2012 IEEE 25th Computer Security Foundations Symposium

Information-flow control for programming on encrypted data

John C. Mitchell, Rahul Sharma, Deian Stefan, and Joe Zimmerman
Stanford University
Stanford, CA
{*mitchell, sharmar, deian, jzim*}@*cs.stanford.edu*

***Abstract*—Using homomorphic encryption and secure multiparty computation, cloud servers may perform regularly structured computation on encrypted data, without access to decryption keys. However, prior approaches for programming on encrypted data involve restrictive models such as boolean circuits, or standard languages that do not guarantee secure execution of all expressible programs. We present an expressive core language for secure cloud computing, with primitive types, conditionals, standard functional features, mutable state, and a secrecy preserving form of general recursion. This language, which uses an augmented information-flow type system to prevent control-flow leakage, allows programs to be developed and tested using conventional means, then exported to a variety of secure cloud execution platforms, dramatically reducing the amount of specialized knowledge needed to write secure code. We present a Haskell-based implementation and prove that cloud implementations based on secret sharing, homomorphic encryption, or other alternatives satisfying our general definition meet precise security requirements.**

***Keywords*-information flow control; secure cloud computing; domain-specific languages; homomorphic encryption; multiparty computation**

I. Introduction

Homomorphic encryption [1], [2], [3] and secure multiparty computation [4], [5], [6], [7] open new opportunities for secure cloud computing on encrypted data. For example, cloud servers could examine encrypted email for spam, without decrypting the email. A cloud server could also potentially compute a route between two endpoints on a public map, and return the encrypted path to a client. This paradigm provides cryptographic confidentiality, because the cloud server never has the keys needed to decrypt or recover private data.

Our goal is to provide a language and programming environment that would allow developers to produce secure cloud applications, without sophisticated knowledge of the cryptographic constructions used. Theoretical approaches for programming on encrypted data involve restrictive models such as boolean circuits, which are not conventional programming models. Programming using a conventional language and a cryptographic library, on the other hand, may allow programmers to write programs that cannot execute, because control flow depends on encrypted values that are not available to the cloud execution platform. Due to the performance costs of computing on encrypted data, realistic computation must involve mixtures of secret (encrypted) and public data. Without information flow restrictions, programs could inadvertently specify leakage of secret data into public variables.

We present an expressive core language for secure cloud computing, with primitive types, conditionals, standard functional features, mutable state, and a secrecy preserving form of general recursion. This language uses an augmented information-flow type system to impose conventional information-flow control and addresses a new problem for information-flow type systems: prevent control-flow leakage to the cloud platform. The language allows programs to be developed and tested using conventional means, then exported to a variety of secure cloud execution platforms, dramatically reducing the amount of specialized knowledge needed to write secure code. Past efforts have produced generally less expressive programming languages (SMC [8], Fairplay [9], SIMAP [10], [11], VIFF [12]) with weaker security proofs (see Section VII).

A core problem is that if an untrusted cloud server is given a program to execute, the server can observe control flow through the program. Therefore, if any conditional branch depends on a secret (encrypted) value, the server must execute both paths and combine results using operations on encrypted data. For example, consider `if` x `then` $y :=$ 4 `else` $y := 5$, where $x : \text{bool}$ and $y : \text{int}$. If x is secret and y is public, then this statement cannot be executed because secret information flows to a public reference; we use conventional information-flow control to prevent this. If both x and y are secret, then this is executed by storing the *secretly computed* value $x \cdot 4 + (1 - x) \cdot 5$ in y. While computing both branches works for a simple if-then-else, this cannot be done for recursive functions, because the set of possible execution paths may be infinite. Therefore, we augment our type system with additional information labels to prevent unbounded recursion on secret values, without restricting computation when control flow relies on public data.

While homomorphic encryption and secure multiparty computation are based on different cryptographic insights and constructions, there is a surprising structural similarity between them. This similarity is also shared by so-called *partially homomorphic* encryption, in which the homomorphism property holds only for certain operations. We capture this similarity in our definition of *secure execution platform.*

DOI 10.1109/CSF.2012.30

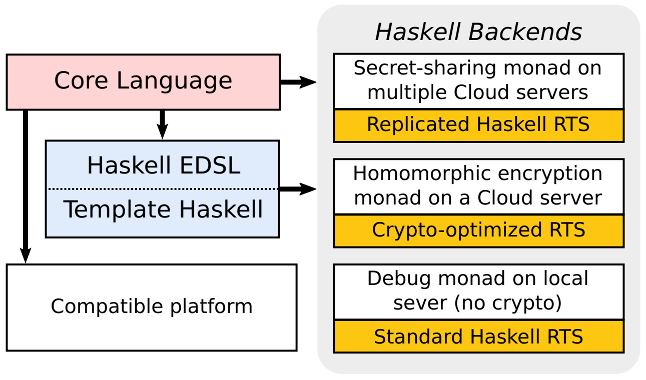

Figure 1. Multiple deployment options using different runtime systems. The EDSL (with the Template Haskell compiler) can be executed on various Haskell-backends. The core language compiler can additionally be compiled to other systems such as SMC and VIFF.

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. One advantage of this approach is that a developer may write conventional code and debug it using standard tools, without committing to a specific form of execution platform security. Another advantage is that as cryptographic constructions for various forms of homomorphic encryption improve, the same code can be retargeted because our language makes correctness independent of choice of secure execution platform.

Our formal definition of secure execution platform allows us to develop a single set of definitions, theorems, and proofs that are applicable to many cryptographic systems. The two main theoretical results are theorems that guarantee the correctness and security of program execution. We state correctness using a reference semantics that expresses the standard meaning of programs, with encryption and decryption as the identity function. The correctness theorem states the cloud execution of a program on encrypted data produces the same output as the conventional execution without encryption. Our security theorem depends on the threat model, because homomorphic encryption and secret sharing are secure in different ways. The security theorem states that no adversary learns the initial secret client values, beyond what is revealed by the program output, because the probability distributions of program behaviors (on different secret inputs) are indistinguishable. We also show that fully homomorphic encryption and a specific secret-sharing scheme meet the definition of secure execution platform, as do somewhat homomorphic schemes when they support the operations actually used in the program.

We develop our results using the honest-but-curious adversary model, commonly used in practical applications (e.g., [13], [10]). However, there are established methods for assuring integrity, using commitments and zero-knowledge techniques [14], as well as for reducing communication overhead [7]. In addition, while we focus on data confidentiality, our current system can also protect confidential algorithms, in principle, by considering code as input data to an interpreter (or "universal Turing machine").

In this article, we first give a motivating example and provide some relevant background on Haskell, our host language, and two representative examples of secure execution platforms: fully homomorphic encryption, and Shamir secret sharing. We then give an overview of the definitions and formal assumptions we require of any secure execution platform (Section IV), followed by a presentation of our core language and its properties (Section V), and a synopsis of our implementation efforts (Section VI). Finally, we discuss related work in Section VII and conclude in Section VIII.

II. Implementation and Motivating Example

The core of our domain-specific language (DSL) is implemented as a Haskell library, an embedded domain-specific language (EDSL). Our implementation includes Shamir secret sharing and fully homomorphic encryption; both use SSL network communication between clients and any number of servers. A preliminary design without recursion, references, or conditionals, and with a different implementation was described in [15].

As highlighted in Figure 1, we provide a Template Haskell compiler, which translates a subset of Haskell syntax to our EDSL, at compile-time. The Template Haskell extension provides syntactic sugar for EDSL combinators, enabling the programmer to use already-familiar syntax when writing code that operates on secret data. We also provide a core-language compiler front-end that directly implements the information-flow type system of this paper; the front-end exports abstract syntax trees (ASTs) of well-typed, labeled expressions, facilitating code generation for various backends, including our Haskell EDSL.

As a working example, we consider the case for email filtering in a cloud setting. More specifically, we consider RE: Reliable Email [16], as shown in Figure 2. With Reliable Email, email from trustworthy senders (as deemed by the receiver) is forwarded directly to the receiver's inbox, bypassing the spam filter. This guarantees that a message from a trustworthy source, which may have been labeled "spam" because of its content, will be received—hence, making email a *reliable* communication medium between trustworthy entities.

A key component of (a simplified) Reliable Email system is a whitelist, containing the email addresses of all senders considered trustworthy by the receiver. For every incoming message, the authentication of the `From` address is verified[1] and checked against the whitelist. If the address is in

[1]Messages must be signed by the sender since forging the `From` address is trivial.

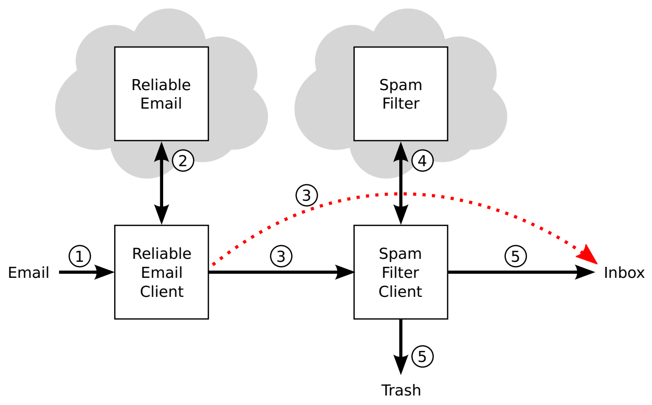

Figure 2. RE: Reliable Email in a cloud setting. Incoming email is forwarded by clients to cloud-based filters (steps 2 and 4). If email is from a trustworthy source as determined by Reliable Email (step 4), it is forwarded directly to the inbox (step 3), otherwise it takes the default path through the spam filter (steps 3-5).

```
-- | Given two secret, hashed email addresses,
-- compute the Hamming distance between them.
hammingDist :: SHA1EmailAddr
            → SHA1EmailAddr → SecIO SecretInt
hammingDist e1 e2 = hammingDist' e1 e2 0

-- | Number of bits in SHA1 digest
sha1DigestLen = 160

-- | Actual distance computation.
-- A SHA1EmailAddr is a list of bits.
hammingDist' e1 e2 = $(compileTHtoEDSL [|
  fix ( λf λi →
          -- Iterate over all bits of an email
      let k = if i < sha1DigestLen -- done?
                then f (i+1)       -- no, next
                else toSecret 0    -- yes
          -- Compute difference between i'th bits
          x = xor (e1 !! i) (e2 !! i)
      in x + k -- Sum of all bit differences
      )
  |])
```

Figure 3. Recursively computing the Hamming distance of two SHA1-hashed email addresses. Recursion with `fix` is used to iterate over all the bits of the hashed email addresses, which are XORed.

the whitelist, the mail is forwarded directly to the inbox, otherwise it is forwarded to the spam filter.

It is important that the sender's whitelist, which is essentially an address book, remain confidential if Reliable Email and the spam filter are executed in the cloud. Hence, in our setting, the hashed email addresses are encrypted (or split into shares, in the case of secret sharing) and the check against the whitelist is done homomorphically. Figure 3 shows a component of the whitelist check in our Haskell DSL: computing the Hamming distance between the hashes of two email addresses. A Hamming distance of zero denotes a match, i.e., the email address is that of a trustworthy entity.

