Taypsi: Static Enforcement of Privacy Policies for Policy-Agnostic Oblivious Computation

QIANCHUAN YE, Purdue University, USA
BENJAMIN DELAWARE, Purdue University, USA

Secure multiparty computation (MPC) techniques enable multiple parties to compute joint functions over their private data without sharing that data with other parties, typically by employing powerful cryptographic protocols to protect individual's data. One challenge when writing such functions is that most MPC languages force users to intermix programmatic and privacy concerns in a single application, making it difficult to change or audit a program's underlying privacy policy. Prior policy-agnostic MPC languages relied on dynamic enforcement to decouple privacy requirements from program logic. Unfortunately, the resulting overhead makes it difficult to scale MPC applications that manipulate structured data. This work proposes to eliminate this overhead by instead transforming programs into semantically equivalent versions that statically enforce user-provided privacy policies. We have implemented this approach in a new MPC language, called Taypsi; our experimental evaluation demonstrates that the resulting system features considerable performance improvements on a variety of MPC applications involving structured data and complex privacy policies.

CCS Concepts: • **Software and its engineering** → **Functional languages**; *Data types and structures*; *Compilers*; • **Security and privacy** → Cryptography.

Additional Key Words and Phrases: Oblivious Computation, Dependent Types, Algebraic Data Types

ACM Reference Format:
Qianchuan Ye and Benjamin Delaware. 2024. Taypsi: Static Enforcement of Privacy Policies for Policy-Agnostic Oblivious Computation. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 144 (April 2024), 30 pages. https://doi.org/10.1145/3649861

1 INTRODUCTION

Secure multiparty computation (MPC) techniques allow multiple parties to jointly compute a function over their private data while keeping that data secure. A variety of privacy-focused applications can be formulated as MPC problems, including secure auctions, voting, and privacy-preserving machine learning [Evans et al. 2018; Hastings et al. 2019; Laud and Kamm 2015]. MPC solutions typically depend on powerful cryptographic techniques, e.g., Yao's Garbled Circuits [Yao 1982] or secret sharing [Beimel 2011], to provide strong privacy guarantees. These cryptographic techniques can be difficult for non-experts to use, leading to the creation of several high-level languages that help programmers write MPC applications [Acay et al. 2021; Darais et al. 2020; Hastings et al. 2019; Liu et al. 2015; Malkhi et al. 2004; Rastogi et al. 2014, 2019; Sweet et al. 2023; Ye and Delaware 2022; Zahur and Evans 2015; Zhang et al. 2013]. While raising the level of abstraction, almost all of these languages intermix privacy and programmatic concerns, requiring the programmers to explicitly enforce the high-level privacy policies within the logic of the application itself, using the secure operations provided by the language. As a consequence, the entire application must be examined

Authors' addresses: Qianchuan Ye, Purdue University, West Lafayette, USA, ye202@purdue.edu; Benjamin Delaware, Purdue University, West Lafayette, USA, bendy@purdue.edu.

ACM 2475-1421/2024/4-ART144
https://doi.org/10.1145/3649861

in order to audit its privacy policy, and an application must be rewritten in order to change its privacy guarantees. This intermixing of policy enforcement and application logic thus makes it difficult to read, write, and reason about MPC applications.

This is particularly true for applications with the sorts of complex requirements that can occur in practice. Within the United States, for example, the Health Insurance Portability and Accountability Act (HIPAA) governs how patient data may be used. HIPAA allows *either* the personally identifiable information (PII) *or* medical data to be shared, but not both. Notably, this policy does not simply specify whether some particular field of a patient's medical record is private or public; rather it is a *relation* that dictates how a program can access and manipulate different parts of every individual record. To conform to this policy, an MPC application must either pay the (considerable) cryptographic overhead of conservatively securing all accesses to the fields of a record, or adopt a more sophisticated strategy for monitoring how data is accessed. These challenges become more acute when dealing with structured data, e.g., lists or trees, whose policies are necessarily more complex. Consider a classifier that takes as input a decision tree and a medical record, each of which is owned by a different party: if the owner of the tree stipulates that its depth may be disclosed, the classification function must use secure operations to ensure that no other information about the tree is leaked, e.g., its spine or the attributes it uses. If the owner of this tree is willing to share such information, however, this function must either be rewritten to take advantage of the new, more permissive policy, or continue to pay the cost of providing stricter privacy guarantees. Thus, most existing MPC languages require users to write different implementations of essentially the same program for each distinct privacy policy.

A notable exception is Taype [Ye and Delaware 2023], a recently proposed language that decouples privacy policies from programmatic concerns, allowing users to write applications over structured data that are agnostic to any particular privacy policy. To do so, Taype implements a novel form of the *tape semantics* proposed by Ye and Delaware [2022]. This semantics allows insecure operations whose evaluation *could* violate a policy to appear in a program, as long as the results of these operations are *eventually* protected. Under tape semantics, such operations are lazily deferred until it is safe to execute them, effectively *dynamically* "repairing" potential leaks at runtime. Using Taype, programmers can thus build a privacy-preserving version of a standard functional program by composing it with a policy, specified as a *dependent type* equipped with security labels, relying on tape semantics to enforce the policy during execution. Unfortunately, while this enforcement strategy disentangles privacy concerns from program logic, it also introduces considerable overhead for applications that construct or manipulate structured data with complex privacy requirements. Thus, this strategy does not scale to the sorts of complex applications that could greatly benefit from this separation of concerns.

This work presents Taypsi, a policy-agnostic language for writing MPC applications that eliminates this overhead by instead transforming a non-secure function into a version that *statically* enforces a user-provided privacy policy. Taypsi extends Taype with a form of dependent sums, which we call Ψ-types, that package together the public and private components of an algebraic data type (ADT). Each Ψ-type is equipped with a set of Ψ-*structures* which play an important role in our translation, enabling it to, e.g., efficiently combine subcomputations that produce ADTs with different privacy policies. Our experimental evaluation demonstrates that this strategy yields considerable performance improvements over the enforcement strategy used by Taype, yielding exponential improvements on the most complex benchmarks in our evaluation suite.

To summarize, the contributions of this paper are as follows:

- We present Taypsi, a version of Taype extended with Ψ-types, a form of dependent sums that enables modular translation of non-secure programs into efficient, secure versions. This

language is equipped with a security type system that offers the same guarantees as Taype: after jointly computing a well-typed function, neither party can learn more about the other's private data than what can be gleaned from their own data and the output of the function.
- We develop an algorithm that combines a program written in the non-secure fragment of Taypsi with a privacy policy to produce a secure private version that statically enforces the desired policy. We prove that this algorithm generates well-typed (and hence secure) target programs that are additionally guaranteed to preserve the semantics of the source programs.
- We evaluate our approach on a range of case studies and microbenchmarks. Our experimental results demonstrate exponential performance improvements over the previous state-of-the-art (Taype) on several complicated benchmarks, while simultaneously showing no performance regression on the remaining benchmarks.

2 OVERVIEW

Before presenting the full details of our approach, we begin with an overview of Taypsi's strategy for building privacy-preserving applications. Consider the simple `filter` function in Figure 1, which drops all the elements in a list above a certain bound.[1] Suppose Alice owns some integers, and wants to know which of those integers are less than some threshold integer belonging to Bob, but neither party wants to share their data with the other. MPC protocols allow Alice and Bob to encrypt their data and then jointly compute `filter` using secure operations, without leaking information about the encrypted data beyond what they can infer from the final disclosed output. One (insecure) implementation strategy is to simply encrypt everyone's integers and use a secure version of the $\leq$ operation to compute the resulting list. Under a standard semi-honest threat model,[2] however, this naive strategy can reveal private information, via the shape of the input and the intermediate program states.

```
data list = Nil | Cons ℤ list

fn filter : list → ℤ → list = λxs y ⇒
  match xs with Nil ⇒ Nil
  | Cons x xs′ ⇒
    if x ≤ y then Cons x (filter xs′ y)
    else filter xs′ y
```

Fig. 1. Filtering a list

As an example, assume Alice's input list is `Cons [2] (Cons [7] (Cons [3] Nil))`, and Bob's input is `[5]`, where square brackets denote secure (encrypted) numbers, i.e., only the owner of the integer can observe its value. By observing that Alice's private data is built from three `Cons`s, Bob can already tell Alice owns exactly 3 integers, information that Alice may want to keep secret. In addition, both parties can learn information from the control flow of the execution of `filter`: by observing which branch of `if` is executed, for example, Bob can infer that the second element of Alice's list is greater than 5. Thus, even if the integers are secure, both parties can still glean information about the other's private data.

The particular policy that a secure application enforces can greatly impact the performance of that application, since the control flow of an application cannot depend on private data. In the case of our example, this means that the number of recursive calls to `filter` depends on the public information Alice is willing to share. If Alice only wants to share the maximum length of her list, for example, its encrypted version must be padded with dummy encrypted values, and a secure version of `filter` must recurse over these dummy elements, in order to avoid leaking information to Bob through its control flow. On the other hand, if Alice does not mind sharing the exact number of integers she owns, the joint computation will not have to go over these values, allowing a secure version of `filter` to be computed more efficiently.

[1] Taypsi supports higher-order functions, but our overview will use this specialized version for presentation purposes.
[2] In this threat model, all parties can see the execution traces produced by a small-step semantics [Ye and Delaware 2022].

2.1 Encoding Private Data and Policies

Taypsi allows Alice and Bob to use types to describe what public information can be shared about their private data (i.e., the policy governing that data), and its type system guarantees that these policies are not violated when jointly computing a function over that data. At a high-level, a privacy policy for a structured (i.e., algebraic) datatype specifies which of its components are private and which can be freely shared. We call this publicly shared information a *public view*, reflecting that it is some projection of the full data. Formally, policies in Taypsi are encoded as *oblivious algebraic data types* (OADTs) [Ye and Delaware 2022], dependent types that take a public view as a parameter. The body of an OADT is the type of the private components of a data type, which are built using *oblivious* (i.e., secure) type formers, e.g., oblivious fixed-width integer ($\widehat{\mathbb{Z}}$) and oblivious sum ($\widehat{+}$).[3] An oblivious sum is similar to a standard sum, but both its tag and "payload" (i.e., component) are obfuscated, so that an attacker cannot distinguish between a left and right injection. Essentially, an OADT is a type-level function that maps the public view of a value to its *private representation*, i.e., the shape of its private component.

```
obliv list≤ (k : ℕ) =
  if k = 0 then 𝟙
  else 𝟙 +̂ ℤ̂ × list≤ (k−1)

obliv list= (k : ℕ) =
  if k = 0 then 𝟙
  else ℤ̂ × list= (k−1)
```

Fig. 2. Oblivious lists with maximum and exact length public views

Figure 2 shows two OADTs for the type `list`: $\widehat{\texttt{list}_{\leq}}$, whose public view is the maximum length of a list, and $\widehat{\texttt{list}_{=}}$, whose public view is the exact length. A public view can be any public data type. We say `list` is the *public type* or public counterpart of the OADTs $\widehat{\texttt{list}_{\leq}}$ and $\widehat{\texttt{list}_{=}}$. The key invariant of OADTs is that private values with the same public view are *indistinguishable* to an attacker, as their private representation is completely determined by the public view. For example, all private lists of type $\widehat{\texttt{list}_{\leq}}\ 2$ have the same private representation, regardless of the actual length of the list: $\widehat{\texttt{list}_{\leq}}\ 2 \equiv \mathbb{1} +̂ \widehat{\mathbb{Z}} \times (\mathbb{1} +̂ \widehat{\mathbb{Z}} \times \mathbb{1})$. Thus, an attacker cannot learn anything about the structure of an OADT, outside of what is entailed by its public view: an empty list, singleton list, or a list with two elements of type $\widehat{\texttt{list}_{\leq}}\ 2$ all appear the same to an attacker.

Conceptually, OADTs generalize the notion of secure fixed-width integers to secure structured data, as illustrated in Figure 3. Every fixed-width integer (of type $\mathbb{Z}$) can be sent to its secure value in $\widehat{\mathbb{Z}}$ by "encryption", and a secure integer can be converted back to $\mathbb{Z}$ by "decryption". In Taypsi, these conversion functions are called *section* (e.g., $\widehat{\mathbb{Z}}$`#s`) and *retraction* (e.g., $\widehat{\mathbb{Z}}$`#r`). The names reflect their expected semantics: applying retraction to the section of a value should produce the same value. Importantly, while the oblivious integer type $\widehat{\mathbb{Z}}$ does not appear to have much structure, it nonetheless has an implicit policy: the public view of an integer is its bit width. If we use 32-bit integers, for example, $\mathbb{Z}$ is the set of all integers whose bit width is 32, and $\widehat{\mathbb{Z}}$ is the set of their "encrypted" values, related by a pair of conversion functions. Similarly, $\widehat{\texttt{list}_{\leq}}\ \texttt{k}$ consists of the secure encodings of lists that have at most `k` elements. Like $\widehat{\mathbb{Z}}$, $\widehat{\texttt{list}_{\leq}}$ is equipped with a section function, $\widehat{\texttt{list}_{\leq}}$`#s`, and a retraction function, $\widehat{\texttt{list}_{\leq}}$`#r`, which convert public values of `list` to their oblivious counterparts and back. Crucially, just as the oblivious integers in $\widehat{\mathbb{Z}}$ are indistinguishable, the elements of $\widehat{\texttt{list}_{\leq}}\ \texttt{k}$ are also indistinguishable.

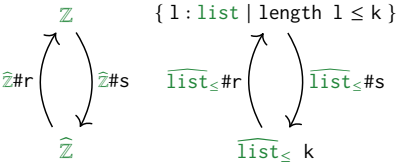

Fig. 3. Public and oblivious types

In the implementation of Taypsi, oblivious values are represented using arrays of secure values. To ensure that attackers cannot learn anything from the "memory layout" of an OADT value, the size of this array is the same for all values of a particular OADT. As an example, the encoding of the list `Cons 10 (Cons 20 Nil)` as an oblivious list of type $\widehat{\texttt{list}_{\leq}}\ 2$ is $\widehat{\texttt{inr}}$ `([10], `$\widehat{\texttt{inr}}$` ([20], ()))`, where $\widehat{\texttt{inr}}$

[3] By convention, we use $\hat{}$ to denote the oblivious version of something.

($\widehat{\texttt{inl}}$) is the oblivious counterpart of standard sum injection `inr` (`inl`). "Under the hood" this oblivious value is represented as an array holding four secure values; in the remainder of this section, we will informally write this value as `[Cons,10,Cons,20]`, where `[Cons]` is a synonym of the tag `[inr]` for readability. As another example, the empty list `Nil` is encoded as $\widehat{\texttt{inl}}$ `()`; it is also represented using an array with four elements, `[Nil,–,–,–]`, where the last three elements are dummy encrypted values (denoted by –). Our compiler uses the type of $\widehat{\texttt{inl}}$ to automatically pad this array with these values, in order to ensure that it is indistinguishable from other private values of $\widehat{\texttt{list}_{\leq}}$ `2`.

2.2 Enforcing Policies

Although using OADTs ensures that the representation of private information does not leak anything, both parties can still learn information by observing the control flow of a program. In order to protect private data from control flow channels, Taypsi provides oblivious operations to manipulate private data safely. One such operation is the atomic conditional `mux`,[4] a version of `if` that fully evaluates *both* branches before producing its final result. To understand why this evaluation strategy is necessary, consider the following example of what would happen if we were to evaluate `mux` like a standard `if` expression:

$$\texttt{mux [true] ([2]} \mathbin{\widehat{+}} \texttt{[3]) [4]} \longrightarrow \texttt{[2]} \mathbin{\widehat{+}} \texttt{[3]} \longrightarrow \texttt{[5]}$$

Even when all the private data (i.e., the integers in square brackets) is hidden, an attacker can infer that the private condition is `true` by observing that `mux` evaluates to the expression in its then branch.

