}}_\alpha(y, \iota)$ and $l_2 \in \hat{\mathcal{E}}_\beta(y, \iota)$

**Indistinguishability conditions** The distributed threat model may generally involve any set of possible combinations of colluding servers. We formalize this by assuming a family $\mathcal{A} \subseteq 2^{\{S_1,\ldots,S_N\}}$ of sets that we refer to as valid sets of untrusted servers. Intuitively, for any $A \in \mathcal{A}$, we assume the cryptographic primitives are intended to provide security even if an adversary has access to all information possessed by servers in $A$.

Since different platforms may provide different security guarantees of their primitives, we assume a generic notion of indistinguishability; for the purposes of our examples, we will restrict our attention to information-theoretic indistinguishability and computational indistinguishability (with respect to some security parameter of the implementation), but our results easily generalize. Using the form of indistinguishability provided by the platform in question, we assume that any two sequences of partial traces are indistinguishable if each pair of corresponding partial traces describes either a primitive operation or a "hiding" operation on two safely-distributed values.[1] More precisely, for all $T(\iota) = (T_1(\iota), \ldots, T_r(\iota))$ and $T'(\iota) = (T'_1(\iota), \ldots, T'_r(\iota))$, if for each $i$, either:

$$\begin{array}{rcl} T_i(\iota) & = & \pi_3(\mathrm{Enc}_\mathrm{S}(y_i, U_\mathcal{R}, \iota)) \quad \text{or} \\ T_i(\iota) & = & \pi_3(\mathrm{Enc}_{\alpha,\beta}(\mathrm{op}_i)(L_{i,1}(\iota), L_{i,2}(\iota), U_\mathcal{R}, \iota)) \\ & & \text{where } L_{i,1}(\iota) \in \hat{\mathcal{E}}_\alpha(y_{i,1}, \iota) \text{ and } L_{i,2}(\iota) \in \hat{\mathcal{E}}_\beta(y_{i,2}, \iota) \\ O(\iota) & = & (\pi_A(\iota), \pi_A(T_1(\iota)), \ldots, \pi_A(T_k(\iota))) \end{array}$$

(and analogously for $O', T'$, substituting $y'_i, y'_{i,1}, y'_{i,2}$ for $y_i, y_{i,1}, y_{i,2}$), then the distributions $O(\hat{\iota})$ and $O'(\hat{\iota})$ are indistinguishable.

▶ **Definition 1.** We say that a platform $(Z, N, M, \mathscr{E}, \mathrm{Enc}, \mathcal{A})$ is a *secure execution platform* for $(Y, (\mathrm{op}_i))$ if it satisfies all of the assumptions of this section.

## 3.1 Framework

We introduce a simple language, $\lambda^{\rightarrow}_{\mathrm{P,S}}$, based on the simply-typed lambda calculus with base values and primitive operations. In addition to standard constructs, expressions in $\lambda^{\rightarrow}_{\mathrm{P,S}}$ may include variables bound at the program level by the `read` construct, representing secret values input by the clients before the body of the program is evaluated; these input variables are represented by capital letters $X$ (in contrast to lambda-bound variables, which use lower-case letters $x$), to emphasize the phase distinction between input processing and evaluation

[1] These values may be either secret (S) or public (P). In the latter case, we still assume that the communication traces are indistinguishable, since a properly implemented protocol should not need to exchange publicly-known information between servers at each operation.

of the program body. Programs in $\lambda^{\rightarrow}_{\mathrm{P,S}}$ may also include `reveal` operations, which specify that the value in question need not be kept secret during the computation. Throughout this section, we assume a set $Y$, primitive operations $(\mathrm{op}_i)$, and a secure execution platform for $(Y, (\mathrm{op}_i))$, as specified in Section 3.

**Listing 1** Syntax for expressions and programs.

$$\begin{array}{lcl} e & ::= & x \mid \lambda x.e \mid e_1\, e_2 \mid \mathrm{op}_i(e_1, e_2) \mid y \in Y \mid X \mid \texttt{reveal}\ e \\ p & ::= & \texttt{read}\ X_1, \ldots, X_r\,; e \end{array}$$

The static semantics (Listing 2) are standard; we assume the two-element security lattice $\{\mathrm{P}, \mathrm{S}\}$, $\mathrm{P} \sqsubseteq \mathrm{S}$, denoting the types of (respectively) public values, which may be revealed to any party (including the servers); and secret values, about which the protocol may reveal no information. Note that we include both the static semantics for expressions ($\Gamma \vdash e : \tau$) and those for values ($\Gamma \vdash_v v : \tau$).

**Listing 2** Static semantics for expressions and values.

$$\frac{}{\Gamma \vdash y : (Y, \mathrm{P})} \qquad \frac{}{\Gamma \vdash X : (Y, \mathrm{S})} \qquad \frac{\Gamma \vdash e : (Y, \mathrm{S})}{\Gamma \vdash \texttt{reveal}\ e : (Y, \mathrm{P})} \qquad \frac{\Gamma[x \mapsto \tau_1] \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \rightarrow \tau_2}$$

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\, e_2 : \tau_2} \qquad \frac{\Gamma \vdash e_1 : (Y, \alpha) \qquad \Gamma \vdash e_2 : (Y, \beta)}{\Gamma \vdash \mathrm{op}_i(e_1, e_2) : (Y, \alpha \sqcup \beta)}$$

$$\frac{y \in Y}{\Gamma \vdash_v (y, \alpha) : (Y, \alpha)} \qquad \frac{\Gamma \vdash \lambda x.e : \tau_1 \rightarrow \tau_2}{\Gamma \vdash_v \lambda x.e : \tau_1 \rightarrow \tau_2}$$

We give a standard dynamic semantics for $\lambda^{\rightarrow}_{\mathrm{P,S}}$ (Listing 3), based on the usual evaluation rules for lambda calculus with primitive operations; to simplify notation, we overload the symbol $\downarrow$ to represent the evaluation judgments for programs, $(\kappa, p) \downarrow (v, \mathcal{O})$, as well as those for expressions, $(\kappa, \rho, e) \downarrow (v, \mathcal{O})$. The environment $\kappa$ represents the initial (secret) values supplied by the client. Operationally, the `read` construct is a no-op, but for clarity we retain it in the syntax, since in the implementation semantics (Listing 5) it will represent the "hiding" and initial transmission of the values from the client to the servers. The `reveal` construct acts as a cast from S to P, and may therefore have side effects in an implementation of $\lambda^{\rightarrow}_{\mathrm{P,S}}$ (as discussed below), but these effects are guaranteed to be benign with respect to functional correctness, since they do not change the first component of the resulting value in the dynamic semantics (Listing 3). We also track a list of "observations", $\mathcal{O}$, throughout the evaluation, holding all values ever supplied to `reveal`; this is important in proving security properties (Theorem 6), as we will show that an appropriately constrained adversary learns nothing except what is entailed by these observations.