This example highlights several key aspects of our Haskell EDSL. First, our language provides various primitives such as `fix` and `toSecret`, that are respectively used for implementing (safe) recursion, and lifting public values to their secret equivalents. Second, the DSL embedding allows the programmer to use existing Haskell features including higher-order functions, abstract data types, lists, etc. Third, the Template Haskell compiler (`compileTHtoEDSL`) allows the programmer to use standard Haskell syntax. Finally, compared to languages with similar goals (e.g., SMCL [11]), where a programmer is required to write separate client and server code, using our EDSL, a programmer needs to only write a single program; we eliminate the client/server code separation by providing a simple runtime system that directs all parties.

Developers who write in the EDSL can take advantage of existing, carefully-engineered Haskell development tools, compilers, and runtime systems. Programmers also have the benefit of sophisticated type-checking and general programming features of Haskell; we use the Haskell type and module system to enforce the correct usage of our secure execution platform. Because our EDSL and compilers are packaged in the form of Haskell libraries, other researchers could use our libraries to implement different programming paradigms over the same forms of cryptographic primitives, or compile our core language to other runtime systems.

III. Background

A. Haskell and EDSLs

Haskell is a purely functional language that is widely used for DSL embedding. Haskell's type system, lazy evaluation strategy (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 languages.

Haskell also supports compile-time meta-programming with Template Haskell [17]. Template Haskell provides a method for reifying Haskell source: converting concrete Haskell syntax to a Haskell data type representing the source abstract syntax tree (AST); and, dually, a method for splicing an AST into Haskell source. Figure 3 shows an example use case of compile-time meta-programming with Template Haskell. The recursive function is reified by enclosing it in '`[|`' brackets: `[| fix ( λf λi →... ) |]`. We use the function `compileTHtoEDSL` to analyze the corresponding AST and generate a new AST composed of EDSL primitives. Finally, the generated AST is spliced in as Haskell source by enclosing it with `$(...)`, providing a definition for `hammingDist'`. Our use of Template Haskell is limited to adding syntactic sugar to our EDSL, so that programmers can work with familiar constructs. An alternative extension, *QuasiQuotes*, can be used to reify arbitrary syntax; in conjunction with

Template Haskell, we can use QuasiQuotes to compile from the core language to our EDSL.

B. 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 encrypted data. More specifically, for a public/secret key pair $\langle \mathrm{pk}, \mathrm{sk} \rangle$ generated by KeyGen, and a plaintext m, if $c = \mathsf{Enc}(\mathrm{pk}, m)$, then $\mathsf{Dec}(\mathrm{sk}, \mathsf{Eval}(\mathrm{pk}, c, f)) = f(m)$ for every $f \in \mathcal{F}$, where $\mathcal{F}$ is some set of functions on plaintexts. We say that the encryption scheme is *homomorphic with respect to the set $\mathcal{F}$ of functions.*

While some homomorphic encryption schemes [18], [19], [20] are *partially homomorphic* – i.e., 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 constructions that are *fully homomorphic*, i.e., homomorphic with respect to all functions of polynomial complexity [1], [2], [21], [3]. 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.

C. Secure multiparty computation

Another approach to computing on ciphertexts makes use of generic two-party or multi-party secure computation [22], [23], [4], [24], [6], [7], [25], [26], [27], in which the client, who has the plaintext m, communicates with the server(s), who have the function f to be computed on m. The standard conditions for secure multiparty computation guarantee that the client learns $f(m)$, while the server (or an adversary compromising some restricted set of servers) learns nothing about m.

In this work, we specifically consider Shamir secret sharing, and the multiparty computation protocol based on it [5]. According to this protocol, a client C shares a secret value a_0 from a prime-order finite field $\mathbb{F}_p$ among N servers. In an (N, k) secret sharing scheme, N servers can jointly perform computations on m and other shared secrets, such that at least k of the N servers must collude to learn anything about m. Letting $a_0 = m$, in Shamir secret sharing, the client C shares 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 *shares* $s_1 = p(1), \ldots, s_N = p(N)$ to servers $S_1, \ldots, S_N$, respectively. The servers can easily execute additions on the values, by adding their shares. Multiplication is more complicated, because multiplication of polynomials increases their degree. The solution involves computing and communicating a new sharing among the servers.

IV. Definitions and assumptions

Before presenting our language design, we summarize the semantic structure used in our analysis; for full details, we refer to reader to Appendix A. As shown below, our semantic structure is sufficient to prove correctness and security theorems for the language we develop, and general enough to encompass secret sharing, homomorphic encryption, and other platforms.

A. Primitive operations

We assume some given sets Y of primitive values, including at least booleans ($Y = \{\mathtt{true}, \mathtt{false}\}$) and the singleton set, "unit" ($Y = \{()\}$). We also assume primitive operations $\mathrm{op}_1, \ldots, \mathrm{op}_r$ on these values, where each operation has its own type: i.e., $\mathrm{op}_i : \mathrm{dom}(\mathrm{op}_i) \to \mathrm{cod}(\mathrm{op}_i)$. (For instance, if op_1 is addition modulo p, then $\mathrm{dom}(\mathrm{op}_1) = (\mathrm{int}, \mathrm{int})$, and $\mathrm{cod}(\mathrm{op}_1) = \mathrm{int}$.) To provide additional flexibility for richer language features, we also assume that for each primitive type Y, we have a primitive branching operator: i.e., there is some $\mathrm{op}_{\mathrm{Br}(Y)}$ such that $\mathrm{op}_{\mathrm{Br}(Y)}(\mathtt{true}, y_1, y_2) = y_1$ and $\mathrm{op}_{\mathrm{Br}(Y)}(\mathtt{false}, y_1, y_2) = y_2$. (In platforms with addition and multiplication, for instance, we might have $\mathrm{op}_{\mathrm{Br}(\mathrm{int})}(b, z_1, z_2) = b \cdot z_1 + (1 - b) \cdot z_2$.) In reality, some platforms might only support branching operators for a subset of primitive types, or none at all (e.g., cryptosystems that are only additively or multiplicatively, rather than fully, homomorphic). However, to simplify the formalism, we assume branching operators for all primitive types; the development proceeds in a similar fashion for simpler platforms, and only requires that additional restrictions be imposed on branching constructs (`if-then-else`).

B. Distributed computing infrastructure

We assume N servers, $S_1, \ldots, S_N$, execute the secure computation on behalf of one client, C.[2] (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 in discrete *communication rounds*. A *communication trace* $T \in \mathcal{T}$ is a sequence of communication rounds. Each round is a tuple $(S, R, m) \in T$ of the sender S, receiver R, and value m. We assume a projection operator, Π_A, that gives the view of a trace for any subset A of the servers, i.e., the messages that those servers see; intuitively, this will represent the view of the protocol by any set of compromised servers that fall under the protocol's threat model.

General form of cryptographic primitives: We work with a two-element security lattice, $\mathrm{P} \sqsubseteq \mathrm{S}$, representing (respectively) "public" values, which are transmitted in the clear and may be revealed to any party; and "secret" values,

[2] The restriction to a single client is for clarity of presentation. The generalization to multiple clients is straightforward, and, indeed, it is common in practice to have N parties in total executing a multiparty computation, each one serving as both a client and a server.

which are encrypted or otherwise hidden, and must remain completely unknown to the adversary. For each primitive type Y, we assume a set $\mathcal{E}_{\mathrm{S}}(Y)$, holding "secret equivalents" of base values in Y; for notational uniformity, we also define $\mathcal{E}_{\mathrm{P}}(Y) = Y$, signifying that the "public equivalent" of a value is just the value itself. We recall that for any two elements (or *labels*) of a lattice, we have a well-defined *join* ($\sqcup$), which corresponds to the *least upper bound* of the two elements (e.g., $\mathrm{P} \sqcup \mathrm{S} = \mathrm{S}$).

We also assume a few standard cryptographic primitives, expressed as *protocol operations* that may operate on initial parameters $\iota \in \mathcal{I}$, generate communication traces among the parties, and/or consume bits from a source of randomness. For clarity, we leave this randomness source implicit, instead considering each operation to produce a distribution over the values in its range (and implicitly lifting the operations to act on distributions over their domains). We regard predicates over these distributions to be true if they hold with probability 1.

The operations we assume are as follows (overloaded for all primitive types Y):

- $\mathrm{Enc}_{\mathrm{S}} : Y \times \mathcal{I} \to \mathcal{E}_{\mathrm{S}}(Y) \times \mathcal{T}$, "hiding" $y \in Y$.
- $\mathrm{Dec}_{\mathrm{S}} : \mathcal{E}_{\mathrm{S}}(Y) \times \mathcal{I} \to Y \times \mathcal{T}$, "unhiding" $\tilde{y} \in \mathcal{E}_{\mathrm{S}}(Y)$.
- $\mathrm{Enc}_{\ell_1,\ldots,\ell_r}(\mathrm{op}_i) : \prod_j \mathcal{E}_{\ell_j}(\mathrm{dom}(\mathrm{op}_i)_j) \times \mathcal{I} \to \mathcal{E}_{\bigsqcup_j \ell_j}(\mathrm{cod}(\mathrm{op}_i)) \times \mathcal{T}$ (when at least one ℓ_j is S), evaluating a primitive operation.

We also assume that Init describes the generation of initial parameters according to some distribution $\mathcal{I}$ (for example, public and secret keys in the case of homomorphic encryption). For notational uniformity, as above, we also define the corresponding operations in the degenerate case of "hiding" public values (operating as the identity on the plaintext values, and yielding empty traces). In addition, we assume a projection operator from the initial parameters onto any subset A of the servers, writing $\Pi_A(\iota)$ to mean, intuitively, the portion of the initial parameters ι that servers in A should receive.

Cryptographic correctness assumptions: We assume the usual encryption and homomorphism conditions, augmented for cryptographic primitives that depend on randomness and that may communicate among servers to produce their result. For every element y of a primitive type Y, and every choice of initial parameters $\iota \in \mathcal{I}$, we assume a family of *safe distributions* $\hat{\mathcal{E}}^{\iota}_{\ell}(y)$: intuitively, any distribution $l \in \hat{\mathcal{E}}^{\iota}_{\ell}(y)$ can safely serve as the "hiding" of y under the initial parameters ι (at secrecy level $\ell \in \{\mathrm{P}, \mathrm{S}\}$). We require that "hiding" a base value must yield a safe distribution; that unhiding ("decryption") is the left-inverse of hiding ("encryption"); and that hiding commutes homomorphically with the primitive operations.

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

Different platforms may provide different security guarantees of their primitives. For example, protocols may specify that distributions are *computationally indistinguishable* (i.e., indistinguishable to a probabilistic polynomial-time adversary), or *information-theoretically indistinguishable* (i.e., identical). For the purposes of this development, we will use the term *indistinguishable* to refer to whichever of the above notions is specified by the secure execution platform. Using this terminology, we require that any two sequences of partial traces are indistinguishable if each pair of corresponding partial traces describes either 1.) a "hiding" operation; 2.) a primitive operation whose public arguments agree (and whose hidden arguments are safely-distributed); or 3.) an "unhiding" operation on values that turn out to be equal.