With the secure semantics of `mux`, however, the following execution trace does not reveal any private information:

$$\begin{aligned}&\texttt{mux ([0]} \mathbin{\widehat{\leq}} \texttt{[1]) ([2]} \mathbin{\widehat{+}} \texttt{[3]) ([4]} \mathbin{\widehat{+}} \texttt{[5])} \longrightarrow \texttt{mux [true] ([2]} \mathbin{\widehat{+}} \texttt{[3]) ([4]} \mathbin{\widehat{+}} \texttt{[5])} \longrightarrow \\ &\texttt{mux [true] [5] ([4]} \mathbin{\widehat{+}} \texttt{[5])} \longrightarrow \texttt{mux [true] [5] [9]} \longrightarrow \texttt{[5]}\end{aligned}$$

Since both branches are evaluated regardless of the private condition, an attacker cannot infer that condition from this execution trace (again, all secure values are indistinguishable to an attacker). Thus, falsifying the condition produces an equivalent trace, modulo the encrypted data:

$$\begin{aligned}&\texttt{mux ([6]} \mathbin{\widehat{\leq}} \texttt{[1]) ([2]} \mathbin{\widehat{+}} \texttt{[3]) ([4]} \mathbin{\widehat{+}} \texttt{[5])} \longrightarrow \texttt{mux [false] ([2]} \mathbin{\widehat{+}} \texttt{[3]) ([4]} \mathbin{\widehat{+}} \texttt{[5])} \longrightarrow \\ &\texttt{mux [false] [5] ([4]} \mathbin{\widehat{+}} \texttt{[5])} \longrightarrow \texttt{mux [false] [5] [9]} \longrightarrow \texttt{[9]}\end{aligned}$$

The security-type system of Taypsi ensures all operations on private data are done in a way that does not reveal any private information, outside the public information specified by the policies.

2.3 *Automatically* Enforcing Policies

Users can directly implement privacy-preserving applications in Taypsi using OADTs and secure operations, but this requires manually instrumenting programs so that their control flow only depends on public information. Under this discipline, the implementation of a secure function intertwines program logic and privacy policies: the secure version of `filter` requires a different implementation depending on whether Alice is willing to share the exact length of her list, or an upper bound on that length. Taype [Ye and Delaware 2023] decouples these concerns by allowing programs to include unsafe computations and repairing unsafe computations at runtime, using a novel form of semantics called *tape semantics*. As an example of this approach, in Taype, a secure implementation of `filter` that allows Alice to only share an upper bound on the size of her list can be written as:

```
fn filter≤ : (k : ℕ) → list≤ k → ℤ̂ → list≤ k =
  λk xs ŷ ⇒ list≤#s k (filter (list≤#r k xs) (ℤ̂#r ŷ))
```

[4]The oblivious version of `if` in Taypsi is called `mux`, not $\widehat{\texttt{if}}$, in order to be consistent with the MPC literature.

The type signature of $\widehat{\texttt{filter}}_{\leq}$ specifies the policy it must follow. Intuitively, its implementation first "decrypts" the private inputs, applying the standard `filter` function to those values, and then "re-encrypts" the filtered list. In this example, the retractions of the private inputs $\widehat{\texttt{xs}}$ and $\widehat{\texttt{y}}$ are unsafe computations that would violate the desired policy if they were computed naively. Fortunately, using the tape semantics prevents this from occurring by deferring these computations until it is safe to do so. Less fortunately, the runtime overhead of dynamic policy enforcement makes it hard to scale private applications manipulating structured data. As one data point, the secure version of `filter` produced by Taype takes more than 5 seconds to run with an oblivious list $\widehat{\texttt{list}}_{\leq}$ with sixteen elements, and its performance grows exponentially worse as the number of elements increases.

To understand the source of this slowdown, consider a computation that filters a private list containing 10 and 20 with integer 15: $\widehat{\texttt{filter}}_{\leq}$ `2 [Cons,10,Cons,20] [15]`. The first step in evaluating this function is to compute $\widehat{\texttt{list}}_{\leq}$`#r 2 [Cons,10,Cons,20]`. Completely reducing this expression leaks information, so tape semantics instead stops evaluation after producing the following computation:[5]

```
mux [false] Nil (Cons (ℤ#r [10]) (mux [false] Nil (Cons (ℤ#r [20]) Nil)))
```

The two `[false]`s are the results of securely checking if the two constructors in the input list are `Nil`. Observe that evaluating either `mux` or $\widehat{\mathbb{Z}}$`#r` would reveal private information, so the evaluation of these operations is deferred. This delayed computation can be thought of as an "if-tree" whose internal nodes are the private conditions needed to compute the final results, and whose leaves hold the result of the computation along each corresponding control flow path. To make progress, tape semantics distributes the context surrounding a delayed computation, `filter` and then $\widehat{\texttt{list}}_{\leq}$`#s` in this example, into each of its leaves; having done so, those leaves can be further evaluated. Importantly, in our example, the leaves of this if-tree are eventually re-encrypted using $\widehat{\texttt{list}}_{\leq}$`#s`. The tape semantics does so in a secure way, so that $\widehat{\mathbb{Z}}$`#r [10]` becomes `[10]` again, and each result list is converted to a secure value of the expected OADT. Once the branches of a `mux` node have been reduced to oblivious values of the same type, the node itself can be securely reduced using the secure semantics of `mux`. Unfortunately, the if-tree produced by the tape semantics can grow exponentially large before its `mux` nodes can be reduced. For example, after applying `filter` to the if-tree produced by $\widehat{\texttt{list}}_{\leq}$`#r`, the resulting if-tree has a leaf corresponding to every possible list that `filter` could produce; the number of these leaves is exponential in the maximum length of the input list. As any surrounding computation, i.e., $\widehat{\texttt{list}}_{\leq}$`#s` in our example, can be distributed to each of these leaves, an exponential number of computations may need to be performed before the if-tree can be collapsed.

To remedy these limitations, this paper proposes to instead compile an insecure program into a secure version that *statically* enforces a specified policy. To do so, we extend Taype, the secure language of Ye and Delaware [2023] with Ψ-*types*, a form of *dependent sums* (or dependent pairs) that packs public views and the oblivious data into a uniform representation. For example, $\Psi\widehat{\texttt{list}}_{\leq}$ is the oblivious list $\widehat{\texttt{list}}_{\leq}$ with its public view: `⟨2, inr ([10], inr ([20], ()))⟩` and `⟨2, inl ()⟩` are elements of type $\Psi\widehat{\texttt{list}}_{\leq}$, corresponding to the examples in the previous section. The first component of this pair-like syntax is a public view and the second component is an OADT whose public view is exactly the first component. This allows users to again derive a private filter function from its type signature:

```
fn filter≤ : Ψlist≤ → ℤ → Ψlist≤ = %lift filter
```

Users no longer need to explicitly provide the public views for either the output or any intermediate subroutines: both are automatically inferred. As a result, the policy specification of $\widehat{\texttt{filter}}_{\leq}$ more directly corresponds to the type signature of `filter`. In addition, specifying policies using Ψ-types

[5]We refer interested readers to Ye and Delaware [2022, 2023] for a complete accounting of tape semantics.

avoids mistakes in the supplied public views: using Taype, if the programmer mistakenly specifies the return type $\widehat{\texttt{list}_\leq}$ `(k-1)` for a secure version of `filter`, for example, the resulting implementation may truncate the last element of the result list. A keyword `%lift` is used to translate the standard non-secure function `filter` to a private version that respects the policy specification.

To understand how this translation works, consider a naive approach where each algebraic data type (ADT) is thought of as an abstract interface, whose operations correspond to the *introduction* and *elimination* forms of the algebraic data type[6]. An ADT, e.g., `list`, as well as any corresponding Ψ-type, e.g., $\Psi\widehat{\texttt{list}_\leq}$ and $\Psi\widehat{\texttt{list}_=}$, are implementations or *instances* of this interface. For example, an interface for list operations is:

```
ListLike t = {Nil : 𝟙 → t; Cons : ℤ̂ × t → t; match : t → (𝟙 → α) → (ℤ̂ × t → α) → α}
```

As long as $\Psi\widehat{\texttt{list}_\leq}$ and $\Psi\widehat{\texttt{list}_=}$ implement this interface, we could straightforwardly translate `filter` to a secure version:

```
fn filter̂≤ : Ψlist̂≤ → ℤ̂ → Ψlist̂≤ = λxs y ⇒
  list̂≤#match xs (λ_ ⇒ list̂≤#Nil ())
    (λ(x, xs′) ⇒
      mux (x ≲̂ y) (list̂≤#Cons x (filter̂≤ xs′ y))
          (filter̂≤ xs′ y))
```

```
fn filter̂= : Ψlist̂= → ℤ̂ → Ψlist̂= = λxs y ⇒
  list̂=#match xs (λ_ ⇒ list̂=#Nil ())
    (λ(x, xs′) ⇒
      mux (x ≲̂ y) (list̂=#Cons x (filter̂= xs′ y))
          (filter̂= xs′ y))
```

This strategy does not rely on unsafe retractions like $\widehat{\texttt{list}_\leq}$`#r`, as private data always remains in its secure form, eliminating the need to defer unsafe computations, which is the source of exponential slowdowns in Taype. Unfortunately, there are several obstacles to directly implementing this strategy. First, an ADT and an OADT may not agree on the type signatures of the abstract interface. `ListLike` fixes the argument types of operations like `Cons` and `match`, meaning that `list` is not an instance of this abstract interface, despite `list` being a very reasonable (albeit very permissive) policy! In general, different OADTs may only be able to implement operations with specific signatures. Second, a private function may involve a mixture of oblivious types. Thus, some functions may need to coerce from one type to a "more" secure version. For example, if the policy of $\widehat{\texttt{filter}_\leq}$ is $\Psi\widehat{\texttt{list}_\leq} \to \mathbb{Z} \to \Psi\widehat{\texttt{list}_\leq}$, its second argument `y` will need to be converted to $\widehat{\mathbb{Z}}$ in order to evaluate `x` $\widehat{\leq}$ `y`. A secure list that discloses its exact length may similarly need to be converted to one disclosing its maximum length. Third, this naive translation results in ill-typed programs, because the branches of a `mux` may have mismatched public views. In $\widehat{\texttt{filter}_\leq}$, for example, the branches of `mux` may evaluate to ⟨`2, [Cons,10,Cons,20]`⟩ and ⟨`1, [Cons,20]`⟩, respectively. Thus, Taypsi's secure type system will (rightly) reject $\widehat{\texttt{filter}_\leq}$ as leaky. Lastly, the signatures that should be ascribed to any subsidiary function calls may not be obvious. Consider the following client of `filter`:

```
fn filter5 : list → list = λxs ⇒ filter xs 5
```

If `filter5` is given a signature $\Psi\widehat{\texttt{list}_\leq} \to \Psi\widehat{\texttt{list}_\leq}$, we would like to use a secure version of the `filter` function with the type $\Psi\widehat{\texttt{list}_\leq} \to \mathbb{Z} \to \Psi\widehat{\texttt{list}_\leq}$, as the threshold argument is publicly known. In general, a function may have many private versions, and we should infer which version to use at each callsite: a recursive function may even recursively call a different "version" of itself.

To solve these challenges, we generalize the abstract interface described above into a set of more flexible structures, which we collectively refer to as Ψ-structures (Section 4). Intuitively, each category of Ψ-structures solves one of the challenges described above. Our translation algorithm (Section 5) generates a set of typing *constraints* for the intermediate expressions in a program. These constraints are then solved using the set of available Ψ-structures, resulting in multiple private versions of the necessary functions and ruling out the infeasible ones, e.g., $\widehat{\texttt{filter}_=}$.

[6]As Taypsi already supports general recursion, we use pattern matching instead of recursion schemes as our elimination forms.

OADT-structure

```
fn list≤#s : (k : ℕ) → list → list≤ k =
  λk xs ⇒
    if k = 0 then ()
    else match xs with Nil ⇒ inl ()
         | Cons x xs′ ⇒
           inr (ℤ#s x, list≤#s (k−1) xs′)
fn list≤#view : list → ℕ = length
```

```
unsafe fn list≤#r : (k : ℕ) → list≤ k → list =
  λk ⇒
    if k = 0 then λ_ ⇒ Nil
    else λxs ⇒
      match xs with inl _ ⇒ Nil
      | inr (x, xs′) ⇒
        Cons (ℤ#r x) (list≤#r (k−1) xs′)
```

Intro/elim-structure

```
fn list≤#Nil : 𝟙 → Ψlist≤ = λ_ ⇒ ⟨0, ()⟩

fn list≤#Cons : ℤ × Ψlist≤ → Ψlist≤ =
  λ(x, ⟨k, xs⟩) ⇒
    ⟨k+1, inr (x, xs)⟩
```

```
fn list≤#match :
  Ψlist≤ → (𝟙 → α) → (ℤ × Ψlist≤ → α) → α =
  λ⟨k, xs⟩ f1 f2 ⇒
    (if k = 0 then λ_ ⇒ f1 ()
     else λxs ⇒
       match xs with inl _ ⇒ f1 ()
       | inr (x, xs′) ⇒ f2 (x, ⟨k−1, xs′⟩)
         : list≤ k → α) xs
```

Join-structure

```
fn list≤#join : ℕ → ℕ → ℕ = max

fn list≤#reshape : (k : ℕ) → (k′ : ℕ) → list≤ k → list≤ k′ = λk k′ ⇒
  if k′ = 0 then λ_ ⇒ ()
  else if k = 0 then λ_ ⇒ inl ()
       else λxs ⇒ match xs with inl _ ⇒ inl ()
                  | inr (x, xs′) ⇒ inr (x, list≤#reshape (k−1) (k′−1) xs′)
```

Fig. 4. Ψ-structures of $\widehat{\texttt{list}_{\leq}}$

Figure 4 presents the methods of each category of Ψ-structures of $\widehat{\texttt{list}_{\leq}}$. The first two methods, $\widehat{\texttt{list}_{\leq}}$#s and $\widehat{\texttt{list}_{\leq}}$#r, are its section and retraction functions, belonging to the OADT-structure category. Unlike Taype, these two functions are not directly used to derive secure implementations of functions. In fact, our type system guarantees that retraction functions are never used in a secure computation, because Taypsi does not rely on tape semantics to repair unsafe computation (the `unsafe fn` keyword tells our type checker that $\widehat{\texttt{list}_{\leq}}$#r is potentially leaky). Our implementation of Taypsi exposes section and retraction functions as part of the API of the secure library it generates, however, so that client programs can conceal their private input and reveal the output of secure computations. This structure also includes a `view` method, which our translation uses to select the public view needed to safely convert a `list` into a $\Psi\widehat{\texttt{list}_{\leq}}$. Figure 4 does not show coercion methods, but the programmers can define a coercion from $\Psi\widehat{\texttt{list}_{=}}$ to $\Psi\widehat{\texttt{list}_{\leq}}$, for example.

The next set of methods belong to the intro-structure and elim-structure category. These introduction ($\widehat{\texttt{list}_{\leq}}$#Nil and $\widehat{\texttt{list}_{\leq}}$#Cons) and elimination ($\widehat{\texttt{list}_{\leq}}$#match) methods construct and destruct private list, respectively. As we construct and manipulate data, these methods build the private version, calculate its public view, and record that view in Ψ-types. Their type signatures are specified by the programmers, as long as the signatures are *compatible* with $\mathbb{Z} \times$ `list` (Section 4).

The `join` and `reshape` methods in the join-structure category enable translated programs to include private conditionals whose branches return OADT values with different public views. As an example, consider the following private conditional whose branches have Ψ-types:

```
mux [true] ⟨2, [Cons,10,Cons,20]⟩ ⟨1, [Cons,20]⟩
```

To build a version of this program that does not reveal `[true]`, Taypsi uses `join` to calculate a common public view that "covers" both branches. In this example, $\widehat{\texttt{list}}_{\leq}$`#join` chooses a public view of 2, as a list with at most one element also has at most two elements. Our translation then uses the `reshape` method to convert both branches to use this common public view. In our example, `[Cons,20]`, an oblivious list of maximum length 1, is converted into the list `[Cons,20,Nil,–]`, which has maximum length 2. Since both branches in the resulting program have the same public view, it is safe to evaluate `mux`: the resulting list is equivalent to ⟨`2, mux [true] [Cons,10,Cons,20] [Cons,20,Nil,–]`⟩. As we will see later, not all OADTs admit join structures, e.g., $\widehat{\texttt{list}}_{=}$, but our translation generates constraints that take advantage of any that are available, failing when these constraints cannot be resolved in a way that guarantees security. Note that these two methods are key to avoiding the slowdown exhibited by Taype's enforcement strategy: they allow functions that may return different private representations to be *eagerly* evaluated, instead of being lazily deferred in a way that requires an exponential number of subcomputations to resolve.