We have the usual type safety theorem (encompassing both progress and preservation):[2]

▶ **Theorem 2** (Soundness for Reference Semantics). *If $\emptyset \vdash e : \tau$, $p =$ `read` $X_1, \ldots, X_r; e$, $\mathrm{FV}(e) \subseteq \{X_1, \ldots, X_r\}$, and $\kappa$ maps each $X_r$ to an element of $Y$, then there exists a value $v$ and an observation sequence $\mathcal{O}$ such that $(\kappa, p) \downarrow (v, \mathcal{O})$ and $\emptyset \vdash_v v : \tau$.*

[2] For space reasons, we omit the proofs of theorems in this section.

**Listing 3** "Reference" dynamic semantics for $\lambda^{\rightarrow}_{\mathrm{P,S}}$.

$$\frac{}{(\kappa,\rho,y)\downarrow((y,\mathrm{P}),\varepsilon)}\qquad\frac{}{(\kappa,\rho,X)\downarrow((\kappa(X),\mathrm{S}),\varepsilon)}\qquad\frac{}{(\kappa,\rho,x)\downarrow(\rho(x),\varepsilon)}\qquad\frac{}{(\kappa,\rho,\lambda x.e)\downarrow(\lambda x.e,\varepsilon)}$$

$$\frac{(\kappa,\rho,e)\downarrow((y,\mathrm{S}),\mathcal{O})}{(\kappa,\rho,\texttt{reveal}\ e)\downarrow((y,\mathrm{P}),\mathcal{O}\|y)}\qquad\frac{(\kappa,\rho,e_1)\downarrow(\lambda x.e,\mathcal{O}_1)\qquad(\kappa,\rho,e_2)\downarrow(v_2,\mathcal{O}_2)\qquad(\kappa,\rho[x\mapsto v_2],e)\downarrow(v,\mathcal{O}_3)}{(\kappa,\rho,e_1\,e_2)\downarrow(v,\mathcal{O}_1\|\mathcal{O}_2\|\mathcal{O}_3)}$$

$$\frac{(\kappa,\rho,e_1)\downarrow((y_1,\alpha),\mathcal{O}_1)\qquad(\kappa,\rho,e_2)\downarrow((y_2,\beta),\mathcal{O}_2)}{(\kappa,\rho,\mathrm{op}_i(e_1,e_2))\downarrow((\mathrm{op}_i(y_1,y_2),\alpha\sqcup\beta),\mathcal{O}_1\|\mathcal{O}_2)}\qquad\frac{(\kappa,\emptyset,e)\downarrow(v,\mathcal{O})}{(\kappa,\texttt{read}\ X_1,\ldots,X_r;e)\downarrow(v,\mathcal{O})}$$

In order to address correctness and security of implementations, we augment the language $\lambda^{\rightarrow}_{\mathrm{P,S}}$ so that there is an additional case for result values, $l\in\mathscr{E}_{\mathrm{S}}(Y,\mathcal{I})$, representing hidden values; we denote this augmented language by $\hat{\lambda}^{\rightarrow}_{\mathrm{P,S}}$. We give a dynamic semantics for $\hat{\lambda}^{\rightarrow}_{\mathrm{P,S}}$ in Listing 5. In contrast to the first, "reference", dynamic semantics for $\lambda^{\rightarrow}_{\mathrm{P,S}}$, the "distributed" semantics for $\hat{\lambda}^{\rightarrow}_{\mathrm{P,S}}$ reflects the steps taken by an actual implementation. We again have the usual type safety theorem for $\hat{\lambda}^{\rightarrow}_{\mathrm{P,S}}$ under the distributed semantics:

▶ **Theorem 3** (Soundness for Distributed Semantics). *If $\emptyset\vdash e:\tau$, $p=\textit{read}(X_1,\ldots,X_r);e$, $\mathrm{FV}(e)\subseteq\{X_1,\ldots,X_r\}$, and $\kappa$ maps each $X_i$ to an element of $Y$, then for all $\iota\in\mathcal{I}$, randomness sources $R\in\mathcal{R}$, there exists a value $w$, a trace $T$, and a randomness source $R'\in\mathcal{R}$ such that $(\kappa,R,p)\Downarrow(w,R',T)$ and $\emptyset\vdash_{tv}w:\tau$.*

**Listing 4** Static semantics for values ("distributed" semantics).

$$\frac{y\in Y\qquad l\in\mathscr{E}_\alpha(y)}{\Gamma\vdash_{tv}(l,\alpha):(Y,\alpha)}\qquad\frac{\Gamma\vdash\lambda x.e:\tau_1\to\tau_2}{\Gamma\vdash_{tv}\lambda x.e:\tau_1\to\tau_2}$$

The reference semantics expresses the standard meaning of programs in $\lambda^{\rightarrow}_{\mathrm{P,S}}$, while the distributed semantics expresses in more detail how an implementation should realize them. Evidently, in a correct system we would expect evaluation to arrive at equivalent results in both cases; this is guaranteed by the following theorem (where the relevant similarity relation is defined in Listing 6):