Definition 1. We say that a system is a *secure execution platform* for the primitive operations (op_i) if it satisfies the conditions summarized above (and detailed in Appendix A).

V. Language design

We present a functional core language, $\lambda^{\rightarrow}_{\mathrm{P,S}}$, whose definition is parameterized by a set of given operations over primitive types. This language is an extension of the simply-typed lambda calculus, with labeled types as used in information flow languages (see, e.g., [28]). Our language design and analysis are valid for any secure execution platform, and are thus parameterized over implementation details such as the number of servers, the form of cryptography used, and the form and extent of communication in the system. From the programmer's standpoint, different cryptographic backends that support the same operations provide the same programming experience.

In order to prove desired correctness and security properties, we formulate both a standard *reference semantics* for $\lambda^{\rightarrow}_{\mathrm{P,S}}$ 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 the distributed semantics is then proved by showing an equivalence with the reference semantics, while security properties are proved by analyzing the information available to the servers throughout the program execution.

A. Syntax

Our core language, $\lambda^{\rightarrow}_{\mathrm{P,S}}$, extends the simply-typed lambda calculus with primitive values, conditionals, mutable stores, and a fixpoint operator. Figure 4 describes the language syntax. Throughout this section, we assume primitive operations (op_i) and a secure execution platform, as specified in Section IV.

Figure 4 $\lambda^{\rightarrow}_{\mathrm{P,S}}$ syntax.

Types	$t ::= Y \mid (\tau \to \tau) \mid Y^{\ell}$ ref
Labeled types	$\tau ::= t^{\ell}$
Values	$v ::= y \mid a \mid \lambda x.e \mid \texttt{fix}\ f.\lambda x.e$
Expressions	$e ::= v \mid x \mid X \mid e\ e$
	$\mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e$
	$\mid \texttt{op}^{t}_{i}(e, \ldots, e) \mid \texttt{ref}\ e \mid !e \mid e := e$
	$\mid \texttt{reveal}\ e$
Programs	$p ::= \texttt{read}(X_1 : Y_1, \ldots, X_n : Y_n); e$

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 lowercase letters x), to emphasize the phase distinction between input processing and evaluation of the program body[3]. 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 remainder of the computation. In addition to decrypting final result values (if they are intended to be public), the `reveal` construct also enables declassification of intermediate results, giving programmers fine-grained control over the tradeoff between performance and total secrecy. We note that references in $\lambda^{\rightarrow}_{\mathrm{P,S}}$ are limited to primitive types, since we later depend on some restricted termination results (and termination need not hold if references of arbitrary type are permitted).

Figure 5 $\lambda^{\rightarrow}_{\mathrm{P,S}}$ syntax (extended).

$$\begin{aligned}
\tilde{v} &::= v \mid \tilde{y} \mid \lambda x.\tilde{e} \mid \texttt{fix}\ f.\lambda x.\tilde{e} \mid a \mid \varphi(\tilde{v}_1, \tilde{v}_2, \tilde{v}_3) \\
\tilde{e} &::= e \mid \tilde{v} \mid x \mid X \mid \tilde{e}\ \tilde{e} \mid \texttt{if}\ \tilde{e}\ \texttt{then}\ \tilde{e}\ \texttt{else}\ \tilde{e} \\
&\quad \mid \texttt{op}^{t}_{i}(\tilde{e}, \ldots \tilde{e}) \mid \texttt{ref}\ \tilde{e} \mid !\tilde{e} \mid \tilde{e} := \tilde{e} \\
&\quad \mid \texttt{reveal}\ \tilde{e}
\end{aligned}$$

In order to reason about the evaluation of programs we extend the language syntax as shown in Figure 5. The previous syntax (Figure 4), is a subset of the extended syntax, and encompasses all expressions that the programmer can write (which we refer to as the "surface syntax"), as well as values a, which range over a countably infinite set of abstract memory locations (as in standard presentations of lambda calculus with mutable references). However, the extensions are necessary to describe values that may result from evaluations in the distributed semantics (described below), despite not being present at the surface level.

In particular, we have a case for possibly-hidden primitive values $\tilde{y} \in \mathcal{E}_{\ell}(Y)$. As described in Section IV, $\tilde{y} \in \mathcal{E}_{\mathrm{S}}(Y)$ is a hidden value, while $\tilde{y} \in \mathcal{E}_{\mathrm{P}}(Y) = Y$ is a publicly-visible value but, for notational uniformity, may be regarded as hidden at the public confidentiality level. The value $\varphi(\tilde{b}, \tilde{v}_2, \tilde{v}_3)$, where $\tilde{b} \in \mathcal{E}_{\mathrm{S}}(\mathrm{bool})$, and $\tilde{v}_2$, $\tilde{v}_3$ are (extended) values intuitively represents a "deferred decision". Because the boolean value $\tilde{b}$ was secret, the system could not decide which of two values was the result of a conditional expression, and thus had to propagate both. For example, the following expression:

$$\varphi(\tilde{b}, \lambda x.0, \lambda x.1)$$

might be the result of a conditional in which the condition evaluates to the secret boolean $\tilde{b}$. Note, however, that a value such as $\varphi(\tilde{b}, 17, 42)$ would never occur, since (as detailed below) the system would be able to evaluate the condition homomorphically on hidden primitive values and produce a hidden primitive value.

B. Static semantics

We include a few representative rules of the static semantics in Figure 6. For the full presentation, we refer the reader to the extended version of this work, available at [29].

Figure 6 Static semantics (selected rules).

$$\frac{\Gamma[x \mapsto \tau_1], \Sigma, C' \vdash e : \tau}{\Gamma, \Sigma, C \vdash \lambda x.\tilde{e} : (\tau_1 \xrightarrow{C'} \tau)^{\ell}}$$

$$\frac{\Gamma[f \mapsto (\tau_1 \xrightarrow{\mathrm{P}} \tau)^{\ell}, x \mapsto \tau_1], \Sigma, C' \vdash \tilde{e} : \tau}{\Gamma, \Sigma, C \vdash \texttt{fix}\ f.\lambda x.\tilde{e} : (\tau_1 \xrightarrow{\mathrm{P}} \tau)^{\ell}}$$

$$\frac{\begin{array}{c}\Gamma, \Sigma, C \vdash \tilde{e} : (\tau_1 \xrightarrow{C'} t^{\ell'})^{\ell} \\ \Gamma, \Sigma, C \vdash \tilde{e}_1 : \tau_1 \qquad \ell \sqsubseteq C' \qquad C \sqsubseteq C' \qquad \ell \sqcup \ell' \sqsubseteq \ell''\end{array}}{\Gamma, \Sigma, C \vdash \tilde{e}\,\tilde{e}_1 : t^{\ell''}}$$

$$\frac{\begin{array}{c}\Gamma, \Sigma, C \vdash \tilde{e}_1 : \mathrm{bool}^{\ell'} \qquad \Gamma, \Sigma, C \sqcup \ell' \vdash \tilde{e}_2 : t^{\ell''} \\ \Gamma, \Sigma, C \sqcup \ell' \vdash \tilde{e}_3 : t^{\ell''} \qquad \ell' \sqcup \ell'' \sqsubseteq \ell\end{array}}{\Gamma, \Sigma, C \vdash \texttt{if}\ \tilde{e}_1\ \texttt{then}\ \tilde{e}_2\ \texttt{else}\ \tilde{e}_3 : t^{\ell}}$$

$$\frac{\begin{array}{c}\Gamma, \Sigma, C \vdash \tilde{e}_1 : (t^{\ell'}\ \mathrm{ref})^{\ell} \\ \Gamma, \Sigma, C \vdash \tilde{e}_2 : t^{\ell'} \qquad C \sqsubseteq \ell' \qquad \ell \sqsubseteq \ell'\end{array}}{\Gamma, \Sigma, C \vdash \tilde{e}_1 := \tilde{e}_2 : \mathrm{unit}^{\ell''}}$$

$$\frac{\Gamma, \Sigma, C \vdash \tilde{e} : Y^{\mathrm{S}}}{\Gamma, \Sigma, \mathrm{P} \vdash \texttt{reveal}\ \tilde{e} : Y^{\ell}}$$

In the typing judgment, Γ, as usual, represents the typing context for lambda-bound variables, while Σ represents the store typing (i.e., if $\Sigma(a) = Y^{\ell}$, then the (extended)

[3]While we enforce this phase distinction to simplify the formalism, it is a straightforward extension to allow input and output operations throughout the program, rather than restricting inputs to the beginning and outputs to the final value.

expression $\tilde{e}$ should be evaluated in a store that maps a to an element of type Y^ℓ).

The key feature of the static semantics is the presence of value labels, $\ell \in \{\mathrm{P}, \mathrm{S}\}$, as well as the context label, $C \in \{\mathrm{P}, \mathrm{S}\}$. The intuitive meaning of these labels is similar to their meaning in standard information flow systems [28], [30]. A value labeled t^ℓ is a value of type t at confidentiality level ℓ. Our `reveal` operator, like traditional declassification operators, indicates that a particular value is allowed to be leaked; statically, it acts as a cast from S to P. A context label C signifies that an expression can be typed in a context in which control flow may depend on values of confidentiality level C. In our language, as in standard information flow systems, this context restriction is used to prevent implicit flows, such as an update to a public memory location inside a secret conditional. However, in our model, we must also regard any deviation in control flow as a publicly visible effect. Thus, not only updates to public memory locations, but also side-effects such as `reveal`, as well as unbounded iteration (or potential nontermination), must be independent of secret values. This makes our system, by necessity, strictly more restrictive than standard termination-insensitive information-flow systems such as JFlow/Jif [31].

One can view these additional restrictions as specifying that the control flow of the program is visible to the adversary, as in an oblivious memory model [32]; but, while intuitively helpful, this analogy does not present a complete picture. In a sense, the purpose of information flow control in our system is dual to its purpose in traditional language-based security: in a traditional system, the machine model permits the implementation of arbitrary operations, and the information flow system must statically rule out code that would leak information; while in our system, the machine model permits only operations that would not leak information (since secret values are encrypted or otherwise hidden), and thus our system must statically rule out code that would not be implementable in such a model.