In summary, to develop a secure application in Taypsi, programmers first implement its desired functionality, e.g., `filter`, in the public fragment of Taypsi, independently of any particular privacy policy. Policies are separately defined as oblivious algebraic data types, e.g., $\widehat{\texttt{list}}_{\leq}$, and their Ψ-structures. Users can then automatically derive a secure version of their application by providing the desired policy in the form of a type signature involving Ψ-types, relying on Taypsi's compiler to produce a privacy-preserving implementation. The type system of Taypsi, like Taype's, provides a strong security guarantee in the form of an obliviousness theorem (Theorem 3.1). This obliviousness theorem is a variant of noninterference [Goguen and Meseguer 1982], and ensures that well-typed programs in Taypsi are *secure by construction*: no private information can be inferred even by an attacker capable of observing every state of a program's execution. Our compilation algorithm is further guaranteed to generate a secure implementation that preserves the behavior of the original program (Theorem 4.8).[7]

The following three sections formally develop the language Taypsi, the Ψ-structures, and our translation algorithm.

3 TAYPSI, FORMALLY

This section presents $\lambda_{\mathsf{OADT}\Psi}$, the core calculus for secure computation that we will use to explain our translation. This calculus extends the existing λ_{OADT} [Ye and Delaware 2022] calculus with Ψ-types, and uses ML-style ADTs in lieu of explicit `fold` and `unfold` operations.[8]

3.1 Syntax

Figure 5 presents the syntax of $\lambda_{\mathsf{OADT}\Psi}$. Types and expressions are in the same syntax class, as $\lambda_{\mathsf{OADT}\Psi}$ is dependently typed, but we use $\mathtt{e}$ for expressions and τ for types when possible. A $\lambda_{\mathsf{OADT}\Psi}$ program consists of a set of *global definitions* of data types, functions and oblivious types. Definitions in each of these classes are allowed to refer to themselves, permitting recursive types and general recursion in both function and oblivious type definitions. We use `x` for variable names, `C` for constructor names, `T` for type names, and $\widehat{\mathtt{T}}$ for oblivious type names. Each constructor of an ADT definition takes exactly one argument, but this does not harm expressivity: this argument is $\mathbb{1}$ for constructors that take no arguments, e.g., `Nil`, and a tuple of types for constructors that have more than one argument, e.g., `Cons` takes an argument of type $\mathbb{Z} \times$ `list`.

[7] Taypsi's formal guarantees (Section 4.7) do not cover equi-termination of the source and target programs: when the public view lacks sufficient information to bound the computation of the original program, the secure version will not terminate, in order to avoid leaking information through its termination behavior.

[8] For simplicity, $\lambda_{\mathsf{OADT}\Psi}$ does not include public sums and oblivious integers, which are straightforward to add.

e,τ ::=				Expressions
| 𝟙 | 𝔹 | $\widehat{\mathbb{B}}$ | τ×τ | τ$\widehat{+}$τ	public & oblivious types	| Ψ$\widehat{\tau}$	Ψ-type
| Π x:τ,τ | λx:τ⇒e	dependent function	| (e,e) | ⟨e,e⟩	pair & Ψ-pair
| () | b | x | T	literals & variables	| π_1 e | π_2 e	product and Ψ-type projection
| let x = e in e	let binding	| $\widehat{\texttt{inl}}$<τ> e | $\widehat{\texttt{inr}}$<τ> e	oblivious sum injection
| e e | C e | $\widehat{\texttt{T}}$ e	applications	| $\widehat{\texttt{match}}$ e with x⇒e|x⇒e	oblivious sum elimination
| if e then e else e	conditional	| match e with $\overline{\texttt{C x⇒e}}$	ADT elimination
| mux e e e	oblivious conditional	| $\widehat{\mathbb{B}}$#s e	boolean section

D ::=		Global Definitions	$\widehat{\omega}$ ::= 𝟙 | $\widehat{\mathbb{B}}$ | $\widehat{\omega}$×$\widehat{\omega}$ | $\widehat{\omega}\widehat{+}\widehat{\omega}$	Obliv. Type Values
| data T = $\overline{\texttt{C τ}}$	algebraic data type	$\hat{v}$ ::= () | [b] | ($\hat{v}$,$\hat{v}$)	Obliv. Values
| fn x:τ = e	(recursive) function	| [inl<$\widehat{\omega}$> $\hat{v}$] | [inr<$\widehat{\omega}$> $\hat{v}$]	
| obliv $\widehat{\texttt{T}}$ (x:τ) = τ	(recursive) obliv. type	v ::= $\hat{v}$ | b | (v,v) | ⟨v,v⟩ | λx:τ⇒e | C v	Values

Fig. 5. $\lambda_{\text{OADT}\Psi}$ syntax with extensions to λ_{OADT} highlighted

In addition to standard types and dependent function types (Π), $\lambda_{\text{OADT}\Psi}$ includes oblivious booleans ($\widehat{\mathbb{B}}$) and oblivious sum types ($\widehat{+}$). The elimination forms of these types are oblivious conditionals mux and oblivious case analysis $\widehat{\texttt{match}}$, respectively. The branches of both expressions must be private and each branch has to be fully evaluated before the expression can take an atomic step to a final result. Boolean section $\widehat{\mathbb{B}}$#s is a primitive operation that "encrypts" a boolean expression to an oblivious version. Oblivious injection $\widehat{\texttt{inl}}$ and $\widehat{\texttt{inr}}$ are the oblivious counterparts of the standard constructors for sums. Other terms are mostly standard, although let bindings (let), conditionals (if) and pattern matching (match) are allowed to return a type, as $\lambda_{\text{OADT}\Psi}$ supports type-level computation.

The key addition over λ_{OADT} is the Ψ-*type*, Ψ$\widehat{\tau}$. It is constructed from a pair expression $\langle\cdot,\cdot\rangle$ that packs the public view and the oblivious data together, and has the same eliminators π_1 and π_2 as normal products. As an example, ⟨3, $\widehat{\texttt{list}_{\leq}}$#s 3 (Cons 1 Nil)⟩ creates a Ψ-pair of type Ψ$\widehat{\texttt{list}_{\leq}}$ with public view 3, using the section function from Figure 4. Projecting out the second component of a pair using π_2 produces a value of type $\widehat{\texttt{list}_{\leq}}$ 3. A Ψ-type is essentially a dependent sum type (Σx:τ,$\widehat{\texttt{T}}$ x), with the restriction that τ is the public view of $\widehat{\texttt{T}}$, and that $\widehat{\texttt{T}}$ x is an oblivious type.

Since $\lambda_{\text{OADT}\Psi}$ has type-level computation, oblivious types have normal forms; oblivious type values ($\widehat{\omega}$) are essentially polynomials formed by primitive oblivious types. We also have the oblivious values of oblivious boolean and sum type. Note that these "boxed" values only appear at runtime, our semantics use these to model encrypted booleans and tagged sums.

3.2 Semantics

Figure 6 shows a selection of the small-step semantics rules of $\lambda_{\text{OADT}\Psi}$ (the full rules are included in the appendix), with judgment $\Sigma \vdash e \longrightarrow e'$. The global context Σ is a map from names to a global definition, which is elided for brevity as it is fixed in these rules. The semantics of $\lambda_{\text{OADT}\Psi}$ is similar to λ_{OADT}, with the addition of S-PsiProj$_1$ (and S-PsiProj$_2$) to handle the projection of dependent pairs, which is simply the same as normal projection. S-Ctx reduces subterms according to the evaluation contexts defined in Figure 6. The first few contexts take care of the type-level reduction of product and oblivious sum type. The type annotation of oblivious injection $\widehat{\texttt{inl}}$ and $\widehat{\texttt{inr}}$ is reduced to a type value first, before reducing the payload. The evaluation contexts for mux capture the intuition that all components of a private conditional have to be normalized to values first to avoid leaking the private condition through control flow channels.

S-OMatchL (and S-OMatchR) evaluates a pattern matching expression for oblivious sums. Similar to mux, oblivious pattern matching needs to ensure the reduction does not reveal private information about the discriminee, e.g., whether it is the left injection or right injection. To do so, we reduce a $\widehat{\texttt{match}}$ to an oblivious conditional that uses the private tag. The pattern variable in

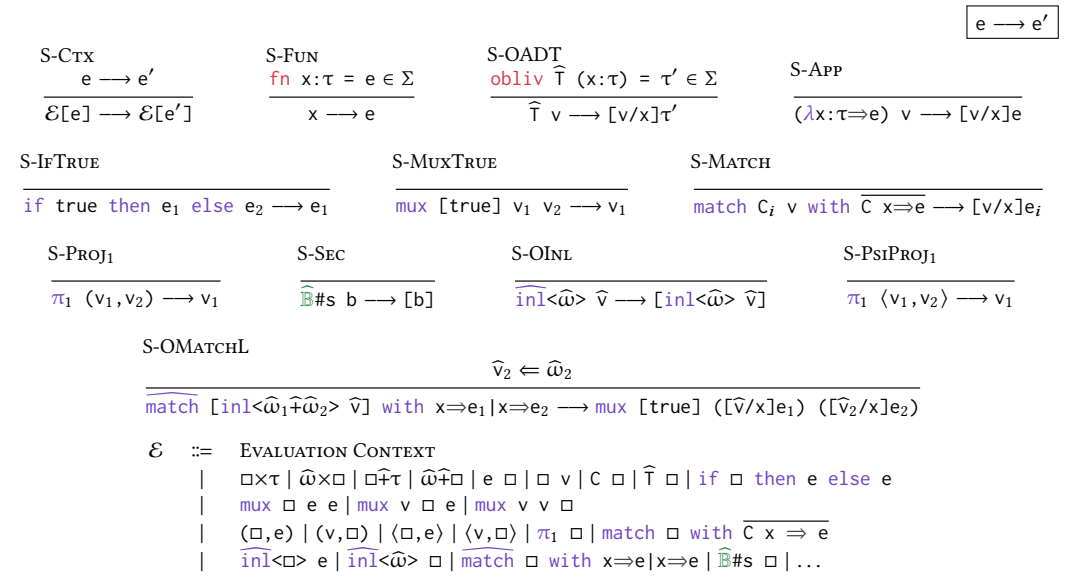

Fig. 6. Selected small-step semantics rules of $\lambda_{\mathsf{OADT}\Psi}$

the "correct" branch is of course instantiated by the payload in the discriminee, while the pattern variable in the "wrong" branch is an arbitrary value of the corresponding type, synthesized from the judgment $\widehat{v} \Leftarrow \widehat{\omega}$, whose definition is in appendix. When evaluating a $\widehat{\texttt{match}}$ statement whose discriminee is $\texttt{[inl<}\widehat{\mathbb{B}}\texttt{+}\widehat{\mathbb{B}}\texttt{×}\widehat{\mathbb{B}}\texttt{> [true]]}$, the pattern variable in the second branch can be substituted by $\texttt{([true],[true])}$, $\texttt{([false],[true])}$, or any other pair of oblivious booleans.

3.3 Type System

Similar to λ_{OADT}, types in $\lambda_{\mathsf{OADT}\Psi}$ are classified by *kinds* which specify how protected a type is, in addition to ensuring the types are well-formed. For example, an oblivious type, e.g., $\widehat{\mathbb{B}}$, kinded by $*^{\mathsf{O}}$, can be used as branches of an oblivious conditional, but not as a public view, which can only be kinded by $*^{\mathsf{P}}$. A mixed kind $*^{\mathsf{M}}$ is used to classify function types and types that consist of both public and oblivious components, e.g., $\mathbb{B}\times\widehat{\mathbb{B}}$. A type with a mixed kind cannot be used as a public view or in private context.

The type system of $\lambda_{\mathsf{OADT}\Psi}$ is defined by a pair of typing and kinding judgments, $\Sigma;\Gamma \vdash e : \tau$ and $\Sigma;\Gamma \vdash \tau :: \kappa$, with global context Σ (which is again elided for brevity) and the standard typing context Γ. Figure 7 presents a subset of our typing and kinding rules; the full rules are in appendix.

The security type system [Sabelfeld and Myers 2003] of $\lambda_{\mathsf{OADT}\Psi}$ enforces a few key invariants. First, oblivious types can only be constructed from oblivious types, which is enforced by the kinding rules, such as K-OSum. Otherwise, the attacker could infer the private tag of an oblivious sum, e.g., $\mathbb{B}\widehat{+}\mathbb{1}$, by observing its public payload. Second, oblivious operations, e.g., $\texttt{mux}$, require their subterms to be oblivious, to avoid leaking private information via control flow channels. T-Mux, for example, requires both branches to be typed by an oblivious type, otherwise an attacker may infer the private condition by observing the result, as in $\texttt{mux [true] true false}$. Third, type-level computation is only defined for oblivious types and cannot depend on private information. Thus, K-If requires both branches to have oblivious kinds, and the condition to be a public boolean. The type $\texttt{mux [true]}\ \mathbb{1}\ \widehat{\mathbb{B}}$ is ill-typed, since the "shape" of the data reveals the private condition.

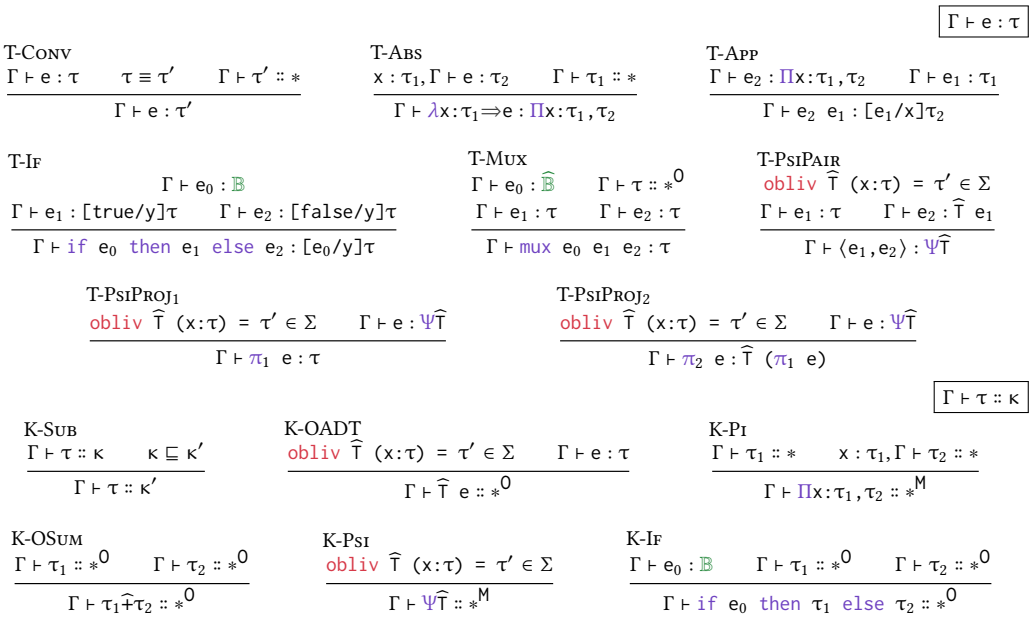

Fig. 7. Selected typing and kinding rules of $\lambda_{\mathsf{OADT}\Psi}$

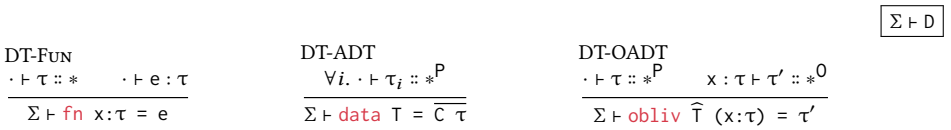

Fig. 8. $\lambda_{\mathsf{OADT}\Psi}$ global definitions typing

The typing rules for Ψ-types are defined similarly to the rules of standard dependent sums. T-PsiPair introduces a dependent pair, where the type of the second component depends on the first component. In contrast to standard dependent sum type, Ψ-type has the restriction that the first component must be public, and the second component must be oblivious. This condition is implicitly enforced by the side condition that $\hat{\tau}$ is an OADT with public view type τ. Figure 8 shows the typing rules for global definitions; DT-OADT prescribes exactly this restriction. The rules for the first and second projection of Ψ-type, T-PsiProj$_1$ and T-PsiProj$_2$, are very similar to the corresponding rules for standard dependent sum types. Observe that a Ψ-type always has mixed kind, as in K-Psi, because it consists of both public and oblivious components.

T-Conv allows conversion between equivalent types, such as `if true then` $\hat{\mathbb{B}}$ `else 𝟙` and $\hat{\mathbb{B}}$. The equivalence judgment $\tau \equiv \tau'$ is defined by a set of *parallel reduction* rules, which we elide here. The converted type is nonetheless required to be well-kinded.