▶ **Theorem 4** (Functional Correctness). *If $\emptyset\vdash e:\tau$, $p=\textit{read}(X_1,\ldots,X_r);e$, $\mathrm{FV}(e)\subseteq\{X_1,\ldots,X_r\}$, $\kappa$ maps each $X_i$ to an element of $Y$, and $(\kappa,p)\downarrow(v,\mathcal{O})$, then for all $R\in\mathcal{R}$, there exist $R'\in\mathcal{R}$, $T$, $w$, and $\iota$ such that $(\kappa,R,p)\Downarrow(T,R',w,\mathcal{O})$ and $v\sim^{\emptyset,\iota}_{\tau}w$.*

Functional correctness expresses that for any well-formed randomness source $R\in\mathcal{R}$, regardless of whether it was in fact generated randomly, the distributed semantics yields the correct answer. It will also be useful to have a correctness theorem expressing the behavior of the system when given a truly random source. In particular, if we regard the values in question as random variables, and assume that at the beginning of the computation they satisfy appropriate safe distributions as given by $\hat{\mathcal{E}}_{\{\mathrm{P,S}\}}(\cdot,\cdot)$, we can show that values remain in such distributions throughout the computation (Theorem 5). In order to state this result, we introduce a similarity relation $\approx^{\Gamma,\iota}_{\tau}$ (Listing 6) to relate values in the reference semantics with their safe distributions.

**Listing 5** "Distributed" dynamic semantics for $\hat{\lambda}^{\rightarrow}_{\mathrm{P,S}}$.

$$\frac{}{(\iota, \Psi, \Delta, R, y) \Downarrow (\varepsilon, R, (y, \mathrm{P}), \varepsilon)} \qquad \frac{}{(\iota, \Psi, \Delta, R, X) \Downarrow (\varepsilon, R, (\Psi(X), \mathrm{S}), \varepsilon)}$$

$$\frac{}{(\iota, \Psi, \Delta, R, x) \Downarrow (\varepsilon, R, \Delta(x), \varepsilon)} \qquad \frac{}{(\iota, \Psi, \Delta, R, \lambda x.e) \Downarrow (\varepsilon, R, \lambda x.e, \varepsilon)}$$

$$\frac{(\iota, \Psi, \Delta, R, e) \Downarrow (T_1, R_1, (l, \mathrm{S}), \mathcal{O}_1) \qquad (y, R_2, T_2) = \mathrm{Dec}_{\mathrm{S}}(l, R_1, \iota)}{(\iota, \Psi, \Delta, R, \texttt{reveal}\ e) \Downarrow (T_1 \| T_2, R_2, (y, \mathrm{P}), \mathcal{O}_1 \| y)}$$

$$\frac{\begin{array}{c}(\iota, \Psi, \Delta, R, e_1) \Downarrow (T_1, R_1, \lambda x.e, \mathcal{O}_1) \\ (\iota, \Psi, \Delta, R_1, e_2) \Downarrow (T_2, R_2, v_2, \mathcal{O}_2) \qquad (\iota, \Psi, \Delta[x \mapsto v_2], R_2, e) \Downarrow (T_3, R_3, v, \mathcal{O}_3)\end{array}}{(\iota, \Psi, \Delta, R, e_1 e_2) \Downarrow (T_1 \| T_2 \| T_3, R_3, v, \mathcal{O}_1 \| \mathcal{O}_2 \| \mathcal{O}_3)}$$

$$\frac{\begin{array}{c}(\iota, \Psi, \Delta, R, e_1) \Downarrow (T_1, R_1, (l_1, \alpha), \mathcal{O}_1) \\ (\iota, \Psi, \Delta, R_1, e_2) \Downarrow (T_2, R_2, (l_2, \beta), \mathcal{O}_2) \qquad (l, T_3, R_3) = \mathrm{Enc}_{\alpha,\beta}(\mathrm{op}_i)(l_1, l_2, R_2, \iota) \\ T = T_1 \| T_2 \| T_3 \qquad \mathcal{O} = \mathcal{O}_1 \| \mathcal{O}_2\end{array}}{(\iota, \Psi, \Delta, R, \mathrm{op}_i(e_1, e_2)) \Downarrow (T, R_3, (l, \alpha \sqcup \beta), \mathcal{O})}$$

$$\frac{\begin{array}{c}(R_0, \iota) = \mathrm{Init}(R) \qquad \forall i \in \{1, \ldots, N\}.\ T_i = \{(C, S_i, \Pi_{\{S_i\}}(\iota)\} \\ \forall j \in \{1, \ldots, r\}.\ (l_j, R_j, T'_j) = \mathrm{Enc}_{\mathrm{S}}(\kappa(X_j), R_{j-1}, \iota) \\ (\iota, \{X_1 \mapsto l_l, \ldots, X_r \mapsto l_r\}, \emptyset, R_r, e) \Downarrow (T, v, R', \mathcal{O}) \qquad T' = T_1 \| \ldots \| T_N \| T'_1 \| \ldots \| T'_r \| T\end{array}}{(\kappa, R, \texttt{read}(X_1, \ldots, X_r); e) \Downarrow (T', v, R', \mathcal{O})}$$

**Listing 6** Similarity relations for functional and statistical correctness.

$$\frac{l \in \mathcal{E}_\alpha(y, \iota)}{(y, \alpha) \sim^{\Gamma,\iota}_{(Y,\alpha)} (l, \alpha)} \qquad \frac{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}{\lambda x.e \sim^{\Gamma,\iota}_{\tau_1 \to \tau_2} \lambda x.e} \qquad \frac{l \in \hat{\mathcal{E}}_\alpha(y, \iota)}{(y, \alpha) \approx^{\Gamma,\iota}_{(Y,\alpha)} (l, \alpha)} \qquad \frac{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}{\lambda x.e \approx^{\Gamma,\iota}_{\tau_1 \to \tau_2} \lambda x.e}$$