In light of these objectives, we include additional restrictions in the static semantics, similar in flavor to some type-and-effect systems [33], [34], in addition to standard information flow control constructs. For example, in the conditional rule, we require that each branch is well-typed at a confidentiality level at least as high as that of the condition. As in other information flow systems, this restriction rules out the canonical example of implicit flow (where s is a secret boolean, and p is a reference to a public integer):

$$\texttt{if } s \texttt{ then } p := 0 \texttt{ else } p := 1$$

Since s is secret, the branches must type-check in a secret context; but the rule for assignment specifies that the label of the reference receiving the assignment is at least as high as that of the surrounding context, which cannot be satisfied by this expression. Our system also rules out the following expression:

$$\texttt{if } s_1 \texttt{ then } (\texttt{reveal } s_2) \texttt{ else } 17$$

The `reveal` operation would cause a publicly-observable side-effect (namely, causing a value to be "unhidden"), and thus it cannot be typed in a secret context. Similarly, our restrictions also rule out the following expression, although the reasoning is more involved:

$$\texttt{fix } f \,.\, \lambda s \,.\, (\texttt{if } s \leq 0 \texttt{ then } 1 \texttt{ else } s * f(s-1))$$

Notably, in the rules for lambda abstraction and recursive function definition, we add a context label above the arrow ($\tau_1 \overset{C'}{\rightarrow} \tau$), signifying that the resulting function, when applied, may generate effects that must be confined to a context C' (independent of the context C in which the term itself is evaluated).[4] In the case of general recursion, this effect label is assumed to be P, since we conservatively assume that any application of a function defined by general recursion (either within its own definition, or as a standalone expression) may generate unpredictable, publicly-observable control flow behavior, including divergence. It is also worth noting that a function's effect context label, C', is only taken into account when the function is applied; for instance, the following expression is well-typed:

$$\texttt{if } s \texttt{ then } (\texttt{fix } f \,.\, \lambda x \,.\, f\ x) \texttt{ else } (\lambda x \,.\, x)$$

The dependence of confidentiality on control flow also raises more subtle issues that necessitate additional typing restrictions. For instance, the following expression clearly does not preserve confidentiality (and thus cannot be implemented on a secure execution platform):

$$(\texttt{if } s \texttt{ then } p_1 \texttt{ else } p_2) := 42$$

The underlying principle in this example is that values escaping a secret context may carry secret information with them, and thus computations that depend on them must be well-typed in a secret context. Indeed, this restriction is reflected in the static semantics. In the assignment rule, the restriction $\ell \sqsubseteq \ell'$ expresses that information could flow from the confidentiality label of the reference itself to the label of its contents: i.e., one can never write to a secret reference cell (a memory location whose identity is secret) if its contents may be public. A similar situation arises with lambda abstractions, as in the following (ill-typed) example:

$$(\texttt{if } s_1 \texttt{ then } (\lambda x \,.\, \texttt{reveal } s_2) \texttt{ else } (\lambda x \,.\, 17))\ ()$$

In the application rule, as above, the restriction $\ell \sqsubseteq C'$ expresses that information could flow from the identity of

[4]For simplicity of presentation, lambda abstractions in our syntax do not carry the traditional type annotations, $\lambda(x : t).e$. In any concrete implementation, we assume that terms are appropriately annotated with (unlabeled) types, so that type-checking becomes tractable while preserving label subtype polymorphism. Since such restrictions permit the typing of strictly fewer terms, our theoretical results still hold.

the function to the context in which it executes: i.e., one can never apply a function whose identity is secret if it might have publicly-observable effects.

C. Dynamic semantics

We now give a standard dynamic semantics for $\lambda^{\rightarrow}_{\mathrm{P,S}}$, the "reference semantics" (Figure 7), extending the evaluation rules for the simply-typed lambda calculus. As described above, the reference semantics gives a precise specification for the evaluation of a program, against which the actual secure implementation (the "distributed semantics") can be compared for correctness.

Figure 7 Reference semantics (selected rules).

$$\frac{(e_0, \mu) \downarrow (\lambda x.e_0', \mu_0, \mathcal{O}_0) \quad (e_1, \mu_0) \downarrow (v_1, \mu_1, \mathcal{O}_1) \qquad (e[v_1/x], \mu_1) \downarrow (v', \mu', \mathcal{O}_2)}{(e_0\, e_1, \mu) \downarrow (v', \mu', \mathcal{O}_0 \| \mathcal{O}_1 \| \mathcal{O}_2)}$$

$$\frac{(e_1, \mu) \downarrow (\mathtt{true}, \mu_1, \mathcal{O}_1) \qquad (e_2, \mu_1) \downarrow (v', \mu', \mathcal{O}_2)}{(\mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3, \mu) \downarrow (v', \mu', \mathcal{O}_1 \| \mathcal{O}_2)}$$

$$\frac{(e_1, \mu) \downarrow (a, \mu_1, \mathcal{O}_1) \qquad (e_2, \mu_1) \downarrow (v_2, \mu_2, \mathcal{O}_2)}{(e_1 := e_2, \mu) \downarrow ((), \mu_2[a \mapsto v_2], \mathcal{O}_1 \| \mathcal{O}_2)}$$

$$\frac{(e, \mu) \downarrow (y, \mu', \mathcal{O})}{(\mathtt{reveal}\ e, \mu) \downarrow (y, \mu', \mathcal{O} \| y)}$$

$$\frac{(e[\kappa(X_1)/X_1 \ldots \kappa(X_n)/X_n], \emptyset) \downarrow (v, \mu, \mathcal{O})}{(\kappa, \mathtt{read}(X_1 : Y_1, \ldots, X_n : Y_n); e) \downarrow (v, \mu, \mathcal{O})}$$

In the reference semantics, the mutable store μ maps addresses (generated by `ref`) to the values they contain, while the environment κ represents the initial (secret) values supplied by the client. The judgment $(\mu, e) \downarrow (v, \mu', \mathcal{O})$ indicates that the expression e, when evaluated in an initial store μ, produces a value v, a final store μ', and a sequence $\mathcal{O}$ of "observations", holding all values ever supplied to `reveal` throughout the evaluation (where the operator $\|$ indicates concatenation of observation sequences). The observation sequences are important in proving security properties (Theorem 2, below), as we will show that an appropriately constrained adversary learns nothing except what is logically entailed by these observations. The `read` construct only serves to bind the initial client-input variables, and is essentially a no-op; for clarity, however, we retain it in the syntax, since in the distributed semantics (Figure 8), it will represent the initial "hiding" operation (potentially including communication between the client and servers).

We give an additional dynamic semantics, the "distributed semantics" (Figure 8), that reflects the actual steps taken by an implementation in terms of a secure execution platform. Values in the distributed semantics may be "hidden" from the server(s) computing on them – shared or otherwise encrypted, according to the primitives of the secure execution platform. Thus, the distributed semantics makes central use of the lifted, or homomorphic, operations of the platform. In particular, we use the lifted primitives, $\mathrm{Enc}_{\ell_1,\ldots,\ell_r}(\mathrm{op}_i)$, to match the execution of the ordinary primitive operations op_i in the reference semantics. It is also worth noting that the distributed semantics implicitly act on probability distributions. As described above, the output of an operation in the secure execution platform is a distribution, and thus, when an operation $f(\tilde{x})$ appears in a semantic rule, it should be considered lifted to act on distributions (i.e., $f(\tilde{x})$ represents the sum of distributions $\sum_{x \in \mathrm{dom}\,\tilde{x}} p_{\tilde{x}}(x) f(x)$, where $p_{\tilde{x}}$ is a probability mass function of the distribution $\tilde{x}$).

The distributed semantics uses the extended form of the expression syntax ($\tilde{e}$), signifying that expressions may contain possibly-hidden primitive values $\tilde{y} \in \mathcal{E}_\ell(Y)$, as well as φ symbols $\varphi(\tilde{b}, \tilde{v}_2, \tilde{v}_3)$ (discussed below). The evaluation judgment $(\tilde{e}, \tilde{\mu}, \iota) \Downarrow (\tilde{v}', \tilde{\mu}', T, \mathcal{O})$ signifies that an expression $\tilde{e}$, when evaluated in an initial store $\tilde{\mu}$ and initialized with platform parameters ι, evaluates to the value $\tilde{v}'$, producing a final store $\tilde{\mu}'$, a communication trace T, and observations $\mathcal{O}$ (again containing all values *ever* supplied to the `reveal` operator). In contrast to the reference semantics, the distributed semantics specifies that `reveal` and `read` operations result in communication between the parties according to the parameters of the secure execution platform: the `reveal` rule specifies that the parties execute an "unhiding", or decryption, $\mathsf{Dec}_{\mathrm{S}}$, of a secret value, while the `read` rule specifies that the client initializes the secure execution platform, distributes to each server its view of the initial parameters ($\Pi_{\{S_i\}}(\iota)$), and executes the "hiding", $\mathsf{Enc}_{\mathrm{S}}$, of the secret values in the client's initial environment κ.

For most of the other rules, we can intuitively regard evaluation in the distributed semantics as proceeding in lock-step with the reference semantics, executing the corresponding operations as provided by the secure execution platform, as long as control flow proceeds independently of secret values. For example, the distributed semantics provides two distinct rules for conditionals (`if-then-else`). When the condition evaluates to a public boolean (i.e., an element of $\mathcal{E}_{\mathrm{P}}(\mathrm{bool}) = \mathrm{bool}$), the evaluation precisely mirrors the reference semantics; observations and communication traces are propagated unchanged.

When control flow does depend on a secret value, however, the distributed semantics yields different behavior. For instance, when a condition evaluates to a secret boolean $\tilde{b}$, the distributed semantics specifies that both branches should be evaluated in the same store $\tilde{\mu}_1$, each generating its own value and resulting store $(\tilde{v}_2, \tilde{\mu}_2)$ and $(\tilde{v}_3, \tilde{\mu}_3)$.[5] The distributed semantics then merges these values and stores,

[5]For technical reasons, we need to specify that $\mathrm{dom}\,\tilde{\mu}_2 \cap \mathrm{dom}\,\tilde{\mu}_3 = \mathrm{dom}\,\tilde{\mu}_1$ to prevent incidental collisions in names of fresh memory locations.

Figure 8 Distributed semantics (selected rules).

$$\frac{\begin{array}{c}\forall j \in \{1,\dots,r\} \,.\, (e_j, \tilde{\mu}_{j-1}, \iota) \Downarrow (\tilde{y}_j, \tilde{\mu}_j, T_j, \mathcal{O}_j) \\ \forall j \in \{1,\dots,r\} \,.\, \tilde{y}_j \in \mathcal{E}_{\ell_j}(Y_j) \\ \mathrm{Enc}_{\ell_1,\dots,\ell_r}(\mathrm{op}_i)(\tilde{y}_1,\dots,\tilde{y}_r,\iota) = (\tilde{y}', T) \\ T' = T_1 \| \cdots \| T_r \| T \qquad \mathcal{O}' = \mathcal{O}_1 \| \cdots \| \mathcal{O}_r\end{array}}{(\mathtt{op}_i(e_1,\dots,e_r), \tilde{\mu}_0, \iota) \Downarrow (\tilde{y}', \tilde{\mu}_r, T', \mathcal{O}')}$$

$$\frac{\begin{array}{c}(\tilde{e}_0, \tilde{\mu}, \iota) \Downarrow (\varphi(\tilde{b}, \tilde{v}_2, \tilde{v}_3), \tilde{\mu}_0, T_0, \mathcal{O}_0) \\ \tilde{b} \in \mathcal{E}_{\mathrm{S}}(\mathrm{bool}) \qquad (\tilde{e}_1, \tilde{\mu}_0, \iota) \Downarrow (\tilde{v}_1, \tilde{\mu}_1, T_1, \mathcal{O}_1) \\ (\tilde{v}_2\ \tilde{v}_1, \tilde{\mu}_1, \iota) \Downarrow (\tilde{v}_2', \tilde{\mu}_2, T_2, \mathcal{O}_2) \\ (\tilde{v}_3\ \tilde{v}_1, \tilde{\mu}_1, \iota) \Downarrow (\tilde{v}_3', \tilde{\mu}_3, T_3, \mathcal{O}_3) \\ (\tilde{v}', T_4) = \Phi^\iota(\tilde{b}, \tilde{v}_2', \tilde{v}_3') \qquad (\tilde{\mu}', T_5) = \Phi^\iota(\tilde{b}, \tilde{\mu}_2, \tilde{\mu}_3)\end{array}}{(\tilde{e}_0\ \tilde{e}_1, \tilde{\mu}, \iota) \Downarrow (\tilde{v}', \tilde{\mu}', T_0 \| \cdots \| T_5, \mathcal{O}_0 \| \cdots \| \mathcal{O}_3)}$$