Note that these rules cannot be used to type check retraction functions, e.g., $\widehat{\texttt{list}_{\leq}}$`#r` from Figure 4, and for good reason: these functions reveal private information. Nevertheless, we still want to check that these sorts of "leaky" functions have standard type safety properties, i.e., progress and preservation. To do so, we use a version of these rules that simply omit some security-related side-conditions about oblivious kinding: removing $\Gamma \vdash \tau :: *^0$ from T-Mux allows the branches of a `mux` to disclose the private condition, for example. The implementation of Taypsi's type checker

uses a "mode" flag to indicate whether security-related side-conditions should be checked. Our implementation ensures that secure functions never use any leaky functions.

3.4 Metatheory

With our addition of Ψ-types, $\lambda_{\mathsf{OADT}\Psi}$ enjoys the standard type safety properties (i.e., progress and preservation), and, more importantly, the same security guarantees as λ_{OADT}:

Theorem 3.1 (Obliviousness). *If* $\mathtt{e}_1 \approx \mathtt{e}_2$ *and* $\cdot \vdash \mathtt{e}_1 : \tau_1$ *and* $\cdot \vdash \mathtt{e}_2 : \tau_2$, *then*
(1) $\mathtt{e}_1 \longrightarrow^n \mathtt{e}_1'$ *if and only if* $\mathtt{e}_2 \longrightarrow^n \mathtt{e}_2'$ *for some* $\mathtt{e}_2'$.
(2) if $\mathtt{e}_1 \longrightarrow^n \mathtt{e}_1'$ *and* $\mathtt{e}_2 \longrightarrow^n \mathtt{e}_2'$, *then* $\mathtt{e}_1' \approx \mathtt{e}_2'$.

Here, $\mathtt{e} \approx \mathtt{e}'$ means the two expressions are indistinguishable, i.e., they only differ in their unobservable oblivious values, and $\mathtt{e} \longrightarrow^n \mathtt{e}'$ means $\mathtt{e}$ reduces to $\mathtt{e}'$ in exactly n steps. This obliviousness theorem provides a strong security guarantee: well-typed programs that are indistinguishable produce traces that are pairwise indistinguishable. In other words, an attacker cannot infer any private information even by observing the execution trace of a program. All these results are mechanized in the Coq theorem prover, including the formalization of the core calculus and the proofs of soundness and obliviousness theorems.

4 Ψ-STRUCTURES AND DECLARATIVE LIFTING

While our secure language makes it possible to encode structured data and privacy policies, and use them in a secure way, it does not quite achieve our main goal yet, i.e., to decouple privacy policies and programmatic concerns. To do so, we allow the programmers to implement the functionality of their secure application in a conventional way, that is using only the public, nondependent fragment of Taypsi. We make this fragment explicit by requiring such programs to have *simple types*, denoted by η, defined in Figure 9. For example, `filter` has simple type $\mathtt{list} \to \mathbb{Z} \to \mathtt{list}$. Programs of simple types are the source programs to our lifting process that translates them to a private version against a policy, which stipulates the public information allowed to disclose in the program input and output. This policy on private functionality is specified by a *specification type*, denoted by θ, defined also in Figure 9. For example, $\widehat{\mathtt{filter}}_{\leq}$ has specification type $\widehat{\Psi\mathtt{list}}_{\leq} \to \widehat{\mathbb{Z}} \to \widehat{\Psi\mathtt{list}}_{\leq}$. Note that dependent types are not directly allowed in specifications, they are instead encapsulated in Ψ-types. Simple types and specification types are additionally required to be well-kinded under empty local context, i.e., all ADTs and OADTs appear in them are defined.

However, not all specification types are valid with respect to a simple type. It is nonsensical to give $\widehat{\mathtt{filter}}$ the specification type $\widehat{\mathbb{Z}} \to \widehat{\mathbb{B}}$, for example. The specification types should still correspond to the simple types in some way: the specification type corresponding to `list` should at least be "list-like". This correspondence is formally captured in the erasure function in Figure 9, which maps a specification type to the "underlying" simple type. For example, $\widehat{\Psi\mathtt{list}}_{\leq}$ is erased to `list`. This function clearly induces an equivalence relation: the erasure $\lfloor\theta\rfloor$ is the representative of the equivalence class. We call this equivalence class a *compatibility class*, and say two types are *compatible* if they belong to the same compatibility class. For example, `list`, $\widehat{\Psi\mathtt{list}}_{\leq}$ and $\widehat{\Psi\mathtt{list}}_{=}$ are in the same compatibility class $[\mathtt{list}]$. This erasure operation is straightforwardly extended to typing contexts, $\lfloor\Gamma\rfloor$, by erasing every specification type in Γ and leaving other types untouched.

Our translation transforms source programs with simple types into target programs with the desired (compatible) specification types. As mentioned in Section 2, this *lifting* process depends on a set of *ψ-structures* which explain how to translate certain operations associated with an OADT.

4.1 OADT Structures

Every global OADT definition $\widehat{\mathtt{T}}$ must be equipped with an *OADT-structure*, defined below.

$\boxed{\lfloor\theta\rfloor}$

SIMPLE TYPES

$\eta \quad ::= \quad \mathbb{1} \mid \mathbb{B} \mid \mathsf{T} \mid \eta\times\eta \mid \eta\to\eta$

$\lfloor\mathbb{1}\rfloor = \mathbb{1} \qquad \lfloor\mathbb{B}\rfloor = \lfloor\widehat{\mathbb{B}}\rfloor = \mathbb{B} \qquad \lfloor\mathsf{T}\rfloor = \mathsf{T}$ where T is an ADT

SPECIFICATION TYPES

$\theta \quad ::= \quad \mathbb{1} \mid \mathbb{B} \mid \widehat{\mathbb{B}} \mid \mathsf{T} \mid \Psi\widehat{\mathsf{T}} \mid \theta\times\theta \mid \theta\to\theta$

$\lfloor\Psi\widehat{\mathsf{T}}\rfloor = \mathsf{T}$ where $\widehat{\mathsf{T}}$ is an OADT for T

$\lfloor\theta\times\theta\rfloor = \lfloor\theta\rfloor\times\lfloor\theta\rfloor \qquad \lfloor\theta\to\theta\rfloor = \lfloor\theta\rfloor\to\lfloor\theta\rfloor$

Fig. 9. Simple types, specification types and erasure

Definition 4.1 (OADT-structure). An OADT-structure of an OADT $\widehat{\mathsf{T}}$, with public view type τ, consists of the following (TAYPSI) type and functions:

- A public type $\mathsf{T} :: *^{\mathsf{P}}$, which is the public counterpart of $\widehat{\mathsf{T}}$. We say $\widehat{\mathsf{T}}$ is an OADT for T.
- A section function $\mathtt{s} : \Pi\mathtt{k}{:}\tau,\mathsf{T}\to\widehat{\mathsf{T}}\ \mathtt{k}$, which converts a public type to its oblivious counterpart.
- A retraction function $\mathtt{r} : \Pi\mathtt{k}{:}\tau,\widehat{\mathsf{T}}\ \mathtt{k}\to\mathsf{T}$, which converts an oblivious type to its public version.
- A public view function $\nu : \mathsf{T}\to\tau$, which creates a valid view of the public type.
- A binary relation $\preccurlyeq$ over values of types T and τ; $\mathtt{v} \preccurlyeq \mathtt{k}$ reads as $\mathtt{v}$ has public view $\mathtt{k}$, or $\mathtt{k}$ is a valid public view of $\mathtt{v}$.

These operations are required to satisfy the following axioms:

- (A-O_1) $\mathtt{s}$ and $\mathtt{r}$ are a valid section and retraction, i.e., $\mathtt{r}$ is a left-inverse for $\mathtt{s}$, given a valid public view: for any values $\mathtt{v} : \mathsf{T}$, $\mathtt{k} : \tau$ and $\widehat{\mathtt{v}} : \widehat{\mathsf{T}}\ \mathtt{k}$, if $\mathtt{v} \preccurlyeq \mathtt{k}$ and $\mathtt{s}\ \mathtt{k}\ \mathtt{v} \longrightarrow^* \widehat{\mathtt{v}}$, then $\mathtt{r}\ \mathtt{k}\ \widehat{\mathtt{v}} \longrightarrow^* \mathtt{v}$.
- (A-O_2) the result of $\mathtt{r}$ always has valid public view: $\mathtt{r}\ \mathtt{k}\ \widehat{\mathtt{v}} \longrightarrow^* \mathtt{v}$ implies $\mathtt{v} \preccurlyeq \mathtt{k}$ for all values $\mathtt{k} : \tau$, $\widehat{\mathtt{v}} : \widehat{\mathsf{T}}\ \mathtt{k}$ and $\mathtt{v} : \mathsf{T}$.
- (A-O_3) ν produces a valid public view: $\nu\ \mathtt{v} \longrightarrow^* \mathtt{k}$ implies $\mathtt{v} \preccurlyeq \mathtt{k}$, given any values $\mathtt{v} : \mathsf{T}$ and $\mathtt{k} : \tau$.

For example, $\widehat{\mathtt{list}_{\leq}}$ is equipped with the OADT-structure with the public type $\mathtt{list}$, section function $\widehat{\mathtt{list}_{\leq}}\#\mathtt{s}$, retraction function $\widehat{\mathtt{list}_{\leq}}\#\mathtt{r}$ and view function $\widehat{\mathtt{list}_{\leq}}\#\mathtt{view}$, all of which are shown in Figure 4. TAYPSI users do not need to explicitly give the public type of an OADT-structure, as it can be inferred from the types of the other functions. The binary relation $\preccurlyeq$ is only used in the proof of correctness of our translation, so TAYPSI users can also elide it. In the case of $\widehat{\mathtt{list}_{\leq}}$, $\preccurlyeq$ simply states the length of the list is no larger than the public view.

4.2 Join Structures

In order for Ψ-types to be flexibly used in the branches of secure control flow structures, our translation must be able to find a common public view for both branches, and to convert an OADT to use this view. To do so, an OADT can *optionally* be equipped with a *join-structure*.

Definition 4.2 (join-structure). A join-structure of an OADT $\widehat{\mathsf{T}}$ for T, with public view type τ, consists of the following operations:

- A binary relation $\sqsubseteq$ on τ, used to compare two public views.
- A join function $\sqcup : \tau\to\tau\to\tau$, which computes an upper bound of two public views[9].
- A reshape function $\wr : \Pi\mathtt{k}{:}\tau,\Pi\mathtt{k}'{:}\tau,\widehat{\mathsf{T}}\ \mathtt{k}\to\widehat{\mathsf{T}}\ \mathtt{k}'$, which converts an OADT to one with a different public view.

such that:

- (A-R_1) $\sqsubseteq$ is a partial order on τ.
- (A-R_2) the join function produces an upper bound: given values $\mathtt{k}_1$, $\mathtt{k}_2$ and $\mathtt{k}$ of type τ, if $\mathtt{k}_1\sqcup\mathtt{k}_2 \longrightarrow^* \mathtt{k}$, then $\mathtt{k}_1 \sqsubseteq \mathtt{k}$ and $\mathtt{k}_2 \sqsubseteq \mathtt{k}$.

[9]It is a bit misleading to call the operation $\sqcup$ "join", as it only computes an upper bound, not necessarily the lowest one. However, it *should* compute a supremum for performance reasons: intuitively, larger public view means more padding.

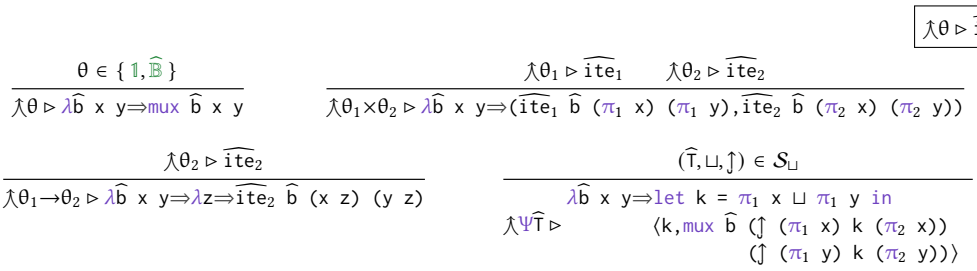

Fig. 10. Mergeability

- (A-R$_3$) the validity of public views is monotone with respect to the binary relation $\sqsubseteq$: for any values $\texttt{v} : \mathsf{T}$, $\texttt{k} : \tau$ and $\texttt{k}' : \tau$, if $\texttt{v} \preccurlyeq \texttt{k}$ and $\texttt{k} \sqsubseteq \texttt{k}'$, then $\texttt{v} \preccurlyeq \texttt{k}'$.
- (A-R$_4$) the reshape function produces equivalent value, as long as the new public view is valid: for any values $\texttt{v} : \mathsf{T}$, $\texttt{k} : \tau$, $\texttt{k}' : \tau$, $\widehat{\texttt{v}} : \widehat{\mathsf{T}}\ \texttt{k}$ and $\widehat{\texttt{v}}' : \widehat{\mathsf{T}}\ \texttt{k}'$, if $\texttt{r}\ \texttt{k}\ \widehat{\texttt{v}} \longrightarrow^* \texttt{v}$ and $\texttt{v} \preccurlyeq \texttt{k}'$ and $\widehat{\downarrow}\ \texttt{k}\ \texttt{k}'\ \widehat{\texttt{v}} \longrightarrow^* \widehat{\texttt{v}}'$, then $\texttt{r}\ \texttt{k}'\ \widehat{\texttt{v}}' \longrightarrow^* \texttt{v}$.

Figure 4 defines the join and reshape functions $\widehat{\texttt{list}_\preccurlyeq}\texttt{\#join}$ and $\widehat{\texttt{list}_\preccurlyeq}\texttt{\#reshape}$. The partial order for this join structure is simply the total order on integers, and the join is simply the maximum of the two numbers. Not all OADTs have a sensible join-structure: oblivious lists using their exact length as a public view cannot be combined if they have different lengths. If such lists are the branches of an oblivious conditional, lifting will either fail or coerce both to an OADT with a join-structure.

Join structures induce a *mergeability* relation, defined in Figure 10, that can be used to decide if a specification type can be used in oblivious conditionals. We say θ is *mergeable* if $\lambda\theta \triangleright \widehat{\texttt{ite}}$, with witness $\widehat{\texttt{ite}}$ of type $\widehat{\mathbb{B}} \to \theta \to \theta \to \theta$. We will write $\lambda\theta$ when we do not care about the witness. This witness can be thought of as a generalized, drop-in replacement of $\texttt{mux}$: we simply translate $\texttt{mux}$ to the derived $\widehat{\texttt{ite}}$ if the result type is mergeable. The case of Ψ-type captures this intuition: we first join the public views, and reshape all branches to this common public view, before we select the correct one privately using $\texttt{mux}$. This rule looks up the necessary methods from the context of join structures $\mathcal{S}_\sqcup$. Other cases are straightforward: we simply fall back to $\texttt{mux}$ for primitive types, and the derivation for product and function types are done congruently.

4.3 Introduction and Elimination Structures

An ADT is manipulated by its introduction and elimination forms. To successfully lift a public program using ADTs, we need structures to explain how the primitive operations of its ADTs are handled in their OADT counterparts. Thus, an OADT $\widehat{\mathsf{T}}$ can *optionally* be equipped with an *introduction-structure* (intro-structure) and an *elimination-structure* (elim-structure), defined below. These structures are optional because some programs only consume ADTs, without constructing any new ADT values (and vice versa): a function that checks membership in a list only requires an elim-structure on lists, for example. Intuitively, the axioms of these structures require the introduction and elimination methods of an OADT to behave like those of the corresponding ADT. This is formalized using a pair of logical refinement relations on values ($\mathcal{V}_n[\![\cdot]\!]$) and expressions ($\mathcal{E}_n[\![\cdot]\!]$); these relations are formally defined in Section 4.6.