▶ **Theorem 5** (Statistical Correctness). *If $\emptyset \vdash e : \tau$, $p = \textbf{\textit{read}}(X_1, \ldots, X_r); e$, $\mathrm{FV}(e) \subseteq \{X_1, \ldots, X_r\}$, $\kappa$ maps each $X_i$ to an element of $Y$, and $(\kappa, p) \downarrow (v, \mathcal{O})$, then there exist $T$ and $w$ such that $(\kappa, U_{\mathcal{R}}, p) \Downarrow (T, U_{\mathcal{R}}, w, \mathcal{O})$ and $v \approx^{\emptyset,\hat{\iota}}_{\tau} w$ (where the semantics judgments are lifted to distributions).*

For security, however, the above results are not sufficient. Rather, we now show that if, during the evaluation of a program in $\lambda^{\rightarrow}_{\mathrm{P,S}}$, an adversary is confined to observing the data visible to a valid subset of untrusted servers $A \in \mathcal{A}$ (represented by their views of the communication trace), then that adversary learns nothing about the initial secret client values that was not already implied by the observations from `reveal`:

▶ **Theorem 6** (Security). *If $\emptyset \vdash e : \tau$, $p = \textbf{\textit{read}}(X_1, \ldots,\ X_r); e$, $(\kappa, U_{\mathcal{R}}, p) \Downarrow (T, U_{\mathcal{R}}, v, \mathcal{O})$, and $(\kappa', U_{\mathcal{R}}, p) \Downarrow (T', U_{\mathcal{R}}, v', \mathcal{O})$, then for all valid sets of untrusted servers $A \in \mathcal{A}$, the distributions $\Pi_A(T)$ and $\Pi_A(T')$ are indistinguishable (in the sense specified by the secure execution platform, as described in Section 3).*

We remark that although the conclusion of this theorem seems simple, it requires some care to set up the proof correctly. In particular, we can proceed by showing inductively that the two evaluation derivations take the same form, with all resulting values, observations, and traces being structurally equal; moreover, all traces can be decomposed into secret components (which, by statistical correctness, must satisfy the hypothesis of the indistinguishability assumption), and public components (which are identical between $T$ and $T'$, since both evaluations yield the same observations). We may then conclude indistinguishability of the projections $\Pi_A(T)$ and $\Pi_A(T')$.

## 3.2 Shamir secret sharing

We now define Shamir secret sharing in the notation of our framework (Section 3), and show that it is a secure execution platform (Definition 1) for addition and multiplication over a finite field, thereby concluding all of the correctness and security results of Section 3.1 as applied to $\lambda^{\rightarrow}_{\mathrm{P,S}}$ with these two primitive operations. Let $N$ be the number of servers executing the computation (i.e., we use an $(N, k)$ sharing). The set of base values $Y$ is the finite field $\mathbb{F}_p$, where $p$ is a parameter of the implementation,[3] equipped with the usual operations of addition and multiplication ($\mathrm{op}_1(x, y) = (x + y) \bmod p$, $\mathrm{op}_2(x, y) = (x \cdot y) \bmod p$). The sets $M$ of messages and $Z$ of random numbers are also defined to be $\mathbb{F}_p$. We define the set of "hidden equivalents" $\mathscr{E}_{\mathrm{S}}(\mathbb{F}_p)$ to be $\mathbb{F}_p^N$; during computations, we will be concerned specifically with inhabitants of $\mathscr{E}_{\mathrm{S}}(\mathbb{F}_p)$ that represent each of the $N$ servers' shares of some base value. Apart from the initial secret sharing, there is no initialization phase, so we let $\mathcal{I}$ be the singleton set $\{()\}$.

The "hiding" and "unhiding" operations are defined using the standard Shamir secret sharing constructions, as described in Section 2. (For brevity, we defer the formal definitions to the extended version of this article[4].) The primitive operations $\mathrm{Enc}_{\mathrm{S,S}}(+)$, $\mathrm{Enc}_{\mathrm{S,P}}(+)$, $\mathrm{Enc}_{\mathrm{P,S}}(+)$, $\mathrm{Enc}_{\mathrm{S,S}}(*)$, $\mathrm{Enc}_{\mathrm{S,P}}(*)$, and $\mathrm{Enc}_{\mathrm{P,S}}(*)$ are defined similarly, following Section 2; We note that each of the secret sharing operations consumes randomness correctly, by definition. Further, since any base value has only one distribution that can result from using uniform randomness (namely, the uniform distribution over all valid sharings), we define the set of safe distributions to contain only this one: $\hat{\mathcal{E}}_{\mathrm{S}}(n, ()) = \{\mathcal{D}(n)\} = \{\mathrm{Enc}_{\mathrm{S}}(n, U_{\mathcal{R}}, ())\}$. We also define $\mathcal{A}$, the family of valid untrusted subsets of the servers, to include exactly those subsets with cardinality less than $k$, and we specify that the system should provide information-theoretic security. Assuming this specification, the required functional correctness, statistical correctness, and indistinguishability properties of the primitives follow from the properties of secret sharing outlined in Section 2. (Again for brevity, we omit proofs of all of these properties, but we refer the reader to the extended version.)

Given that the operations of Shamir secret sharing satisfy all of the required properties (as enumerated in Section 3), we can conclude that Shamir secret sharing is a secure execution platform for $(+, \times)$, and thus all of the results of Section 3.1 hold of programs in $\lambda^{\rightarrow}_{\mathrm{P,S}}$ when it is given the semantics of Shamir secret sharing. In particular, functional correctness (Theorem 4) takes on the flavor of a "SIMD" property, stating that the evaluation of a program on $N$ servers results in $N$-tuples in the "distributed" semantics (a share for each server) being produced in lock-step with their equivalents (the shared value) in the "reference" semantics. Moreover, the security result (Theorem 6) now guarantees the desired secrecy property for the entire language: if the adversary can observe the data from at most $k$ of the servers, then even with unbounded computational resources, it cannot distinguish between any two initial secret value environments, except to the extent that they cause different values to be provided to explicit "`reveal`" directives in the program.

[3] In practice, it is more useful to have programs act on integers rather than elements of a finite field. This can be done via a static analysis that infers the largest possible integer value that can arise during the execution, given bounds on the input values.