$$\frac{(\tilde{e}, \tilde{\mu}, \iota) \Downarrow (\tilde{y}, \tilde{\mu}', T_1, \mathcal{O}) \qquad (\tilde{y}', T_2) = \mathrm{Dec}_{\mathrm{S}}(\tilde{y}, \iota)}{(\mathtt{reveal}\ \tilde{e}, \tilde{\mu}, \iota) \Downarrow (\tilde{y}', \tilde{\mu}', T_1 \| T_2, \mathcal{O} \| \tilde{y}')}$$

$$\frac{\begin{array}{c}(\tilde{e}_1, \tilde{\mu}, \iota) \Downarrow (\mathtt{true}, \tilde{\mu}_1, T_1, \mathcal{O}_1) \\ (\tilde{e}_2, \tilde{\mu}_1, \iota) \Downarrow (\tilde{v}', \tilde{\mu}', T_2, \mathcal{O}_2) \\ T = T_1 \| T_2 \qquad \mathcal{O} = \mathcal{O}_1 \| \mathcal{O}_2\end{array}}{(\mathtt{if}\ \tilde{e}_1\ \mathtt{then}\ \tilde{e}_2\ \mathtt{else}\ \tilde{e}_3, \tilde{\mu}, \iota) \Downarrow (\tilde{v}', \tilde{\mu}', T, \mathcal{O})}$$

$$\frac{\begin{array}{c}\tilde{b} \in \mathcal{E}^\iota_{\mathrm{S}}(\mathrm{bool}) \\ (\tilde{e}_1, \tilde{\mu}, \iota) \Downarrow (\tilde{b}, \tilde{\mu}_1, T_1, \mathcal{O}_1) \\ (\tilde{e}_2, \tilde{\mu}_1, \iota) \Downarrow (\tilde{v}_2, \tilde{\mu}_2, T_2, \mathcal{O}_2) \\ (\tilde{e}_3, \tilde{\mu}_1, \iota) \Downarrow (\tilde{v}_3, \tilde{\mu}_3, T_3, \mathcal{O}_3) \\ \mathrm{dom}\,\tilde{\mu}_2 \cap \mathrm{dom}\,\tilde{\mu}_3 = \mathrm{dom}\,\tilde{\mu}_1 \\ (\tilde{v}', T_4) = \Phi^\iota(\tilde{b}, \tilde{v}_2, \tilde{v}_3) \qquad (\tilde{\mu}', T_5) = \Phi^\iota(\tilde{b}, \tilde{\mu}_2, \tilde{\mu}_3) \\ T = T_1 \| \cdots \| T_5 \qquad \mathcal{O} = \mathcal{O}_1 \| \mathcal{O}_2 \| \mathcal{O}_3\end{array}}{(\mathtt{if}\ \tilde{e}_1\ \mathtt{then}\ \tilde{e}_2\ \mathtt{else}\ \tilde{e}_3, \tilde{\mu}, \iota) \Downarrow (\tilde{v}', \tilde{\mu}', T, \mathcal{O})}$$

$$\frac{\begin{array}{c}(\tilde{e}_1, \tilde{\mu}, \iota) \Downarrow (\tilde{v}_1, \tilde{\mu}_1, T_1, \mathcal{O}_1) \\ (\tilde{e}_2, \tilde{\mu}_1, \iota) \Downarrow (\tilde{v}, \tilde{\mu}_2, T_2, \mathcal{O}_2) \\ (\tilde{\mu}', T') = \mathrm{update}^\iota(\tilde{\mu}_2, \tilde{v}_1, \tilde{v}_2)\end{array}}{(\tilde{e}_1 := \tilde{e}_2, \tilde{\mu}, \iota) \Downarrow ((), \tilde{\mu}', T_1 \| T_2 \| T', \mathcal{O}_1 \| \mathcal{O}_2)}$$

$$\frac{\begin{array}{c}\iota = \mathrm{Init}() \\ T_0 = \{(C, S_i, \Pi_{\{S_i\}}(\iota)) : 1 \le i \le N\} \\ \forall j \in \{1,\dots,r\}.(\tilde{v}_j, T_j') = \mathrm{Enc}_{\mathrm{S}}(\kappa(X_j), \iota) \\ (e[\tilde{v}_1/X_1 \dots \tilde{v}_r/X_r], \emptyset, \iota) \Downarrow (\tilde{v}', \tilde{\mu}', T', \mathcal{O}) \\ T = T_0 \| T_1' \| \dots \| T_r' \| T'\end{array}}{(\kappa, \mathtt{read}(X_1 : Y_1, \dots X_r : Y_r); e) \Downarrow (\tilde{v}', \tilde{\mu}', T, \mathcal{O})}$$

according to the secret boolean $\tilde{b}$, using the Φ^ι function (Figure 9). For example, if $\tilde{v}_2$ and $\tilde{v}_3$ are (hidden) primitive values representing, respectively, 17 and 42, and the boolean $\tilde{b}$ is a (hidden) representation of `true`, then the result:

$$\Phi^\iota(\tilde{b}, \tilde{v}_2, \tilde{v}_3) = \mathrm{Enc}_{\mathrm{S,S,S}}(\mathrm{op}_{\mathrm{Br(int)}})(\tilde{b}, \tilde{v}_2, \tilde{v}_3, \iota)$$

will be a hidden representation of 17, along with whatever traces were produced by the execution of the protocol operation $\mathrm{Enc}_{\mathrm{S,S,S}}(\mathrm{op}_{\mathrm{Br(int)}})$. On the other hand, if $\tilde{v}_2$ and $\tilde{v}_3$ are non-primitive values – e.g., lambda abstractions – then the join function, Φ^ι, is unable to arithmetize the branch immediately, since the arguments to which the abstractions will be applied are not available. Thus, Φ^ι wraps these operands in the special "deferred decision" symbol φ:

$$\Phi^\iota(\tilde{b}, \tilde{v}_2, \tilde{v}_3) = \varphi(\tilde{b}, \tilde{v}_2, \tilde{v}_3)$$

Figure 9 Definition of the merge function Φ^ι (for values).

$$\Phi^\iota(\tilde{b}, \tilde{v}_2, \tilde{v}_3) = \begin{cases} (\tilde{v}_2, \varepsilon) \text{ if } \tilde{v}_2 \in \mathcal{E}_{\mathrm{P}}(Y) \text{ and } \tilde{v}_2 = \tilde{v}_3 \\ \mathrm{Enc}_{\ell_1,\ell_2,\ell_3}(\mathrm{op}_{\mathrm{Br}(Y)})(\tilde{b}, \tilde{v}_2, \tilde{v}_3, \iota) \\ \qquad \text{if } \tilde{b} \in \mathcal{E}_{\ell_1}(\mathrm{bool}), \\ \qquad\quad \tilde{v}_2 \in \mathcal{E}_{\ell_2}(Y), \tilde{v}_3 \in \mathcal{E}_{\ell_3}(Y) \\ (\varphi(\tilde{b}, \tilde{v}_2, \tilde{v}_3), \varepsilon) \text{ otherwise} \end{cases}$$

The contents of each memory address in the pair of stores are merged by Φ^ι in the same fashion (arithmetization, for primitive values; and wrapping by φ symbols, for non-primitive values). It is worth noting that since stores contain only primitive values, this process can never cause a memory location to contain a φ symbol.

Conversely, when values wrapped in φ symbols appear as results of evaluated subexpressions (e.g., in the rule for application of a φ symbol), the distributed semantics takes the value from each branch, uses it to evaluate the entire expression (inductively), and merges the results (again using the Φ^ι function). For example, if $\tilde{e}_1$ evaluates to $\varphi(\tilde{b}, \lambda x.0, \lambda x.1)$, and $\tilde{e}_2$ evaluates to 17, then the application $\tilde{e}_1\,\tilde{e}_2$ causes the subexpressions $(\lambda x.0)\,17$ and $(\lambda x.1)\,17$ to be evaluated (returning 0 and 1), and finally executes the merge $\Phi^\iota(\tilde{b}, 0, 1)$ (which, if $\tilde{b}$ is a hidden representation of `true`, will produce a hidden primitive value representing 0).

The case of assignment to a secret reference is similar. Since it is secret, the reference might evaluate to a tree of nested φ symbols (e.g., $\varphi(\tilde{b}_1, \varphi(\tilde{b}_2, a_1, a_2), a_3)$), rather than a single address. In this case, using the update operator, we perform the update recursively on the references from the left and right branches of the φ symbol, then join the resulting (updated) stores as above.

D. Theoretical results

We now present the two main results of our system: theorems that guarantee the correctness and security of an execution. For clarity, we omit the proofs here, since they depend on additional notation and lemmas not presented in this article. We refer the reader to the extended version of this work (available at [29]) for the full definitions and proofs.

Theorem 1 (Correctness)**.** *For any program p and initial input value environment κ, if $\vdash p : \tau$ and $(\kappa, p) \downarrow (y, \mu', \mathcal{O})$*

for some $y \in Y$, then for some mutable store $\tilde{\mu}'$ and communication trace T, $(\kappa, p) \Downarrow (\tilde{y}, \tilde{\mu}', T, \mathcal{O})$, with $\tilde{y} \in \hat{\mathcal{E}}^{\iota}_{\ell}(y)$ for some ℓ.

The correctness theorem expresses that any well-typed source program, when it evaluates according to the reference semantics to some primitive value, will also evaluate according to the distributed semantics to the same primitive value (possibly hidden), yielding identical observations. Since the reference semantics expresses the standard meaning of programs in $\lambda^{\rightarrow}_{\mathrm{P,S}}$, this theorem guarantees that the distributed semantics give a correct implementation for any well-typed source program.

The proof of this theorem involves a few key lemmas:

- *Conditional purity* for the distributed semantics: if an (extended) expression is well-typed in a secret context, then its evaluation terminates, yields no observations, and produces no publicly-observable effects on the mutable store. The proof of this lemma makes extensive use of the control flow restrictions of the static semantics.
- *Type preservation* for the distributed semantics: if an (extended) expression has type τ, and it evaluates to some value v, then v also has type τ (including preservation of confidentiality labels).
- *Correctness for arbitrary expressions*, i.e., the main correctness theorem, generalized in a standard fashion to appropriately "related", rather than identical, expressions, stores, and values.

Theorem 2 (Security). *If $(\kappa_a, p) \Downarrow (\tilde{v}'_a, \tilde{\mu}'_a, T_a, \mathcal{O})$ and $(\kappa_b, p) \Downarrow (\tilde{v}'_b, \tilde{\mu}'_b, T_b, \mathcal{O})$, then for all valid adversarial views $A \in \mathcal{A}$, the distributions $\Pi_A(T_a)$ and $\Pi_A(T_b)$ are indistinguishable*[6].