Definition 4.3 (intro-structure). An intro-structure of an OADT $\widehat{\mathsf{T}}$ for ADT T, with global definition $\texttt{data}\ \mathsf{T} = \overline{\texttt{C}\ \eta}$, consists of a set of functions $\widehat{\texttt{C}}_i$, each corresponding to a constructor $\texttt{C}_i$. The type of $\widehat{\texttt{C}}_i$ is $\theta_i \to \Psi\widehat{\mathsf{T}}$, where $\lfloor \theta_i \rfloor = \eta_i$ (note that DT-ADT guarantees that η_i is a simple type). The particular θ_i an intro-structure uses is determined by the author of that structure. Each $\widehat{\texttt{C}}_i$ is required to logically

$$\frac{}{\theta \rightarrowtail \theta \triangleright \lambda \mathtt{x} \Rightarrow \mathtt{x}} \qquad \frac{}{\mathbb{B} \rightarrowtail \widehat{\mathbb{B}} \triangleright \lambda \mathtt{x} \Rightarrow \widehat{\mathbb{B}}\#\mathtt{s}\ \mathtt{x}} \qquad \frac{(\widehat{\mathsf{T}}, \mathsf{T}, \mathtt{s}, \mathtt{r}, \nu, \preccurlyeq) \in \mathcal{S}_\omega}{\mathsf{T} \rightarrowtail \Psi\widehat{\mathsf{T}} \triangleright \lambda \mathtt{x} \Rightarrow \langle \nu\ \mathtt{x}, \mathtt{s}\ (\nu\ \mathtt{x})\ \mathtt{x} \rangle} \qquad \frac{\uparrow : \Psi\widehat{\mathsf{T}} \rightarrow \Psi\widehat{\mathsf{T}}' \in \mathcal{S}_\uparrow}{\boxed{\theta \rightarrowtail \theta' \triangleright \uparrow}\ \Psi\widehat{\mathsf{T}} \rightarrowtail \Psi\widehat{\mathsf{T}}' \triangleright \uparrow}$$

$$\frac{\theta_1 \rightarrowtail \theta_1' \triangleright \uparrow_1 \qquad \theta_2 \rightarrowtail \theta_2' \triangleright \uparrow_2}{\theta_1 \times \theta_2 \rightarrowtail \theta_1' \times \theta_2' \triangleright \lambda \mathtt{x} \Rightarrow (\uparrow_1(\pi_1\ \mathtt{x}), \uparrow_2(\pi_2\ \mathtt{x}))} \qquad \frac{\theta_1' \rightarrowtail \theta_1 \triangleright \uparrow_1 \qquad \theta_2 \rightarrowtail \theta_2' \triangleright \uparrow_2}{\theta_1 \rightarrow \theta_2 \rightarrowtail \theta_1' \rightarrow \theta_2' \triangleright \lambda \mathtt{x} \Rightarrow \lambda \mathtt{y} \Rightarrow \uparrow_2(\mathtt{x}\ (\uparrow_1 \mathtt{y}))}$$

Fig. 11. Coercion

refine the corresponding constructor (A-I$_1$): given any values $\mathtt{v} : \lfloor\theta\rfloor$ and $\mathtt{v}' : \theta$, if $(\mathtt{v}, \mathtt{v}') \in \mathcal{V}_n[\![\theta]\!]$, then $(\mathtt{C}_i\ \mathtt{v}, \widehat{\mathtt{C}}_i\ \mathtt{v}') \in \mathcal{E}_n[\![\Psi\widehat{\mathsf{T}}]\!]$.

Definition 4.4 (elim-structure). An elim-structure of an OADT $\widehat{\mathsf{T}}$ for ADT T, with global definition $\mathtt{data}\ \mathsf{T} = \overline{\mathtt{C}\ \eta}$, consists of a family of functions $\widehat{\mathtt{match}}_\alpha$, indexed by the possible return types. The type of $\widehat{\mathtt{match}}_\alpha$ is $\Psi\widehat{\mathsf{T}} \rightarrow \overline{(\theta \rightarrow \alpha)} \rightarrow \alpha$, where $\lfloor\theta_i\rfloor = \eta_i$ for each θ_i in the function arguments corresponding to alternatives. Each $\widehat{\mathtt{match}}_\alpha$ is required to logically refine the pattern matching expression, specialized with ADT T and return type α. The sole axiom of this structure (A-E$_1$) only considers return type α being a specification type: given values $\mathtt{v}_i : \eta_i$, $\langle \mathtt{k}, \widehat{\mathtt{v}} \rangle : \widehat{\mathsf{T}}\ \mathtt{k}$, $\lambda \mathtt{x} \Rightarrow \mathtt{e}_i : \lfloor\theta_i\rfloor \rightarrow \lfloor\alpha\rfloor$ and $\lambda \mathtt{x} \Rightarrow \mathtt{e}_i' : \theta_i \rightarrow \alpha$, if $\mathtt{r}\ \mathtt{k}\ \widehat{\mathtt{v}} \longrightarrow^* \mathtt{C}_i\ \mathtt{v}_i$ and $(\lambda \mathtt{x} \Rightarrow \mathtt{e}_i, \lambda \mathtt{x} \Rightarrow \mathtt{e}_i') \in \mathcal{V}_n[\![\theta_i \rightarrow \alpha]\!]$ then $([\mathtt{v}_i/\mathtt{x}]\mathtt{e}_i, \widehat{\mathtt{match}}\ \langle \mathtt{k}, \widehat{\mathtt{v}} \rangle\ \overline{(\lambda \mathtt{x} \Rightarrow \mathtt{e}')}) \in \mathcal{E}_n[\![\alpha]\!]$.

The types of the oblivious introduction and elimination forms in these structures are only required to be compatible with the public counterparts. The programmers can choose which specific OADTs to use according to their desired privacy policy. Figure 4 shows the constructors and pattern matching functions for $\widehat{\mathtt{list}}_\preccurlyeq$.

The elim-structure of an OADT consists of a family of destructors, whose return type α does not necessarily range over all types. For example, $\widehat{\mathtt{match}}_\alpha$ of $\widehat{\mathtt{list}}_\preccurlyeq$, $\widehat{\mathtt{list}}_\preccurlyeq\#\mathtt{match}$ in Figure 4, requires α to be a mergeable type, due to the use of $\widehat{\mathtt{match}}$, which imposes a restriction similarly to $\mathtt{mux}$. Such constraints on α are automatically inferred and enforced.

4.4 Coercion Structures

As discussed in Section 2, we may need to convert an oblivious type to another, either due to a mismatch from input to output, or due to its lack of certain structures. For example, $\widehat{\mathtt{list}}_=$ does not have join structure, so if the branches of an oblivious conditional has type $\Psi\widehat{\mathtt{list}}_=$, they should be coerced to $\Psi\widehat{\mathtt{list}}_\preccurlyeq$, when such a coercion is available.

Two compatible OADTs may form a *coercion-structure*, shown below.

Definition 4.5 (coercion-structure). A coercion-structure of a pair of compatible OADTs $\widehat{\mathsf{T}}$ and $\widehat{\mathsf{T}}'$ for T, with public view type τ and τ' respectively, consists of a coercion function $\uparrow$ of type $\Psi\widehat{\mathsf{T}} \rightarrow \Psi\widehat{\mathsf{T}}'$. The coercion should produce an equivalent value (A-C$_1$): given values $\mathtt{v} : \mathsf{T}$, $\langle \mathtt{k}, \widehat{\mathtt{v}} \rangle : \Psi\widehat{\mathsf{T}}$ and $\langle \mathtt{k}', \widehat{\mathtt{v}}' \rangle : \Psi\widehat{\mathsf{T}}'$, if $\mathtt{r}\ \mathtt{k}\ \widehat{\mathtt{v}} \longrightarrow^* \mathtt{v}$ and $\uparrow\langle \mathtt{k}, \widehat{\mathtt{v}} \rangle \longrightarrow^* \langle \mathtt{k}', \widehat{\mathtt{v}}' \rangle$, then $\mathtt{r}\ \mathtt{k}'\ \widehat{\mathtt{v}}' \longrightarrow^* \mathtt{v}$.

This structure only defines the coercion between two Ψ-types. Figure 11 generalizes the coercion relation to any (compatible) specification types. We say θ is *coercible* to θ' if $\theta \rightarrowtail \theta' \triangleright \uparrow$, with witness $\uparrow$ of type $\theta \rightarrow \theta'$. We may write $\theta \rightarrowtail \theta'$ when we do not care about the witness. The rules of this relation are straightforward. The context of coercion structures $\mathcal{S}_\uparrow$ and the context of OADT structures $\mathcal{S}_\omega$ are used to look up the necessary methods in the corresponding rules. The rule for coercing a function type is contravariant. Note that we can always coerce a public type to an OADT by running the section function, and the public view can be selected by the view function in the OADT structure.

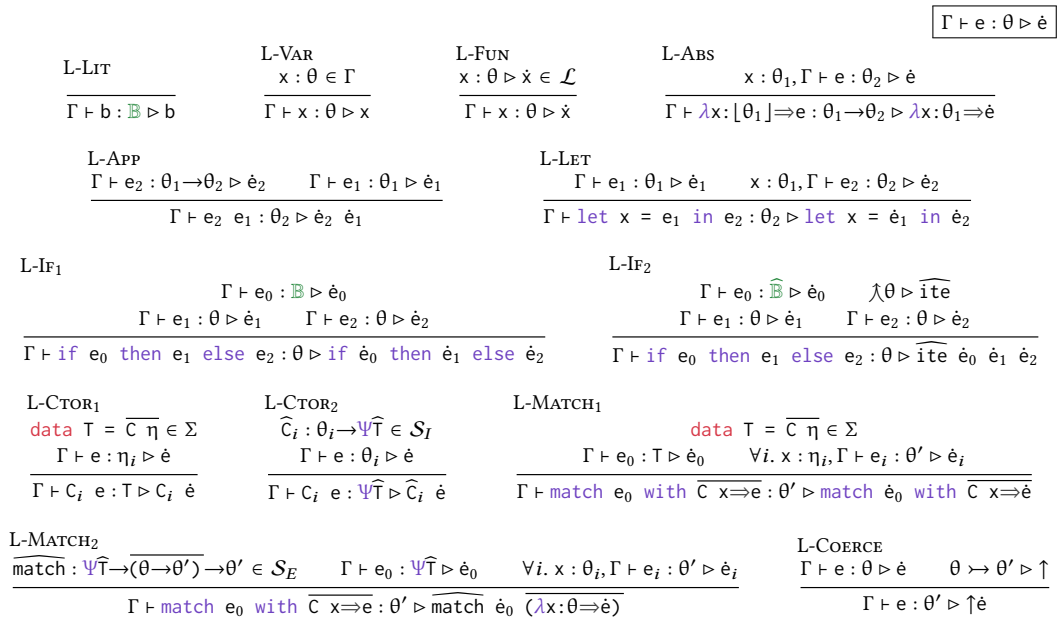

Fig. 12. Selected declarative lifting rules

4.5 Declarative Lifting

With these Ψ-structures, we define a declarative lifting relation, which describes what the lifting procedure is allowed to derive at a high level. This lifting relation is given by the judgment $\mathcal{S}; \mathcal{L}; \Sigma; \Gamma \vdash e : \theta \triangleright \dot{e}$. It is read as the expression e of type $\lfloor\theta\rfloor$ is lifted to the expression $\dot{e}$ of target type θ, under various contexts. The Ψ-structure context $\mathcal{S}$ consists of the set of OADT-structures ($\mathcal{S}_\omega$), join-structures ($\mathcal{S}_\sqcup$), intro-structures ($\mathcal{S}_I$), elim-structures ($\mathcal{S}_E$) and coercion-structures ($\mathcal{S}_\uparrow$), respectively. The global definition context Σ is the same as the one used in the typing relation. The local context Γ is also similar to the one in the typing relation, but it keeps track of the target types of local variables instead of source types. Finally, the lifting context $\mathcal{L}$ consists of entries of the form $x : \theta \triangleright \dot{x}$, which associates the global function x of type $\lfloor\theta\rfloor$ with a generated function $\dot{x}$ of the target type θ. A single global function may have multiple target types, i.e., multiple private versions, either specified by the users or by the callsites. For example, $\mathcal{L}$ may contain $\texttt{filter} : \Psi\widehat{\texttt{list}_\leq} \to \widehat{\mathbb{Z}} \to \Psi\widehat{\texttt{list}_\leq} \triangleright \widehat{\texttt{filter}_1}$ and $\texttt{filter} : \Psi\widehat{\texttt{list}_=} \to \widehat{\mathbb{Z}} \to \Psi\widehat{\texttt{list}_\leq} \triangleright \widehat{\texttt{filter}_2}$.

Figure 12 shows a selection of rules of the declarative lifting relation (the full rules are in appendix). We elide most contexts as they are fixed, and simply write $\Gamma \vdash e : \theta \triangleright \dot{e}$ for brevity. Most rules are simply congruences and similar to typing rules. L-Fun outsources the lifting of a function call to the lifting context. L-If$_2$ handles the case when the condition is lifted to an oblivious boolean by delegating the translation to the mergeability relation. Similarly, L-Ctor$_2$ and L-Match$_2$ query the contexts of the intro-structures and elim-structures, and use the corresponding instances as the drop-in replacement, when we are constructing or destructing Ψ-types. Lastly, L-Coerce coerces an expression nondeterministically using the coercion relation.

This lifting relation in Figure 12 only considers one expression. In practice, the users specify a set of functions and their target types to lift. The result of our lifting procedure is a lifting context $\mathcal{L}$ which maps these functions and target types to the corresponding generated functions, as well as any other functions and the inferred target types that these functions depend on. The global context

$$\boxed{\mathcal{V}_n[\![\theta]\!]}$$

$$\mathcal{V}_n[\![\mathbb{1}]\!] = \mathcal{V}_n[\![\mathbb{B}]\!] = \mathcal{V}_n[\![\mathsf{T}]\!] = \{\ (v, v')\ \mid 0 < n \implies v = v'\ \} \qquad \mathcal{V}_n[\![\widehat{\mathbb{B}}]\!] = \{\ (b, \lfloor b' \rfloor)\ \mid 0 < n \implies b = b'\ \}$$

$$\mathcal{V}_n[\![\Psi\widehat{\mathsf{T}}]\!] = \left\{\ (v, \langle k, \widehat{v}\rangle)\ \middle|\ 0 < n \implies r\ k\ \widehat{v} \longrightarrow^* v\ \right\}$$

$$\mathcal{V}_n[\![\theta_1 \times \theta_2]\!] = \left\{\ ((v_1, v_2), (v_1', v_2'))\ \middle|\ (v_1, v_1') \in \mathcal{V}_n[\![\theta_1]\!] \wedge (v_2, v_2') \in \mathcal{V}_n[\![\theta_2]\!]\ \right\}$$

$$\mathcal{V}_n[\![\theta_1 \to \theta_2]\!] = \left\{\ (\lambda x{:}\lfloor\theta_1\rfloor \Rightarrow e, \lambda x{:}\theta_1 \Rightarrow e')\ \middle|\ \forall i < n. \forall (v, v') \in \mathcal{V}_i[\![\theta_1]\!]. ([v/x]e, [v'/x]e') \in \mathcal{E}_i[\![\theta_2]\!]\ \right\}$$

$$\boxed{\mathcal{E}_n[\![\theta]\!]}$$

$$\mathcal{E}_n[\![\theta]\!] = \left\{\ (e, e')\ \middle|\ \forall i < n. \forall v'.\ e' \longrightarrow^i v' \implies \exists v.\ e \longrightarrow^* v \wedge (v, v') \in \mathcal{V}_{n-i}[\![\theta]\!]\ \right\}$$

Fig. 13. A logical relation for refinement

Σ is also extended with the definitions of the generated functions. To make this more clear, we say a lifting context is *derivable*, denoted by $\vdash \mathcal{L}$, if and only if, for any $x : \theta \triangleright \dot{x} \in \mathcal{L}$, $\mathtt{fn}\ x{:}\lfloor\theta\rfloor = e \in \Sigma$ and $\mathtt{fn}\ \dot{x}{:}\theta = \dot{e} \in \Sigma$ for some e and $\dot{e}$, such that $\mathcal{S}; \mathcal{L}; \Sigma; \cdot \vdash e : \theta \triangleright \dot{e}$. In other words, any definitions of the lifted functions in $\mathcal{L}$ can be derived from the lifting relation in Figure 12. Note that the derivation of a function definition is under a lifting context with possibly an entry of this function itself. This is similar to the role of global context in type checking, as Tayps supports mutually recursive functions. The goal of our algorithm (Section 5) is then to find such a derivable lifting context that includes the user-specified liftings.

4.6 Logical Refinement

The correctness of the lifting procedure is framed as a *logical refinement* between expressions of specification types and those of simple types; this relationship is defined as a step-indexed logical relation [Ahmed 2006]. As is common, this relation is defined via a pair of set-valued type denotations: a value interpretation $\mathcal{V}_n[\![\theta]\!]$ and an expression interpretation $\mathcal{E}_n[\![\theta]\!]$. We say an expression e' of type θ refines e of type $\lfloor\theta\rfloor$ (within n steps) if $(e, e') \in \mathcal{E}_n[\![\theta]\!]$. In other words, e' preserves the behavior of e, in that if e' terminates at a value, e must terminate at an equivalent value. The equivalence between values is dictated by $\mathcal{V}_n[\![\theta]\!]$.