[4] Available at `http://eprint.iacr.org/2011/561`.

## 3.3 Fully homomorphic encryption

In addition to secure multiparty computation, a variety of homomorphic encryption schemes can also serve as secure execution platforms for standard primitive operations. In particular, we will now show that any fully homomorphic encryption scheme, and notably Gentry's scheme [11] (under the appropriate cryptographic assumptions), is a secure execution platform for addition and multiplication over the ring $\mathbb{Z}_{2^k}$, achieving security against a computationally-bounded adversary.

In fully homomorphic encryption, the number of servers, $N$, is 1; the client simply sends encrypted values to the server, and the server performs the computation homomorphically, returning the encrypted result. Although traditionally the operations provided under fully homomorphic encryption would be a complete set of circuit gates, in order to provide a better analogy with secret sharing we define the set of base values $Y$ to be the ring $\mathbb{Z}_{2^k}$, and the operations $(\mathrm{op}_1, \mathrm{op}_2)$ to be addition and multiplication in the ring. The initialization step is just $\mathrm{Init} = \mathsf{KeyGen}(\lambda)$ generating the public/private key pair,[5] where $\lambda$ is the security parameter to the system.

To begin the computation, the client sends the public key to the server (i.e., $\Pi_{\{S_1\}}(\iota)$ here is $\Pi_{\{S_1\}}((\mathrm{sk}, \mathrm{pk})) = \mathrm{pk}$), then encrypts all of the initial values one bit at a time and sends the corresponding ciphertexts to the server (i.e., $\mathrm{Enc}_\mathrm{S}(b_k b_{k-1} \cdots b_1, (\mathrm{sk}, \mathrm{pk})) = (\Psi, \{(C, S_1, \Psi)\})$ where $\Psi = (\mathsf{Enc}(\mathrm{pk}, b_1), \ldots, \mathsf{Enc}(\mathrm{pk}, b_k))$). During the computation, the server itself performs additions and multiplications on the ciphertexts by homomorphically evaluating the corresponding circuits, producing no communication trace with the client (i.e., $\mathrm{Enc}_{\mathrm{S,S}}(\mathrm{op})(\Psi_1, \Psi_2, (\mathrm{sk}, \mathrm{pk})) = (\mathsf{Eval}(\mathrm{pk}, \mathrm{op}, \Psi_1, \Psi_2), \varepsilon)$); when one of the operands is a public value (i.e., $\mathrm{Enc}_{\mathrm{S,P}}$, $\mathrm{Enc}_{\mathrm{P,S}}$), the server simply "hides" it using $\mathsf{Enc}(\mathrm{pk}, \cdot)$, and then uses $\mathrm{Enc}_{\mathrm{S,S}}$. For `reveal` operations, the server sends back to the client a tuple of ciphertexts to be decrypted, and the corresponding plaintexts (bits of some base value) are returned to the server (i.e., $\mathrm{Dec}_\mathrm{S}(\Phi, (\mathrm{sk}, \mathrm{pk})) = (n, \{(S_1, C, \Phi), (C, S_1, n)\})$ where $n = \sum_{i=1}^{k} b_i 2^i$, $b_i = \mathsf{Dec}(\mathrm{sk}, \Phi_i)$). Finally, given these operations, we note that the set $\mathscr{E}_\mathrm{S}(Y)$ of possible "hidden" values should be defined as the set of $k$-tuples of ciphertexts, while the set $M$ of messages in $M$ consists of ciphertexts, plaintexts, and $k$-tuples of ciphertexts.

Functional correctness of the primitives follows directly from the homomorphic properties of the encryption scheme. For statistical correctness, we can trivially define a safe distribution to be any distribution $l \in \hat{\mathcal{E}}_\mathrm{S}(y, \iota)$. Indistinguishability is then immediate for partial traces derived from $\mathrm{op}_i$, since these operations produce empty traces. For the other partial traces (i.e., the initial encryptions $\mathrm{Enc}_\mathrm{S}$), indistinguishability follows from CPA-security of the encryption scheme, since the only values in the traces are the encryptions of each of the bits of the secret client inputs.

Thus, fully homomorphic encryption is a secure execution platform for $(+, \times)$, and as above, all of the results of Section 3.1 hold of programs in $\lambda^{\rightarrow}_{\mathrm{P,S}}$ when it is given the semantics of fully homomorphic encryption (now obtaining security guarantees against a computationally bounded adversary).

## 4 Implementation

We implemented the language of Section 3 as an EDSL in Haskell. Our implementation framework consists of a module that defines the language interface, and SMC and FHE

[5] For clarity, we elide the randomness sources in this section.

libraries that implement the interface combinators. In this section we detail the EDSL and underlying libraries.

## 4.1 Haskell Secure Cloud Computing EDSL

Our EDSL defines a generic interface, extending the language given in Listing 1. We use the type alias `BType` to denote the base type $Y$, and `LType` to denote the hidden, or lifted, type $\mathscr{E}_{\mathrm{S}}(Y)$. Additionally, we provide `SIO`, a "secret" `IO` monad, which is used to carry out IO operations and thread platform state (e.g., $\mathcal{R}$ and $\mathcal{T}$ of Section 3) through a given computation.

As previously mentioned, we use Haskell type classes to overload the operators core to the EDSL syntax. As many library functions have side effects (e.g., the SMC multiplication requires network communication) we prefix the EDSL operators with '`.`', and functions with '`s`', as to avoid name collisions with the standard `Prelude` library that is implicitly imported by every Haskell module. Below we detail some of the core aspects of our EDSL. However, we note that, compared to SMCL and other, similar, DSLs, we do not provide any loop constructs—our Haskell embedding allow a programmer to use existing high-order constructs (including general recursion) to create very powerful application-specific loop constructs.

**Primitive operations** Secure addition, subtraction and multiplication operators are defined using the multi-parameter type class `EDSLArith`. The use of multi-parameter type classes allows us to define instances of the operators with operands of mixed secrecy types (e.g., addition of a public and hidden type). In a similar fashion, we provide standard comparison operators, and a random number generator (RNG) interface. The RNG implementation is, however, limited to SMC following [25].