The security theorem expresses that if two executions of the same program, each with its own secret initial inputs, yield the same observations, then their communication traces (as distributions) are indistinguishable to any adversary that receives only the information available to servers in A. In other words, the theorem guarantees that any suitably constrained adversary learns nothing about the initial secret client values that was not already logically entailed by the observations from `reveal`. It is impossible for the programmer to unintentionally leak information due to the distributed implementation.

The security theorem is proved by instantiation of the following, more general lemma:

> *Safety of traces*: If two expression/store pairs are "equivalent to a public observer", i.e., they are identical except for secret primitive values, and evaluate to two corresponding traces and observation sequences, then neither observation sequence is a proper prefix of the other; and, if the observation sequences are identical, then the resulting value/store pairs are equivalent, and the pair of traces is "safe" (in the sense of the secure execution platform).

Intuitively, as long as two evaluations continue to produce matching observations, then their control flow must match, and the expressions and traces must remain indistinguishable to the adversary. In a sense, this is the best generic result one could expect, since evaluation can depend arbitrarily on public values (and thus can differ arbitrarily once any differing values are declassified).

[6]In the sense specified by the secure execution platform in Section IV.

E. Examples of secure execution platforms

Since we have defined our language in terms of the assumptions of a secure execution platform, it is also important to note that the examples we describe (Shamir secret sharing and fully homomorphic encryption) indeed satisfy these assumptions. We now give an overview of the arguments that establish these facts. Since the arguments proceed from known properties of Shamir secret sharing and fully homomorphic encryption, and are not central to our presentation, we refer the reader to the extended version of this work (available at [29]) for a more detailed treatment.

In Shamir secret sharing, N servers execute the computation in a distributed manner, using an (N, k) sharing scheme. The primitive values are expressed as elements of a finite field $\mathbb{F}_p$. "Hidden" values are represented by N-tuples of field elements, representing each server's share of the value, and the primitive arithmetic and logical operations (including branching), as well as the initial sharing and decryptions. Intermediate values and communication traces are produced according to the rules of Shamir secret sharing. Against a valid adversarial set of fewer than k servers, the primitives of Shamir secret sharing provide information-theoretic security.

On the other hand, in fully homomorphic encryption, the execution proceeds on a single server, $N = 1$. Except for the initial "hiding" (encryption and sending to the server), and "unhiding" (returning values to the client for decryption) upon `reveal` operations, the operations of the platform are executed on the server according to the definition of the cryptosystem, and produce no communication traces. In this case, the trace for the entire evaluation consists of the encrypted values and the revealed plaintexts, and computational indistinguishability (up to revealed values) follows from the security of the cryptosystem.

VI. Implementation

Our implementation consists of three parts. First, we implemented the constructs of the secure execution platform as an embedded domain-specific language in Haskell. Specifically, our EDSL framework consists of a module defining these constructs as a set of combinators, as well as secure multi-party computation (SMC) and fully homomorphic encryption (FHE) libraries implementing the combinators. Second, we implemented a lightweight compiler in Template Haskell, which removes syntactic sugar and translates a subset of Haskell to our EDSL.[7] Finally, we implemented a compiler front-end for $\lambda_{\mathrm{P,S}}^{\rightarrow}$, producing both type and label information for core language programs. In this section, we present these implementations, evaluate their performance, and describe our experience in writing applications.

A. A Haskell EDSL for secure execution platforms

Our EDSL is composed of several type classes [35], which define primitives of a secure execution platform (e.g., `(.+)` for addition) in terms of actions in a *secure* I/O monad. In a debug setting, this "secure" monad is simply Haskell's `IO` monad. However, the secure monad used in implementing a real secure execution platform is more complex. We use monad transformers [36] to stack additional functionalities, in a modular fashion, on top of Haskell's `IO`. Specifically, we implement the following functionalities:

- *RNG:* used for implementing random number generation. Specifically, the transformer provides access to the cryptographically strong, deterministic random bit generator (DRBG) of the Haskell crypto-api library.
- *State:* used for threading library-specific state through computations (including the initial parameters ι). For example, the FHE library uses this state to store the public and private keys used in the secure computation.
- *MPI/RPC* (message passing interface and remote procedure calls): used to enable communication among the client and servers. We use the SSL protocol to provide secure, authenticated party-to-party channels.

The constructs for both of our examples of secure execution platforms, SMC and FHE, can be built using this secure monad. For example, the SMC multiplication combinator `(.*)` uses the RNG and MPI functionalities to generate and communicate a new secret sharing among the servers. On the other hand, the FHE multiplication combinator relies on the State functionality to store the key needed by the homomorphic evaluation function (from the Gentry-Halevi implementation of FHE [37], [38]) that performs the actual multiplication. Furthermore, we note that although our implementation currently treats only SMC and FHE, the secure monad can easily be extended with other monad transformers to add features required by other platforms. The flexibility and modularity of monads makes this embedding approach especially attractive.

For both secure execution platforms, SMC and FHE, our EDSL implements secure addition `(.+)`, subtraction `(.-)` and multiplication `(.*)`, bitwise and logical operators (and, or, and exclusive-or), and comparison operations (equality `(.==)` and inequality `(./=)` testing, less-than `(.<)`, greater-than `(.>)`, and so on). We also implement branching operators, `sif-sthen-selse`, in terms of arithmetization, as described above: `sif b sthen x selse y` becomes `b .* x + (1 .- b) .* y`.

For Shamir secret sharing, our implementation of the above primitives is essentially a direct translation of the protocol into a Haskell implementation. In the case of fully homomorphic encryption, our library extends the Gentry-Halevi implementation, presented in [37], [38]. Their C++ implementation 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 by providing support for k-bit homomorphic addition, multiplication, comparison and equality testing functions. In integrating the extended C++ FHE library with our Haskell framework, we implemented C wrappers for the basic FHE operations, and various library functions. The EDSL primitives are implemented as foreign calls to the corresponding C functions, using Haskell's Foreign Function Interface (FFI). Our design hides the low-level C details from the programmer, in addition to adding garbage collection support to the underlying FHE library.

B. Extending the EDSL with Template Haskell

Our Template Haskell compiler takes a Haskell AST, enclosed in Template Haskell quotes `[| ... |]`, and outputs a transformed AST, which is spliced into the surrounding code using Template Haskell's `$(...)`. The compiler removes syntactic sugar, performs label inference and static label checks, and translates Haskell library operators, such as `<=`, to our EDSL operators `(.<=)`, in addition to inserting type annotations and explicit conversions from public to secret values. An example use of this compiler is shown in Figure 3. In our implementation efforts, this compiler serves as an intermediate step between the EDSL (i.e., using the secure execution platform primitives directly) and the full core language $\lambda_{\mathrm{P,S}}^{\rightarrow}$.

C. Core language

We also implement a compiler front-end for our core language, $\lambda_{\mathrm{P,S}}^{\rightarrow}$, adapting standard approaches [39] to perform type inference and type checking according to the rules of Section V. In ongoing work, we are continuing to improve the compiler front-end, and extending our development to a full $\lambda_{\mathrm{P,S}}^{\rightarrow}$ compiler using the Template Haskell and QuasiQuotes extensions.

D. Evaluation

Our experimental setup consists of 6 machines, interconnected on a local Gig-E network, with each machine

[7]Notably, the implementation adds constructs for arrays of public length, product and sum types, and bounded iteration primitives. While these features are very useful in practice, we omit them in our present theoretical analysis, as they would further complicate the formalism without providing additional insight.

	Addition	Multiplication	Comparison	Assignment
Public	0.003 sec	0.006 sec	0.004 sec	0.003 sec
Secret	0.006 sec	1.71 sec	406.8 sec	3.28 sec

Table I
MICRO-BENCHMARKS FOR SMC ($N = 5, k = 2$). WE MEASURE 1000 OPERATIONS ON RANDOMLY GENERATED VALUES.

containing two Intel Xeon E5620 (2.4GHz) processors, and 48GB of RAM. Our SMC implementation uses arithmetic modulo the largest 32-bit prime.

We measured the performance of several SMC core primitives. In particular, we measure the cost of addition (`.+`), multiplication (`.*`), and comparison (`.>=`) of two secret integers and compare them to the corresponding public operations. Similarly, we compare the cost of assignment to a memory location in secret and public conditionals. Table I summarizes these results. We observe that SMC additions, multiplications, and comparisons on secret operands are roughly $1.8\times$, $300\times$ and $91000\times$ slower than the corresponding operation on public values[8]. We note that the time difference in the additions of public and secret values (since addition does not involve network communication) serves as a measurement for the performance overhead incurred by our secret monad, i.e., our library imposes a $1.8\times$ overhead. Assignments are roughly $1000\times$ slower, reflecting the fact that each assignment in a secret context consists of an application of a branching operator $\text{op}_{\text{Br(int)}}$ (which, in the case of SMC, consists of two multiplications, a subtraction, and an addition).

We also measure the cost of using our framework to implement a portion of Reliable Email. Specifically, we measure the cost of checking if an email address is present in a whitelist of 100 random entries. Our results indicate that the average time of checking such a secret list is about 2.4 minutes.

Although the overhead for secret computations seems formidable, we note that our prototype implementation uses a very simple multi-party computation protocol, which incurs a round of communication for each multiplication operation. There exist more efficient protocols that use a constant number of rounds to execute an arbitrary (though pre-specified) sequence of operations [40]. Since the overhead from network latency is significant, the efficiency of our system should improve substantially upon adapting our implementation to use a more round-efficient scheme for sub-computations whose operations are known in advance (i.e., do not depend on values from `reveal` operations). Likewise, we anticipate a significant speedup when independent computations (such as the 100 email address comparisons above) are performed in parallel. Moreover, given the wide interest in secure cloud computing, we expect the performance of the underlying cryptographic primitives, and thereby our EDSL, to improve in the near future.

[8]We omit performance evaluations for fully homomorphic encryption, as past results have shown that the time complexity of existing schemes renders experimental results of limited value [15].

VII. RELATED WORK

Our static semantics is similar to many standard type systems for information flow control (for example, the system described in the work of Pottier and Simonet [30]). However, as described in Section V-B, our system is designed for a dual purpose: rather than ensuring that any expressible program is free of information leaks, our system ensures that any expressible program (which, by definition, cannot leak information on a secure execution platform) is nevertheless implementable in our model. In addition, we need substantially different restrictions to deal with the problem of control flow leakage.

Li and Zdancewic's seminal work [41], [42] presents the first implementation of information flow control as a library, in Haskell. They enforce information flow dynamically, and consider only pure computations; our system relies on strong static guarantees, and addresses a language with side effects. Subsequently [43], Tsai et al. addressed the issue of internal timing for a multi-threaded language; since in our model the servers can only observe their own execution, timing attacks are not relevant. More closely related, Russo et al. [44] present a static information flow library, SecIO, that is statically enforced using Haskell type classes. They prove *termination-insensitive* non-interference for a call-by-name λ-calculus. Our type system enforces substantially stricter requirements on control flow. Nevertheless, their system is complementary to ours, and SecIO could be used to implement some of our static restrictions.