Figure 13 shows the complete definition of the logical relation. All pairs in the relations must be closed and well-typed, i.e., their interpretations have the forms:

$$\mathcal{V}_n[\![\theta]\!] = \{\ (v, v')\ \mid \cdot \vdash v : \lfloor\theta\rfloor \wedge \cdot \vdash v' : \theta \wedge \dots\ \}$$
$$\mathcal{E}_n[\![\theta]\!] = \{\ (e, e')\ \mid \cdot \vdash e : \lfloor\theta\rfloor \wedge \cdot \vdash e' : \theta \wedge \dots\ \}$$

For brevity, we leave this requirement implicit in Figure 13.

The definitions are mostly standard. The most interesting case is the value interpretation of Ψ-type: we say the pair of a public view and an oblivious value of an OADT is equivalent to a public value of the corresponding ADT when the oblivious value can be retracted to the public value. Intuitively, an encrypted value is equivalent to the value it decrypts to. The base cases of the value interpretation are also guarded by the condition that we still have steps left, i.e., greater than 0. This requirement maintains the pleasant property that the interpretations $\mathcal{V}_0[\![\theta]\!]$ and $\mathcal{E}_0[\![\theta]\!]$ are total relations on closed values and expressions, respectively, of type θ. The proof also uses a straightforward interpretation of typing context, $\mathcal{G}_n[\![\Gamma]\!]$, whose definition is in appendix.

This relation also gives rise to a semantic characterization of the lifting context. We say a lifting context is n-valid, denoted by $\vDash_n \mathcal{L}$, if and only if, for any $x : \theta \triangleright \dot{x} \in \mathcal{L}$, $(x, \dot{x}) \in \mathcal{E}_n[\![\theta]\!]$. If $\vDash_n \mathcal{L}$ for any n, we say $\mathcal{L}$ is valid, denoted by $\vDash \mathcal{L}$. The validity is essentially a semantic correctness of $\mathcal{L}$.

4.7 Metatheory of Lifting

The first key property of the lifting relation is well-typedness, which guarantees the security of translated programs, thanks to Theorem 3.1.

Theorem 4.6 (Regularity of declarative lifting). *Suppose* $\mathcal{L}$ *is well-typed and* $\mathcal{S}; \mathcal{L}; \Sigma; \Gamma \vdash e : \theta \triangleright \dot{e}$. *We have* $\Sigma; \lfloor \Gamma \rfloor \vdash e : \lfloor \theta \rfloor$ *and* $\Sigma; \Gamma \vdash \dot{e} : \theta$.

Our lifting relation ensures that lifted expressions refine source expressions in fewer than n steps, as long as every lifted program in $\mathcal{L}$ is also semantically correct in fewer than n steps. As is common in logical relation proofs, this proof requires a more general theorem about open terms.

Theorem 4.7 (Correctness of declarative lifting of closed terms). *Suppose* $\mathcal{S}; \mathcal{L}; \Sigma; \cdot \vdash e : \theta \triangleright \dot{e}$ *and* $\vDash_n \mathcal{L}$. *We have* $(e, \dot{e}) \in \mathcal{E}_n[\![\theta]\!]$.

Finally, Theorem 4.8 provides a strong result of the correctness of our translation. Any lifting context that is derived using the rules of Figure 12 is semantically correct. In other words, if every pair of source program and lifted program in $\mathcal{L}$ are in our lifting relation, they also satisfy our refinement criteria.

Theorem 4.8 (Correctness of declarative lifting). $\vdash \mathcal{L}$ *implies* $\vDash \mathcal{L}$.

Our notion of logical refinement only provides partial correctness guarantees, as can be seen in the definition of $\mathcal{E}_n[\![\cdot]\!]$. As a result, the lifting relation does not guarantee equi-termination: it is possible that a lifted program will diverge when the source program terminates. This can occur when an `if` is replaced by a `mux`: since the latter fully executes both branches, this effectively changes the semantics of a conditional from a lazy evaluation strategy to an eager strategy. Using a public value to bound the recursion depth in order to guarantee termination is a common practice in data-oblivious computation, for the reasons discussed in Section 2. While the public view of an OADT naturally serves as a measure in many cases, including all of the case studies and benchmarks in our evaluation, in theory it is possible for a user to provide a policy to a function that results in a nonterminating lifted version. In this situation, users must either specify a different policy, or rewrite the functions to recurse on a different argument, e.g., a fuel value.

5 ALGORITHMIC LIFTING

Figure 14 presents the overall workflow of our lifting algorithm. This algorithm starts with a set of *goals*, i.e., pairs of source functions tagged with the `%lift` keywords and their desired specification types. We then run our lifting algorithm on all the functions in these goals, as well as any functions they depend on, transforming each of these functions to an oblivious version parameterized by *typed macros* and type variables, along with a set of constraints over these type variables. After solving the constraints, we obtain a set of type assignments for each function. Note that a single function may have multiple type assignments, one for each occurence in a goal and callsite. For example, `filter` may have the type assignment for the goal $\Psi\widehat{\texttt{list}_{\leq}} \to \widehat{\mathbb{Z}} \to \Psi\widehat{\texttt{list}_{\leq}}$ generated by `%lift`, and the assignment for $\Psi\widehat{\texttt{list}_{\leq}} \to \mathbb{Z} \to \Psi\widehat{\texttt{list}_{\leq}}$ generated by the call in `filter5` from Section 2. Finally, we generate the private versions of all the lifted functions by instantiating their type variables and expanding away any macros. The lifting context from the last section is simply these lifted functions and their generated private versions.

The lifting algorithm is defined using the judgment $\Sigma; \Gamma \vdash e : \eta \sim \chi \triangleright \dot{e} \mid C$. It reads as the source expression e of type η is lifted to the target expression $\dot{e}$ whose type is a *type variable* χ as a placeholder for the specification type, and generates constraints C. The source expression e is required to be in *administrative normal form* (ANF) [Flanagan et al. 1993], which is guaranteed by our type checker. In particular, type annotations are added to `let`-bindings, and the body of every

`let` is either another `let` or a variable. Importantly, this means the last expression of a sequence of `let` must be a variable. The output of this algorithm is an expression $\dot{e}$ containing macros (which will be discussed shortly), and the constraints C. Unlike the declarative rules, this algorithm keeps track of the source type η, which is used to restrict the range of the type variables. Consequently, every entry of the typing context Γ has the form $x : \eta \sim X$, meaning that local variable x has type η in the source program and type X in the target program. For example, after the lifting algorithm has processed the function arguments of `filter` in Figure 1, the typing context contains entries $xs : \texttt{list} \sim X_1$ and $y : \mathbb{Z} \sim X_2$, with freshly generated type variables X_1 and X_2.

The typed macros, defined in Figure 15, are an essential part of the output of the lifting algorithm, and permit a form of ad-hoc polymorphism, that allows the algorithm to cleanly separate constraint solving from program generation. These macros take types as parameters and elaborate to expressions, under the contexts $\mathcal{S}$, $\mathcal{L}$ and Σ implicitly. These macros are effectively thin "wrappers" of their corresponding language constructs and the previously defined relations. The conditional macro `%ite`, for example, corresponds to the `if` expression, but the condition may be oblivious. The constructor macro `%C` is a "smart" constructor that may construct a Ψ-type. The pattern matching macro `%match` is similar to `%C` but for eliminating a type compatible with an ADT. Lastly, `%↑` and `%x` is simply a direct wrapper of the mergeable relation and the lifting context $\mathcal{L}$, respectively. Note that the derivation of these macro are completely determined by the type parameters.

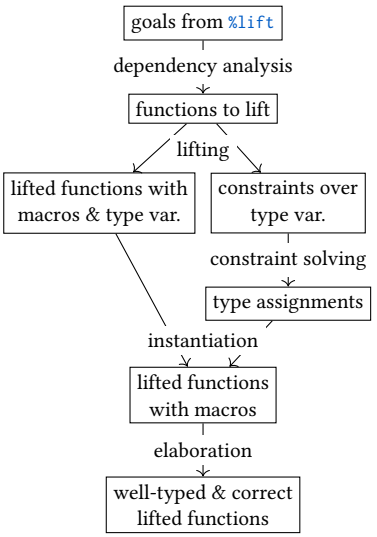

Fig. 14. Translation pipeline

Figure 16 defines the constraints used in the algorithm, where θ^+ is the specification types extended with type variables. The constraint $X \in [\eta]$ means type variable X belongs to the compatibility class of η. In other words, $\lfloor X \rfloor = \eta$. Each macro is accompanied by a constraint on its type parameters. These constraints mean that the corresponding macros are resolvable. More formally, this means they can elaborate to some expressions according to the rules in Figure 15 for any expression arguments. As a result, after solving all constraints and concretizing the type variables, all macros in the lifted expression $\dot{e}$ can be fully elaborated away.

Figure 17 shows a selection of lifting algorithm rules. Coercions only happen when we lift variables, as in A-Var. This works because the source program is in ANF, so each expression is bound to a variable which has the opportunity to get coerced. For example, the argument to a function or constructor, in A-App and A-Ctor, is always a variable in ANF, and recursively lifting it allows the application of A-Var. On the other hand, the top-level program is always in `let`-binding form, whose last expression is always a variable too, allowing coercion of the whole program. However, not all variables are subject to coercions: the function x_2 in A-App, the condition x_0 in A-If and the discriminee x_0 in A-Match are kept as they are, for example. Coercing these variables would be unnecessary and undesirable. For example, coercing the condition in a conditional only makes the generated program more expensive: there is no reason to coerce from $\mathbb{B}$ to $\widehat{\mathbb{B}}$, and use `mux` instead of `if`. Another key invariant we enforce in our algorithmic rules is that every fresh variable is "guarded" by a compatibility class constraint. For example, in A-Abs, the freshly generated variables X_1 and X_2 belong to the classes η_1 and η_2, respectively. This constraint ensures that every type variable can be finitely enumerated, as every compatibility class is a finite set, bounded by the number of available OADTs. As a result, constraint solving in our context is decidable. Finally, if an

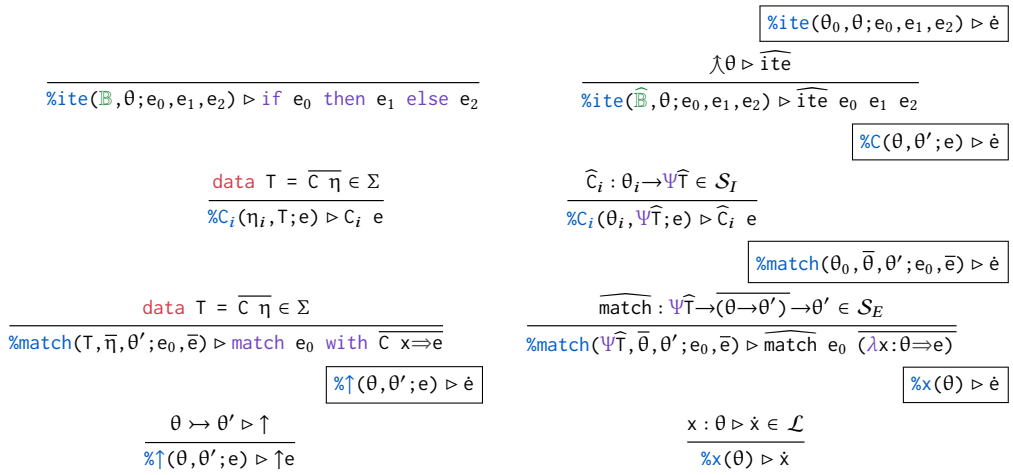

Fig. 15. Typed macros

Constraints

$$c \quad ::= \quad \mathtt{X} \in [\eta] \mid \theta^+ = \theta^+ \mid \mathtt{\%ite}(\theta^+,\theta^+) \mid \mathtt{\%C}(\theta^+,\theta^+) \mid \mathtt{\%match}(\theta^+,\overline{\theta^+},\theta^+) \mid \mathtt{\%\uparrow}(\theta^+,\theta^+) \mid \mathtt{\%x}(\theta^+)$$

Fig. 16. Constraints

expression is translated to a macro, a corresponding constraint is added to ensure this macro is resolvable.

We use the judgment $\mathcal{S};\mathcal{L};\Sigma;\sigma \vDash C$ to mean the assignment σ satisfies a set of constraints C, under the context of Ψ-structure, lifting context and global definition context. The constraints generated by our lifting algorithm use type variables $\mathtt{X}$ as placeholders for the target type of the function being lifted. To solve a goal with a particular target type θ, we add a constraint to C that equates the placeholder with the stipulated type, i.e., $\mathtt{X} = \theta$. Our constraint solver then attempts to find type assignments that satisfy the constraints in C; the resulting assignment is used to generate private versions of all the functions in the set of goals, as well as the accompanying lifting context.

At a high level,[10] our solver reduces all constraints, except for function call constraints (`%x`), to quantifier-free formulas in a finite domain theory, which can be efficiently solved using an off-the-shelf solver. Function call constraints are recursively solved once their type arguments have been concretized by discharging the other constraints. When a function call constraint is unsatisfiable, we add a new refutation constraint and invoke the solver again to find a new instantiation of type parameters. As an example of this process, in order to ascribe `filter` the type $\Psi\widehat{\mathtt{list}_=} \to \widehat{\mathbb{Z}} \to \Psi\widehat{\mathtt{list}_\leq}$, we first add the constraint $\mathtt{X} = \Psi\widehat{\mathtt{list}_=} \to \widehat{\mathbb{Z}} \to \Psi\widehat{\mathtt{list}_\leq}$ to the constraints generated by the lifting algorithm $\cdot \vdash \ldots : \mathtt{list} \to \mathbb{Z} \to \mathtt{list} \sim \mathtt{X} \triangleright \dot{e} \mid C$. Solving the other constraints may concretize the type variable of function call constraint $\mathtt{\%filter(X)}$, i.e., the type of the recursive call to `filter`, to $\mathtt{\%filter}(\Psi\widehat{\mathtt{list}_=} \to \widehat{\mathbb{Z}} \to \Psi\widehat{\mathtt{list}_\leq})$. Recursively solving this subgoal assuming the original goal is solved, i.e., extending the lifting context with the original goal, results in immediate success, as the subgoal is simply in the lifting context. On the other hand, if the type of the recursive call is instantiated as $\mathtt{\%filter}(\Psi\widehat{\mathtt{list}_=} \to \widehat{\mathbb{Z}} \to \Psi\widehat{\mathtt{list}_=})$, the same constraints generated by lifting `filter` are solved, with an additional constraint $\mathtt{X} = \Psi\widehat{\mathtt{list}_=} \to \widehat{\mathbb{Z}} \to \Psi\widehat{\mathtt{list}_=}$. However, this set of constraints is unsatisfiable, as $\widehat{\mathtt{list}_=}$ has no join structure, so we add a refutation constraint to the context

[10]The full details of our constraint solver are given in the appendix.