We leverage Haskell's strong type system (and `newtype` declaration) to provide a hidden Boolean type. Specifically, we introduce `BoolLType` as a wrapper for `LType`, hiding the constructor from the programmer (to avoid unsafe coercions). However, we provides basic Boolean arithmetic and logic operators, including bit–and, bit–or, bit–exclusive-or, $\wedge, \vee$, and $\neg$. Directly, our EDSL can be used to enforce safety of conditionals on hidden values. Specifically, we provide the construct `sif` $c$ `sthen` $x$ `selse` $y$, which is implemented by safe arithmetization (i.e., $c \cdot x + (1 - c) \cdot y$, that preserves/restores types). In addition to type-safety, this allows writing code using familiar syntax. For example, we can write the max function simply as: `max x y = sif (x .<= y) sthen y selse x`.

**Hiding and unhiding functions** Further using type classes, we define the `EDSLHide` class which declares `hide`, a Haskell function corresponding to $\mathrm{Enc}_{\mathrm{S}}$; `hide` maps values to their secret equivalent. Dually, we declare `reveal` and the `EDSLReveal` type class that implements the functionality of $\mathrm{Dec}_{\mathrm{S}}$ of Section 3; `reveal` maps hidden, or secret, values to their public equivalent.

**User I/O** We provide three combinators for interacting with users: `uRead`, `uWrite`, and `uPutStrLn`. `uRead` is used to request a user for input; the user responds by sending a hidden value to the server(s). Dually, `uWrite` is used to send a hidden value to the user, who then locally unhides the value. Observe, that, using

```
withUsers  :: (BType → SIO α) → SIO [α]
withUsers_ :: (BType → SIO α) → SIO ()
```

**Figure 4** Iterating over users

this construct, a programmer can write a program that reveals results only to clients. Finally, `uPutStrLn` is used to print a string on a user's terminal. To execute IO actions on all the connected user clients, we provide `withUsers` and `withUsers_`, shown in Figure 4. The former executes a function on all the clients, returning a list of results, while the latter discards the results (useful, e.g., when executing `uWrite`).

## 4.2 SMC & FHE Library Implementations

In this section we present our SMC and FHE libraries, which instantiate our EDSL with the secure execution platform respectively based on Shamir secret sharing and the Gentry-Halevi FHE implementation [14, 13]. In our framework, each program, such as that of Figure 2, that is executed by the Cloud server parties is actually an `SIO` action. Hence, all the configuration details (e.g., which clients are connected, or the identity of the executing server) are transparent and abstracted into this underlying monad and EDSL constructs. A programmer need only provide an initial configuration that specifies the participating server and client parties, in addition to the program. The same program and configuration is copied to all the Could servers—in the SMC case, the servers execute in a network-SIMD fashion, while in the FHE case the server executes in a standard (network-SISD) fashion. Clients, on the other hand, are event-based: they await instructions from the server(s) and simply respond accordingly. Below, we detail the core base and hidden types, and library-specific details on parties and the execution environment.

**Secure Multi-party Computation** To implement Shamir secret sharing, we define a base type `Zp` that represents elements of $\mathbb{F}_p$ as a wrapper for the Haskell's arbitrary precision `Integer` type with the standard operators corresponding to their finite field counterparts. Directly, we define a share, or hidden, type (`Share`) as a record enclosing a party number and share value, each of type `Zp`. Shamir secret sharing functions, described in Section 2, are shown in Figure 5, where the type `SMCScheme` is used to encode the $(N, k)$-scheme. Here, `share` breaks an element into shares, while `reconstruct` takes a list of shares and constructs the corresponding element. We highlight that `share` returns an `SIO` action: the function requires a RNG (we use a cryptographically secure deterministic random bit generator) to break an element into its shares, while `reconstruct` is pure. Further, we highlight, that, compared to the semantics of Listing 5, the RNG in part of underlying monad and not explicitly passed to functions.

```
share :: SMCScheme → Zp → SIO [Share]
reconstruct :: [Share] → Zp
```

**Figure 5** Shamir sharing constructs

As previously mentioned, our implementation relies on the notion of party, which we realize using the data type `Party`. A `Party` has an identifier, a network address (hostname, port, SSL certificate), and two typed communication channels: an inbox and outbox. After setting up a mutually-authenticated connection, parties can exchange message using the inbox/outbox channels. Specifically, parties can exchange messages of several forms: (i) a response (constructed with `RespShare`) when sending a server party a share from either a client or another server party, (ii) a request (`ReqShare`) when requesting a client for input, (iii) a reconstruct (`ReconstrShare`) when sending a client a share, who then combines all the received shares to reconstruct the hidden value, (iv) a print (`PrintStr`) when writing a string message to the user's terminal, and (v) a disconnect (`Disconnect`) when the computation has terminated, or failed. We found these message forms to be sufficient when implementing the core Shamir secret sharing EDSL constructs.

Each server party executes an SMC computation in two steps. First, each server listens for incoming connections from other server or client parties. Upon accepting a connection from a party it spawns two threads: a thread that reads incoming network messages and writes them to the inbox channel, and a thread that block-reads the outbox channel, serializes the message, and writes them to the network. Second, when all the servers are interconnected and every client is connected to all the servers, the server parties execute the EDSL program in lock-step, or SIMD fashion. The underlying monad abstracts-away and manages all the configuration details, such as to which party or channel a share should be sent. Of course, the configuration details are queried and used by constructs such as the multiplication operator `(.*)`.

**Fully Homomorphic Encryption** The Gentry-Halevi C++ implementation [14, 13] provides several functions, including a public/private key pair generation function, encryption/decryption functions, a recrypt (ciphertext refreshing) function and simple single-bit homomorphic arithmetic operators. We extend their implementation with $k$-bit homomorphic addition, multiplication, comparison and equality testing functions. To integrate the (extended) C++ FHE library into our Haskell framework, we further implemented C wrappers for the basic FHE operations, and various library functions—calling foreign functions in Haskell is accomplished using the Foreign Function Interface (FFI), which is currently best suited for interfacing with C.