Vaughan presents an extension to Aura, called AuraConf [45], which provides an information flow language with cryptographic support. AuraConf allows programmers with knowledge of access control to implement general decentralized information flow control. The AuraConf system also builds on earlier work by Vaughan and Zdancewic, in which they develop a decentralized label model for cryptographic operations [46]. In addition, Fournet et al., in work on the CFlow system [47], analyze information flow annotations on cryptographic code. In contrast to these approaches, our system abstracts away the cryptography primitives from the language, allowing programmers without specialized knowledge to write secure applications for the cloud.

Systems for secure computations include SCET [48], with focus on economic applications and secure double auctions; FairplayMP [49], a specification language SFDL that is converted to primitive operations on bits; Sharemind [50], for multiparty computations on large datasets; VIFF [12], a basic language embedded in Python and API to cryptographic primitives. These systems implement cryptographic protocols, without proving the more comprehensive correctness and security properties.

The closest work to ours is SMCL [11], an imperative-style DSL with similar goals (notably, SMCL aims to enable programmers to use secure multiparty computation without detailed knowledge of the underlying protocol), as well as some similar constructs (notably, the behavior of SMCL's `open` construct is very similar to our `reveal`). However, our work improves on existing efforts in several respects. While the papers on SMCL do exhibit correctness and security properties, but they do not formally define some crucial aspects: notably, the execution model of the platform, and the security properties required of its primitives so that security for the entire system can be guaranteed. Unlike SMCL, our system also generalizes to other platforms beyond SMC. In addition, our system provides significantly richer language constructs (encompassing both imperative and functional features).

In a previous paper [15], we described a restricted language without recursion, mutable stores, and conditionals. As in this paper, we proved correctness and security for programs written in the language executing on a secure execution platform. Drawing on our experience of working with the EDSL in our previous work, we implemented the Template Haskell compiler and the compiler frontend for our core language. As far as we know, we are the first to formalize and prove correctness and security properties for a unified language framework, providing rich language features such as recursion, mutable stores, and conditionals, and encompassing a wide range of cryptographic schemes for computation on encrypted data.

VIII. Conclusions

We present an expressive core language for secure cloud computing, with primitive types, conditionals, standard functional features, mutable state, and a secrecy preserving form of general recursion. This language uses an augmented information-flow type system to impose conventional information-flow control and prevent previously unexplored forms of control-flow leakage that may occur when the execution platform is untrusted. The language allows programs to be developed and tested using conventional means, then exported to a variety of secure cloud execution platforms, dramatically reducing the amount of specialized knowledge needed to write secure code. We prove correctness and confidentiality for any platform meeting our definitions, and note two examples of such platforms: fully homomorphic encryption, as well as a multi-party computation protocol based on Shamir secret sharing.

The implementation of our language 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. On the other hand, implementation in Haskell is not an inherent feature of our core language; other languages with functional features, such as Scala or F# (or even object-oriented languages such as Java, with some additional implementation effort) would also be reasonable choices. Our core language could also be used to develop secure libraries that can be safely called from other languages, providing the strong security guarantees of our DSL in an unrestricted multi-language programming environment.

In future work, we plan to extend our theoretical framework to other secure execution platforms that can provide stronger guarantees, such as security against active adversaries. We will also explore the possibility of mechanically verifying that a particular implementation realizes our distributed semantics. 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 more efficient partially homomorphic encryption schemes.

Acknowledgements

We thank the anonymous reviewers, as well as Amit Levy and Jérémy Planul, for helpful feedback on earlier versions of this paper. 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. Deian Stefan and Joe Zimmerman are further supported by the Department of Defense (DoD) through the National Defense Science & Engineering Graduate Fellowship (NDSEG) Program.

References

[1] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *STOC*, 2009, pp. 169–178.

[2] C. Gentry, "Computing arbitrary functions of encrypted data," *Commun. ACM*, vol. 53, no. 3, pp. 97–105, 2010.

[3] M. van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan, "Fully homomorphic encryption over the integers," in *EUROCRYPT*, 2010, pp. 24–43.

[4] M. Ben-Or, S. Goldwasser, and A. Wigderson, "Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract)," in *STOC*, 1988, pp. 1–10.

[5] R. Gennaro, M. O. Rabin, and T. Rabin, "Simplified VSS and fact-track multiparty computations with applications to threshold cryptography," in *PODC*, 1998, pp. 101–111.

[6] R. Cramer, I. Damgård, and U. M. Maurer, "General secure multi-party computation from any linear secret-sharing scheme," in *EUROCRYPT*, 2000, pp. 316–334.

[7] M. Naor and K. Nissim, "Communication preserving protocols for secure function evaluation," in *STOC*, 2001, pp. 590–599.

[8] M. C. Silaghi, "SMC: Secure multiparty computation language," http://www.cs.fit.edu/~msilaghi/SMC/tutorial.html, 2004.

[9] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella, "Fairplay - secure two-party computation system," in *USENIX Security Symposium*, 2004, pp. 287–302.

[10] 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/.

[11] J. D. Nielsen and M. I. Schwartzbach, "A domain-specific programming language for secure multiparty computation," in *PLAS*, 2007, pp. 21–30.

[12] I. Damgård, M. Geisler, M. Krøigaard, and J. B. Nielsen, "Asynchronous multiparty computation: Theory and implementation," in *Public Key Cryptography*, 2009, pp. 160–179.

[13] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan, "CryptDB: protecting confidentiality with encrypted query processing," in *SOSP*, 2011, pp. 85–100.

[14] O. Goldreich, S. Micali, and A. Wigderson, "How to play any mental game or a completeness theorem for protocols with honest majority," in *STOC*, 1987, pp. 218–229.

[15] A. Bain, J. C. Mitchell, R. Sharma, D. Stefan, and J. Zimmerman, "A domain-specific language for computing on encrypted data (invited talk)," in *FSTTCS*, 2011, pp. 6–24.

[16] S. Garriss, M. Kaminsky, M. J. Freedman, B. Karp, D. Mazières, and H. Yu, "Re: Reliable email," in *NSDI*, 2006.

[17] T. Sheard and S. L. P. Jones, "Template meta-programming for Haskell," *SIGPLAN Notices*, vol. 37, no. 12, pp. 60–75, 2002.

[18] D. Boneh, E.-J. Goh, and K. Nissim, "Evaluating 2-DNF formulas on ciphertexts," in *TCC*, 2005, pp. 325–341.

[19] C. Gentry, S. Halevi, and V. Vaikuntanathan, "A simple BGN-type cryptosystem from LWE," in *EUROCRYPT*, 2010, pp. 506–522.

[20] Y. Ishai and A. Paskin, "Evaluating branching programs on encrypted data," in *TCC*, 2007, pp. 575–594.

[21] N. P. Smart and F. Vercauteren, "Fully homomorphic encryption with relatively small key and ciphertext sizes," in *Public Key Cryptography*, 2010, pp. 420–443.

[22] A. C.-C. Yao, "Protocols for secure computations (extended abstract)," in *FOCS*, 1982, pp. 160–164.

[23] Y. Lindell and B. Pinkas, "A proof of security of Yao's protocol for two-party computation," *J. Cryptology*, vol. 22, no. 2, pp. 161–188, 2009.

[24] Y. Ishai and E. Kushilevitz, "Randomizing polynomials: A new representation with applications to round-efficient secure computation," in *FOCS*, 2000, pp. 294–304.

[25] Y. Ishai, E. Kushilevitz, and A. Paskin, "Secure multiparty computation with minimal interaction," in *CRYPTO*, 2010, pp. 577–594.

[26] I. Damgård, Y. Ishai, and M. Krøigaard, "Perfectly secure multiparty computation and the computational overhead of cryptography," in *EUROCRYPT*, 2010, pp. 445–465.

[27] B. Applebaum, Y. Ishai, and E. Kushilevitz, "From secrecy to soundness: Efficient verification via secure computation," in *ICALP (1)*, 2010, pp. 152–163.

[28] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, pp. 5–19, 2003.

[29] D. S. J.C. Mitchell, R. Sharma and J. Zimmerman, "Information-flow control for programming on encrypted data," Cryptology ePrint Archive, Report 2012/205, 2012, http://eprint.iacr.org/.

[30] F. Pottier and V. Simonet, "Information flow inference for ML," in *POPL*, 2002, pp. 319–330.

[31] A. C. Myers, "JFlow: Practical mostly-static information flow control," in *POPL*, 1999, pp. 228–241.

[32] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious RAMs," *J. ACM*, vol. 43, no. 3, 1996.

[33] J. M. Lucassen and D. K. Gifford, "Polymorphic effect systems," in *POPL*, 1988, pp. 47–57.

[34] D. Marino and T. D. Millstein, "A generic type-and-effect system," in *TLDI*, 2009, pp. 39–50.

[35] C. V. Hall, K. Hammond, S. L. P. Jones, and P. Wadler, "Type classes in Haskell," in *ESOP*, 1994, pp. 241–256.

[36] S. Liang, P. Hudak, and M. P. Jones, "Monad transformers and modular interpreters," in *POPL*, 1995, pp. 333–343.

[37] C. Gentry and S. Halevi, "Implementing Gentry's fully-homomorphic encryption scheme," Manuscript; see https://researcher.ibm.com/researcher/view_page.php?id=1579, 2010.

[38] 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.

[39] L. Damas and R. Milner, "Principal type-schemes for functional programs," in *POPL*, 1982, pp. 207–212.

[40] D. Beaver, S. Micali, and P. Rogaway, "The round complexity of secure protocols (extended abstract)," in *STOC*, 1990, pp. 503–513.

[41] P. Li and S. Zdancewic, "Encoding information flow in Haskell," in *CSFW*, 2006, p. 16.

[42] P. Li and S. Zdancewic, "Arrows for secure information flow," *Theor. Comput. Sci.*, vol. 411, no. 19, pp. 1974–1994, 2010.

[43] T.-C. Tsai, A. Russo, and J. Hughes, "A library for secure multi-threaded information flow in Haskell," in *CSF*, 2007, pp. 187–202.

[44] A. Russo, K. Claessen, and J. Hughes, "A library for lightweight information-flow security in Haskell," in *Haskell*, 2008, pp. 13–24.

[45] J. A. Vaughan, "Auraconf: a unified approach to authorization and confidentiality," in *TLDI*, 2011, pp. 45–58.

[46] J. A. Vaughan and S. Zdancewic, "A cryptographic decentralized label model," in *IEEE Symposium on Security and Privacy*, 2007, pp. 192–206.

[47] C. Fournet, J. Planul, and T. Rezk, "Information-flow types for homomorphic encryptions," in *ACM Conference on Computer and Communications Security*, 2011, pp. 351–360.

[48] P. Bogetoft, I. B. Damgard, T. Jakobsen, K. Nielsen, J. Pagter, and T. Toft, "Secure computing, economy, and trust: A generic solution for secure auctions with real-world applications," BRICS, Tech. Rep. RS-05-18, 2005.

[49] A. Ben-David, N. Nisan, and B. Pinkas, "FairplayMP: a system for secure multi-party computation," in *ACM Conference on Computer and Communications Security*, 2008, pp. 257–266.