$\boxed{\Gamma \vdash e : \eta \sim X \rhd \dot{e} \mid C}$

A-Lit
$$\frac{}{\Gamma \vdash b : \mathbb{B} \sim X \rhd b \mid X = \mathbb{B}}$$

A-Var
$$\frac{x : \eta \sim X \in \Gamma}{\Gamma \vdash x : \eta \sim X' \rhd \%\uparrow(X, X'; x) \mid \%\uparrow(X, X')}$$

A-Fun
$$\frac{\texttt{fn}\ x{:}\eta\ =\ e \in \Sigma}{\Gamma \vdash x : \eta \sim X \rhd \%x(X) \mid \%x(X)}$$

A-Abs
$$\frac{X_1, X_2 \text{ fresh} \qquad x : \eta_1 \sim X_1, \Gamma \vdash e : \eta_2 \sim X_2 \rhd \dot{e} \mid C}{\Gamma \vdash \lambda x{:}\eta_1 \Rightarrow e : \eta_1 \to \eta_2 \sim X \rhd \lambda x{:}X_1 \Rightarrow \dot{e} \mid X_1 \in [\eta_1], X_2 \in [\eta_2], X = X_1 \to X_2, C}$$

A-App
$$\frac{X_1 \text{ fresh} \qquad x_2 : \eta_1 \to \eta_2 \sim X \in \Gamma \qquad \Gamma \vdash x_1 : \eta_1 \sim X_1 \rhd \dot{e}_1 \mid C}{\Gamma \vdash x_2\ \ x_1 : \eta_2 \sim X_2 \rhd x_2\ \ \dot{e}_1 \mid X_1 \in [\eta_1], X = X_1 \to X_2, C}$$

A-Let
$$\frac{X_1 \text{ fresh} \qquad \Gamma \vdash e_1 : \eta_1 \sim X_1 \rhd \dot{e}_1 \mid C_1 \qquad x : \eta_1 \sim X_1, \Gamma \vdash e_2 : \eta_2 \sim X_2 \rhd \dot{e}_2 \mid C_2}{\Gamma \vdash \texttt{let}\ x{:}\eta_1\ =\ e_1\ \texttt{in}\ e_2 : \eta_2 \sim X_2 \rhd \texttt{let}\ x{:}X_1\ =\ \dot{e}_1\ \texttt{in}\ \dot{e}_2 \mid X_1 \in [\eta_1], C_1, C_2}$$

A-If
$$\frac{x_0 : \mathbb{B} \sim X_0 \in \Gamma \qquad \Gamma \vdash e_1 : \eta \sim X \rhd \dot{e}_1 \mid C_1 \qquad \Gamma \vdash e_2 : \eta \sim X \rhd \dot{e}_2 \mid C_2}{\Gamma \vdash \texttt{if}\ x_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \eta \sim X \rhd \%\texttt{ite}(X_0, X; x_0, \dot{e}_1, \dot{e}_2) \mid \%\texttt{ite}(X_0, X), C_1, C_2}$$

A-Ctor
$$\frac{\texttt{data}\ T\ =\ \overline{C\ \eta} \in \Sigma \qquad X_i \text{ fresh} \qquad \Gamma \vdash x : \eta_i \sim X_i \rhd \dot{e} \mid C}{\Gamma \vdash C_i\ \ x : T \sim X \rhd \%C_i(X_i, X; \dot{e}) \mid X_i \in [\eta_i], \%C_i(X_i, X), C}$$

A-Match
$$\frac{\texttt{data}\ T\ =\ \overline{C\ \eta} \in \Sigma \qquad \overline{X} \text{ fresh} \qquad x_0 : T \sim X_0 \in \Gamma \qquad \forall i.\ x : \eta_i \sim X_i, \Gamma \vdash e_i : \eta' \sim X' \rhd \dot{e}_i \mid C_i}{\Gamma \vdash \texttt{match}\ x_0\ \texttt{with}\ \overline{C\ x \Rightarrow e} : \eta' \sim X' \rhd \%\texttt{match}(X_0, \overline{X}, X'; x_0, \overline{\dot{e}}) \mid \overline{X \in [\eta]}, \%\texttt{match}(X_0, \overline{X}, X'), \overline{C}}$$

Fig. 17. Selected algorithmic lifting rules

that forces the solver to not generate this assignment again. In general, the type of the recursive call to `filter` may be concretized to any types compatible with $\texttt{list} \to \mathbb{Z} \to \texttt{list}$. The number of such compatible types is bounded, as the number of arguments of this function and the number of OADTs are themselves bounded. The function `filter` has $3 \times 2 \times 3 = 18$ possible type assignments. In the worst case scenario, the algorithm eventually terminates after exhausting all 18 combinations.

The lifting algorithm enjoys a soundness theorem with respect to the declarative lifting relation. As a result, our algorithm inherits the well-typedness and correctness properties of the declarative version. The statement of this theorem follows how the algorithm is used: if the generated constraints, equating the function type variable with the specification type, are satisfiable by the type assignment σ, instantiating the lifted expression with σ and elaborating the macros results in a target expression that is valid under the declarative lifting relation:

Theorem 5.1 (Soundness of algorithmic lifting). *Suppose* $\Sigma; \cdot \vdash e : \eta \sim X \rhd \dot{e} \mid C$. *Given a specification type* θ, *if* $\mathcal{S}; \mathcal{L}; \Sigma; \sigma \vDash X = \theta, C$, *then* $\sigma(\dot{e})$ *elaborates to an expression* $\dot{e}'$, *such that* $\mathcal{S}; \mathcal{L}; \Sigma; \cdot \vdash e : \theta \rhd \dot{e}'$.

The proof of this theorem is available in the appendix.

6 IMPLEMENTATION AND EVALUATION

Our compilation pipeline takes as input a source program, including any OADTs, Ψ-structures, and macros (e.g., `%lift`), in the public fragment of Taypsi and privacy policies (i.e., security-type signatures) for all target functions. After typing the source program using a bidirectional type

checker, our lifting pass generates secure versions of the specified functions and their dependencies, using Z3 [de Moura and Bjørner 2008] as its constraint solver. The resulting Taypsi functions are translated into Oil [Ye and Delaware 2023], an ML-style functional language equipped with oblivious arrays and secure array operations: OADTs are converted to serialized versions which are stored in secure arrays, and all oblivious operations are translated into secure array operations. After applying some optimizations, our pipeline outputs an OCaml library providing secure implementations of all the specified functions, including section and retraction functions for encrypting private data and decrypting the results of a joint computation. After linking this library to a *driver* that provides the necessary cryptographic primitives (i.e., secure integer arithmetic), programmers can build secure MPC applications on top of this API. The following evaluation uses a driver implemented using the popular open-source EMP toolkit [Wang et al. 2016].

Optimizations. Our implementation of Taypsi implements three optimizations which further improve the performance of the programs it generates.[11] The *reshape guard* optimization instruments reshape instances to first check if the public views of two private values are identical, omitting the reshape operation if so. The *memoization* optimization caches the sizes of the private representation of data in order to avoid recalculating this information, which is needed to create and slice oblivious arrays. The final, *smart array* optimization supports zero-cost array slicing and concatenation, and eliminates redundant operations over the serialized representation of oblivious data. One observation underlying this optimization is that evaluating a `mux` whose branches are encrypted versions of publicly-known values is unnecessary: `mux [b] (𝔹̂#s true) (𝔹̂#s false)` is equivalent to `[b]`, for example. This situation frequently occurs in map-like functions, where the constructor used in each branch of a function is publicly known. Under the hood, the serialized encoding of the result of `map` uses a boolean tag to indicate which constructor was used to build it, i.e., `Nil` or `Cons`; this boolean is determined by the tag of the input list, e.g., `mux [tag] [true] [false]`. Of course, the tag used in each branch is publicly known: `map` always returns `Nil` if the input list is empty, and returns a `Cons` otherwise. Thus, we can safely reuse the `[tag]` of the input list to label the result of `map`, for similar reasons as the previous example. The smart array optimization exploits this observation by marking when section functions are applied to public values instead of, for example, immediately evaluating `𝔹̂#s true` to the encrypted value `[true]`. Then, when performing a `mux`, the smart array first checks if both branches are "fake" private values, safely reducing the `mux` to its private condition if so, without actually performing any cryptographic operations.

Our evaluation considers the following research questions:

RQ1 How does the performance of Taypsi's transformation-based approach compare to the dynamic enforcement strategy of Taype?

RQ2 What is the compilation overhead of Taypsi's translation strategy?

6.1 Microbenchmark Performance

To answer **RQ1**, we have evaluated the performance of a set of microbenchmarks compiled with both Taypsi and Taype. Both approaches are equipped with optimizations that are unique to their enforcement strategies: Taypsi's reshape guard optimization is not applicable to Taype, and Taype features an *early tape* optimization that does not make sense for Taypsi.[12] Our evaluation also includes a version of Taype that implements Taypsi's smart array optimization (Taype-SA), in order to provide a comparison of the two approaches at their full potential.

[11] Our appendix describes each of these optimizations in more detail.

[12] Taype also implements a tupling optimization, but this is analogous to Taypsi's memoization optimization.

Benchmark	Taype (ms)	Taype-SA (ms)	Taypsi (ms)	
elem_1000†	8.15	8.11	8.02	(98.47%, 98.89%)
hamming_1000†	15.09	15.21	14.46	(95.79%, 95.04%)
euclidean_1000†	67.43	67.55	67.32	(99.84%, 99.66%)
dot_prod_1000†	66.12	66.19	66.41	(100.43%, 100.33%)
nth_1000†	11.98	12.05	12.04	(100.54%, 99.93%)
map_1000	2139.55	5.07	5.14	(0.24%, 101.44%)
filter_200	**failed**	**failed**	86.86	(N/A, N/A)
insert_200	5796.69	88.92	88.07	(1.52%, 99.04%)
insert_list_100	**failed**	**failed**	4667.66	(N/A, N/A)
append_100	4274.7	45.09	44.18	(1.03%, 97.99%)
take_200	169.07	3.05	3.09	(1.83%, 101.15%)
flat_map_200	**failed**	**failed**	7.3	(N/A, N/A)
span_200	13529.34	124.79	91.22	(0.67%, 73.09%)
partition_200	**failed**	**failed**	176.49	(N/A, N/A)
elem_16†	446.81	459.1	404.9	(90.62%, 88.19%)
prob_16†	13082.52	12761.7	12735.16	(97.34%, 99.79%)
map_16	4414.69	262.14	215.67	(4.89%, 82.27%)
filter_16	8644.14	452.04	433.7	(5.02%, 95.94%)
swap_16	**failed**	**failed**	4251.36	(N/A, N/A)
path_16	**failed**	6657.07	894.88	(N/A, 13.44%)
insert_16	83135.81	8093.81	1438.87	(1.73%, 17.78%)
bind_8	21885.65	494.98	532.86	(2.43%, 107.65%)
collect_8	**failed**	**failed**	143.38	(N/A, N/A)

Fig. 18. Running times for each benchmark in milliseconds. The Taypsi column also reports the percentage of running time relative to Taype and Taype-SA. A **failed** entry indicates the benchmark either timed out after 5 minutes or exceeded the memory bound of 8 GB. List and tree benchmarks appear above and below the double line, respectively.

Our benchmarks are a superset of the benchmarks from Ye and Delaware [2023]. Figure 18 presents the experimental results.[13] These experiments fix the public views of private lists and trees to be their maximum length and maximum depth, respectively; the suffix of each benchmark name indicates the public view used. The benchmarks annotated with † simply traverse the data type in order to produce a primitive value, e.g., an integer; these include membership (`elem`), hamming distance (`hamming`), minimum euclidean distance (`euclidean`), dot product (`dot_prod`), secure index look up (`nth`) and computing the probability of an event given a probability tree diagram (`prob`). The programs generated by Taype, Taype-SA and Taypsi all exhibit similar performance on these benchmarks. The remaining benchmarks all construct structured data values, i.e., the sort of application on which Taypsi is expected to shine. In addition to standard list operations, the list benchmarks include insertion into a sorted list (`insert`) and insertion of a list of elements into a sorted list (`insert_list`) (both lists have public view 100). The tree examples include a filter function that removes all nodes (including any subtrees) greater than a given private integer (`filter`), swapping subtrees if the node matches a private integer (`swap`), computing a subtree reached following a list of "going left" and "going right" directions (`path`), insertion into a binary search tree (`insert`), replacing the leaves of a tree with a given tree (`bind`), and collecting all nodes smaller than a private integer into a list (`collect`).

[13] All results are averaged across 5 runs, on an M1 MacBook Pro with 16 GB memory. All parties run on the same host with local network communication.

Benchmark	No Smart Array (ms)		No Reshape Guard (ms)		No Memoization (ms)	
elem_1000	18.37	(2.29x)	8.06	(1.0x)	17.76	(2.21x)
hamming_1000	51.73	(3.58x)	14.53	(1.01x)	35.5	(2.46x)
euclidean_1000	79.07	(1.17x)	67.31	(1.0x)	76.36	(1.13x)
dot_prod_1000	87.77	(1.32x)	66.15	(1.0x)	77.33	(1.16x)
nth_1000	22.69	(1.88x)	12.18	(1.01x)	20.53	(1.7x)
map_1000	2106.43	(409.89x)	139.91	(27.23x)	37.71	(7.34x)
filter_200	5757.28	(66.29x)	93.93	(1.08x)	114.7	(1.32x)
insert_200	255.43	(2.9x)	94.61	(1.07x)	89.32	(1.01x)
insert_list_100	22806.87	(4.89x)	5186.07	(1.11x)	4771.28	(1.02x)
append_100	4226.32	(95.66x)	50.79	(1.15x)	61.77	(1.4x)
take_200	169.45	(54.91x)	12.92	(4.19x)	4.68	(1.52x)
flat_map_200	5762.63	(789.08x)	16.99	(2.33x)	60.03	(8.22x)
span_200	5924.1	(64.95x)	99.83	(1.09x)	120.09	(1.32x)
partition_200	11528.0	(65.32x)	185.16	(1.05x)	231.06	(1.31x)
elem_16	433.73	(1.07x)	404.05	(1.0x)	402.15	(0.99x)
prob_16	13019.56	(1.02x)	12746.24	(1.0x)	12731.89	(1.0x)
map_16	4410.84	(20.45x)	635.18	(2.95x)	213.96	(0.99x)
filter_16	8674.71	(20.0x)	1131.02	(2.61x)	440.16	(1.01x)
swap_16	8671.52	(2.04x)	5471.4	(1.29x)	4246.39	(1.0x)
path_16	9108.54	(10.18x)	1083.21	(1.21x)	888.95	(0.99x)
insert_16	19101.36	(13.28x)	2151.83	(1.5x)	1432.92	(1.0x)
bind_8	19647.83	(36.87x)	870.93	(1.63x)	534.3	(1.0x)
collect_8	11830.6	(82.51x)	152.29	(1.06x)	186.92	(1.3x)

Fig. 19. Impact of turning off the smart array (No Smart Array), reshape guard (No Reshape Guard), and public view memoization (No Memoization) optimizations. Each column presents running time in milliseconds and the slowdown relative to that of the fully optimized version reported in Figure 18.

Dynamic policy enforcement either fails to finish within 5 minutes or exceeds an 8 GB memory bound on almost half of the last set of benchmarks, due to the exponential blowup discussed in Section 2. For those benchmarks that do finish, Taypsi's enforcement strategy results in a fraction of the total execution time compared to Taype. Compared to the version of Taype using smart arrays, Taypsi still performs comparably or better, although the gap is somewhat narrowed: functions like `map` do not suffer from exponential blowup, so these benchmarks benefit mostly from the smart array optimization. In summary, these results demonstrate that a static enforcement strategy performs considerably better than a dynamic one on many benchmarks, and works roughly as well on the remainder.

6.2 Impact of Optimization

To evaluate the performance impact of Taypsi's three optimizations, we conducted an ablation study on their effect. The results, shown in Figure 19, indicate that our smart array optimization is the most important, providing up to almost 800x speedup in the best case. As suggested by Figure 18, this optimization also helps significantly with the performance of Taype, although not enough to outweigh the exponential blowup innate in its dynamic approach. The other optimizations also improve performance, albeit not as significantly. As our memoization pass caches public views of arbitrary type, we have also conducted an ablation study for these list and tree examples using ADT public views instead, e.g., using Peano number to encode the maximum length of a list. In this study, we observe up to 9 times speed up in list examples, with minimal regression in tree examples. The full results of this study are included in the appendix.

6.3 Compilation Overhead

To measure the overhead of Taypsi's use of an external solver to resolve constraints, we have profiled the compilation of a set of larger programs drawn from Taype's benchmark suite, plus an additional secure dating application.[14] The first two benchmark suites (List and Tree) in Figure 20 include all the microbenchmarks from previous section. The next benchmark, List (stress), consists of the same microbenchmarks as List with 5 additional list OADTs. The purpose of this synthetic suite is to examine the impact of the number of OADTs on the search space. The remaining benchmarks represent larger, more realistic applications which demonstrate the expressivity and usability of Taypsi.

The last three columns of Figure 20 report the results of these experiments: total compilation time (Tot), time spent on constraint solving (Slv) and the number of solver queries (#Qu). The group of columns in the middle of the table describes features that can impact the performance of our constraint-based approach: the number of functions (#Fn) being translated, the number of atomic types (#Ty), and the total number of atomic types used in function types (#At). For example, the List benchmark features 7 atomic types: public and oblivious booleans, integers and lists, as well as an unsigned integer type (i.e. natural numbers). The number of atomic types in the function `filter : list → ℤ → list` is 3. In the worst case scenario, our constraint solving algorithm will explore every combination of types that are compatible with this signature, resulting in the constraints associated with `filter` being solved $2 * 2 * 2 = 8$ times. Exactly how many compatible types the constraint solving algorithm explores depends on many factors: the user-specified policies, the complexity of the functions, the calls to other functions and so on. We chose these 3 metrics as a coarse approximation of the solution space. Our results show that the solver overhead is quite minimal for most benchmarks, and in general solving time per query is low thanks to our encoding of constraints in an efficiently decidable logic.