Similar to the SMC case, we define a base and hidden type. Specifically, we define the base type (`ZZ`) as a simple wrapper for Haskell's `Int`, bounding it to $k$-bits. The hidden, encrypted, type is a wrapper for a C pointer (to a vector of "big integers") that allows for simple calling of the C/C++ FHE functions from Haskell. Although this adds the additional complexity of performing garbage collection of the C-allocated big integers, it allows us to use the optimzied C/C++ FHE functions when implementing the EDSL combinators such as the addition operator `(.+)`.

To support a practical Cloud-oriented FHE library, we require the separation of client and server code, and we thus provide functions that serialize and deserialize encrypted values. Directly, this allows for transmission of encrypted values over the network. From a networking perspective the FHE setting is a special case of SMC with $N = 1$. Hence, the FHE notion of party is similar to that of SMC described above, though it additionally requires associating public-private keys with a computation. However, among other differences, compared to SMC, where only server communication is necessary in unhiding, or decrypting, a value, in the FHE setting, communication with a client is necessary. These details are, of course, abstracted into the underlying `SIO` monad and corresponding EDSL constructs (e.g., `reveal`) and thus transparent to the programmer.

## 4.3 Performance Evaluation

Our SMC library, including the EDSL interface, and comparison protocol of [4], was implemented in roughly 1300 lines of Haskell code. Our FHE library was implemented in about 1200 lines of Haskell, and 650 lines of C/C++ code extending the Gentry-Halevi implementation. To evaluate the performance of these implementation we also implemented various programs, including the Clock-Auction, and mall benchmark suite of [24]. The suite consists of 3 programs that compute the sign of a quadratic polynomial: (i) the *ideal* program operates solely on hidden values, (ii) the *pragmatic* program operates on mixed-secrecy values—all values are secrec except for the evaluation point and the result of the polynomial evaluation, (iii) the *public* program operates solely on public values.

**Table 1** Performance benchmarks for SMC and FHE, where the security parameter $\lambda^{\text{toy}}$ corresponds to a "toy" security level (a lattice of with dimension 128). Tests with realistic parameters are currently unfeasible.

| Scheme | Ideal | Pragmatic | Public |
|---|---|---|---|
| SMC $(3, 1)$ | 0.97 sec | 3.3 ms | $< 1$ ms |
| SMC $(5, 2)$ | 1.02 sec | 3.3 ms | $< 1$ ms |
| SMC $(7, 3)$ | 1.04 sec | 3.3 ms | $< 1$ ms |
| FHE $\lambda^{\text{toy}}$ | 17.6 min | 5.3 min | $< 1$ ms |

Table 1 presents our results for various SMC configurations and a "toy" FHE configuration. The SMC implementation uses arithmetic modulo the largest 32-bit prime, while the FHE implementation operates on 8-bit integers. Our experimental setup consisted of 7 machines, interconnected on a local Gig-E network, each machine containing two Intel Xeon E5620 (2.4GHz) processors and 48GB of RAM. Similar to the results of SMCL [24], we observe that the SMC pragmatic version is an order of magnitude faster than the ideal. Compared to their results, our system is significantly faster; however, this is not a meaningful comparison because we are using newer generation of hardware. More importantly, we note that the performance results of both the ideal and pragmatic SMC benchmarks highlight the usability of our SMC implementation for real-world applications.

## 5 Related work

Among several projects demonstrating potential applications of secure multiparty computations, SCET [5], with its focus on economic applications, implemented secure double auction. In Fairplay [22], programs written in SFDL were converted to primitive operations on bits. Fairplay was restricted to only two parties; this drawback was removed in FairplayMP. Sharemind [3] aimed at general multiparty computation on large datasets, supporting three players and providing security against a passive adversary.

VIFF [8] provides a basic language embedded in Python and API calls to cryptographic primitives. It provides Shamir and pseudorandom secret sharing as options to the programmer. VIFF can be seen as a system for expert programmers to build complex cryptographic protocols. Indeed, VIFF has been used for building distributed implementations of RSA and AES. In contrast, our EDSL is for writing applications by nonexpert programmers, and permits one to write at a substantially higher level of abstraction than that of the cryptographic primitives. Moreover, compared to Python, Haskell has a natural advantage as a host for EDSLs; as a functional language, Haskell allows extensive static reasoning about programs, performs a variety of optimizations, and has lightweight multithreading capabilities. On the other hand, our EDSL can complement systems such as VIFF by targeting it as a platform, providing a higher-level abstraction layer over its powerful and efficient cryptographic primitives.

From a theoretical standpoint, the systems discussed above are generally concerned with implementing cryptographic protocols, without proving the more comprehensive correctness and security properties we consider. The closest work is SMCL [24], an imperative-style DSL. The papers on SMCL contain proofs of correctness and security properties, but they do not formally define a crucial aspect: the requirements on the side-effects produced by primitive operations so that security can be guaranteed. Our system is also implemented as an EDSL, rather than as a standalone language, so it can leverage the full power of Haskell and its type system. In addition, unlike SMCL, our system easily generalizes to other cryptographic schemes. As far as we know, we are the first to formalize and prove correctness and security properties for a unified language framework which encompasses a wide range of cryptographic schemes for computation on encrypted data, in particular Shamir secret sharing and fully homomorphic encryption.

## 6 Conclusions

We present the design, foundational analysis, implementation, and performance benchmarks for an embedded domain-specific language that allows programmers to develop code that can be run on different secure execution platforms with different security guarantees. We prove functional correctness and confidentiality for any *secure execution platform* meeting our definitions and then show that a specific secret-sharing scheme and fully homomorphic encryption both meet our definition. Our language allows developers to produce a single program that can be executed on different secure execution platforms, making the deployment decisions after development according to security and performance requirements.