[50] D. Bogdanov, S. Laur, and J. Willemson, "Sharemind: A framework for fast privacy-preserving computations," in *ESORICS*, 2008, pp. 192–206.

APPENDIX

A. Semantic structure and assumptions

In this appendix, we elaborate on the the definitions and assumptions that constitute a secure execution platform (omitted above for brevity).

1) Distributed computing infrastructure: We assume N servers, $S_1, \dots, S_N$, execute the secure computation on behalf of one client, C. (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 $A \subseteq \{S_1, \dots, S_N\}$ 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:

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

General form of cryptographic primitives: We work with a two-element security lattice, $\mathrm{P} \sqsubseteq \mathrm{S}$, 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. For each primitive type $Y \in \mathcal{Y}$, we assume a set $\mathcal{E}_{\mathrm{S}}(Y)$, holding "secret equivalents" of base values in Y; for notational uniformity, we also define $\mathcal{E}_{\mathrm{P}}(Y) = Y$, signifying that the "public equivalent" of a value is just the value itself. Similarly, we assume, for any $y \in Y$, a set $\mathcal{E}_{\mathrm{S}}(y) \subset \mathcal{E}_{\mathrm{S}}(Y)$, holding the "secret equivalents" of y (with $\mathcal{E}_{\mathrm{P}}(y) = \{y\}$); we assume that the sets $\{\mathcal{E}_\alpha(y) : y \in Y\}$ form a partition of $\mathcal{E}_\alpha(Y)$. We recall that for any two elements (or *labels*) of a lattice, we have a well-defined *join* ($\sqcup$), which corresponds to the *least upper bound* of the two elements (e.g., $\mathrm{P} \sqcup \mathrm{S} = \mathrm{S}$).

We also assume a few standard cryptographic primitives, expressed as *protocol operations* that may operate on initial parameters $\iota \in \mathcal{I}$, generate communication traces among the parties, and/or consume bits from a source of randomness. For clarity, we leave this randomness source implicit, instead considering each operation to produce a distribution over the values in its range (and implicitly lifting the operations to act on distributions over their domains). We regard predicates over these distributions to be true if they hold with probability 1.

The operations we assume are as follows (overloaded for all primitive types Y):

- $\mathrm{Enc}_{\mathrm{S}} : Y \times \mathcal{I} \to \mathcal{E}_{\mathrm{S}}(Y) \times \mathcal{T}$, "hiding" $y \in Y$.
- $\mathrm{Dec}_{\mathrm{S}} : \mathcal{E}_{\mathrm{S}}(Y) \times \mathcal{I} \to Y \times \mathcal{T}$, "unhiding" $\tilde{y} \in \mathcal{E}_{\mathrm{S}}(Y)$.
- $\mathrm{Enc}_{\ell_1,\dots,\ell_r}(\mathrm{op}_i) : \prod_j \mathcal{E}_{\ell_j}(\mathrm{dom}(\mathrm{op}_i)_j) \times \mathcal{I} \to \mathcal{E}_{\bigsqcup_j \ell_j}(\mathrm{cod}(\mathrm{op}_i)) \times \mathcal{T}$ (when at least one ℓ_j is S), evaluating a primitive operation.

We also assume that Init describes the generation of initial parameters according to some distribution $\mathcal{I}$ (for example, public and secret keys in the case of homomorphic encryption). For notational uniformity, as above, we also define the corresponding operations in the degenerate case of "hiding" public values (operating as the identity on the plaintext values, and yielding empty traces):

- $\mathrm{Enc}_{\mathrm{P},\dots,\mathrm{P}}(\mathrm{op}_i)(y_1, \dots, y_r, \iota) = (\mathrm{op}_i(y_1, \dots, y_r), \varepsilon)$
- $\mathrm{Enc}_{\mathrm{P}}(y, \iota) = (y, \varepsilon)$
- $\mathrm{Dec}_{\mathrm{P}}(y, \iota) = (y, \varepsilon)$

We also write Dec as shorthand for $\mathrm{Dec}_{\mathrm{P}}$ or $\mathrm{Dec}_{\mathrm{S}}$, as appropriate based on the domain (i.e., Dec acts as $\mathrm{Dec}_{\mathrm{P}}$ on Y, and acts as $\mathrm{Dec}_{\mathrm{S}}$ on $\mathcal{E}_{\mathrm{S}}(Y)$). In addition, we assume a projection operator from the initial parameters onto any server or set of servers, writing:

$$\Pi_A(\iota) = (\Pi_{\{S_{a_1}\}}(\iota), \dots, \Pi_{\{S_{a_k}\}}(\iota))$$

(where $A = \{S_{a_1}, \dots, S_{a_k}\}$) to mean, intuitively, the portion of the initial parameters $\iota \in \mathcal{I}$ that servers in A should receive.

In addition, we assume that equality is efficiently decidable on any universe Y of primitive values; that the label ℓ and universe Y of a value in $\mathcal{E}_\ell(Y)$ are efficiently computable from the value itself (e.g., by tagging, when the underlying sets are the same); and that there is some canonical ordering on the universes.

Cryptographic correctness assumptions: We assume the usual encryption and homomorphism conditions, augmented for cryptographic primitives that depend on randomness and that may communicate among servers to produce their result. For every element y of a primitive type Y, and every choice of initial parameters $\iota \in \mathcal{I}$, we assume a family of *safe distributions* $\hat{\mathcal{E}}^{\iota}_{\ell}(y)$ over $\mathcal{E}_{\ell}(y)$: intuitively, any distribution $l \in \hat{\mathcal{E}}^{\iota}_{\ell}(y)$ can safely serve as the "hiding" of y under the initial parameters ι (at secrecy level $\ell \in \{\mathrm{P}, \mathrm{S}\}$). We require that "hiding" a base value must yield a safe distribution:

- $\pi_1(\mathrm{Enc}_{\ell}(y, \iota)) \in \hat{\mathcal{E}}^{\iota}_{\ell}(y)$

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

- $\pi_1(\mathrm{Dec}_{\ell}(\pi_1(\mathrm{Enc}_{\ell}(y, \iota)), \iota)) = y$
- $\pi_1(\mathrm{Enc}_{\ell_1,\ldots,\ell_r}(\mathrm{op}_i)(l_1, \ldots, l_r, \iota)) \in \hat{\mathcal{E}}_{\bigsqcup_j \ell_j}(\mathrm{op}_i(y_1, \ldots, y_r), \iota)$ whenever $l_j \in \hat{\mathcal{E}}^{\iota}_{\ell_j}(y_j)$

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

Different platforms may provide different security guarantees of their primitives. For example, protocols may specify that distributions are *computationally indistinguishable* (i.e., indistinguishable to a probabilistic polynomial-time adversary), or *information-theoretically indistinguishable* (i.e., identical). For the purposes of this development, we will use the term *indistinguishable* to refer to whichever of the above notions is specified by the secure execution platform. Using this terminology, we require that any two sequences of partial traces are indistinguishable if each pair of corresponding partial traces describes either 1.) a "hiding" operation; 2.) a primitive operation whose public arguments agree (and whose hidden arguments are safely-distributed); or 3.) an "unhiding" operation on values that turn out to be equal. More precisely, we say that the pair of communication rounds $T_j(\iota), T'_j(\iota)$ is *safe*, denoted $\textsc{Safe}(\iota, T_j(\iota), T'_j(\iota))$, if it satisfies any of the following conditions:

1) $T_j(\iota) = \pi_2(\mathrm{Enc}_{\mathrm{S}}(y_j, \iota))$, and $T'_j(\iota) = \pi_2(\mathrm{Enc}_{\mathrm{S}}(y'_j, \iota))$ (for some Y, and $y_j, y'_j \in Y$). In this case, we say that $T_j(\iota), T'_j(\iota)$ constitute a "safe hiding" (denoted $\textsc{SafeEnc}(\iota, T_j(\iota), T'_j(\iota))$).
2) $T_j(\iota) = \pi_2(\mathrm{Enc}_{\ell_1,\ldots,\ell_r}(\mathrm{op}_i)(\tilde{y}_1, \ldots, \tilde{y}_r, \iota))$ and $T'_j(\iota) = \pi_2(\mathrm{Enc}_{\ell_1,\ldots,\ell_r}(\mathrm{op}_i)(\tilde{y}'_1, \ldots, \tilde{y}'_r, \iota))$ where for each k, either:
   - $\ell_k = \mathrm{S}$, and for some Y_k, we have $\tilde{y}_k \in \hat{\mathcal{E}}^{\iota}_{\mathrm{S}}(y_k)$ and $\tilde{y}'_k \in \hat{\mathcal{E}}^{\iota}_{\mathrm{S}}(y'_k)$, with $y_k, y'_k \in Y_k$.
   - $\ell_k = \mathrm{P}$, and for some Y_k, we have $\tilde{y}_k, \tilde{y}'_k \in Y_k$, and $\tilde{y}_k = \tilde{y}'_k$.

   (and the analogous conditions for $T'_j(\iota)$). In this case, we say that $T_j(\iota), T'_j(\iota)$ constitute a "safe primitive operation" (denoted $\textsc{SafeOp}(\iota, T_j(\iota), T'_j(\iota))$).
3) $T_j(\iota) = \pi_2(\mathrm{Dec}_{\mathrm{S}}(c_j, \iota)), T'_j(\iota) = \pi_2(\mathrm{Dec}_{\mathrm{S}}(c'_j, \iota))$ and $\pi_1(\mathrm{Dec}_{\mathrm{S}}(c_j, \iota)) = \pi_1(\mathrm{Dec}_{\mathrm{S}}(c'_j, \iota))$. In this case, we say that $T_j(\iota), T'_j(\iota)$ constitute a "safe unhiding" (denoted $\textsc{SafeDec}(\iota, T_j(\iota), T'_j(\iota))$).

We extend the predicate $\textsc{Safe}$ to pairs of entire traces if each component is safe: i.e., $\textsc{Safe}(\iota, T_1, T_2)$ if $|T_1| = |T_2|$ and $\textsc{Safe}(\iota, T_1(j), T_2(j))$ for all j. Finally, we consider partial traces $T(\iota) = (T_1(\iota), \ldots, T_m(\iota))$ and $T'(\iota) = (T'_1(\iota), \ldots, T'_m(\iota))$, and the corresponding adversarial views:

- $O(\iota) = (\Pi_A(\iota), \Pi_A(T_1(\iota)), \ldots, \Pi_A(T_k(\iota)))$
- $O'(\iota) = (\Pi_A(\iota), \Pi_A(T'_1(\iota)), \ldots, \Pi_A(T'_k(\iota)))$

We require the following indistinguishability condition: if $\textsc{Safe}(\iota, T_j(\iota), T'_j(\iota))$ for every j, then the distributions $O(\mathrm{Init}())$ and $O'(\mathrm{Init}())$ are indistinguishable.

Definition 1. We say that the system $(N, \mathcal{I}, \mathrm{Init}, \mathcal{E}, \hat{\mathcal{E}}, M, \mathrm{Enc}, \mathrm{Dec}, \mathcal{A})$ is a *secure execution platform* for (op_i) if it satisfies all of the conditions specified above.