Suite	#Fn	#Ty	#At	#Qu	Tot (s)	Slv (s)
List	20	7	70	84	0.47	0.081
Tree	14	9	44	31	0.47	0.024
List (stress)	20	12	70	295	3.45	2.8
Dating	4	13	16	10	0.58	0.019
Medical Records	20	19	58	51	0.48	0.072
Secure Calculator	2	9	6	5	1.34	0.013
Decision Tree	2	13	6	16	0.28	0.016
K-means	16	11	68	86	1.62	0.95
Miscellaneous	11	7	42	47	0.26	0.065

Fig. 20. Impact of constraint solving on compilation.

7 RELATED WORK

The problem of secure computation was first formally introduced by Yao [1982], who simultaneously proposed garbled circuits as a solution. Subsequently, a number of other solutions have been proposed [Evans et al. 2018; Hazay and Lindell 2010]. Solutions categorized as multiparty computation are usually based on cryptographic protocols, e.g., secret-sharing [Beimel 2011; Goldreich et al. 1987; Maurer 2006]. Outsourced computation is another type of secure computation that includes both cryptographic solutions, e.g., fully homomorphic encryption [Acar et al. 2018; Gentry 2009], and solutions based on virtualization [Barthe et al. 2014, 2019] or secure processors [Hoekstra 2015].

Taypsi features a *security-type system* [Sabelfeld and Myers 2003; Zdancewic 2002] based on the type system of λ_{OADT}. While most security-type systems tag types with labels classifying the sensitivity of data, our dependent type system tags kinds instead. The obliviousness guarantee provided by Theorem 3.1 is a form of *noninterference* [Goguen and Meseguer 1982] that generalizes

[14]The full details of the additional case study can be found in the appendix.

memory trace obliviousness (MTO) [Liu et al. 2013]. MTO considers traces of memory accesses, while the traces in Theorem 3.1 include *every* intermediate program state under a small-step operational semantics. This reflects MPC's stronger threat model, in which all parties can observe the complete execution of a program, including each instruction executed. As a consequence, our type system also protects against timing channels, similar to other *constant-time* languages [Cauligi et al. 2019].

Numerous high-level programming languages for writing secure multiparty computation applications have been proposed [Hastings et al. 2019]. Most prior languages either do not support structured data, or require all structural information to be public, e.g., Obliv-C [Zahur and Evans 2015] and ObliVM [Liu et al. 2015]. To the best of our knowledge Taype is the only existing language for MPC applications that natively supports decoupling privacy policies from program logic. On the other hand, there are many aspects of MPC tackled by prior languages that we do not consider here. Wysteria and Wys* [Rastogi et al. 2014, 2019], for example, focus on *mixed-mode computation* which allows certain computation to be executed locally. Symphony [Sweet et al. 2023], a successor of Wysteria, supports first-class shares and first-class party sets for coordinating many parties, enabling more reactive applications. Darais et al. [2020] developed λ_{obliv}, a probabilistic functional language for oblivious computations that can be used to safely implement a variety of cryptography algorithms, including Oblivious RAM.

Several prior works have considered how to compile secure programs into more efficient secure versions. Viaduct [Acay et al. 2024, 2021] is a compiler that transforms high-level programs into secure distributed versions by intelligently selecting an efficient combination of protocols for subcomputations. The HyCC toolchain [Büscher et al. 2018] similarly transforms a C program into a version that combines different MPC protocols to optimize performance. The HACCLE toolchain [Bao et al. 2021] uses staging to generate efficient garbled circuits from a high-level language. Compiler techniques, e.g., vectorization, have been studied for optimizing fully homomorphic encryption (FHE) applications [Cowan et al. 2021; Dathathri et al. 2020; Malik et al. 2023, 2021; Viand et al. 2023].

Jeeves [Yang et al. 2012] and Taypsi have a shared goal of decoupling security policies from program logic. While they both employ a similar high-level strategy of relying on the language to automatically enforce policies, their different settings result in very different solutions. In Jeeves' programming model, each piece of data is equipped with a pair of high- and low-level views: a username, for example, may have a high confidentiality view of "Alice", but a low view of "Anonymous". The language then uses the view stipulated by the privacy policy and current execution context, ensuring that information is only visible to observers with the proper authority. In the MPC setting, however, no party is allowed to observe the private data of other parties. Thus, no party can view all the data necessary for the computation, making it impossible to compute a correct result by simply replacing data with some predetermined value, like "Anonymous".

8 CONCLUSION

Secure multiparty computation allows joint computation over private data from multiple parties, while keeping that data secure. Previous work has considered how to make languages for MPC more accessible by allowing privacy requirements to be decoupled from functionality, relying on dynamic enforcement of polices. Unfortunately, the resulting overhead of this strategy made it difficult to scale applications manipulating structured data. This work presents Taypsi, a policy-agnostic language for oblivious computation that transforms programs to instead *statically* enforce a user-provided privacy policy. The resulting programs are guaranteed to be both well-typed, and hence secure, and equivalent to the source program. Our experimental results show this strategy yields considerable performance improvements over prior approaches, while maintaining a clean separation between privacy and programmatic concerns.

DATA-AVAILABILITY STATEMENT

An artifact containing our implementation of Taypsi, its source code, and the source for all the benchmarks in our experiments with instructions is publicly available [Ye and Delaware 2024]. The appendix is included in the auxiliary material.

ACKNOWLEDGMENTS

We thank Raghav Malik, Prasita Mukherjee, Tarindu Jayatilaka, Patrick LaFontaine, and the anonymous reviewers for their detailed comments and suggestions. We also thank Milind Kulkarni and Ashish Kundu for many stimulating discussions about this work. The material in this paper is based on work partially supported by Cisco Systems under award #23013611.

REFERENCES

Abbas Acar, Hidayet Aksu, A. Selcuk Uluagac, and Mauro Conti. 2018. A Survey on Homomorphic Encryption Schemes: Theory and Implementation. *ACM Computing Surveys (CSUR)* 51, 4 (July 2018), 79:1–79:35. https://doi.org/10.1145/3214303

Coşku Acay, Joshua Gancher, Rolph Recto, and Andrew C. Myers. 2024. Secure Synthesis of Distributed Cryptographic Applications (Technical Report). https://doi.org/10.48550/arXiv.2401.04131 arXiv:2401.04131 [cs]

Coşku Acay, Rolph Recto, Joshua Gancher, Andrew C. Myers, and Elaine Shi. 2021. Viaduct: An Extensible, Optimizing Compiler for Secure Distributed Programs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 740–755. https://doi.org/10.1145/3453483.3454074

Amal Ahmed. 2006. Step-Indexed Syntactic Logical Relations for Recursive and Quantified Types. In *Programming Languages and Systems (Lecture Notes in Computer Science)*, Peter Sestoft (Ed.). Springer, Berlin, Heidelberg, 69–83. https://doi.org/10.1007/11693024_6

Yuyan Bao, Kirshanthan Sundararajah, Raghav Malik, Qianchuan Ye, Christopher Wagner, Nouraldin Jaber, Fei Wang, Mohammad Hassan Ameri, Donghang Lu, Alexander Seto, Benjamin Delaware, Roopsha Samanta, Aniket Kate, Christina Garman, Jeremiah Blocki, Pierre-David Letourneau, Benoit Meister, Jonathan Springer, Tiark Rompf, and Milind Kulkarni. 2021. HACCLE: Metaprogramming for Secure Multi-Party Computation. In *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 2021)*. Association for Computing Machinery, New York, NY, USA, 130–143. https://doi.org/10.1145/3486609.3487205

Gilles Barthe, Gustavo Betarte, Juan Campo, Carlos Luna, and David Pichardie. 2014. System-Level Non-Interference for Constant-Time Cryptography. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*. Association for Computing Machinery, Scottsdale, Arizona, USA, 1267–1279. https://doi.org/10.1145/2660267.2660283

Gilles Barthe, Gustavo Betarte, Juan Diego Campo, and Carlos Luna. 2019. System-Level Non-Interference of Constant-Time Cryptography. Part I: Model. *Journal of Automated Reasoning* 63, 1 (June 2019), 1–51. https://doi.org/10.1007/s10817-017-9441-5

Amos Beimel. 2011. Secret-Sharing Schemes: A Survey. In *Coding and Cryptology (Lecture Notes in Computer Science)*, Yeow Meng Chee, Zhenbo Guo, San Ling, Fengjing Shao, Yuansheng Tang, Huaxiong Wang, and Chaoping Xing (Eds.). Springer, Berlin, Heidelberg, 11–46. https://doi.org/10.1007/978-3-642-20901-7_2

Niklas Büscher, Daniel Demmler, Stefan Katzenbeisser, David Kretzmer, and Thomas Schneider. 2018. HyCC: Compilation of Hybrid Protocols for Practical Secure Computation. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. ACM, New York, NY, USA, 847–861. https://doi.org/10.1145/3243734.3243786

Sunjay Cauligi, Gary Soeller, Brian Johannesmeyer, Fraser Brown, Riad S. Wahby, John Renner, Benjamin Grégoire, Gilles Barthe, Ranjit Jhala, and Deian Stefan. 2019. FaCT: A DSL for Timing-Sensitive Computation. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. Association for Computing Machinery, Phoenix, AZ, USA, 174–189. https://doi.org/10.1145/3314221.3314605

Meghan Cowan, Deeksha Dangwal, Armin Alaghi, Caroline Trippel, Vincent T. Lee, and Brandon Reagen. 2021. Porcupine: A Synthesizing Compiler for Vectorized Homomorphic Encryption. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 375–389. https://doi.org/10.1145/3453483.3454050

David Darais, Ian Sweet, Chang Liu, and Michael Hicks. 2020. A Language for Probabilistically Oblivious Computation. *Proceedings of the ACM on Programming Languages* 4, POPL (Jan. 2020), 1–31. https://doi.org/10.1145/3371118 arXiv:1711.09305

Roshan Dathathri, Blagovesta Kostova, Olli Saarikivi, Wei Dai, Kim Laine, and Madan Musuvathi. 2020. EVA: An Encrypted Vector Arithmetic Language and Compiler for Efficient Homomorphic Computation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 546–561. https://doi.org/10.1145/3385412.3386023

Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems (Lecture Notes in Computer Science)*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer, Berlin, Heidelberg, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

David Evans, Vladimir Kolesnikov, and Mike Rosulek. 2018. A Pragmatic Introduction to Secure Multi-Party Computation. *Foundations and Trends® in Privacy and Security* 2, 2-3 (2018), 70–246. https://doi.org/10.1561/3300000019

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. *ACM SIGPLAN Notices* 28, 6 (June 1993), 237–247. https://doi.org/10.1145/173262.155113

Craig Gentry. 2009. Fully Homomorphic Encryption Using Ideal Lattices. In *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing (STOC '09)*. Association for Computing Machinery, New York, NY, USA, 169–178. https://doi.org/10.1145/1536414.1536440

J. A. Goguen and J. Meseguer. 1982. Security Policies and Security Models. In *1982 IEEE Symposium on Security and Privacy*. 11–11. https://doi.org/10.1109/SP.1982.10014

O. Goldreich, S. Micali, and A. Wigderson. 1987. How to Play ANY Mental Game. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing (STOC '87)*. Association for Computing Machinery, New York, New York, USA, 218–229. https://doi.org/10.1145/28395.28420

M. Hastings, B. Hemenway, D. Noble, and S. Zdancewic. 2019. SoK: General Purpose Compilers for Secure Multi-Party Computation. In *2019 2019 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 479–496. https://doi.org/10.1109/SP.2019.00028

Carmit Hazay and Yehuda Lindell. 2010. *Efficient Secure Two-Party Protocols: Techniques and Constructions*. Springer, Berlin ; London.

Matthew E Hoekstra. 2015. Intel SGX for Dummies (Intel SGX Design Objectives). https://www.intel.com/content/www/us/en/develop/blogs/protecting-application-secrets-with-intel-sgx.html

Peeter Laud and Liina Kamm (Eds.). 2015. *Applications of Secure Multiparty Computation*. Number volume 13 in Cryptology and Information Security Series. IOS Press, Amsterdam, Netherlands.

Chang Liu, Michael Hicks, and Elaine Shi. 2013. Memory Trace Oblivious Program Execution. In *2013 IEEE 26th Computer Security Foundations Symposium*. 51–65. https://doi.org/10.1109/CSF.2013.11

C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi. 2015. ObliVM: A Programming Framework for Secure Computation. In *2015 IEEE Symposium on Security and Privacy*. 359–376. https://doi.org/10.1109/SP.2015.29

Raghav Malik, Kabir Sheth, and Milind Kulkarni. 2023. Coyote: A Compiler for Vectorizing Encrypted Arithmetic Circuits. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 118–133. https://doi.org/10.1145/3582016.3582057

Raghav Malik, Vidush Singhal, Benjamin Gottfried, and Milind Kulkarni. 2021. Vectorized Secure Evaluation of Decision Forests. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 1049–1063. https://doi.org/10.1145/3453483.3454094

Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. 2004. Fairplay - a Secure Two-Party Computation System. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13 (SSYM'04)*. USENIX Association, USA, 20.

Ueli Maurer. 2006. Secure Multi-Party Computation Made Simple. *Discrete Applied Mathematics* 154, 2 (Feb. 2006), 370–381. https://doi.org/10.1016/j.dam.2005.03.020

A. Rastogi, M. A. Hammer, and M. Hicks. 2014. Wysteria: A Programming Language for Generic, Mixed-Mode Multiparty Computations. In *2014 IEEE Symposium on Security and Privacy*. 655–670. https://doi.org/10.1109/SP.2014.48

Aseem Rastogi, Nikhil Swamy, and Michael Hicks. 2019. Wys*: A DSL for Verified Secure Multi-Party Computations. In *Principles of Security and Trust (Lecture Notes in Computer Science)*, Flemming Nielson and David Sands (Eds.). Springer International Publishing, 99–122. https://doi.org/10.1007/978-3-030-17138-4_5

A. Sabelfeld and A.C. Myers. 2003. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications* 21, 1 (Jan. 2003), 5–19. https://doi.org/10.1109/JSAC.2002.806121

Ian Sweet, David Darais, David Heath, William Harris, Ryan Estes, and Michael Hicks. 2023. Symphony: Expressive Secure Multiparty Computation with Coordination. *The Art, Science, and Engineering of Programming* 7, 3 (Feb. 2023), 14:1–14:55. https://doi.org/10.22152/programming-journal.org/2023/7/14

Alexander Viand, Patrick Jattke, Miro Haller, and Anwar Hithnawi. 2023. HECO: Fully Homomorphic Encryption Compiler. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, USA, 4715–4732. https://www.usenix.org/conference/usenixsecurity23/presentation/viand

Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. 2016. EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit.

Jean Yang, Kuat Yessenov, and Armando Solar-Lezama. 2012. A Language for Automatically Enforcing Privacy Policies. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL '12*. ACM Press, Philadelphia, PA, USA, 85. https://doi.org/10.1145/2103656.2103669

Andrew C. Yao. 1982. Protocols for Secure Computations. In *23rd Annual Symposium on Foundations of Computer Science (Sfcs 1982)*. 160–164. https://doi.org/10.1109/SFCS.1982.38

Qianchuan Ye and Benjamin Delaware. 2022. Oblivious Algebraic Data Types. *Proceedings of the ACM on Programming Languages* 6, POPL (Jan. 2022), 51:1–51:29. https://doi.org/10.1145/3498713

Qianchuan Ye and Benjamin Delaware. 2023. Taype: A Policy-Agnostic Language for Oblivious Computation. *Proceedings of the ACM on Programming Languages* 7, PLDI (June 2023), 147:1001–147:1025. https://doi.org/10.1145/3591261

Qianchuan Ye and Benjamin Delaware. 2024. Taypsi: Static Enforcement of Privacy Policies for Policy-Agnostic Oblivious Computation: OOPSLA24 Artifact. Zenodo. https://doi.org/10.5281/zenodo.10701642

Samee Zahur and David Evans. 2015. *Obliv-C: A Language for Extensible Data-Oblivious Computation.* Technical Report 1153. https://eprint.iacr.org/2015/1153

Stephan Arthur Zdancewic. 2002. *Programming Languages for Information Security.* Ph. D. Dissertation. Cornell University, USA.

Yihua Zhang, Aaron Steele, and Marina Blanton. 2013. PICCO: A General-Purpose Compiler for Private Distributed Computation. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS '13)*. Association for Computing Machinery, New York, NY, USA, 813–826. https://doi.org/10.1145/2508859.2516752

Received 21-OCT-2023; accepted 2024-02-24