As a programming language, our embedded DSL, implemented as a Haskell library, allows developers to use standard Haskell software development environments. Programmers also have the benefit of sophisticated type-checking and general programming features of Haskell because we rely only on the Haskell type discipline to enforce information flow and other restrictions; there are no unexpected *ad hoc* code restrictions. Our Haskell implementation also provides more flexible data structures than previous work because our information-flow constraints make secrecy-preserving operations on such such structures possible. In future work, we plan to improve the expressiveness of the programming language through more sophisticated information-flow typing of recursive and iterative constructs, for example. In addition, we plan to apply our framework to other secure execution platforms that can provide stronger guarantees, such as security against active adversaries. We will also explore the possibility of proving formally that a particular implementation realizes our secret semantics, possibly in a mechanically-verified fashion. Finally, we plan to develop more sophisticated implementation techniques, possibly leveraging Template Haskell meta-programming, such as automatically producing code that is optimized for particular forms of partially homomorphic encryption with better performance.

**Acknowledgments** This work was supported by DARPA PROCEED, under contract #N00014-11-1-0276-P00002, the National Science Foundation, and the Air Force Office of Scientific Research. D. Stefan and J. Zimmerman are further supported by the Department of Defense (DoD) through the National Defense Science & Engineering Graduate Fellowship (NDSEG) Program.

## References

1 B. Applebaum, Y. Ishai, and E. Kushilevitz. From secrecy to soundness: Efficient verification via secure computation. In *ICALP (1)*, pages 152–163, 2010.
2 M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *STOC*, pages 1–10, 1988.
3 D. Bogdanov, S. Laur, and J. Willemson. Sharemind: A framework for fast privacy-preserving computations. In *ESORICS*, pages 192–206, 2008.

**4** P. Bogetoft, D. L. Christensen, I. Damgard, M. Geisler, T. Jakobsen, M. Krøigaard, J. D. Nielsen, J. B. Nielsen, K. Nielsen, J. Pagter, M. Schwartzbach, and T. Toft. Multiparty computation goes live. Cryptology ePrint Archive, Report 2008/068, 2008. `http://eprint.iacr.org/`.

**5** P. Bogetoft, I. B. Damgård, T. Jakobsen, K. Nielsen, J. Pagter, and T. Toft. Secure computing, economy, and trust: A generic solution for secure auctions with real-world applications. Technical Report RS-05-18, BRICS, 2005.

**6** D. Boneh, E.-J. Goh, and K. Nissim. Evaluating 2-dnf formulas on ciphertexts. In *TCC*, pages 325–341, 2005.

**7** R. Cramer, I. Damgård, and U. M. Maurer. General secure multi-party computation from any linear secret-sharing scheme. In *EUROCRYPT*, pages 316–334, 2000.

**8** I. Damgård, M. Geisler, M. Krøigaard, and J. B. Nielsen. Asynchronous multiparty computation: Theory and implementation. In *Public Key Cryptography*, pages 160–179, 2009.

**9** I. Damgård, Y. Ishai, and M. Krøigaard. Perfectly secure multiparty computation and the computational overhead of cryptography. In *EUROCRYPT*, pages 445–465, 2010.

**10** R. Gennaro, M. O. Rabin, and T. Rabin. Simplified vss and fact-track multiparty computations with applications to threshold cryptography. In *PODC*, pages 101–111, 1998.

**11** C. Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, pages 169–178, 2009.

**12** C. Gentry. Computing arbitrary functions of encrypted data. *Commun. ACM*, 53(3):97–105, 2010.

**13** C. Gentry and S. Halevi. Gentry-Halevi implementation of a fully-homomorphic encryption scheme. `https://researcher.ibm.com/researcher/files/us-shaih/fhe-code.zip`.

**14** C. Gentry and S. Halevi. Implementing gentry's fully-homomorphic encryption scheme. In *EUROCRYPT*, pages 129–148, 2011.

**15** C. Gentry, S. Halevi, and V. Vaikuntanathan. A simple bgn-type cryptosystem from lwe. In *EUROCRYPT*, pages 506–522, 2010.

**16** O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game or a completeness theorem for protocols with honest majority. In *STOC*, pages 218–229, 1987.

**17** Embedded domain-specific languages in haskell. `http://www.haskell.org/haskellwiki/Research_papers/Domain_specific_languages`.

**18** Y. Ishai and E. Kushilevitz. Randomizing polynomials: A new representation with applications to round-efficient secure computation. In *FOCS*, pages 294–304, 2000.

**19** Y. Ishai, E. Kushilevitz, and A. Paskin. Secure multiparty computation with minimal interaction. In *CRYPTO*, pages 577–594, 2010.

**20** Y. Ishai and A. Paskin. Evaluating branching programs on encrypted data. In *TCC*, pages 575–594, 2007.

**21** Y. Lindell and B. Pinkas. A proof of security of yao's protocol for two-party computation. *J. Cryptology*, 22(2):161–188, 2009.

**22** D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay - secure two-party computation system. In *USENIX Security Symposium*, pages 287–302, 2004.

**23** M. Naor and K. Nissim. Communication preserving protocols for secure function evaluation. In *STOC*, pages 590–599, 2001.

**24** J. D. Nielsen and M. I. Schwartzbach. A domain-specific programming language for secure multiparty computation. In *PLAS*, pages 21–30, 2007.

**25** T. Nishide and K. Ohta. Multiparty computation for interval, equality, and comparison without bit-decomposition protocol. In *Public Key Cryptography*, pages 343–360, 2007.

**26** A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE JOURNAL ON SELECTED AREAS IN COMMUNICATIONS*, 21(1):2003, 2003.

**27** M. C. Silaghi. Smc: Secure multiparty computation language. `http://www.cs.fit.edu/~msilaghi/SMC/tutorial.html`, 2004.

**28** N. P. Smart and F. Vercauteren. Fully homomorphic encryption with relatively small key and ciphertext sizes. In *Public Key Cryptography*, pages 420–443, 2010.

**29** M. van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan. Fully homomorphic encryption over the integers. In *EUROCRYPT*, pages 24–43, 2010.

**30** A. Yao. Protocols for secure computations (extended abstract). In *FOCS*, pages 160–164, 1982.